Strategies for Quality and Performance Improvement of Hardware Verification and Synthesis Algorithms

Mahmoud A. M. S. Elbayoumi

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Engineering

December 9, 2014

Blacksburg, Virginia

Keywords: Binary Decision Diagram, Satisfiability(SAT),VLSI, Bounded Model Checking, logic synthesis, Multi-threading

Strategies for Performance and Quality Improvement of Hardware Verification and Synthesis Algorithms

Mahmoud A. M. S. Elbayoumi

(ABSTRACT)

According to Moore's law, Integrated Chips (IC) doubles its capacity every 18 months. This causes an exponential increase of the available area, and hence,the complexity of modern digital designs. This consistent enormous gross challenges different research areas in Electronic Design Automation (EDA). Thus, various EDA applications such as equivalence checking, model checking, Automatic Test Pattern Generation (ATPG), functional Bi-decomposition, and technology mapping need to keep pace with these challenges. In this thesis, we are concerned with improving the quality and performance of different EDA algorithms particularly in area of hardware verification and synthesis.

First, we introduce algorithms to manipulate Reduced Ordered Binary Decision Diagrams (ROBDD) on multi-core machines. In order to perform multiple BDD operations concurrently, our algorithm uses a breadth-first search (BFS). As ROBDD algorithms are memory-intensive, maintaining locality of data is an important issue. Therefore, we propose the usage of Hopscotch hashing technique for both Unique Table and BFS Queues to improve the construction time of ROBDD on the parallel platform. Hopscotch hashing technique not only improves the locality of the manipulating data, but also provides a way to cache recently performed BDD operation. Consequently, The time and space usage can be traded off.

Secondly, we used static implications to enhance the performance of SAT-based Bounded Model Checking (BMC) problem. we propose a parallel deduction engine to efficiently utilize low-cost off-shelf multi-core processors to compute the implications. With this engine, we can significantly reduce the computational processing time in analyzing the deduced implications. Secondly, we formulate the clause filter problem as an elegant set-covering problem. Thirdly, we propose a novel greedy algorithm based on the Johnsons algorithm to find the optimal set of clauses that would accelerate BMC solution.

Thirdly, we proposed a novel synthesis paradigm to achieve timing-closure called *Timing-Aware CUt Enumeration (TACUE)*. In TACUE, optimization is conducted through three aspects: First, we propose a new divide-and-conquer strategy that generates multiple sub-cuts on the critical parts of the circuit. Secondly, cut enumeration have been applied in two cutting strategies. In the *topology-aware* cutting strategy, we preserve the general topology of the circuit by applying TACUE in only self-contained cuts. Meanwhile, the *topology-masking* cutting strategy investigates circuit cuts beyond their current topology. Thirdly, we proposed an efficient parallel synthesis framework to reduce computation time for synthesizing TACUE sub-cuts. We conducted experiments on large and difficult industrial benchmarks.

Finally, we proposed the first scalable SAT-based approaches for Observability Dont Care (ODC) clock gating. Moreover we intelligently choose those inductive invariants candidates such that their validation will benefit the purpose in clock-gating-based low-power design.

To the soul of my father Atef Elbayoumi

Acknowledgments

I extend my gratitude to my advisor, Professor Michael Hsiao for his consistent guidance and continuous support throughout my research. I am so grateful to have opportunities discussing my research works with him and be able to obtain remarkable guidance, high expertise in the digital electronic design automation. Without the help from him, this dissertation would not have been completed or written.

I would like to thank Dr. Sedki M. Riad, Dr. Mark M. Shimozono, Dr. Sandeep K. Shukla, Dr. Mustafa Y. El-Nainay and Dr. Chao Wang to serve on my committee and spend precious time on the dissertation review and oral presentation. I would like also to thank all the PROACITVE members for their suggestions on my research and dissertation.

Last but not the least, I would like to express my deep gratitude to my family members: parents Atef Elbayoumi and Ahlam Elsayed, and my wife Lamiaa Elsayed, for their love and constant support, without which I would never make this work possible. I sincerely appreciate all the help.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

According to Moore's law, the feature size of transistors is shrinking every 18 months. Meanwhile, the complexity of the logic design is increasing exponentially. These advances have challenged circuits developer to ensure that their products are synthesized efficiently and are free of bugs and defects.

Binary Decision Diagrams (BDDs)-based and Satisfiability (SAT)-based algorithms have been used in logic circuits synthesis, testing and verification [1], [2], [3], [4]. In this chapter, we introduce the motivation for our work, thesis objectives, outlines and publications.

1.1 Motivations

In this section we will discuss the motivation of our work. First, we will discuss the motivation for utilizing multi-core platforms. Secondly, the motivation for concurrent BDD package will be discussed. thirdly, we will discuss the motivation for SAT-based bounded model checking. In addition, we will discuss the motivation for timing aware and low-power logic synthesis.

1.1.1 Motivation of parallelization

Nowadays, computing has seen a steady shift toward parallel computing. This shift is mainly because of three reasons [5]. First, operating frequency has hit a wall. This is because increasing the operating frequency may induce excessive power consumption. Secondly, Instruction Level Parallelism (ILP) has reached its limit due to constraints limited by data, control, and structural resources. Thirdly, memory manufacturers tend to trade delay with storage; thus, the degradation due to memory access time outweighs any improvement in processor operating frequency. As a result, manufacturers are interested in looking for alternatives to these traditional approaches for increasing performance. Thus, increasing the number of processors (cores) on a die to improve the performance is one of the mainstream to boost the performance[6]. This lead to a low-cost (but powerful) parallel computing platform. Accordingly, these platform need to be utilized efficiently to leverage the opportunities in many sequential EDA algorithms [7].

1.1.2 Motivation for Binary Decision Diagrams

Boolean algebra is an essential mathematical tool in various fields of computer science and engineering. Many problems can be modeled as sequence of Boolean expressions. For example, VLSI design automation [8], artificial intelligence [9], and network system analysis [10] are typical examples of these problems. Consequently, the efficient representation and manipulation of Boolean function are requirements in many algorithms in various area especially in VLSI design automation.

Among many Boolean representations [11], Binary Decision Diagrams (BDDs) are considered one of the most efficient ways for representation and manipulation of Boolean functions [11]. Since its introduction in 1986 [12], efficient constructions of Reduced Ordered Binary Decision Diagrams (ROBDD) have penetrated many areas of computer aided VLSI design, including fault simulation [1], circuit synthesis [2], Automatic Test Pattern Generation (ATPG) [3], circuit verification [4], just to name a few. Three major approaches have been proposed to construct ROBDD: depth-first search (DFS) [13], breadth-first search (BFS) [14] and hybrid BFS-DFS [15]. Although DFS has a low associated memory overheads and large potential to be optimized (i.e., by using Computed Tables (CT)), it has poor memory locality. Meanwhile, BFS construction approach preserve memory locality and it has a large potential to be parallelized; however, it has associated overheads in storing temporary nodes (stored in queues).

Recently, concurrent hashing techniques (e.g., Hopscotch hashing [16]) have been proposed to guarantee efficient constant lookup and deletion times in the hash table. In addition, it shows a superior performance on traditional hashing techniques; i.e., chained hashing, linear hashing and Cuckoo hashing [17], even when the hash table is 90% full. It depends mainly on keeping those nodes, hashed to the same bucket, in a restricted nearby space of the memory. In doing so, it has benefits of low access time of the main memory cache, and guarantees constant worst-case lookup time. This can play a critical role in improving the performance in the access of BDD nodes, which are typically stored and indexed via hash tables. Thus, in a sense, while construction algorithms for BDDs are extremely difficult to parallelize, we exploit concurrency in hashing algorithms, with tremendous payoffs. As we have new architectures, we need to investigate and develop new algorithm to handle BDDs on these architecture to efficiently utilize the available off shelf devices.

1.1.3 Motivation for SAT-based bounded model checking

Two classes of approaches have been proposed to verify IC designs: simulation-based and formal verification techniques. While simulation-based verification can better accommodate large designs, it cannot offer guarantees [18]. On the other hand, formal methods explore all corners in theory and thus are complete. In that sense, formal verification can be regarded as a promising alternative for certain types of designs and properties.

Model checking is a formal verification method, which computes and traverses the reachable states of the design to determine whether the target property is upheld. In 1999, bounded model checking (BMC) was proposed [19] based on a Satisfiability (SAT) formulation and gives model checking a great push forward. BMC has enjoyed considerable success with the employment of propositional SAT solvers in recent years [20]. It has been widely applied in bug hunting and property checking within a bounded sequential depth [21]. In Chapter 4, we will presents a formulation for static implication selection to boost the performance of BMC problem.

1.1.4 Motivation for timing-aware logic synthesis

Logic synthesis is the automated generation of an optimized logic networks (in terms of delay [22], area and/or power [23]) from another unoptimized/sub-optimized logic network. Over the years, many researchers have been interested in optimizing certain dedicated hardware components. For example, Sklyarov *et al.* [24] had proposed a novel technique for synthesis parallel hierarchical finite state machine; Roy *et al.* [25] had proposed a customizable prefix graph structures that yield adders with optimal performance-area trade-off. Other researchers had developed various synthesis algorithms for dedicated platforms. For example, Uma *et al.* [26] had explored constraint synthesis optimization technique for targeted FPGA device. Another research aims toward building general methodologies that is capable of efficiently synthesizing both control and data-path components [27].

This thesis fits in the last group, that is, general timing-aware synthesis. In other words, we do not target a specific hardware structure or platform. Instead, we propose a general framework for delay-optimization logic synthesis.

1.1.5 Motivation for low-power logic synthesis

Power consumption is one of the major considerations in modern chip design. This was not only driven by the stringent power budget in mobile and tablet industry [28], but also driven by the need to reduce power consumption in high end servers as well. This need comes as the fact that increasing power consumption of computing systems has started to limit their performance growth [29].

The two major sources of power consumption in CMOS circuits are static and dynamic power consumption. According to data provided by large chip manufacturer [29], devices such as CPUs could have their dynamic power consumption contributing to 70% of the total power. Various power reduction techniques have been proposed and implemented in all levels of the computing system, from software and architecture levels [30] to circuit levels [31]. At the circuit level, many clock-gating and power-gating techniques had been proposed [32], [33]-[34]. In Chapter 6, we will present a novel scalable invariant-directed power aware synthesis technique.

1.2 Dissertation Objectives

Dissertation objectives are as follows:

- Improve BDDs construction time with concurrent Hopscotch Hashing techniques.
- Formulate a criterion for filtering static Implications to help SAT-BMC problem.
- propose a parallel implication deduction engine.
- propose an timing-Aware CUt Enumeration (TACUE) algorithm to improve timing closure in hard circuit instances.
- propose an efficient parallel synthesis framework for TACUE.
- propose the first method for SAT-based ODC power-aware synthesis.

1.3 Dissertation Outlines

This thesis is organized as follows:

- Chapter 2 introduces the necessary background for the techniques used in this dissertation. It introduces the fundamentals of BDDs, including its concepts, algorithms. In addition, SAT-based Bounded Model Checking (BMC) is introduced. Static logic implications and their various types will also presented. Preliminaries for timing-aware and low-power logic synthesis is introduced.

- Chapter 3 introduces the detailed BDDs construction with Hopscotch hashing. We will introduce efficient resizing and garbage collector mechanism. We will presents The results for these algorithms.

- Chapter 4 introduces the details of filtering algorithm of static implications. In addition, it will present the parallel framework for generating static implications. Finally, the results will be presented.

- Chapter 5 introduces the details of our timing-aware synthesis algorithm. We apply TACUE in two different cutting strategies. In addition, we will present the parallel framework for generating the synthesized sub-cuts. Finally, the results will be presented.

- Chapter 6 introduces the details of our Low-power synthesis algorithm. First, we introduce our problem and our proposed algorithm related concepts. Secondly, our scalable algorithm along with heuristics is introduced. Thirdly, Results and experiments are presented. Finally, our work is concluded.

- Chapter 7 presents the future work for this thesis. We propose to extend algorithms presented in Chapter 3 to very large BDDs and partition BDDs. In addition, we propose to filter Potential Inductive invariants to assist SAT solver to efficiently solve BMC problems. Finally, we propose to apply both BDDs and SAT techniques we have investigated to logic synthesis area.

1.4 Publications

As of today, the work presented in this dissertation has resulted in the following publications:

- Mahmoud Elbayoumi, Michael Hsiao and Mustafa ElNainay , Novel SAT-based Invariant-Directed Low-Power Synthesis, ISQED 2015, March 2015, Santa Clara, CA, USA. (Accepted)
- Mahmoud Elbayoumi, Mihir Choudhury, Victor N. Kravets, Andrew Sullivan, Michael S. Hsiao, Mustafa Y. ElNainay, TACUE: A Timing-Aware Cuts Enumeration Algorithm for Parallel Synthesis. DAC 2014, San Francisco, CA, USA.
- Mahmoud Elbayoumi, Michael Hsiao and Mustafa ElNainay , Selecting Critical Implications with Set Covering Formulation for SAT-based Bounded Model Checking, ICCD 2013, Oct. 2013, Asheville, NC, USA.
- Mahmoud Elbayoumi, Michael Hsiao and Mustafa ElNainay , Set-Cover-based Critical Implications Selection to Improve SAT-based Bounded Model Checking Extended Abstract. GLSVLSI 2013, May 2013, Paris, France.
- Mahmoud Elbayoumi, Michael Hsiao and Mustafa ElNainay , A Novel Concurrent Cache-friendly Binary Decision Diagram Construction For Multi-Core Platforms, ACM Design, Automation & Test in Europe (DATE 2013), March 2013, Grenoble, France.

Chapter 2

Background

In this chapter, we present preliminaries about work accomplished in the dissertation. First, We define Boolean Functions. Secondly, we introduce Binary Decision Diagrams (BDDs). Moreover, we introduce the concept of static implications and inductive invariants. Next, we introduce satisfiability (SAT)-based Bounded Model Checking (BMC) and Property-directed Model checking. Furthermore, BDD bi-decomposition and time-driven logic bi-decomposition are presented as they been utilized as synthesis optimization engine for sub-cuts produced by TACUE. In addition, dominant cuts is used in *topology-aware* cutting strategy and in our *low-power* synthesis algorithm. Finally, we introduce clock-gating synthesis and rarity random simulation.

2.1 Boolean Functions

The Boolean function of n variables is defined as [11]:

$$f(x) : \mathbf{B}^n \mapsto \mathbf{B} \tag{2.1}$$

Where $\mathbf{B} = \{0, 1\}$. The set $\mathbf{B}$ is the set of Boolean values whose elements are sometimes referred to a 'true' or 'false' instead of 1 and 0. For any value of n, we have always 2^n possible Boolean functions. Boolean functions are used to express relations between different Boolean variables. A Boolean expression is composed of Boolean variables, $x_1, x_2, \cdots, x_n$, Boolean values; 'true(1)' and 'false(0)' and also the Boolean operators: conjunction $\wedge$, disjunction $\vee$, negation $\neg$, implication $\Rightarrow$, and bi-implication $\Leftrightarrow$. An example of Boolean expressions is shown in Eq. (2.2).

$$x_1 \Leftrightarrow x_2 \vee \neg\, x_3 \tag{2.2}$$

A Boolean expression describes how to determine a Boolean output value based on logical calculations on some Boolean variables and values. The sequence of assignments of values for Boolean variables is referred to as a truth assignment and is written as Eq. (2.3).

$$[1/x_1, 0/x_2, 1/x_3] \tag{2.3}$$

Eq. (2.3) means that Boolean value 'true(1)' is assigned to x_1 and x_3, and Boolean value 'false(0)' is assigned to x_1. For example, Eq. (2.2) is evaluated to 0 for the truth assignment in Eq. (2.3). Meanwhile, it is evaluated to 1 for assignment $[1/x_1, 0/x_2, 0/x_3]$.

Two Boolean functions are said to be equivalent, if they yield the same output for all truth assignment. A Boolean function is said to be *tautology*, if it is evaluated to 1 for all truth assignment. On contrary, *contradiction* is a Boolean function whose output always is evaluated to 0 for all truth assignments. A *satisfiable* Boolean function is the one whose output is 1 for at least one truth assignment.

Testing Boolean function for satisfiability, and checking for equivalence is a common procedure in EDA. These problems involves extensive handling of Boolean functions. classical methods have been used to represent Boolean functions (i.e., truth table, Karnaugh maps, and prime cubes). However, their time and memory requirements are growing exponentially with number of input for many common functions [12]. Moreover, many of these classical methods may produce multiple representation for the same Boolean function. In addition, in cases, the size is not exponential, simple operations (i.e., negation) would produce problems with exponential size [12]. Thus, many of the EDA tasks are requiring a solution to NP-complete or co-NP-complete problems [12]. Accordingly, There is a need for an efficient way to represents and manipulates Boolean functions; such that, the size of the representation is reasonable and exponential computation will be avoided in most of cases.

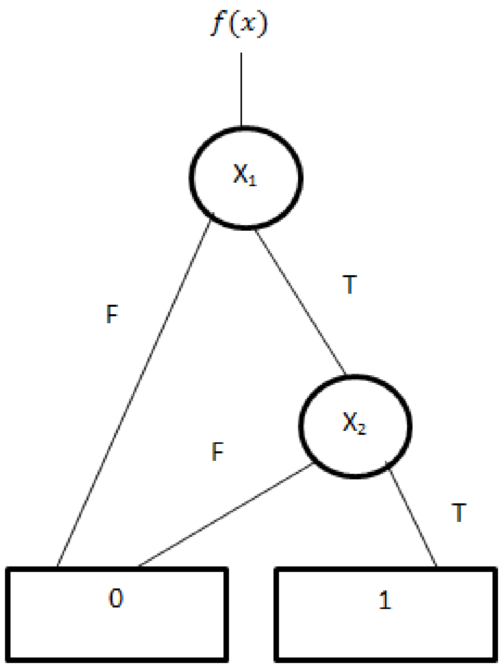

Figure 2.1: Illustration of ROBDD.

2.2 Binary Decision Diagrams

BDDs are graphs represents a Boolean function as depicted in Fig. 2.1. ROBDD of $f = x_1.x_2$. The is one node for each level. node x_1 is in level 2 and node x_2 is in level 1. There is two terminal nodes "0" and "1". Each node has two edges; Then (or **true**) edge, and Else (or **false**) edge. As mentioned before, BDDs are first introduced by Lee in 1959 [35]. However, they did not become popular until Bryant proposed efficient algorithm to manipulate them [12]. Since then, BDDs and their variants [36] have been extensively exploited in various applications in many area of VLSI automation.

A BDD is a directed acyclic graph with two terminal nodes; so called 0-terminal node and

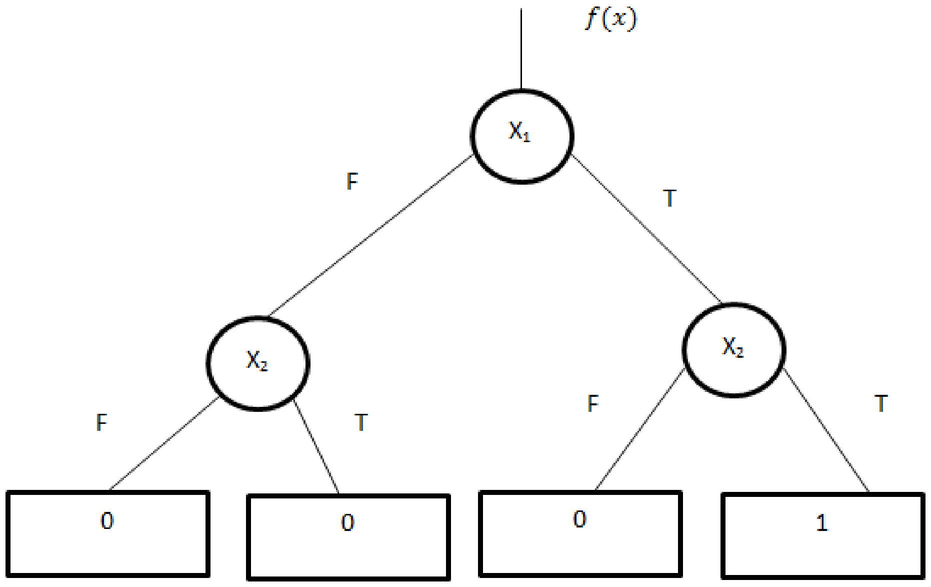

Figure 2.2: The original tree for $f = x_1.x_2$.

1-terminal node [13]. Each non-terminal node has an index to identity an input variable of the Boolean function and has two outgoing edges, called the 0-edge and 1-edge (see Fig. 2.1).

An Ordered BDD (OBDD) is a BDD where input variables appear in a fixed order in all paths of the graph and no variable appears more than once in a path. In this dissertation, we will use natural numbers $1, 2, \cdots$ for the indices of the input variables, and every non-terminal node as an index less than its descendant nodes.

A compact OBDD is derived by reducing a binary tree graph as depicted in 2.2. In the binary tree, 0-terminals and 1-terminals represent logic values 0 and 1, and each node represents the *Shannon's expansion* of the Boolean function [11]:

Usually BDDs are constructed using a ternary operator so called If-Then-Else (ITE) operator. It is defined as follows

$$ITE(x, y, z) = x.y + \overline{x}.z \tag{2.4}$$

ITE operator is applied recursively as in Eq. 2.5 to obtain any BDD:

$$ITE(f, g, h) = ITE(x_i, ITE(f_{x_i}, g_{x_i}, h_{x_i}), ITE(f_{\overline{x_i}}, g_{\overline{x_i}}, h_{\overline{x_i}})) \tag{2.5}$$

Where $f_{x_i}, g_{x_i}, h_{x_i}$ are the positive cofactor of f, g and h with respect to x_i. $f_{\overline{x_i}}, g_{\overline{x_i}}, h_{\overline{x_i}}$ are the negative cofactor of f, g and h with respect to x_i. The ITE operator is applied recursively until it reach one of the base case as follows (Eq. 2.6):

$$\begin{aligned} ITE(f, 1, 0) &= f \\ ITE(f, 0, 1) &= \overline{f} \\ ITE(1, f, g) &= f \\ ITE(0, f, g) &= g \\ ITE(f, g, g) &= g \end{aligned} \tag{2.6}$$

Where $\overline{f}$ is the complement of function f. The following reduction rules give a Reduced Ordered BDD (ROBDD):

- Eliminate all the redundant nodes whose two edges point to the same node.

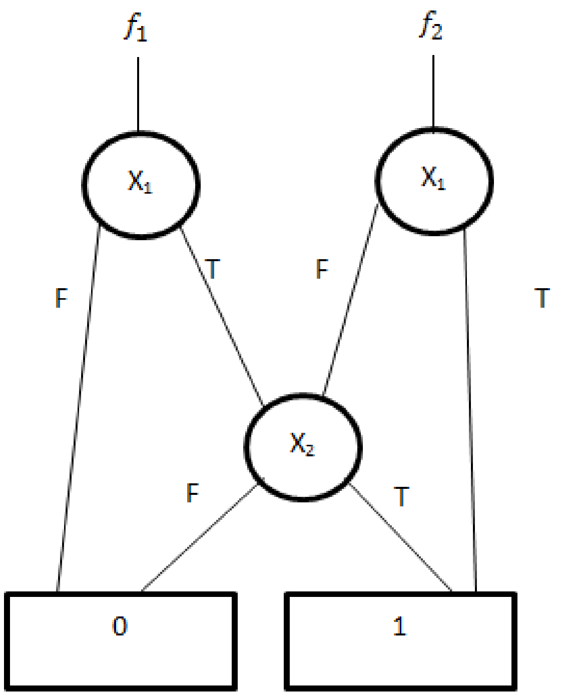

Figure 2.3: Two shared BDDs. $f_1 = x_1.x_2$, $f_2 = x_1 + x_2$

- share all the equivalent sub-graphs.

A set of BDDs representing multiple functions can be united into a graph that consists of BDDs [37] sharing their sub-graphs with each other as shown in Fig. 2.3. This sharing property saves time and space to have duplicate BDDs. When the isomorphic sub-graphs are completely shared, two equivalent nodes never coexist.

In the shared BDD environment, the following advantages are obtained:

- Equivalence checking can be performed immediately by just looking at the root nodes.
- reusing duplicate BDD can save time and memory as the operation will be converted to copy a pointer to the root node.

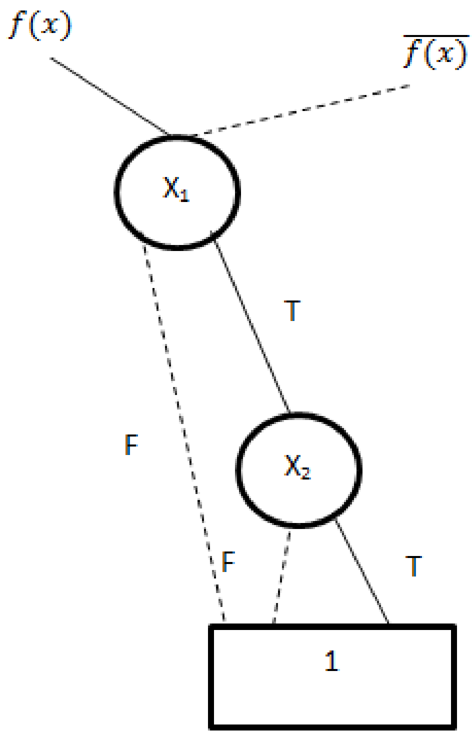

Figure 2.4: The original tree for $f = x_1.x_2$.

2.2.1 Complemented Edges

In order to reduce memory, complemented edges are used. It is used to represents the complement of the function by complementing the pointing value to this function. This is because ROBDDs of a function f and its complement $\overline{f}$ are only differ in one aspect: The values of their sink node are interchanged. Rudnell *et al.* [13] showed that by restricting the complemented edge at the *Else* edge only, the ROBDDs become canonical. Fig. 2.4 depicts the ROBDD of $f = x_1.x_2$ and $\overline{f}$ with the complemented edge.

2.2.2 Depth-first Search vs. Breadth-First search

Two major approaches have been proposed to construct ROBDD; Depth-First (DF) and Breadth-First (BF). While DF has low associated memory overheads and large potential to be optimized (i.e.; by using computed tables), it has low potential to be parallelized. Meanwhile, BF construction approach has a large potential to be parallelized; however, it associates with overheads in storing temporary nodes (stored in queues).

2.2.3 Unique and Computed Tables

Almost all BDD packages uses hash tables. Hash tables are used for two purposes. First, they guarantee strong canonicity of the constructed BDDs. In other words, hash tables will not allow duplicated nodes, and hence sub-graphs; so, every function will be represented by single node. Second, they are used as a software cache for newly calculated subfunctions. In other words, packages usually need an additional hash table (usually called Computed Table (CT)) to store the result of recent BDD operation. So, before any BDD operation is evaluated, it is checked if its result is stored in CT. In the following sections, We will present an introduction to static implications and inductive invariants.

2.3 Static implications and inductive invariants

Static implication and inductive invariants is introduced in this section. In addition, the concept of implication graph and filtering of static implication is presented.

2.3.1 Static Logic Implications

Static logic implications [38], [39], [40] describe relationship between different gates in a CUV that hold for all the states of the sequential circuits (whether it is legal or illegal states) . It can be obtained by asserting and propagating logic '0' and '1', to every gate in CUV. We have different types of static logic implications, direct, indirect, extended Backward, Extended Forward and Justification Frontiers Implications. We define $impl[G, v, t]$ as the set of all implications resulting by assigning a value v to gate G in time frame t, where $v \in \{0, 1\}$. In addition, $(G_1, v) \rightarrow (G_2, w, t)$ means assigning gate G_1 value v implies a value w on gate G_2 in time frame t.

Direct and Indirect Implications

A direct implication is the result of assigning the value on the target gate in their direct fan-out gates (forward implications) and direct fan-in gates (backward implications). They are based on controlling values of gates. The controlling values The direct implications are of two types: 1) forward implications, and 2) backward implications. direct implications could be formulated mathematically as:

$$impl[N, v] \equiv impl[N, v] \cup [LogicSimulate(impl[N, v])] \tag{2.7}$$

Where $LogicSimulate()$ refers to performing logic simulation with direct implications asserted on the gates.

Extended Backward Implications

Extended Backward Learning (EBL) [41] is performed on unjustified implicants (which are gates whose inputs are not sufficient to justify the output) in the implication list of target gate (which is the gate we need to find its implications). The extended backward implications are computed by considering both the target gate N and the unjustified output specified gates in the implication list of the target gate. There are generally four cases depending on the gate type (**AND, OR, XOR, XNOR**) of the gate where the target signal is the output [42].

Extended Forward Implications

Extended Forward Implications is proposed in [43] to identify un-testable faults for ATPG. The motivation for Extended Forward Learning (EFL) is to push the envelope of implication of I-Frontier gates (gates that don't have sufficient values at its inputs to determine its output). In other words, EFL tries to extend implications beyond the point forward implications reach.

Justification Frontiers Implications

Justification Frontier (JFron) Learning is proposed in [39]. This technique extends the EBL technique by justifying an unjustified gate to its Immediate Justification Frontier. The Immediate Justification Frontier (IJF) of a gate's assignment can be obtained by taking the immediate input justification scenarios for that assignment and recursively extending them backward until all the gates achieve their non-controlling values or a primary input is reached.

Contrapositive principle

Contrapositive principle is used to deduce more implications from CUV. It states that if $x \to y$ is true, then $\overline{y} \to \overline{x}$ must also be true. When representing an implication in clausal form, the contrapositive relation is automatically included.

2.3.2 Implication graph

An implication graph $G(V, E)$ is a digraph used to store logical implications among the set of nodes, V, and relations among nodes with directed edges, E. In the implication graph, every node represents one gate with one assignment (i.e., $a = 0$ is a node that represents gate a with value 0). Accordingly, as every gate can only set to either 0 or 1, thus $|V| = 2N$, where N is number of gates in the CUV. E is the set of all edges in G. $e \in E$ between nodes x and y iff $x \to y$.

2.3.3 Filtering of Static Logic Implications

The classification of constraint clauses can be regarded as a filter: only those that meet the criteria are taken and added as constraints to the formula. However, such simple filters may inadvertently leave out some key critical relations or fail to remove those less important clauses. As far as we know, there are two major work that are related to enhancing the performance of SAT-based BMC. One of the early work aims to tune SAT solvers presents in [44], [45]. Strichman presents many optimization techniques for BMC-specific application, one of them is related to the work presented in this dissertation, he used semi-symmetry property of the checked Conjunctive Normal Formula (CNF) (due to replication of the transitive relation function over multiple timeframes) to deduce more clauses. Our approach outperforms this technique in two aspects. First, we already duplicate the deduced implication through different timeframe without taking care of the initial state assignment and its corresponding cone of influence (contrary to [44]). Secondly, the previous technique does not provide a criterion for selecting important clauses.

Another work based on And-Inverter-Graph (AIG) is proposed in [46]. The authors present an algorithm that dynamically reduces the transition relation by detecting and merging equivalent nodes. Kuehlmann *et al.* [47] extend the approach by identifying more equivalent nodes using observability don't cares (ODCs). Case *et al.* [48] further extend this technique for sequential circuits by exploiting sequential ODCs. The main drawback in such approaches is that they cannot avoid a potentially large number of SAT solver calls. Also, the evaluation of ODCs can be computationally expensive.

2.3.4 Inductive invariants

Sequential circuits can be modeled as nite-state transition system $S : (i, x, I, T)$. Where I is the initial condition,$T(i, x, x')$ is the transition relation over a set of input variables i , x is the set of internal state variables and x' is the set of next state internal variables. A relation $P(x)$ is called an inductive invariant in S if there is not any sequence of states from I that could violate P. P does not necessarily need to satisfy T.

2.4 Model Checking

Model checking is a field of science (and engineering) concerned with deciding whether a design complies with its specifications. Algorithm [49], Inductive Model Checking [50], deductive [51] and symbolic state exploration techniques [52] have been used in the past. Inductive invariants and model checkers interact in various ways. For example, inductive invariant candidates could be used to strengthen the model checker itself [53], [54]. On the other hand, model checkers could be used to prove the validity of inductive invariant candidates. These proved inductive invariants could then be used to improve synthesis or help solving the verification problem [55]. We introduce below two types of model checking related to our work.

2.4.1 SAT-based Bounded Model Checking Problem

In Bounded Model Checking (BMC), the Circuit Under Verification (CUV) is checked against a certain property p on a k time-frame window. SAT-based Bounded Model Checking is performed by constructing the following Conjunctive Normal Formula (CNF) Boolean Formula (Eq. 2.8).

$$\varphi = I(s_0) \wedge \bigwedge_{0 \leq i < k} T(s_i, s_{i+1}) \wedge \parallel \phi \parallel_k \tag{2.8}$$

where: $T(s_i, s_{i+1})$ is the transitive relation; hence, $\bigwedge_{0 \leq i < k} T(s_i, s_{i+1})$ is k-timeframe Iterative Logic Array (ILA) of CUV; $I(s_0)$ is a legal initial state; $\parallel \phi \parallel_k$ is a monitor circuit for some property p (which we want to check). The monitor circuit is constructed such that φ will be satisfiable if and only if the CUV would violate p. Fig 2.5 depicts a typical BMC instance, where k unrolled transitions are shown , and the property assertions in each unrolled time frame.

Given a BMC instance φ, an implicit exploration of the reachable states to verify whether the property assertion is satisfiable within k frames is performed by a SAT solver. If φ is satisfiable, a trace (counter-example) of length less than or equal to k starting from the initial state is generated which exposes the violation of p in the CUV. Otherwise, the satisfiability of the instance indicates that no counter-example of length k from the given initial state exists. In this case, it is necessary to increase the bound of k (via incremental SAT-based BMC [56]). When k exceeds the sequential length of CUV and the instance is still un-satisfiable,

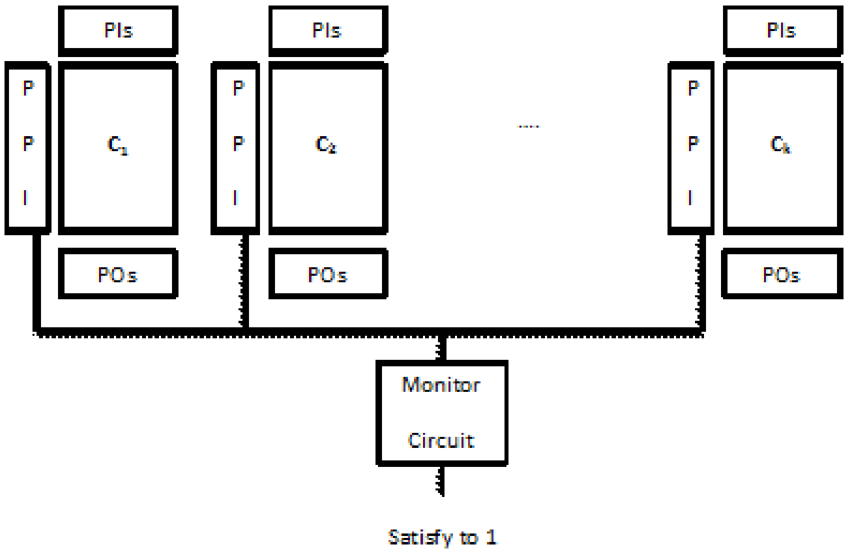

Figure 2.5: BMC illustration.

it can be concluded that the CUV holds p. To avoid unrolling all the way to the sequential depth when p is a true property, BMC with induction can be used [57].

2.4.2 Property-directed reachability model checking

Property-directed reachability (PDR), aka IC3, is a complete (and one of the fastest) model checker tool used in model checking [58] and sequential equivalence [59]. According to Hardware Model Checking Competition 2014 results (HWMCC14 [60]), many of the fastest (publically available model checkers) are based on IC3. This motivates us to use PDR as the underlying model checker in our algorithm. PDR strengthens a property P by incrementally generating inductive invariants that have to be valid if P is valid. It incrementally iterates until P is relative inductive to the set of generated invariants [61]. We have to note that any other complete model checker could be used in our proposed low-power algorithm in Chapter 6.

2.5 Logic synthesis techniques and dominant cuts

BDD bi-decomposition and time-driven logic bi-decomposition are utilized as synthesis optimization engine for sub-cuts produced by TACUE. In addition, dominant cuts is used in *topology-aware* cutting strategy. Thus, we briefly introduce them.

2.5.1 BDD Bi-decomposition

Boolean function bi-decomposition is pervasive in logic synthesis [62]. It consists of decomposing Boolean function $f(X)$ into the form of $f(X) = h(f_A(X_A, X_C), f_B(X_B, X_C))$, under variable partition $X = \{X_A|X_B|X_C\}$. The quality of bi-decomposition is mainly determined by the quality of variable partitions, as an optimal solution results in simpler subfunctions f_A and f_B [63], [64].

An efficient BDD bi-decomposition is proposed in [65]. It starts by building the BDD of each output. Then, it recursively decomposes each output BDD to two smaller logically related BDDs. However, the complexity of BDD bi-decomposition rests in achieving a good variable partition for the given logic function. Thus, we use a fast, scalable algorithm [66] for obtaining provably optimum variable partitions for bi-decomposition of Boolean functions by constructing an undirected graph called the blocking edge graph (BEG) [66].

2.5.2 Time-Driven Logic Bi-Decomposition

Time-driven logic bi-decomposition is proposed in [67]. It synthesizes a timing-aware circuit by first bi-decomposing the Boolean representation of the circuit, then it re-balances the functions using a tree-height reduction technique. Kravets *et al.* [68] had proposed a general symbolic decomposition template for logic synthesis that uses information-theoretical properties of a function to infer its decomposition patterns. Using this template, the decomposition is done in a Boolean domain unrestricted by the representation of a function, which enables superior implementation choices driven by additional technological constraints. Bi-decomposition technique in [67] is applied iteratively on the decomposed templates.

2.5.3 Vertex Dominator and Dominant Cuts

Definitions

Circuit graph: a circuit can be represented by a directed acyclic graph (DAG) $G(V, E)$ in which each gate is represented by a vertex $v \in V$ and each wire connecting two gates is represented by an edge $e \in E$ as depicted in Fig. 2.6. *Sink vertices*: The set of outputs SO of a circuit labeled as *sink vertices.* In Fig. 2.6, only vertex m is labeled as *sink vertex.*

Source vertices: The set of inputs of a circuit labeled as *Source vertices.* In Fig. 2.6, vertices a, b, c and d are *source vertices.*

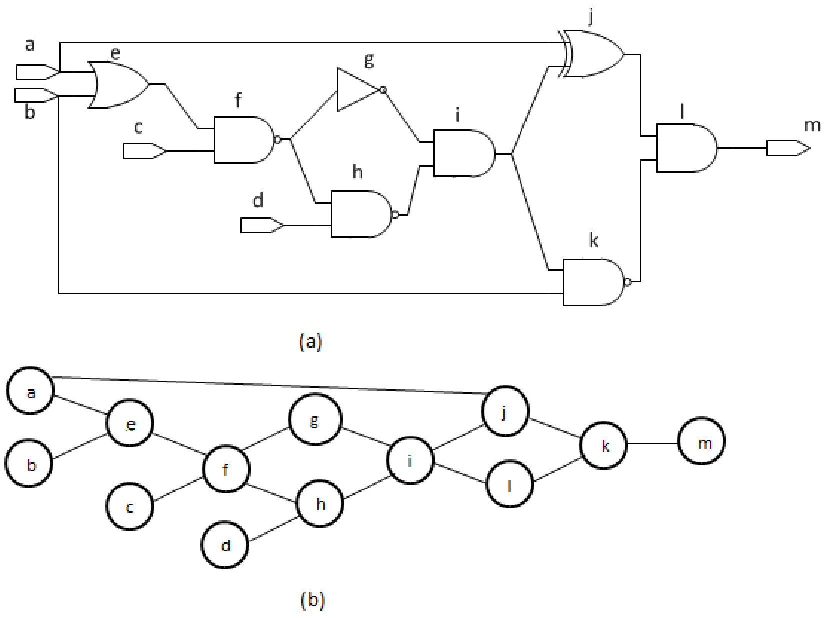

Figure 2.6: Circuit graph illustration.

Vertex dominator: a node $v \in V$ dominates a node $u \in V$ if every path from u to any node in SO must go through v.

Dominating set: a set $D(v) \subseteq V$ is called a dominating set for vertex v iff every vertex $u \in D$ is a vertex dominator of v.

Cut: a cut in a circuit is defined by a root vertex $r \in SO$ (a sink node) and a set of boundary vertices $S \subseteq V$, such that any path from an input to r must path through at least one of the vertices in S. The cut can be written as a 2-tuple (r, S).

Dominant cut: a cut $C = (a, S)$ in which all paths from S to a never pass through any vertex outside the fan-in cone of node a. Thus, dominant cut is a self-contained cut.

Fig. 2.7 depicts the algorithm for finding vertex dominator [69]. The algorithm starts with

```
1: CalDomSets()
2: **for all** node $a$ **do**
3:     $D(a) = \{a\}$
4: **end for**
5: **for** Level $i = N - 1$ **to** 1 **do**
6:     **for all** gate $g$ in level $i$ **do**
7:         calculate $D(g)$ as defined in Eq. 2.9
8:     **end for**
9: **end for**
```

Figure 2.7: Dominant sets calculation algorithm.

initializing all dominant sets (lines 2 to 4). It calculates the dominant sets for all nodes other than the output nodes. Let N be the number of levels in the circuit. It starts from level N backward to the inputs (at level 1). It calculates dominant sets level by level (line 5 to 9) using the following formula:

$$D(a) = \begin{cases} \{a\} & a \in SO \\ \{a\} \cup \bigcap_{b \in FO} D\{b\} & otherwise \end{cases} \tag{2.9}$$

Here, FO is the set of fanout nodes for node a. The complexity of finding dominant sets is linear to the number of gates. The dominant cuts could be computed directly after calculating dominator sets.

2.5.4 Clock-gating power-aware synthesis

During normal circuit operation, clock-gating reduces dynamic power consumption by eliminating signal transitions that are not required to correctly compute the output signal. One way to achieve that is to identify ODC [32]. A single output function $f(x_1, .., x_n)$ can be decomposed into sub-blocks, such as C and D blocks. When $C = c$ (where $c\{0, 1\}$ is set to the controlling value of f), f can be completely determined by block C alone and thus block D can be safely switched off. As the goal is to save power, block D is refrained from changing by clock gating its inputs. f decomposition could be as simple as enumerating all possible **AND/OR** gates in small circuits [70]. Recent techniques use BDDs to increase the opportunities in the medium-sized circuits [71].

2.5.5 Rarity simulation

Simulation is an obvious way to eliminate invalid inductive invariant candidates. However, conventional random and constrained simulations usually saturate too fast for large circuits and thus unable to disprove many of invalid inductive invariant candidates [72]. This is because random simulation does not take into account the properties of the circuit. Constrained random simulation mandates the use of a SAT-solver or an Automatic Test Generation Pattern (ATPG) engine to extract knowledge (such as reachable states) and thus can be very slow. On the other hand, rarity simulation [72] uses heuristics to identify rare states, and simulation proceeds from those rare states. This allows the engine to quickly remove many of invalid inductive invariant candidates.

2.6 Conclusion

In this chapter, we have introduced the preliminaries of the rest of the thesis. We have presented Boolean functions and the basics of BDD construction algorithm in the first two sections. Static implications are introduced. Moreover, Model checking problem are introduced. Finally, we introduce logic synthesis, dominant cuts is presented. In the next chapter, we will presents the detailed implementation of our concurrent BDD package.

Chapter 3

Concurrent Cache-friendly Binary Decision Diagram Package For Multi-core Platforms

In this chapter, a literature suvery on BDD are presented in the first section. We present an overview for Hopscotch hashing and how we adapt it to our BDD package (i.e., our concurrent version and resizing technique). Then, we present the package framework and our Garbage collector. Finally, we present the result for our package.

3.1 Literature Survey

Currently, BDD packages depend on chained hash tables. Although they are efficient in terms of memory usage, they exhibit poor cache performance due to dynamic allocation and indirections of data. Moreover, they are less appealing for concurrent environments as they need thread-safe garbage collectors. Furthermore, to take advantage of the benefits from multi-core platforms, it is best to re- engineer the underlying algorithms, such as whether traditional depth-first search (DFS) construction, breadth-first search (BFS) construction, or a hybrid BFS with DFS would be best. In this chapter , we introduce a novel BDD package friendly to multicore platforms that builds on a number of heuristics [73]. Firstly, we restructure the Unique Table (UT) using Hopscotch hashing to improve caching performance. Hashing plays a critical role and Hopscotch hashing is also concurrency-friendly. Secondly, we re-engineer the BFS Queues with hopscotch hashing. Thirdly, we propose a novel technique to utilize BFS Queues to work as a Computed Table (CT) simultaneously. Finally, we propose a novel incremental Mark-Sweep Garbage Collector (GC). We report results for both BFS and hybrid BFS-DFS construction methods. With these techniques, even with a single-threaded BDD, we were able to achieve a speedup of up to $8\times$ compared to a conventional single-threaded CUDD package. When two-threads are launched, another $1.5\times$ speedup is obtained.

Prior work relevant to our proposed BDD package comes from four lines of research: sequential depth-first/breadth-first BDD algorithm, RAM-based BDD algorithms, disk-based

BDD algorithms and pointer reduction techniques for minimizing memory usage. These approaches are often overlap. Binary Decision Diagrams (BDDs) was introduced by Lee [35] and Akers [74] to efficiently represent Boolean functions. Reduced Ordered Binary Decision Diagrams (ROBDDs) was proposed by Bryant [12] to assure a canonical form of a Boolean Function. ROBDDs are BDDs with fixed variable ordering in all its branches and they do not have any repeated sub-graph. Bryant prove that two Boolean Functions are equivalent iff they have the same ROBDDs structure. The restriction of a fixed variable order and the removing of any repeated sub-graph in ROBDDs enable Bryant to develop efficient manipulation algorithms for ROBDDs.

As hash tables enforce the uniqueness property of its nodes, Bryant introduced an efficient way to integrate the reduction algorithm with the manipulation algorithm of the BDDs by using hash tables. However, the main disadvantage of the Bryant approach is that the ROBDDs are stored in disjunctive memory areas. Therefore, the memory overheads are increased. For this reason, Minato [37], [75] propose Shared Ordered Binary Decision Diagrams (SOBDDs). SOBDDs use single Hash table to store all ROBDDs. By this way, similar sub-graphs are utilized efficiently across different ROBDDs. SOBDDs will be used as the basis for implementing our BDD package.

BF algorithms have been used mainly to handle large BDDs. Ochi *et al.* [76] described efficient algorithms for BDDs too large to fit in main memory on the commodity architectures available at that time. Ashar and Cheong [77] improved the performance of BDD algorithms used in [76] by removing redundant nodes, and hence their BDD are more compact. Another line of research exploits memory hierarchy [14]. In this approach, super-scalarity and pipelining are exploited to improve BDD algorithms.

Another approach uses network of workstations to distribute BDD operations. Ranjan *et al.* proposed a BF algorithms for a network of workstation [78]. However, the proposed algorithms are sequential. In other words, the computation is carried in one workstation at a time. Stornetta et al. [79] proposed a parallel BDD package based on DF algorithms. Although, their mechanism provides an efficient work balance. However, their algorithms incur large communication overhead. Milvang *et al.* proposed a package [80] based on the ideas in [78]. However, they utilized all workstations in the network at the same time. The main problem of their approach is that the work is unbalance. Yang *et al.* [81] proposed a hybrid BF/DF algorithms for parallel construction of BDDs for shared memory multiprocessors and distributed shared memory (DSM) systems. Their algorithm relies on frequent synchronization of global data structures and on exclusive access to critical section of the code which make their approach inefficient in terms of scalability. Binachi *et al.* [82] proposed a parallel BDD package for Multiple-Instruction Multiple Data (MIMD) systems. Their approach exhibits efficient work balance. However, the communication overhead is a bottleneck.

Other research takes the approach to handle large BDDs which have very large number of nodes that can't be placed in a single hash table. Minato *et al.* [83] used bit streaming technique to store BDDs in secondary storage medium. By assuming that the machine has enough storage, their technique never causes memory over flow or swap out. A recent study [84] proposed utilizing a parallel disk of a cluster or a storage area network (SAN). They use

Roomy library to overcome the latency issue of using disk. Their technique are based on BFS. Our research fills the gab that utilizes cache in a single machine. but can be extended with other direction to distributed and/or large BDDs. Some of the public domain BDD packages available is **CUDD** [85], **CAL** [78], **ABCD** [86] and **TUDD**.

Hash tables are one of the most thoroughly researched data structures as they have many applications in computer science and engineering. It also plays important rule in constructing BDDs (they are used in Unique Tables(UTs) and Computed Tables(CTs)). There are various types of hashing technique. For example Chained hashing [87], Linear probing [87] Cuckoo hashing [17] and Hopscotch hashing [16].

3.2 Hopscotch Hashing: Overview and adaption to our package

Hopscotch hashing is a recently proposed hashing technique [16]. It is based on multi-phase probing displacement techniques. Hopscotch hashing preserves and utilizes data locality, hence it has been shown to outperform all other well-known hashing techniques, including chaining, cuckoo hashing, and linear probing. Moreover, it also guarantees a fixed worst case fetching time.

3.2.1 Basic Operations

The hash table is an array of buckets, each of which is a set/list of items. Due to cache memory, the cost of finding an element in the nearby buckets is almost the same. So, Hopscotch hashing utilizes the notion of neighborhood around any given bucket. Hopscotch hashing tries to insert elements of the same bucket in the neighborhood of the bucket. In other words, it may place the new item in a neighboring bucket if necessary. If it does not find any empty location for the new element, it tries to shuffle and move other bucket elements to provide a space for this new element.

Thus, given that the hash table has a single hashing function $h()$, any hashed element will be found in the corresponding bucket b_h or in one of the next H-1 entries after b_h, where H is a constant referred to as the hopscotch neighborhood length. In order to accelerate the operation, a bitmap contains hop information for each bucket is stored. Each bit in the hop information for bucket b_h represents an entry of the nearby entries of b_h. For each bucket b_h, if any of the H entries, starting from bucket b_h, has been mapped to b_h, then the corresponding bit in the hop information for bucket b_h is set to one. Otherwise, it is set to zero as depicted in Fig. 3.1. In this example, $H = 4$, and hop information for bucket 233 is 1001. Accordingly, nodes x and z belong to bucket 233.

The pseudo-code for *contains()* and *findOrAdd()* methods are shown in Fig. 3.2 and Fig. 3.3, respectively. In Fig. 3.2, the *contains()* method takes three arguments and returns $TRUE$ when the item exists in the hash bucket and $FALSE$ otherwise. The input arguments

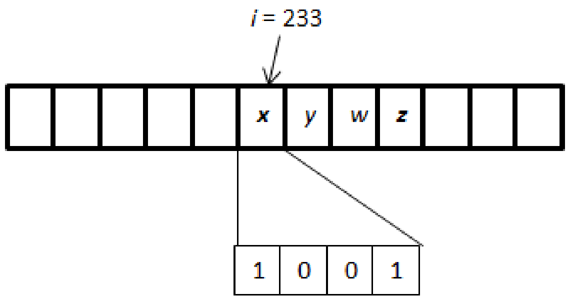

Figure 3.1: Hop Information Illustration.

are defined as follows: 1) *node* is the node item that we are searching for, 2) if the item already exists, *ptr* stores a pointer to it, otherwise it is undefined, 3) *lck* is a Boolean flag to determine whether the method will use locks or not. In this *contains()* method, the hash value is calculated in lines 2 & 3, hop information is then fetched, and decoded to corresponding entry in the hash table (not shown in Figure). Finally, the algorithm loops in every corresponding entry to find whether the required item exists or not (lines 13-24). Additional details will be provided in Section 3.3.

Fig. 3.3 lists the pseudo-code for the *findOrAdd()* method. In the add method, to add an item x with hash value $h(x) = i$: 1 - starting from entry i and probe the successive entries until an empty entry is found at index j (line 5). The *findNearestEmptyLocation()* method finds the nearest empty location and passes it by value to *emptyLoc* and return $TRUE$. If it reaches the end of the table without finding any empty locations, it returns $FALSE$ and the $resize()$ method will be called to reallocate the bucket. 2 - if an empty entry is within $H-1$ from index i, then the procedure places the item, update hop information for bucket i, and return (lines 20 & 21). 3 - otherwise, location j is too far from index i. and the algorithm will try to swap items between i and j to make a space for the new item (lines 9-16) as following. First, it tries to find an item y that lies between i and j and is hashed within H-1 entries below j (line 10). Secondly, move y to an entry at index j (lines 11-15). Now, the old entry for y is available. We repeat this operation until the available entry is within H-1 of entry at index i. Finally, add x at the available entry, update the hop information for i, and return. *contains()* methods complete in constant time in the worst case.

3.2.2 Concurrent version and adaptation to BDDs

As depicted in Fig. 3.4, the concurrent version of Hopscotch hashing partitions the nodes table into M number of segments. Each segment is associated with a read/write lock. $SegmentSize$ refers to the length of each segment in the node table. Segment table stores the lock for each division. The segment size (S) is the length of each partition in the node table corresponding to each segment. We choose $S > H$ to reduce the maximum number of segments needed to be handled in any operation to two segments. For example, consider Fig. 3.2. The $contains$() method calculates the segment of the final possible item in the neighborhood of the bucket (variable $NextSeg$ in line 6). If $NextSeg$ is different from the segment of the first item of the bucket ($FirstSeg$), the segment is locked (lines 8 & 9). Otherwise, only $FirstSeg$ segment is locked (line 11).

```
contains($node$, $ptr$, $lck$)
$hashString$ = getHashString($node$)
$hashCode$ = getHashCode($hashString$)
**if** $lck$ **then**
    $FirstSeg$ = getSegment($hashCode$)
    $NextSeg$ = getSegment($hashCode + BucketSize - 1$)
    lock Segment at $FirstSeg$
    **if** $FirstSeg \neq NextSeg$ **then**
        lock Segment at $NextSeg$
    **end if**
**end if**
**while** There is another node in the bucket $hashCode$ **do**
    **if** node.key = Current location key **then**
        ptr = calculate current location pointer
        **if** $lck$ **then**
            unlock Segment at $FirstSeg$
            **if** $FirstSeg \neq NextSeg$ **then**
                unlock Segment at $NextSeg$
            **end if**
        **end if**
        **return true**
    **end if**
**end while**
**if** $lck$ **then**
    unlock Segment at $FirstSeg$
    **if** $FirstSeg \neq NextSeg$ **then**
        unlock Segment at $NextSeg$
    **end if**
**end if**
**return false**
```

Figure 3.2: *contains*() method.

3.2.3 Resizable Unique table

The unique table (UT) is implemented as a hash table using the hopscotch hashing technique described above. To preserve data locality, it is represented by an array of buckets, where each bucket contains data as depicted in Fig. 3.5. Each bucket in UT consists of 16 bytes.

```
1: findOrAdd($node$, $ptr$)
2: if contains($node$, $ptr$, true) then
3:     return true
4: end if
5: if !findNearestEmptyLocation($emptyLoc$, $segment$) then
6:     resize()
7: end if
8: calculate $hashCode$ for $node$
9: while $emptyLoc - hashCode > BucketSize - 1$ do
10:    search for a node within $emptyLoc - BucketSize + 1$ and $emptyLoc$ whose bucket is
       within the same range.
11:    if a node suitable for swapping then
12:        perform swapping and update $emptyLoc$ and $segment$ and hop information for the
           bucket.
13:    else
14:        resize()
15:    end if
16: end while
17: if contains($node$, $ptr$, false) then
18:     return true
19: else
20:     add $node$ at $emptyLoc$
21:     update hop information for $emptyLoc$
22: end if
```

Figure 3.3: *findOrAdd()* method.

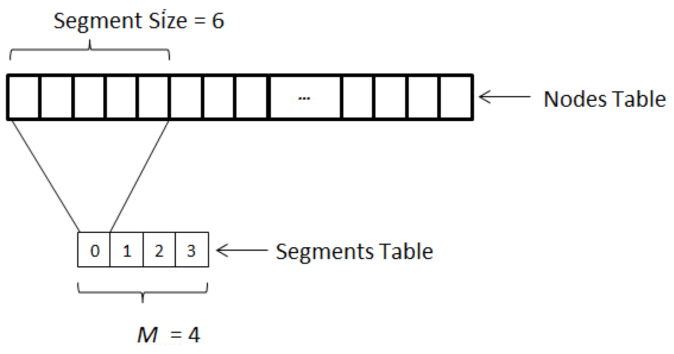

Figure 3.4: Hopscotch Concurrency Illustration.

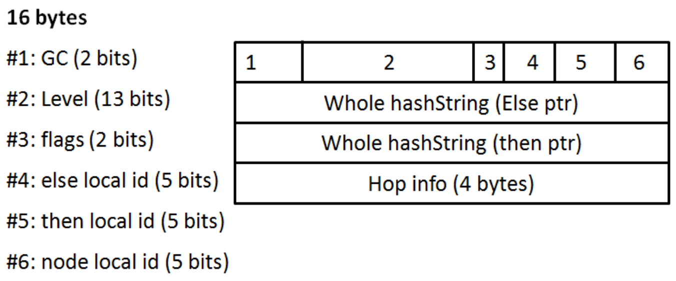
16 bytes
#1: GC (2 bits)
#2: Level (13 bits)
#3: flags (2 bits)
#4: else local id (5 bits)
#5: then local id (5 bits)
#6: node local id (5 bits)
1
2
3
4
5
6
Whole hashString (Else ptr)
Whole hashString (then ptr)
Hop info (4 bytes)

Figure 3.5: UT Bucket Layout.

We have 4 bytes to store bitmap for bucket hop information. In addition, we have two 4-bytes to store hashstring for $Then$ and $Else$. The remaining 4 bytes are used to pack six fields; 1) Garbage Collector Mark (2-bits), 2) $Index$: which is node index, 3) $flags$: which is used for internal operation of UT, 4) $Else$ local id, 5) $Then$ local id and 6) node internal id (within the hashed bucket).

Each bucket contains hop information bitmap, a $Then$ pointer, a $Else$ Pointer, an $Index$, some Flags, and a GC mark field (described in Section 3.3.4). The initial size of the UT is selected to be a power of two, in order to improve the performance of UT resizing operation. The hashing function is calculated as follows:

$$h(x) = f(T, E, m) \mod n \tag{3.1}$$

where T is the $Then$ pointer, E is the $Else$ pointer, m is the node index, and $f(T, E, Index)$ is a stream of bits (called hash string) which is function of $Then$ pointer, $Else$ pointer and $Index$. Our package is a pointer-free package; that is, the pointers of the BDD nodes are not the physical memory addresses of the node. We take this approach for mainly two reasons. First of all, it is to provide a platform-independent implementation. Secondly, Hopscotch

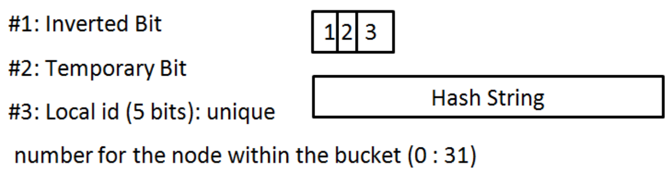
#1: Inverted Bit
#2: Temporary Bit
#3: Local id (5 bits): unique number for the node within the bucket (0 : 31)
1
2
3
Hash String

Figure 3.6: Pointer Layout.

hashing usually swaps nodes between entries. So, physical-address pointers will add more overhead due to the need to update pointers in each swap. Figure 3.6 depicts our BDD pointer. It consists of four fields: (1) hash string (which is defined in Eq. 3.1), (2) inverting bit, (3) temporary bit (it is used to indicate whether the pointer is used in the UT or CT) and (4) local pointer (which is a unique identifier for the node within the bucket's neighborhood).

When a new node is inserted into the UT, it searches for other nodes within the bucket first, and assigns a new local pointer to the new node as illustrated in Figure 3.7. Nodes x and y belong to bucket 462. Location 465 is empty (*emptyLoc* appears in gray). Nodes x and y have local ids equal 0 and 2 respectively. Thus, node at *emptyLoc* will get the smallest available local id assigned to any node belongs to this bucket; that is, one. When UT is not able to insert a new node within its bucket neighborhood, *resize*() is called (as shown in Fig. 3.3 at line 14).

In order to save time and avoid memory explosion, we propose an incremental resizing technique for our UT. As depicted in Fig. 3.8, *resize*() Method consists of three phases. The first and third phases are done by the *Master* thread, while the second thread is processed by both *Master* and *Slaves* threads. When *resize*() is called, the first thread

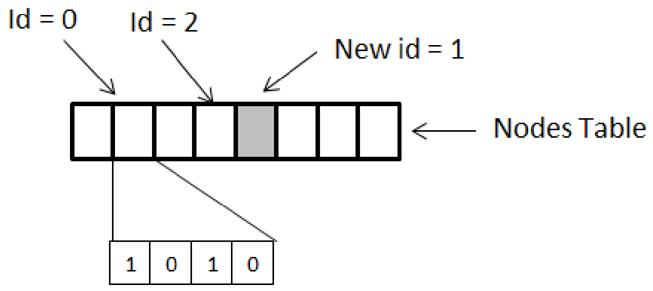

Figure 3.7: ID assignments.

call is considered as the *Master* thread, and any other thread executed by the Master is called a *Slaves* thread. In the first phase (see Fig. 3.8, lines 2-7), the *Master* will allocate a continuous memory block with the size identical to the size of the original UT as shown in Fig. 3.9. Assume the initial UT size is x. After one $resize()$ method call the size will double; so, the size will increase by another x (and the old partition will remain). After another call for $resize()$ method, the size will doubles again; and hence, UT size will increased by $2x$. Note that any other *Slave* thread that enters while *Master* thread is executing the first phase, will be blocked while trying to acquire the lock (line 1 of Fig. 3.8).

In the second phase, all *Master* and *Slave* threads are cooperating in table rehashing. Each thread will take a segment and rehash it until all segments are rehashed (lines 8-12 of Fig. 3.8). Note that each pointer of any node will not change due to UT resizing. This is because that the UT stores the hash string instead of the hash values. In addition, by restricting the size of UT to be a multiple of two, any node will be rehashed to the same location or will be shifted by the old size of the table as depicted in as depicted in Fig. 3.10. Item at location A will remain in the same place or it will move to location B, which be is far from

```
1: resize()
2: acquire master lock.
3: if this is the master thread then
4:     acquire lock on all segments.
5:     allocates new space, updates necessary parameters variables
6: end if
7: release master lock.
8: while there is a segment doesn't rehashed yet do
9:     for all $i$ such that $i \in current\ segment$ do
10:        rehash node at location i.
11:    end for
12: end while
13: acquire master lock.
14: if this is the master thread then
15:     release all segments lock
16:     notify all sleeping threads
17: else
18:     sleep
19: end if
20: release master lock.
```

Figure 3.8: *resize()* method.

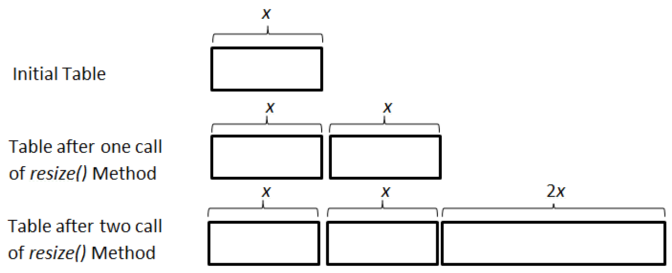
x
Initial Table
x
x
Table after one call of resize() Method
x
x
2x
Table after two call of resize() Method

Figure 3.9: UT resizing illustration.

location A by x (where x is the size of the UT before calling $resize()$). In the third phase (lines 14-20 in Fig. 3.10), all $Slave$ threads will sleep until the $Master$ finishes.

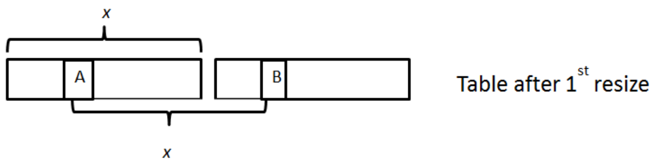

Figure 3.10: UT Rehash Illustration.

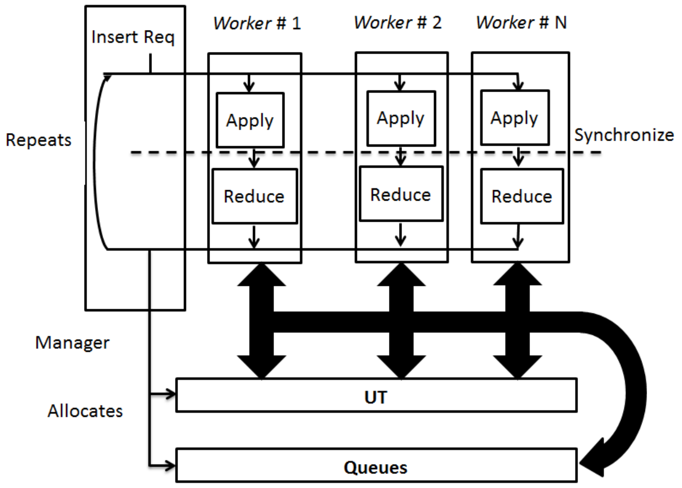

Figure 3.11: BDD package framework.

3.3 Package Framework

The overall package Framework is illustrated in Fig. 3.3. It consists mainly of 1) *Master*, 2) *Slaves*, 3) *UT*, and *Queues*.

3.3.1 Manager

As depicted in Fig. 3.12, *manager* is responsible for allocation and initiation of all necessary components to built a BDD from a netlist circuit (lines 2-4). It schedules BDD request into queues (Line 6). In addition, it synchronizes between workers (lines 8 & 9).

```
1: Read circuit netlist.
2: Initialize and allocate UT, Queues, and other internal variables.
3: Create worker threads.
4: **for all** circuit levels **do**
5:     insert available BDD request for this level.
6:     **while** there is a scheduled request in this level **do**
7:         wait until workers finish executing apply() method.
8:         wait until workers finish executing reduce() method.
9:     **end while**
10: **end for**
```

Figure 3.12: *Manager* main method.

3.3.2 Worker Threads

Worker threads are responsible of performing BDD basic construction operations as depicted in Fig. 3.13. Fig. 3.11 depicts how *Manager* synchronizes among *workers*. *Manager* allocates *UT* and *Queues*, insert BDD requests, then creates $N - worker$ threads. *Worker* threads execute *apply*() method concurrently. *Manager* synchronizes between *worker* threads until all threads finish executing *apply*(); Then, *worker* threads execute *reduce*() method. These operations repeat until all circuit gates are constructed.

manager waits until all *workers* finish the *apply*() method (line 4 in Fig. 3.13), then it allows *workers* to begin in *reduce*() method(lines 6 in Fig. 3.13).

```
1: loop
2:     while there is a scheduled request in this level do
3:         apply().
4:         wait until all other workers finish apply() method.
5:         reduce().
6:         wait until all other workers finish reduce() method.
7:     end while
8: end loop
```

Figure 3.13: *Worker* main method.

Fig. 3.14 depicts *apply*() method pseudo-code. As *apply*() method reads nodes from UT, we remove all locks while *apply*() fetch nodes from UT.

Fig. 3.15 shows the method *reduce*(). As the *reduce*() method reads requests from *Queues*, we remove all locks while *reduce*() fetch nodes from *Queues*.

3.3.3 Queues and Computed Tables

While *worker* threads construct BDDs, *apply*() and *reduce*() methods require to access temporary request often (see lines 4,12, 22 in Fig. 3.14 and lines 4, 29, 34, 37 in Fig. 3.15). In order to have an efficient memory access, we propose to implement *Queues* as a hash Table and an array of lists as depicted in Fig. 3.16. Array of lists is used to keep track of every request on each level. : Queues consist of one large hash table (implemented with

```
apply()
for $i = 1$ to $MaxIndex$ do
  while there is a scheduled request in Queues in level i do
    dequeue request from Queues.
    get $F$, $G$, $H$, and $R$ from request.
    if $R$ isn't fully processed then
      $x = topIndex(F, G, H)$.
      if terminalCase($F_x$,$G_x$,$H_x$,$result$) then
        $R.setThen(result)$.
      else
        $putInStandardTriple(F_x, G_x, H_x)$.
        $Queues.findOrAdd(F_x, G_x, H_x, result)$.
        $R.setThen(result)$.
      end if
      if $terminalCase(F_{\overline{x}}, G_{\overline{x}}, H_{\overline{x}}, result)$ then
        $R.setElse(result)$.
      else
        $putInStandardTriple(F_{\overline{x}}, G_{\overline{x}}, H_{\overline{x}})$.
        $Queues.findOrAdd(F_{\overline{x}}, G_{\overline{x}}, H_{\overline{x}}, result)$.
        $R.setElse(result)$.
      end if
      $Queues.putRequest(F, G, H, R)$
      if max. nodes limit is reached for level i then
        break.
      end if
    end if
  end while
end for
```

Figure 3.14: *apply*() method.

Hopscotch hashing), and array of N lists, where N is number of levels. Each list contains the pointers of the request in hash table corresponding to certain level. Hash table utilization in implementing *Queues* have another motivation. We can utilize the hash table as a CT. Dated requests remains in the hash Table as long as a new request needed to be stored in the same location. Accordingly, if a request ($R = \{F, G, H\}$) is previously evaluated and still in the hash table, its forwarded pointer is fetched.

3.3.4 Garbage Collectors

Garbage Collector (GC) is based on a lock-free Mark-and-Sweep approach. We use two bits to represent a mark. Any node can be marked as 1) a permanent node, 2) an updated node, or 3) a dated node. Permanent nodes has its unique mark. We use two other marks to represent updated and dated marks. Before a level is constructed, mark is set and all nodes represent gates in circuit are marked as Permanent. All new nodes inserted will be marked with the Updated Mark. Any node that has a Dated Mark may be overwritten with a new nodes if it is needed to be inserted in the same location of the old node.

3.4 Results

The proposed package is implemented in C++ and is tested with a *BDDtest*. *BDDtest* creates a $Manager$, which in sequence creates $Workers$ and allocates UT, $Queues$, and other necessary data. *BDDtest* also create BDDs for every gate in the circuit. For sequential circuits (Table 3.2), *BDDtest* builds the transition relation of the circuit (which takes a longer time for construction). The transition relation for the circuit is the conjunction of all transition relations for each state element s_i, i.e., $(\bigwedge_{\forall i} s_i \oplus \delta_i)$. The experiments were run on Core 2 machine with 4 GB of RAM and Ubuntu as the Operating system. CUDD 2.4.2 [88] is used as comparison. We report results on a number of circuits from ISCAS85, ISCAS89, and ITC99 to test our proposed BDD construction. The results are reported in Tables 3.1 & 3.2. Table 3.1 reports the results for combination circuits. Table 3.2 reports the results of creating the transition relation for sequential circuits.

Table 3.1: Experimental Results - Combinational Circuits

Circuit	**CUDD**	**BR-1**	**BR-2**	**Resizing**	**BR-CT**	**HBR**	**HCT**	**GC**
c432	0.455	0.411	0.277	0.386	0.246	0.510	0.300	0.381
c1355	33.206	7.929	5.030	6.803	4.888	114.722	20.737	5.785
c1908	11.542	1.622	1.037	1.292	1.001	4.980	1.963	1.759
c5315	16.364	2.911	2.430	2.593	2.144	28.612	12.952	4.234

In Tables 3.1 & 3.2, for each circuit, we first report the execution time taken by CUDD, followed by our package with BR-1 (Basic Run with 1 thread). We define Basic Run as a run that does not include UT resizing, CT utilization, BFS-DFS hybrid approach, nor GC. In other words, BR-1 is a BFS without UT resizing and CT is not utilized. BR-1 is followed by BR-2 (Basic run with 2 threads). Note that starting from column three, 2 threads are used. The fourth column (Resizing) reports BR-2 with 6 resizing operations. The fifth column (BR-CT) reports the BR with computed table. The sixth column (HBR) reports the DFS-BFS hybrid approach with Queue size ranging from 1 to 40% (percentage depends on the circuit). The seventh column (HCT) reports the time for DFS-BFS hybrid approach with CT (we use the same configuration as in column six). Finally, the eighth column (GC) reports the time of BR-2 with Garbage collection.

According to Tables 3.1 & 3.2, our base approach (with BR-1) achieved a speedup ranging from 2× to 8× compared with CUDD. When we use two threads (BR-2), we achieved another 1.5× speedup on average. For example, consider circuit b12rst_1_3_new_s, the original CUDD

Table 3.2: Experimental Results - Transition Relation Construction of Sequential Circuits

Circuit	**CUDD**	**BR-1**	**BR-2**	**Resize**	**BR-CT**	**HBR**	**HCT**	**GC**
s298_1_2_new_s	2822.1	1585.4	847.8	950.2	838.6	10159.4	1012.4	1044.9
s298_2_3_new_s	2202.2	646.0	437.7	525.2	365.1	875.3	529.6	660.7
s298_2_4_new_s*	2760.9	395.5	228.6	274.4	270.0	581.6	278.2	353.0
s400_1_2_new_s*	1071.2	369.3	275.7	281.2	270.7	1498.3	380.8	342.9
s444_1_2_new_s*	2430.3	335.7	183.3	247.4	167.6	786.2	266.7	268.7
b12rst_1_2_new_s*	214.1	49.6	30.9	31.2	30.0	401.7	150.0	69.1
b12rst_1_3_new_s*	714.3	88.1	64.1	76.9	64.0	268.6	67.4	100.4
b12rst_1_4_new_s*	734.4	120.4	84.6	88.8	70.5	338.5	105.4	156.0
b12rst_1_5_new_s*	680.8	94.1	65.7	81.7	53.7	248.0	84.3	82.3
b12rst_2_3_new_s*	853.9	173.0	120.2	146.4	97.8	819.0	321.9	194.5
b12rst_2_4_new_s*	864.7	109.0	78.4	114.0	76.9	231.5	151.0	110.9
b12rst_2_5_new_s*	913.5	174.7	107.6	141.2	90.1	933.2	271.0	124.1

(*) only a portion of the circuit is tested, not the whole circuit.

took 714 seconds to construct the transition relation for this circuit, and our BR-1 took only 88 seconds. This is a speedup of 8.11×. BR-2 reduces the time further to 64 seconds. In a few cases, having two threads allowed us to achieve nearly 2× speedup, such as s298_1_2_new_s, where the execution time was reduced from 1585 seconds to 847 seconds. When resizing is performed for 6 times, we still obtained speedups higher than BR-1. Accordingly, we can conclude that, resizing has a little impact on the performance, because we utilize all threads to perform the resizing operation. When comparing the results under columns HBR and HCT, we observed that utilizing the CT with a hybrid approach provides a large improvement in the performance when DFS-BFS hybrid approach is exploited. This is because many requests have been replicated during DFS-BFS hybrid, and hence, CT becomes very vital. When we use the DFS-BFS hybrid approach (column 6, HBR), the performance is degraded compared to BR-2 in all cases. Also, it degraded in most of cases compared to CUDD (i.e., s298_1_2_new_s), since no computed table is used (while CT is used in CUDD). However, when we use DFS-BFS hybrid approach with Computed Table (HCT), the performance is enhanced. For example, in circuit b12rst_2_5_new_s, although the performance of HCT is degraded by 2.52 compared with BR-2, we obtained 3.37× speed-up compared with CUDD and 3.44× over the one with CT (column HBR). Finally, garbage collection incurs some overhead, but can be useful when memory usage is high, as in b12rst_1_2.

In all circuits, our results show that the BFS based construction of the BDDs, as opposed to hybrid BFS-DFS, achieves superior performance on multi-core platforms. Of course, these depend on the type of hashing algorithms used.

3.5 Conclusion

In this chapter, we have introduced a novel cache-friendly Multi-threaded BDD Package to construct and manipulate ROBDDs on a multi-core platform. As BDD algorithms are memory intensive, maintaining locality of data is important to reduce cost of memory loads and stores. Furthermore, our algorithm offers concurrency to enhance performance on multi-core platforms. We propose the usage of concurrent Hopscotch hashing technique for both the Unique Table and the BFS Queues to improve the performance of BDD construction. Hopscotch hashing not only improves the locality of the manipulating data, but also provides a way to cache recently performed BDD operation. Moreover, it is concurrency friendly. Consequently, the time and space usage can be traded off. With our approach, even with a single-threaded implementation, we were able to achieve a speed-up of up to $8\times$ compared to a conventional single-threaded CUDD package. When two-threads are launched, another $1.5\times$ speed-up is obtained.

```
reduce()
for $i = MaxIndex$ to 1 do
  while there is a scheduled request in Queues in level i do
    dequeue request from Queues.
    if $R$ isn't forwarded then
      if $Then$ of $R$ is processed in $apply()$) then
        if $Then$ points to a $Queues$ node then
          get $R$ in $Queues$ corresponding to $Then$.
          if $R.isForwarded(result)$ then
            $Then = result$.
          end if
        end if
      end if
      if $Else$ of $R$ is processed in $apply()$) then
        if $Else$ points to a $Queues$ node then
          get $R$ in $Queues$ corresponding to $Then$.
          if $R.isForwarded(result)$ then
            $Else = result$.
          end if
        end if
      end if
      if either $Then$ or $Else$ points to a $Queues$ node then
        update $R$ with new $Then$ and $Else$.
      else if $Then = Else$ then
        forward $R$ to $Then$.
      else if $Then$ is inverted then
        invert $Then$, and $Else$.
        create UT $node$ with $Then$, $Else$ and $i$.
        $result = UT.findOrAdd(node)$
        invert $result$.
        forward R to result
      else
        create UT $node$ with $Then$, $Else$ and $i$.
        $result = UT.findOrAdd(node)$
        forward R to result
      end if
      return $R$ back to the $Queues$
    end if
  end while
end for
```

Figure 3.15: *reduce*() method.

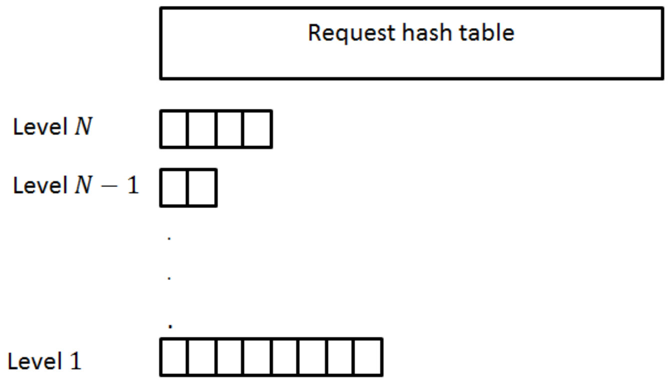

Figure 3.16: Queues Structure.

Chapter 4

Selecting Critical Implications with Set-Covering Formulation for SAT-based Bounded Model Checking

In this Chapter, a literature survey in static implication utilization in BMC problem is presented. A parallel deduction engine is presented in the second section. Set-Cover based formulation of Static implication filtering for BMC is presented in the third section. A greedy approach based on this formulation is presented in the fourth section. Finally, results and conclusion are presented.

4.1 Literature Survey and Contributions

The effectiveness of SAT-based Bounded Model Checking (BMC) critically relies on the deductive power of the BMC instance. Although implication relationships have been used to help SAT solver to make more deductions, frequently an excessive number of implications has been used. Too many such implications can result in a large number of clauses that could potentially degrade the underlying SAT solver performance.

One method to improve BMC is to embed clauses in the formula such that the deductive power of the instance is increased. To this end, there has been low-cost learning techniques based on a combination of binary resolution and static logic implications to learn sequential relations that may span several time-frames in the circuit have been proposed [38], [39]. In these methods, all the learned relations are globally true, which means that the relations learned over a small window of the unrolled circuit can be readily replicated throughout the unrolled transitions of the circuit. These relations, when added as constraint clauses to the original Conjunctive Normal Form (CNF) for the BMC instance, aid the SAT solver in deducing a larger set of implied literals at a given decision point. As a result, they can help to prune the search space. However, adding an unnecessarily large number of static implication clauses may degrade the performance, as DPLL would now have to iterate on a large number of clauses. Therefore, investigation of a method that can classify the implication clauses as useful or not-useful would be a tremendous benefit.

As far as we know, there is no well-defined criteria for a classification strategy of static logic implication clauses. The only filtering technique used for static logic implication is proposed in [39], where they the metric is based on extended implications [38]-[39]. Other simple methods may use the distance; that is, the distance between the nodes in question in a clause. Larger distances may be preferred as they relate signals that are further apart. The classification can be regarded as a filter, only those that meet the criteria are taken and added as constraints to the formula [53],[54].

Our contributions are summarized as follows.

- We first propose a framework for a parallel deduction engine to reduce implication learning time. Our parallel deduction engine can achieve a 5.7× speedup on a 36-core machine.

- We propose a novel set-covering technique for optimal selection of constraint clauses. This technique depends on maximizing the number of literals that can be deduced by the SAT solver during the BCP (Boolean Constraint Propagation) operation. By selecting only those critical implications, our strategy improves BMC by another 1.74× against the case where all extended implications were added to the BMC instance. Compared with the original BMC without any implication clauses, up to 55.32× speedup can be achieved.

4.2 Parallel Deduction Engine

We have based our parallel deduction framework on the observation that the implication of each node can be deduced independently from other nodes. Thus, we can design a simple parallel algorithm to perform this operation. Our parallel deduction engine computes the implication of each node in the implication graph independently. It then shares the learnt implications among all nodes, and iterates until fixed point is reached.

Our parallel framework engine is constructed as follows: we have two types of logical threads, (1) one Master and (2) M Worker threads. As depicted in Fig. 4.1, the master thread is responsible for allocating the implication graph (at line 1), creating other worker threads (at line 2), synchronizing between them, and interchanging and sharing learnt clauses between different nodes (at line 5 and 6). Fig. 4.1 shows the pseudo-code for the Master thread.

Fig. 4.2 lists the pseudo-code for the worker thread. Each worker thread takes a node in the implication graph and deduces its implications using techniques described in Section 2.3.1. These operations are repeated until all nodes have been processed. The workers will wait until the Master thread has shared all learnt clauses from other nodes. It repeats the routine until no other edge is added, i.e., a fixed-point has been reached.

As depicted in Fig. 4.3, the *deduce*() method iterates on the implication graph. First, it deduces direct implications for a node (at line 3), then, it deduces indirect implications (at line 4). Thirdly, it deduces Extended Backward implications (EBI) (at line 5). Fourthly, it deduces Extended Forward Implications (EFI) (at line 6). In the fifth step, it deduces Justi-

```
1: allocate Implication graph nodes (2N)
2: create $M$ worker threads
3: repeat
4:     wait until ALL worker finishes
5:     share learnt clauses
6: until no other clause have been added to the implication graph
```

Figure 4.1: Master thread.

```
1: repeat
2:     repeat
3:         deduce()
4:     until no other shared clause is added
5: until no other clause is added
```

Figure 4.2: Worker thread.

fication Frontier Implications (at JFron) (line 8). It then repeats until no more implications are added to the implication graph.

4.3 Set-Cover Problem

An instance (X, F) of the set-cover problem consists of a finite set X and a family F of subsets S of X, such that every element of X belongs to at least one subset in F [89]:

```
1: deduce()
2: repeat
3:     deduceDirectImplication()
4:     deduceIndirectImplication()
5:     deduceEBI()
6:     deduceEFI()
7:     deduceJFron()
8: until no other clause have been added to the implication graph
```

Figure 4.3: Deduce() method.

$$X = \bigcup_{S \in F} S \tag{4.1}$$

If a subset S is a non-empty set (contains atleast one element), we say that a subset $S \in F$ covers its elements. The set-covering problem is to find a minimum-sized subset $C \subseteq F$ whose members cover all of X:

$$X = \bigcup_{S \in C} S \tag{4.2}$$

We say that any C satisfying Equation (4.2) covers X. The set-cover problem is known as an NP-hard combinatorial problem [90]. Approximation algorithms based on greedy algorithms [91], linear programming and network flow algorithm [91] had been proposed. Dutta [91] has

```
1: Greedy-Set-Cover($X$,$F$)
2: $U = X$
3: $C = \emptyset$
4: while $U \neq \emptyset$ do
5:     select an $S \in F$ that maximizes $|S \cap U|$
6:     $U = U \backslash S$
7:     $C = C \cup \{S\}$
8: end while
9: return $C$
```

Figure 4.4: Standard Greedy-Set-Cover Algorithm [89].

conducted a comparative study and has shown that greedy algorithms can often provide a near-optimal solution to the set-cover problem.

A standard greedy algorithm is depicted in Fig. 4.4. The main idea behind this greedy approach is to find greedily the subsets that covers maximum number of elements in F. U is initialized with the family of subsets X (line 2) and the cover C is initially empty (line 3). The algorithm greedily chooses a subset S that have maximum number of elements (line 5). It removes these elements from U (line 6) and adds the subset S to the cover U (line 7). This process is repeated until all elements are covered. In other words, U became an empty set (line 4). Its approximation ratio is proven to be $\ln|X| - \ln\ln|X| + \Theta(1)$ [92].

4.4 Greedy Set-Cover Formulation and Algorithm

Our clause selection criteria is based on selecting the clauses that can bring about the maximum number of assignments during BCP operation in the SAT solver. That is, our target is to select the smallest set of implication clauses, when added to the original formula, can help produce a maximal number of implications during BCP operation.

Before we formally define the problem, the following definitions are needed.

- Definition I: Let i be a node in the implication graph corresponding to a gate g with a value v. Node i could also be rewritten as (g, v).
- Definition II: S_i is a set of all signal assignments implied from asserting node i in the CUV (deduced from some or all implication techniques stated in Section 2.3.1).
- Definition III : M_i is a set of all signal assignments implied from asserting node i in the CUV deduced using *only* direct and Indirect implications.
- Definition IV : V is a set of all nodes in the implication graph, $|V| = 2G$, where G is number of gates in CUV.

Problem statement: Given an implication graph $G(V, E)$ for the circuit, for each node $i \in V$, we have a set of implications S_i (direct, indirect and extended implications) and M_i (direct and indirect implications only). Note that $M_i \subseteq S_i$. As BCP can deduce direct and indirect implication efficiently, we want to select the minimum set of nodes $k \in S_i$ such that their direct and indirect implications (M_k) cover S_i. Formally, let M be the family of direct and indirect implication of nodes in S_i; that is, $M = \bigcup_{w \in S_i} M_w$. So, the problem is to find a minimum-sized subset $C \subseteq M$ whose members cover all of S_i, and hence, $S_i = \bigcup_{M_k \in M} M_k$. According to Eq. [4.1], (S_i, M) is a set-cover instance for clause selection.

Example 1 : (Basic Idea) Let us consider that we have a CUV containing gates d, f, g, l, m, n, r and t. Without loss of generality, we assume that the deduced implications are in the same time frame. Let signal assignments $(d, 1)$, $(f, 0)$, $(g, 1)$, $(l, 0)$, $(m, 1)$, $(n, 0)$, $(r, 0)$ and $(t, 0)$ be labeled as nodes $1, 2, 3, 4, 5, 6, 7$ and 8, respectively. Let $S_7 = \{1, 2, 3, 4, 5, 6, 7, 8\}$. Let the set of direct and indirect implications for each of the nodes be, $M_1 = \{2, 7, 8\}$, $M_2 = \{1, 4, 6\}$, $M_3 = \{5\}$, $M_4 = \{7, 8\}$, $M_5 = \{2, 6, 7, 8\}$, $M_6 = \{4, 7\}$, $M_7 = \{1, 3, 6\}$, and $M_8 = \{3, 5\}$. So, $M = \{M_1, M_2, M_3, M_4, M_5, M_6, M_7, M_8\}$.

Suppose we want to find the cover for (S_{S_7}, M); that is, we want to find the smallest set of clauses that can provide the most number of implications for node 7. We apply the greedy strategy; M_5 has the maximum size (of 4), the greedy algorithm will choose it and remove nodes $2, 6, 7$ and 8 from the other sets, and we will have 4 remaining non-empty sets: $M_2 = \{1, 4\}$, $M_3 = \{5\}$, $M_7 = \{1, 3\}$, $M_8 = \{3, 5\}$. Now, the algorithm has the choice between M_7 and M_8 (as both are of max. size of 2). It will pick M_8, and hence it will remove nodes 3 and 5 from all other set, and we will have 2 remaining non-empty sets. $M_2 = \{1, 4\}$ and $M_7 = \{1\}$. Greedy choice will pick M_2 (as it is of max. size of 2). Now, all sets are empty, and implications $7 \to 2$, $7 \to 5$, $7 \to 8$ are chosen to cover all the transitive relations of S_7.

Example 2 : (Contrapositive Principle Consideration): As the greedy algorithm selects the clauses, it may remove its contrapositive relation(s) from other sets. Referring to Example 1 above, when $7 \rightarrow 5$ is chosen, the greedy algorithm will check if $\neg 5 \rightarrow \neg 7$ exits in $S_{\neg 5}$. If the contrapositive clause exists, it will remove it and all its direct and indirect implications. In other words, $S_{\neg 5} = S_{\neg 5} \setminus M_{\neg 7}$. This not only removes unnecessary clauses, but also reduces the calculation time for set Cover of all subsequent nodes.

Algorithm

Fig. 4.5 describes our greedy approach for the selection of clauses. FL_i is a filtered list for node i. The filtering process iterates on each non-constant node in the implication graph. It gets the implication with largest $|M_j|$ (line 5) and adds node j to FL_i (line 6). Then, it removes direct and indirect implication from S_i, $M_j \forall j \in V$. In line 12, it removes the contrapositive implication (and all its direct and indirect implications from $S_k : \ k \in v, k \neq i$).

4.5 Results

The proposed parallel deduction engine and the set-covering selection framework have been developed with C++ and the performance was evaluated on SGI UV system [93]. We have used 36 dedicated 2.66 GHz Intel Xeon cores, with total of 190.8 GB of RAM, running SUSE Linux 11. We have compiled our program with g++ under -O3 option. Since most of the BMC instance available online is in CNF form including the initial state, the unrolled

```
1: for all node $i = (N, v)$ in the implication graph do
2:     $FL_i = \emptyset$
3:     if $i$ is not a constant node then
4:         repeat
5:             get node $j = (M, v') \in S_i$ such that $|M_j|$ is max.
6:             $FL_i = FL_i \cup \{j\}$
7:             $S_i = S_i \setminus M_j$
8:             for all node $k \neq j$ do
9:                 $M_k = M_k \setminus M_j$
10:            end for
11:            get node $l = (M, \overline{v'})$ and node $t = (N, \overline{v})$
12:            $S_l = S_l \setminus M_t$
13:        until all clauses in $S_i$ is covered
14:    end if
15: end for
```

Figure 4.5: Greedy-Set-Cover Filtering Algorithm.

instance, and the monitor [94], [95], it is difficult to extract a single time-frame of only the circuit. Thus, we have developed a set of BMC benchmark based on ISCAS89 circuits that require a large computational cost to solve (as depicted in last column (orig.) in Table 4.6).

Table 4.1: Parallel Deduction Engine **without** Implication Sharing between Nodes

ckt	Case A			Case B			Case C			Case D		
	Seq.	Par.	s-u	Seq.	Par.	s-u	Seq.	Par.	s-u	Seq.	Par.	s-u
s298	0.015	0.005	2.91	0.358	0.066	5.39	0.839	0.157	5.34	1.788	0.313	5.72
s344	0.011	0.005	2.22	0.150	0.032	4.64	0.542	0.100	5.42	1.047	0.192	5.46
s382	0.533	0.098	2.96	0.533	0.098	5.44	1.736	0.314	5.53	0.519	1.460	5.68
s400	0.024	0.006	4.01	0.607	0.109	5.59	1.958	0.341	5.75	3.403	0.598	5.69
s444	0.628	0.120	3.34	0.631	0.117	5.38	0.631	0.119	5.70	6.858	1.174	5.84
s526	1.015	0.204	3.40	1.012	0.191	5.29	1.009	0.198	5.66	1.015	0.204	5.65
s820	0.085	0.024	3.55	3.684	0.640	5.76	9.053	1.595	5.72	17.333	3.122	5.55

ckt: circuit. Case A: Direct and Indirect Implication only. Case B: Case A + EBI. Case C: Case B + EFI. Case D: Case C + JFron. Seq. : sequential run with 1 thread. Par. : parallel run with 36 threads. s-u: speed-up ratio.

The results for the parallel deduction engine are first reported in Tables 4.1 and 4.2. We have run deduction engine in different scenarios. We conducted different deduction approaches with and without sharing of the implications among the nodes, reported in Tables 4.2 and 4.1, respectively. The speedup is computed against the single-processor scenario.

In Table 4.1, we have different cases. In case A, we have run the deduction engine for only direct and indirect implications. In case B, we have run the deduction engine with direct, indirect and EBI. In case C, we have run the deduction Engine with direct, indirect, EBI

Table 4.2: Parallel Deduction Engine **with** Implication Sharing between Nodes

ckt	Case B			Case C			Case D		
	Seq.	Par.	spd-up	Seq.	Par.	spd-up	Seq.	Par.	spd-up
s298	0.726	0.138	5.264	1.682	0.326	5.159	3.554	0.661	5.702
s344	0.224	0.050	4.522	0.634	0.117	5.428	1.208	0.226	5.354
s382	0.973	0.181	5.375	0.973	0.189	5.479	0.972	0.185	5.540
s444	1.773	0.358	4.950	12.380	2.173	5.677	12.380	2.173	5.697
s526	2.768	0.520	5.319	7.394	1.344	5.504	16.109	2.832	5.687
s820	11.733	2.220	5.286	43.411	7.701	5.637	83.642	15.009	5.573

Case B: Direct and Indirect Implication only+ EBI. Case C: Case B + EFI. Case D: Case C + JFron. Case E: Case D + JEnum. Seq. : sequential run with 1 thread. Par. : parallel run with 36 threads. spd-up: speed-up ratio.

and EFI implications. In case D, we have run the deduction engine with direct, indirect, EBI, EFI and JFron implications. For each case, we report the sequential run time (seq.), the parallel deduction time (par.) with 36 threads, and the speed-up ratio (spd-up). In Case A, the speed up varies from 2.22× (in s344) to 4.02× (in s400) with average speedup 2.84×. As we utilize more extended learning, we achieved more speed up. In case B, the the speed up varies from 4.64× (in s344) to 5.76× (in s820) with average speedup 5.16×. In case C, the the speed up varies from 5.42× (in s344) to 5.75× (in s400) with average speedup 5.2×. In case D, the the speed up varies from 5.46× (in s344) to 5.84× (in s44) with average speedup 5.6×. As we utilize more extended learning, more speedup was achieved because of the following reason. Direct and indirect implications are deduced faster than extended implications. This causes more threads to finish at the same time. Accordingly, this would cause a contention on choosing which thread should deduce the next node, and hence reduces the speedup from parallelism.

In Table 4.2, we have different cases when the clauses are shared between nodes, namely cases B, C, and D, as in Table 4.1. For each case, we likewise report the sequential run time (seq.), the parallel deduction time (par.) with 36 threads, and the speed-up ratio (spd-up). In Case B, the speed up varies from 4.522× (in s344) to 5.375× (in s382) with average speedup 5.031×. As we utilize more extended learning, we get more speed up. In case C, the the speed up varies from 5.159× (in s298) to 5.677× (in s444) with average speedup 5.346×. In case D, the the speed up varies from 5.354× (in s344) to 5.702× (in s298) with average speedup 5.401×. As we utilize more extended learning, we get more speed up for the same reason in the previous case (Table 4.1)

In Table 4.4, we report the time taken by our set-cover approach to select the clauses. In other words, we want to see what fraction of all the learned implications need to be added to the formula such that we can still achieve the same deductive power had we included all the learned clauses. We list the selection time for the two approaches (with/without sharing clauses). In each approach, we show time required in each case (cases are defined in the same way as in Tables 4.1 and 4.2). For Case B, the selection time for s298 is 0.60s when no clauses are shared, and .62s when clauses are shared. For Case C, the selection time for s444 is 2.22s when no clauses are shared, and 2.27s when clauses are shared. For Case D, the selection time for s820 is 10.39s when no clauses are shared, and 10.82s when clauses are shared. We conclude that the selection time is almost the same for both approaches and for every case. Also, we notice that the selection time is also very low, compared to the SAT-solving time, which will be reported subsequently in Table 4.6.

In Table 4.5, we report the percentage of the clauses after applying our filtering algorithms on all the learned clauses (i.e., using our set-cover filter and extended implication filtering [39]). We list the selection time for the two approaches (with/without sharing clauses). In each approach, we show the percentage of the clauses in each case (cases are defined in the same way as Tables 4.1 and 4.2). For Case B, the percentage of clauses for s298 is 34.99% with SC filtering and 20.69% with EI filtering when no sharing is used. However, this percentage is reduced with clauses sharing (12.43% with SC filtering and 7.89% with EI filtering). For Case C, the percentage for s400 is 29.25% with SC filtering and 22.05% with EI filtering when no sharing is used. However, this percentage is reduced with clauses sharing (8.66% with SC filtering and 5.61% with EI filtering). For Case D, the percentage for s526 is 34.74% with SC filtering and 25.88% with EI filtering when no sharing is used. However, this percentage is reduced with clauses sharing (11.28% with SC filtering and 6.87% with EI filtering). We notice that the number of clauses from SC filtering is always larger than the number of clauses from EI filtering whether we apply sharing or not. However, smaller is not necessarily better. Taking into account that the performance of BMC is enhanced with SC filter (discussed next in Table 4.6). We conclude that SC filtering takes into account those important clauses that EI filtering does not, which in turn reduces the search space.

In Table 4.3, we report the size of used formulas in our experiments. Number of variables varies from 376001 for s382 circuits to 780001 for s344 circuits. which provides a large search space for SAT solver. In addition, no. of clauses varies from 1037422 for s382 to 2209183 for s832. These formulas uses long time to solve (see Table 4.6 the last column "orig").

In Table 4.6, we report the time used to solve the BMC instances. We have used Minisat 2.0 [96] to solve each CNF formula. k is the number of unrolled timeframes. We report the time for the two approaches (with/without sharing clauses between nodes). In each approach, we report the different cases. In every case, we list the solving time using the set-covering selection is used (SC) as well as using the Extended implication clauses is used (EI). In addition, the time to solve the original CNF formula without any added static implication clauses is reported (Orig.). Consider circuit s298, unrolled for 4000 time-frames. The original solving time was 2107 seconds. With our set-cover filtering, the solving time varied from 15 to 17 seconds. On the other hand, the Extended Implication filtering varied between 57 and 59 seconds. For all circuits, we obtain an average of 1.74$\times$ speedup due to our selection algorithm (compared with extended implication approach). Also, we gain an average 63.52$\times$ speed-up compared to the original CNF formula. Moreover, we have an average 55.32$\times$ overall speed-up against the original problem.

Table 4.3: Formulas Size

ckt	no. of variables	no. of clauses
s298	568001	1615465
s344	780001	1923587
s382	376001	1037422
s400	388001	1081400
s444	422001	1175324
s526	446001	1384940
s832	658001	2209183
s1196	575001	1602774
s1238	554001	1613771

ckt: circuit.

4.6 Conclusion

In this chapter, we have three contributions. First, we propose a parallel framework to deduce different static implications. We show that we could gain a 5.6× speed-up on 36 core machine. Second, we formulate the clause selection problem as a set-cover problem. Third, we propose a novel low-cost greedy set-covering based selection algorithm for choosing static implication. In comparison with extended implication approach, we achieve an average

Table 4.4: Clause Selection Time

ckt	# IN	# OUT	# FF	no shared clauses			Clauses shared between nodes		
				Case B	Case C	Case D	Case B	Case C	Case D
s298	3	6	14	0.60	0.62	0.62	0.62	0.62	0.63
s344	9	11	15	0.88	0.91	0.92	0.90	0.90	0.91
s382	3	6	21	1.43	1.44	0.92	1.53	1.50	1.56
s400	3	6	21	1.63	1.64	1.65	1.70	1.68	1.651
s444	3	6	21	2.07	2.22	2.25	2.25	2.27	2.29
s526	3	6	21	2.03	2.04	2.08	2.01	2.04	2.05
s820	18	19	5	9.98	10.26	10.39	10.06	10.71	10.82

ckt: circuit. # IN : no. of inputs. # OUT: no. of outputs. # FF: no. of state elements. Case B: Direct and Indirect Implication only+ EBI. Case C: Case B + EFI. Case D: Case C + JFron.

1.74× speed up due to our selection algorithm. Moreover, we have an average 55.32× overall speed-up compared to the original problem.

Table 4.5: Remaining Clause Percentage(%)

ckt	no shared clauses						Clauses shared between nodes					
	Case B		Case C		Case D		Case B		Case C		Case D	
	SC	EI	SC	EI	SC	EI	SC	EI	SC	EI	SC	EI
s298	34.99	20.69	35.11	21.19	33.29	20.75	12.43	7.89	9.49	6.14	9.45	6.23
s344	30.60	19.55	31.57	23.25	31.95	23.71	10.67	7.19	15.98	12.69	18.69	14.93
s382	27.33	19.37	27.96	20.82	26.85	20.81	11.00	6.17	8.27	5.07	8.28	5.20
s400	28.62	20.64	29.25	22.05	27.26	21.91	11.48	6.87	8.66	5.61	8.66	5.75
s444	31.41	20.47	33.82	26.12	32.70	25.50	6.40	3.67	10.21	7.39	10.14	7.33
s526	35.08	24.08	35.33	24.94	34.74	25.88	10.96	5.92	11.19	6.17	11.28	6.87

ckt: circuit. Case B: Direct and Indirect Implication only+ EBI. Case C: Case B + EFI. Case D: Case C + JFron. SC. : set-covering selection approach. EI: extended implication approach.

Table 4.6: BMC Time

ckt	k	no shared clauses						Clauses shared between nodes						Orig.
		Case B		Case C		Case D		Case B		Case C		Case D		
		SC	EI	SC	EI	SC	EI	SC	EI	SC	EI	SC	EI	
s298	4000	17.29	57.84	17.39	59.08	17.67	59.39	17.51	58.3	17.62	59.6	15.29	59.63	2107.92
s344	4000	242.86	527.96	243.98	585.5	243.69	593.53	255.60	588.62	257.7	595.68	255.4	585.34	1910.38
s382	2000	33.88	46.6	32.78	44.41	18.08	41.74	15.74	47.10	26.4	44.97	25.77	41.61	1365.38
s400	2000	24.37	44.37	31.79	50.02	44.16	34.33	24.75	52.10	21.73	48.20	31.88	43.95	1303.74
s444	2000	26.27	36.060	28	37.11	27.18	37.79	26.71	36.06	26.32	36.05	26.36	38.03	2145.82
s526	2000	7.82	9.94	3.54	4.12	3.53	4.1	6.76	10.72	5.36	1.19	5.31	1.21	2004.53
s832	2000	5582	5915	6183	5072	5579	5034	2196	3305	1413	1754	889.1	897	6323.62
s1196	1000	256	494	470	890	257	1163	223	287	184	1179	237	1103	923
s1238	1000	203	274	211	365	190	377	225	199	208	278	203	171	1079

Case B: Direct and Indirect Implication only+ EBI. Case C: Case B + EFI. Case D: Case C + JFron. SC. : set-covering selection approach. EI: extended implication approach. Orig. : original CNF without any implication clauses.

Chapter 5

TACUE: A Timing-Aware Cuts Enumeration Algorithm for Parallel Synthesis

In this chapter, literture survey on timing-aware synthesis and our contributions is presented. Secondly, TACUE algorithm is presented in the second section. In the third section, we apply TACUE algorithm to different cutting approaches: *topology-aware* cuts and *topology-masking* cuts. We propose an efficient parallel synthesis framework for TACUE cuts in the fourth section. Our results are presented in fifth section. In the last section, the chapter is concluded.

5.1 Literature Survey and Contributions

Achieving timing-closure has become one of the hardest tasks in logic synthesis due to the required stringent timing constraints over very large circuit designs. In this Chapter, we propose a novel synthesis paradigm to achieve timing-closure called Timing-Aware CUt Enumeration (TACUE) [97]. In TACUE, optimization is conducted through three aspects: First, we propose a new divide-and-conquer strategy that generates multiple sub-cuts on the critical parts of the circuit. Secondly, cut enumeration have been applied in two cutting strategies. Thirdly, we proposed an efficient parallel synthesis framework to reduce computation time for synthesizing TACUE sub-cuts. We conducted experiments on large and difficult industrial benchmarks.

As far as we know, there are two major bodies of work that are related to delay-optimization with cuts generation. The first was proposed by Baneres *et al.* [98], in which a dominator-based partitioning technique is used to find topologically ordered clusters in the circuit-under-optimization (CUO), followed by logic restructuring on these clusters. The second timing-aware work conducts cuts enumeration on *And-Inverter-Graphs (AIGs)*. For example, Chatterjee *et al.* [99] had proposed a cut factorization scheme to enumerate bounded size cuts up to 16 inputs. Their technique is usually used in technology mapping and re-writing. Martinello *et al.* [100] had extended the concept of factor cuts to KL-cuts, where K is number of inputs and L is number of outputs in a circuit cut. Because factor cuts are not restricted to convex cuts [101], an uncontrollable amount of area increase may result. This increase in area would lead, in many cases, to undesirable degradation in circuit timing behavior. Although KL-cuts extend the application of factor cuts to peep-hole optimization and regularity extraction, however, they still suffer from having a restricted number of inputs and work exclusively with *AIGs*.

Our work is uniquely different from the previous work in timing-aware synthesis from several perspectives. Unlike Baneres work [98] that only groups nodes in the critical paths and generates a single solution, we enumerate sub-cuts in dominator-based partitions. As a result, we are less likely to be stuck at local optima as our approach explores more possible solutions to improve the timing behavior of CUO. Secondly, their approach allows the grouping of different dominant cuts, which can result in a significant increase of area, thereby indirectly degrade the timing performance of the optimized solution. On the contrary, we propose two different cutting strategies, one that aims to preserve the topological structure of CUO and the other one would investigate other possible topologies. Such strategies allow for more control on the optimization choices. In contrast with Chatterjee work [99] on factor cuts, our approach could handle larger cuts (TACUE have been tested on up to 60 inputs sub-cuts). In addition, our technique runs on a general circuit graph. Thus, each vertex represents a general Boolean function (not just "*AND*" function as in *AIG*). Subsequently, our method can be applied in all synthesis stages and it is not restricted only to technology mapping stage. Thirdly, previous experiments were conducted on fairly small circuits (ISCAS'85, ISCAS'89, ITC'99 and some other small circuits). In our case, we conducted experiments on very large industrial benchmarks. In addition, previous methods only optimized their circuits using outdated SIS tool [102]. In contrast, we first apply extensive optimization techniques (i.e. BooleDozer [103], ABC [104], SIS ...etc.) before using our synthesis framework to show that our synthesis framework exhibits a superior outcome for very-large very-hard-to-optimize circuit instances.

Our contributions are summarized as follows.

- we propose a novel *Timing-Aware CUts Enumeration (TACUE)* algorithm to generate timing critical sub-cuts in CUO.
- we apply our TACUE algorithm in two different cutting strategies. 1) *Topology-aware* cutting strategy, in which we preserve the general topology (i.e., connectivity) of the circuit, and 2) *topology-masking* cutting strategy, in which we relax this constraint and allow the connectivity to change in CUO.
- we propose an efficient parallel synthesis framework for applying different synthesis optimization techniques in the generated TACUE sub-cuts.

5.2 Timing-Aware Cut Enumeration

Definitions:

Vertex slack: the slack of a vertex v is defined as follows:

$$s_v = t_r - t_a \tag{5.1}$$

where t_r is the required arrival time and t_a is actual arrival time.

Direct children of a cut: for a cut $C = (a, S)$, it is the set of direct descendant of the root a. For example, in Fig. 5.3, nodes b, f and g are the direct children of cut $(a, \{a\})$.

Convex subgraph: A subgraph, S, of graph G is convex if for any pair of vertices $v, w \in S$, all the shortest paths from v to w in G are fully contained in S.

Convex Cut: A convex cut of a graph $G = (V, E)$ is a partition of V into V_1 and V_2 such that both sub-graphs of G induced by V_1 and V_2 are convex.

Basic idea of our algorithm: we target to generate time-critical sub-cuts from bigger cuts. In other words, given a cut in CUO, our objective is to enumerate, heuristically, sub-cuts in the critical paths of CUO. These sub-cuts will be passed later to various logic synthesis optimization techniques. If they successfully find better solutions, a heuristic is used to select the optimal choice among them. Then, the optimal choice will be admitted to CUO, and hence, this would contribute to the overall timing closure of CUO.

TACUE algorithm for enumerating critical sub-cuts is shown in Fig. 5.1. TACUE starts with a critical cut C which is required to be enumerated. TACUE uses Breadth-first (BF) approach for enumerating C. First, TACUE creates the base sub-cuts, which contains the root with all direct children by calling *createBaseCut* (line 2). TACUE adds the base cut to a queue (line 3). Then it loops until the queue becomes empty (line 4-10). It dequeues a sub-cut from the queue (line 5) and calculates the possible combinations that could be conducted on the boundary of this sub-cut (line 6). Finally, it enumerates all combinations

```
1:  TACUE($C$)
2:  $c_{base}$ = createBaseCone($C$)
3:  $queue$.enq($c_{base}$)
4:  while queue.size() $> 0$ do
5:      $v = queue$.deque()
6:      $combCount$ = CalCombCount($v$.boundary())
7:      for $i = 1$ to $combCount$ do
8:          nPairEnum($v, i, queue$)
9:      end for
10: end while
```

Figure 5.1: TACUE algorithm outlines.

using *nPairEnum* function (line 7-9).

Fig. 5.2 depicts the algorithm for *nPairEnum* function. *nPairEnum* enumerates all i combinations at the boundary of a sub-cut v and adds it to *queue*. We measure the criticality of a vertex by *vertex slack* which is defined in Eq. 5.1. *nPairEnum* enumerates only the critical vertices, that is, it has a cut-off value on vertex slack. Thus, vertices with slack larger than certain threshold will not be added to the enumerated sub-cuts list.

Fig. 5.3 illustrates our approach in which TACUE generates the time-critical sub-cuts. Let the slack cut-off value be 1. TACUE takes the original cut t_O (the cut that needs to be enumerated as depicted in Fig. 5.3.a) and a set of tuning parameters. The cut-enumeration

```
nPairEnum($v, i, queue$)
$c$ = getCritVertex($v$)
if $i = 1$ then
    for $j = 1$ to $c$.length() do
        $cnew$ = createNewCut($v, c[i]$)
        $queue$.enq($cnew$)
    end for
else
    for $j = 1$ to $c$.length() do
        $vnew$ = createNewCut($v, c[i]$)
        nPairEnum($vnew$, $i - 1$, $queue$)
    end for
end if
```

Figure 5.2: Enumerates all n combinations at the boundary of a sub-cut.

algorithm starts with the root a of t_O and enumerates sub-cuts in a BF manner. The vertices of cut t_O are labeled with nodes $a, b, \ldots, etc.$ The direct children of a are g, b and f. Without loss of generality, let the slack of vertex g be 3 (which is greater than the cut-off slack value). Then, vertex g and all of its children will not be included in any enumerated sub-cuts. Now, TACUE will enumerate all possible combinations for the other direct children nodes b and f. As vertex b has a slack value of zero (b is in a critical path, and all critical vertices have a zero slack) and is a direct child of a, it will be included in any enumerated sub-cuts. For

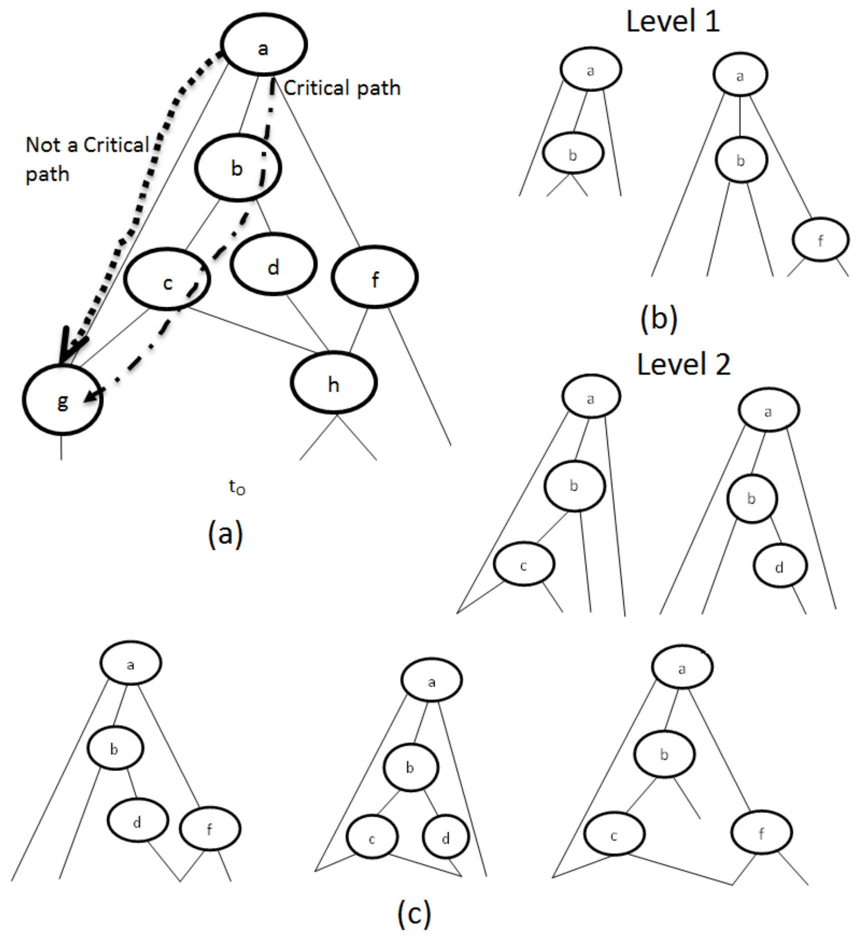

Figure 5.3: Cut enumeration illustration. (a) the original cut, (b) sub-cuts generated in the first level, and (c) sub-cuts generated in the second level.

node f, let us assume that its slack is 1, thus it is not on the critical path. However, its slack is smaller than the cut-off value, thus, it will be included in the future sub-cuts. However, because it is not on a critical path, it does not need to be included in every sub-cut. Thus, at this level, we have two sub-cuts $(a, \{b\})$ and $(a, \{b, f\})$ as depicted in Fig. 5.3.b.

In the second round, the cut-enumeration algorithm will enumerate cuts in the next level for all cuts in the first level as depicted in Fig. 5.3.c. It starts with cut $(a, \{b\})$, the direct children of this cut are c, d and f. We have 7 combinations, The children vertices will be enumerated one by one. The resultant sub-cuts will be $(a, \{b, c\})$, $(a, \{b, d\})$ and $(a, \{b, f\})$. The next step would be enumerating them two by two. The resultant sub-cuts will be $(a, \{b, c, d\})$, $(a, \{b, c, f\})$ and $(a, \{b, d, f\})$. Finally we consider three nodes at a time. Thus, all three direct children will be taken altogether in sub-cut $(a, \{b, c, d, f\})$. This cut is not shown in the Fig. 5.3 because the cut-off limit for generated sub-cuts is 8 (which is a user input). In the case that the cut-off limit is increased, this final cut and sub-cuts in deeper levels will be generated and added.

5.3 Applications of cut Enumeration

In the previous section, we described how TACUE takes a cut C and enumerates timing-critical sub-cuts from C. However, we did not describe how to generate cuts used by TACUE. In this section, we describe two divide-and-conquer strategies to generate these cuts.

5.3.1 Topology-Aware Cuts

In some cases, we need to preserve the connectivity of the CUO. This is useful when we start with an initially "good" topology, and we would want to keep the same topology to be used later.

```
1: GenerateStructAwareCuts()
2: GenDomCuts()
3: $CL$ = FilterCuts()
4: **for all** $C$ in $CL$ **do**
5:     TACUE($C$)
6: **end for**
```

Figure 5.4: Topology-Aware cuts generation algorithm.

One way to achieve connectivity preservation in CUO is by restricting changes to be made only inside each dominant cut. This intuition is motivated by the self-contained nature of dominant cuts, that is, any vertex in a dominant cut does not fanout to any vertex outside that cut. In other words, we model each dominant cut as a single super node and we restrict the change of logic to occur only inside these super nodes. The main benefit from this restriction is to keep the general connectivity of circuit nearly the same, which is helpful when the CUO already has a ”good” topology.

Fig. 5.4 depicts *topology-aware* cuts generation algorithm. It starts by generating dominant cuts (line 2). Then the critical dominant cuts are enumerated (line 3). Finally, it iterates on all filtered cuts and enumerates them (lines 4-5).

5.3.2 Topology-Masking Cuts

Fig. 5.5 depicts the *topology-masking* cuts generation algorithm. Contrary to the *topology-aware* cutting strategy, this strategy does not take CUO connectivity preservation into consideration. Because we may start with an initially poor timing-performance CUO or locally optimized CUO. The algorithm starts with identifying critical outputs of CUO and generate cuts from these outputs (line 2). These cuts have the critical sink vertices as a root and all its fan-in source vertices as the boundary vertices. Secondly, it enumerates sub-cuts from these critical cuts (lines 4-5). If a sub-cut is being accepted as a new solution, it is committed to CUO. The aforementioned process is repeated at the critical boundary vertices of the new committed sub-cut (line 7). For example, consider the cut t_O in Fig. 5.3.a, t_O has a root a. Suppose that, without losing the generality, the sub-cut $(a, \{b, c\})$ had been accepted and committed to CUO. Thus, $GenerateStructMaskCuts$ starts at the direct children of the sub-cut $(a, \{b, c\})$, which in this case are d, f, g and h. $GenerateStructMaskCuts$ identifies the critical vertices (from the slack value of each vertex), then it applies TACUE on them. As g and h are not critical (as slack value is $2 <$ *cut-off slack* value), they are pruned. Thus, sub-cuts are enumerated from vertices d and f. If it is unable to find a better cut, $GenerateStructMaskCuts$ identifies the critical direct children vertices (in this case, b is the only critical direct children of a). These direct children are enumerated in order to explore a better solution for them. $GenerateStructMaskCuts$ repeats on critical boundary vertices of the committed sub-cuts until it reaches the source vertices.

```
1: GenerateStructMaskCuts()
2: $CL$ = GetCritOutCuts()
3: **repeat**
4:     **for all** $C$ in $CL$ **do**
5:         TACUE($C$)
6:     **end for**
7:     $CL$ =getNextCritCuts()
8: **until** $CL$ is NOT empty
```

Figure 5.5: Topology-Masking cuts generation algorithm.

5.4 Parallel Synthesis Framework

In this section, we first describe a sequential strategy for TACUE, followed by a naive and optimized parallel strategies for TACUE in a synthesis framework.

5.4.1 Sequential Algorithm

Fig. 5.6 depicts a sequential algorithm for TACUE. It enumerates sub-cuts individually (line 4). Then, it applies synthesis optimization (e.g., BDD bi-decomposition and time-driven logic bi-decomposition) to each sub-cut (line 5). It iterates this process on all sub-cuts until it reaches the stopping criteria. Due to the sequential nature of the flow, it may take an excessive amount of time to reach a good solution.

```
1: SeqSynth()
2: $CL$ = getCuts()
3: for all $C$ in $CL$ do
4:     $SC$ = enumOneCut($C$)
5:     synthesize($SC$)
6: end for
```

Figure 5.6: Sequential Synthesis Algorithm

5.4.2 Parallel Framework - Naive Approach

Fig. 5.7 extends the sequential algorithm to a naive parallel implementation. First, It generates all sub-cuts in one level (line 5). Secondly, it applies synthesis optimization algorithms in all of these sub-cuts in parallel (line 6). Thirdly, it repeats these steps to the next level until it reaches the stopping criteria. The main problem in this approach is that we cannot guarantee load balance in each level. For instance, if we may have 4 workers and only one sub-cut in a level, we will end up having 3 idle workers on that sub-cut.

5.4.3 Parallel Framework - Optimized Approach

In order to have a well-balanced parallel framework, we propose to split the cut enumeration step from parallel synthesis. This is based on noting that cut enumeration only takes a small fraction of time compared to the synthesis step. Thus, we will not have a tangible performance degradation if we enumerate the cuts sequentially. Meanwhile, we boost the

```
1: NaivParSynth()
2: $CL$ = getCuts()
3: **for all** $C$ in $CL$ **do**
4:    **repeat**
5:       $SC$ = enumDirChildCut($C$)
6:       ParSynth($SC$)
7:    **until** reach stopping criteria
8: **end for**
```

Figure 5.7: Naive Parallel Synthesis Algorithm

performance of the major part of our framework by having a full parallelism in the synthesis optimization stage.

Fig. 5.8 lists our proposed parallel framework algorithm, and Fig. 5.9 illustrates the framework pictorially. As shown in Fig. 5.8, sub-cuts are first enumerated sequentially. Next, they are evenly distributed among different workers (which achieves a well-balanced work load). Thirdly, each worker applies different synthesis optimization techniques on sub-cuts assigned to it. All successful sub-cuts are sent back to the master process, and the master determines which sub-cut would be committed to the original circuit. The criteria used for this try to decrease the number of levels while maintaining a limited percentage area increase.

```
1: ParrSynth()
2: $CL$ = EnumerateCuts()
3: for all $C$ in $CL$ do
4:     ParSynth($C$)
5: end for
```

Figure 5.8: Sophisticated parallel Synthesis Algorithm

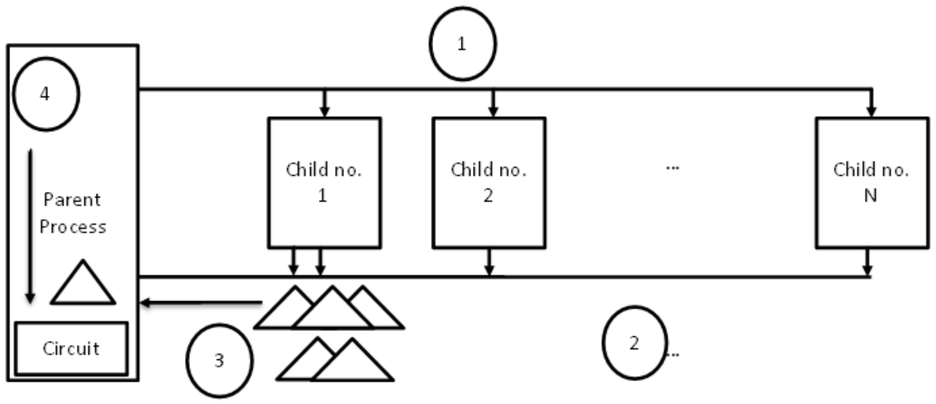

Figure 5.9: Parallel synthesis Framework.

5.5 Results

The proposed TACUE algorithm and parallel synthesis framework have been developed with C++ and the performance was evaluated on 11 dedicated 2.7 GHz Intel Xeon cores, running a 64-bit Linux distribution. We have compiled our program with g++ under -O3 option. We have used large industrial benchmarks to evaluate our work. The characteristics for our benchmarks are reported in Table 5.1. We applied many optimizations before applying TACUE (i.e., SIS, ABC, BooleDozer . . . etc.). In doing so, we guarantee that our benchmarks are already "well optimized", making further optimization harder. In addition, the benchmarks are also hard to synthesize under their timing constraints. In other words, timing closure was not yet achieved even though they had gone through a sophisticated timing-aware optimization (see Table 5.1 for the original worst time slack values before applying our work).

Table 5.1 lists the characteristics of the industrial benchmarks used. These data are reported after applying many state-of-art industrial and public-access optimization techniques but before applying TACUE. For each circuit, the number of inputs (IN) is first listed, followed by the number of outputs (OUT), the number of sequential elements (DFF), the total number of gates (GATES), the number of levels (LV), the total area (AR) for the combinational part of the benchmarks (measured in number of basic unit cells). Finally, the worst slack (measured in picoseconds) is reported in the last column. Note that all of the circuits have a negative worst slack, which mean that, **the current state of art synthesis tools could not achieve timing-closure on these designs.**

5.5.1 Topology-Aware Cuts

Table 5.2 reports the number of levels after applying TACUE using the *topology-aware* cuts. We apply TACUE for 2 iterations (Itr. # 1 and Itr. # 2) with BDD bi-decomposition (BD) and time-driven logic synthesis (TD). The last column reports the *Maximum Level Reduction Percentage (MLRP)* for each case. The results showed that we can reduce the number of

Table 5.1: Circuit Statistics

ckt	IN	OUT	DFF	GATES	LV	AR	WS
ia0	279	624	968	12339	27	48602	-36.335
ia1	229	517	870	146608	26	47110	-28.619
ia2	204	508	840	127823	26	39667	-19.344
ib0	253	626	963	149566	26	48234	-37.644
ib1	201	515	867	146530	27	46763	-26.999
ib2	176	505	841	127782	27	39126	-18.384

levels by 14.81% for *topology-aware* cuts. Table 5.3 reports the corresponding circuit area. The last column reports *Maximum Area Increase Percentage (MAIP)* for each case. Our approach showed that TACUE has a very slight area increase of only 0.475%.

Table 5.4 reports the number of dominant cut (DC) computed and the number of accepted dominant cuts (AC). AC is defined as the number of dominant cuts that the synthesis algorithm had successfully reduced its number of levels. We report results for 3 iterations. For each iteration we run both BDD Bi-decomposition and time-driven logic bi-decomposition. We always had a higher acceptance rate in TD case over BD because TD tends to better optimize the cut in terms of delay. In addition, the number of AC decreases with the increasing number of iterations because the *topology-aware* cutting strategy restricts TACUE to preserve the topology. Thus, we do not have a large room for changing the design struc-

Table 5.2: No. of Level Reduction with Topology-Aware Cuts

ckt	Itr. # 1		Itr. # 2		MLRP (%)
	BD	TD	BD	TD	
ia0	25	24	25	23	14.81
ia1	25	25	24	23	11.53
ia2	25	24	23	23	11.53
ib0	25	24	24	23	11.53
ib1	25	24	25	23	14.81
ib2	25	24	24	23	14.81

ture.

5.5.2 Topology-Masking Cuts

Table 5.5 reports the number of levels after applying TACUE with *topology-masking* cuts. TACUE was applied for 3 iterations for both BDD BD and TD as before. The results show that we could get up to 22.22% reduction (compare with 14.81% for *topology-masking* cuts in Table 5.2) in the number of levels. This is due to that we did not require TACUE to preserve the topology.

Table 5.6 reports the percentage area increase from the topology-masking technique. We

Table 5.3: Area Report for Topology-Aware Cuts

ckt	Itr. # 1		Itr. # 2		MAIP (%)
	BD	TD	BD	TD	
ia0	48698	48630	48668	48616	0.1975
ia1	47148	47213	47174	47231	0.2569
ia2	39656	39676	39666	39713	0.1160
ib0	48304	48223	48302	48195	0.1451
ib1	46771	46899	46782	46962	0.4256
ib2	39216	39312	39227	39275	0.475

noticed that TD perform poorly (especially with the increase in number of iterations) from the area point of view (up to 44.7% area increase). This is because we allow for changing the topology, and TD had a large acceptance rate. On the other hand, BD synthesis would reduce number of levels (up to 22%) while maintaining a adequate area increase (7.21% in circuit ib2). This increase in area will be reflected on the physical synthesis stages as it will be discussed later. We also noticed that, if we restrict the number of iteration in this stage to one, we would get enhancement on both logic and physical synthesis stages. This is because that number of levels is highly reduced from the first iteration with a limited percentage area increase.

Table 5.4: Dominant Cut Statistics for Topology-Aware Cuts

ckt	Itr. # 1				Itr. # 2				Itr. # 3			
	BD		TD		BD		TD		BD		TD	
	DC	AC	DC	AC	DC	AC	DC	AC	DC	AC	DC	AC
ia0	710	60	710	149	790	14	812	60	784	3	817	24
ia1	653	56	653	144	720	14	772	56	719	1	771	20
ia2	634	35	634	118	640	7	669	45	641	2	709	18
ib0	675	61	675	161	721	8	773	65	745	3	766	29
ib1	593	55	593	136	628	16	688	70	652	8	717	28
ib2	617	41	617	131	630	8	674	48	636	2	683	17

5.5.3 Impacts of TACUE on Physical Synthesis

We have run our physical synthesis tool on the circuits optimized by TACUE for *topology-aware* cuts. We report the worst slack in Table 5.7. The *Maximum Worst Slack Increase Percentage (MWSIP)* is reported in the last column. We could gain 21.16% increase of worst slack on average and 45.72% in the best case.

The impact of TACUE on physical synthesis is also investigated. Table 5.8 reports the worst slack after each iteration. Results shows that we could gain 18.54% increase on the worst slack on average and 31.23% in the best case.

Table 5.5: No. of Level Reduction with Topology-Masking Cuts

ckt	Itr. # 1		Itr. # 2		Itr. # 3		MLRP (%)
	BD	TD	BD	TD	BD	TD	
ia0	26	24	23	22	23	21	22.22
ia1	25	23	23	21	23	21	19.23
ia2	25	23	23	21	23	21	19.23
ib0	24	23	23	21	23	21	19.23
ib1	25	24	23	21	22	21	22.22
ib2	25	24	23	22	22	21	22.22

We also manually conduct experiments with different tuning parameters to optimize the overall flow of our synthesis tool. Table 5.9 depicts the best results we could obtain for each benchmark. We report worst slack and area in each benchmark for base case (BC) and best case (BSC). In addition, we report the synthesis technique (ST) we had utilize in the best case. We have 3 cases in which BD is superior to TD and other 3 cases in which TD is superior to BD. Moreover, we report the cutting technique (CT) we had used (TA: for *topology-aware* and TM: for *topology-masking*). The results shows that TM is generally superior to TA. The Worst Slack Reduction percentage (WSRP) show that we could obtain an average 21.98% and up to 45.72% in the best case (for ia2). Results on *Level Reduction Percentage (LRP)* show that we have an average of 9.37% LRP with minimum of 3.84% (in

Table 5.6: Area Report for Topology-Masking Cuts

ckt	Itr. # 1		Itr. # 2		Itr. # 3		MAIP
	BD	TD	BD	TD	BD	TD	
ia0	48889	49658	50829	60527	51773	66322	36.5
ia1	47753	52981	48127	61422	49415	67599	43.5
ia2	39702	39757	40113	48385	40447	54049	36.3
ib0	49299	54703	51004	62566	51376	68565	42.15
ib1	47267	50691	49022	58715	50326	67698	44.7
ib2	39315	39352	40891	48683	41957	52999	35.5

ia1) and 14.81% (in ia0). Results for *Area Increase Percentage* (AIP in the last column) show that we have an average of 1.869% increase in area with min. increase of 0.2569% (in ia2) and maximum area increase of 6.52% (in ia0).

5.5.4 Parallel Synthesis Framework

Table 5.10 reports the time required for TACUE and our parallel Synthesis framework to optimize our benchmarks. The time is reported for one iteration of *topology-aware* cuts and our synthesis method is time-driven logic bi-decomposition. We report time for 1, 2, 4, 6, 10 and 20 processes. We gain an average speed-up of 2.18×, 3.99×, 5.42×, 7.12× and

Table 5.7: Worst-Slack Report for Topology-Aware Cuts

ckt	Itr. # 1		Itr. # 2		Itr. # 3		Itr. # 4		MWSIP (%)
	BD	TD	BD	TD	BD	TD	BD	TD	
ia0	-32.602	-29.72	-31.342	-31.237	-33.355	-31.046	-28.579	-29.501	21.35
ia1	-28.213	-29.103	-26.842	-26.579	-28.333	-29.157	-26.648	-29.61	7.13
ia2	-15.803	-20.81	-14.583	-10.499	-13.019	-15.39	-15.395	-14.361	45.72
ib0	-34.172	-35.763	-42.46	-32.997	-34.076	-33.973	-33.525	-39.275	10.94
ib1	-24.881	-22.204	-22.861	-25.984	-26.114	-26.074	-25.387	-20.398	24.45
ib2	-17.331	-21.352	-15.19	-20.382	-15.81	-24.511	-17.126	-19.164	17.37

8.78× on 2, 4, 6, 10 and 20 processes, respectively. The speed-up showed that we have a good work balance. The reason for the super-linear speed for 2 processors and almost linear for 4 processors is that these are conducted on quad-core processors. Thus, in case of 2 and 4 processes we get benefit from locality and advanced parallelism features on the same processor. Our results show that we still get noteworthy speed-up for 6 and 10 processes. This is due to the fact that TACUE takes a small fraction of the computation. In addition, the way we organize the parallelism framework is efficient.

Table 5.8: Worst-Slack Report for Topology-Masking Cuts

ckt	Itr. # 1		Itr. # 2	Itr. # 3	Itr. # 4	MWSIP
	BD	TD	BD	BD	BD	
ia0	-35.447	-31.306	-30.485	-28.368	-30.714	21.92
ia1	-26.293	-36.491	-29.989	-31.504	-26.372	8.13
ia2	-18.039	–	-19.145	-13.302	-16.102	31.23
ib0	-32.189	-40.093	-35.058	-36.468	-35.822	14.49
ib1	-25.369	-29.457	-23.856	-24.208	-27.641	11.64
ib2	-16.405	-14	-16.88	-19.526	-18.854	23.85

5.6 Conclusion

In this chapter, we presented a novel paradigm to accomplish timing closure of very large, previously optimized circuits. In order to tackle the scalability problem in our industrial benchmarks, we propose a divide-and-conquer heuristic, which we call Time-Aware CUt Enumeration (TACUE) algorithm. The basic idea behind TACUE is to generate many well-chosen sub-cuts along the critical paths of the circuits. We apply different synthesis techniques to these sub-cuts, and we choose the best solution in terms of delay and area. Some circuits start with a "good" topology, while others do not have this feature. Thus, sometimes we need to make a decision whether we want to keep the current topology or not.

Table 5.9: Physical Synthesis Best Results Summary

ckt	WS		AREA		CT	ST	# of itr.	WSRP(%)	LRP(%)	AIP (%)
	BC	BSC	BC	BSC						
ia0	-36.335	-28.368	48602	51773	TM	BD	3	21.93	14.81	6.52
ia1	-28.619	-26.293	47110	47753	TM	BD	1	8.13	3.84	1.36
ia2	-19.344	-10.499	39667	39713	TA	TD	2	45.72	11.53	0.2569
ib0	-37.644	-32.189	48234	49299	TM	BD	1	14.49	3.84	2.21
ib1	-26.999	-22.204	46763	46899	TA	TD	1	17.76	11.11	0.2908
ib2	-18.384	-14	39126	39352	TM	TD	1	23.85	11.11	0.5776

Accordingly, we have proposed two different cutting strategies to handle this issue. Finally, we also proposed an efficient parallel synthesis framework for TACUE. Significant reductions in worst slack was achieved with only a slight to moderate area overhead. Although TACUE, with synthesis framework, seems to be sequential in nature on first impression, we could come up with an elegant way to separate these data dependencies and sharing. The results show that we could gain almost linear and sometimes super-linear speedups.

Table 5.10: parallelism Results (time measured in seconds)

ckt/np.	1	2	4	6	10	20
ia0	1781.05	859.67	437.57	326.325	264.32	205.6
ia1	2015.77	935.08	515.96	351.55	303.3	227.6
ia2	1377.05	600.56	333.41	248.99	183.47	157.7
ib0	1735.25	903.72	497.57	368.13	280.8	208.3
ib1	1670.9	732.9	404.19	310.09	219.92	193.05
ib2	1588.7	658.23	374.2	277.12	197.1	168.14

Chapter 6

Novel SAT-based Invariant-Directed Low-Power Synthesis

In this chapter, we present our algorithm for low-power synthesis. A literature survey for clock-gating algorithm is presented in the first section. Secondly, a motivating example is illustrated. We present heuristics to speedup inductive invariant generation, invalidation and proving is followed. Finally, we present results and conclusion.

6.1 Literature Survey and Contributions

Dynamic power consumption is a critical concern in the design of both high performance and low-power circuits. Clock-gating is one of the most efficient and prominent approaches to reduce dynamic power. In this Chapter, (1) we propose the first scalable SAT-based approaches for Observability Dont Care (ODC) clock gating; (2) we intelligently choose those inductive invariants candidates such that their validation will benefit the purpose in clock-gating-based low-power design. Our approach shows an average 23.2 % reduction in dynamic power with an average 9.5% increase in area [105].

Clock-gating could be classified into two main approaches [33]. In one approach, Observability Dont Cares (ODCs) are used to gate state elements that are not observed by the circuit outputs [32]. In the other approach, stability conditions (STC) are used to gate state elements that hold stable value for two or more consecutive time frames [106].

In the ODC-based clock gating approach, many algorithms have been proposed. In one technique [71], the combinational part of the circuit is first decomposed, and combinational ODCs are identified and blocked whenever appropriate [70]. However, this approach has scalability limitations as it depends on Binary Decision Diagram (BDD) decompositions. As far as we know, the only ODC-based scalable clock-gating algorithm is proposed in [32]. Benini *et al.* [32] proposed a scalable clock gating at the Register Transfer Level (RTL). It considers the data inputs of a steering module (i.e., multiplexers and tri-state elements). When a particular line is selected, the rest of the lines become unobservable for computing the other signal. The ODCs are propagated backwards to maximize the set of signals that should be gated. The main drawback of such approach is that it may miss many potential parts for optimization, and it is restricted to only the combinational part (not sequential) of the circuit.

In the STC-based clock gating approach, Hurst [107] proposed the first SAT-based STC clock gating to address the scalability of STC-based clock-gating problem. However, he did not explore the maximum gating condition which is the main component of STC-based technique. This is because the monolithic approach is used in dealing with model checking. Therefore, they provide an ad-hoc method for clock-gating. Moreover, the lack of computing the overall signal probability of their clock gating condition adds a limitation to their approach. Lin *et al.* [34] proposed a SAT-based STC clock gating technique based on interpolation computation for maximum clock gating calculation. Thus, this would restrict the scalability of the approach. In general, extracting STCs are shown to be a challenging task [33],[106]. Wiener *et al.* [33] use data mining to generate inductive invariant candidates. However, they did not prove them formally. Instead, they use ”Human Experts” to judge on the correctness of those candidates. This approach has two major drawbacks. First, a ”human expert” may not be available or it may be error-prone and this approach does not scalable well. In addition, they depend heavily on randomized constrained simulation which can be computationally expensive.

Logic synthesis has also been used as a preprocessing stage to reduce the complexity of formulas expressed in the CNF [108]. In doing so, it can help reducing the complexity of the SAT solver, such as the original version of IC3 [61]. However, the use of this idea was limited to the combinational part of the circuit as the SAT-formula (in these approaches) usually represents only a couple of unrolled timeframes.

Our technique could be depicted as a generalization of ODC-based techniques, in that it decomposes the functions using sequential ODC, which are not included in the previous combinational ODC. Thus, it offers more opportunities in clock-gating more logic. One uniqueness about our approach is that it does not depend on specific type of model checkers [34]. Rather, any model checker could be used with our framework. In addition, our algorithm is easily parallelizable, so the capabilities offered by recent low-cost, high performance computing platform could be leveraged on running our algorithm. Moreover, as we work in a pre-optimized circuit, our algorithm showed that it had the capabilities to save power even in cases where we only have small windows for optimization. The scalability of our algorithm allows us to finish quickly even for the largest benchmark circuits. Finally, there is a wide range of applications in ASIC [31] and FPGA [109] in which our algorithm is applicable. Our contributions can be distinguished from the previous work as follows:

- Unlike previous techniques, our method decomposes logic functions in an entirely new way, by using inductive invariants that relate signals in the circuit. This gives more opportunities especially in highly optimized circuits.
- Unlike algorithms proposed in [70], [71], we use SAT- based toolsets [61],[58], which are more scalable than BDD-based techniques, thereby providing new opportunities for power saving.
- We have used SAT-based formal methods to prove inductive invariants instead of depending on ”human experts” as proposed in [33].
- Techniques proposed in [107], [34] are STC-based, whereas our technique is an ODC-based. However, we have used efficient heuristics including efficient filtering of those

invalid invariants (with the aid of incremental saturated rarity simulation) as we select which inductive invariant candidates to prove.

- Unlike all of the approaches above (and the original IC3 [61]), we address the scalability problem of the model checking problem in a new way. In the original IC3, combinational synthesis techniques are used to reduce the size of the SAT formula. In addition, the learnt invariants are used from one proved property to another. However, model checkers would have to deal with the whole circuit. In our approach, instead of using this monolithic approach in which combinational synthesis is performed in all inductive invariant candidates, we propose a local (but efficient) approach in which we apply combinational and sequential synthesis on each inductive invariant candidate (not the whole set of candidates). We found that this approach highly reduces the number of state elements and literals, and synergistically reduces the run time for model checker.

6.2 basic Idea, Motivating Example and Heuristics

Dynamic power dissipation of a CMOS circuit is calculated as follows [70]:

$$P_{dynamic}(n) = \frac{1}{2} \times f \times V_{dd}^2 \times \sum_{i=1}^{n}(c_i s_i) \tag{6.1}$$

where n is the total number of gates, f is the clocking frequency, V_{dd} is the supply voltage, c_i is the load capacitance for gate i and s_i is the switching activity for gate i. Switching

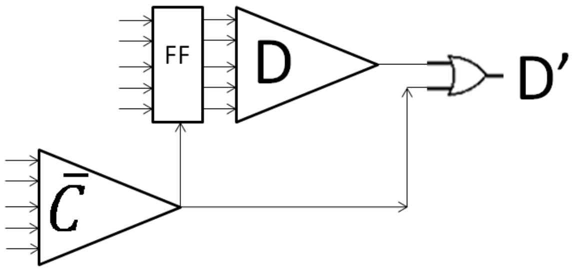

Figure 6.1: Our Proposed clock-gating.

activity is the average number of transitions ($0 \rightarrow 1$ and $1 \rightarrow 0$) a gate i switches per unit time. Reducing switching activities is an effective way to lower dynamic power.

Consider an invariant $(C + D)$ which by definition holds for all reachable states from initial state I. According to this invariant, if $C = 0$ then D has to be 1. Thus, whenever C is 0, signal D can be set to 1 directly and its fan-in cone could be safely blocked. Fig. 6.1 depicts this idea. The total power saving in this case would be:

$$P_{saved} = p(C = 0) \times P(s(B)) \tag{6.2}$$

where $p(C = 0)$ is the probability that C is set to 0 and $s(B)$ in the set of nodes in the transitive fan-in cone of D. As noticed from Equation 6.2, the amount of power saving is directly proportional to the probability of negation of signal C. One issue has to be taken care of is when part of the logic in the fan-in of D is used in other node that is not in D [70].

Although the problem statement is simple, achieving it is hard due to the following reasons. First, generating all inductive invariant candidates would be prohibitive as we would have very huge number of candidates $O(n^2)$ in large circuits containing n signals. Secondly, many of these candidates might be invalid, and this would overwhelm the model checker later [72]. Thirdly, even if we have a good approach to filter invalid invariant candidates, we still have the problem of prove a potentially large number of survived candidates, which usually is a computationally expensive step. In the next sections, we discuss our approach to tackle these problems.

6.3 Inductive Invariant Candidate Generation

A circuit of n gates would have $O(n^2)$ 2-literal potential inductive invariant candidates. Generating and proving this large number of candidates would be expensive. Thus, we propose two heuristics to reduce this cost. First, we consider a signal C only if it has a very high probability of occurrence. The motivation is that even if we have a valid invariant with small probability, we will likely not benefit from it for gating the corresponding block. The second heuristic is that we only choose D variables as the roots of dominant cuts. As we stated earlier, the blocked part is the only part of logic that is completely contained in the transitive fan-in of signal D. When candidate invariants are proven, if a dominant cut is totally contained in the transitive fan-in of another dominant cut, they will be merged. However, at this stage we do not have the information to determine whether this would happen or not. Thus, we choose to select D from the roots of the dominant cuts (as they are the maximally self-contained cuts). In doing so, dominant cuts allow us to abstract the circuit and to reduce the number of D variables needed to be considered. In case we have a valid invariant, we would be able to block the entire dominant cut, and any other invariants inside this dominant cut need not to be considered. On the other hand, if we have an invariant that involves a variable internal to this cut (but not at the root), more refinements could be used later to determine which invariant candidates are needed according to the proved invariants from the dominant cut.

With these two heuristics, the number of candidates needs to be proven is significantly reduced. In addition, nodes that seem to have constant values during simulation are eliminated from C and D variables sets. This would have two benefits: first, it reduces the number of inductive invariant candidates. Secondly, in case it is truly a constant node, it would produce inconsistency later. For example, if C is a constant 1, it will have signal probability equal to 1 (from random simulation). In case this variable is chosen, it would generate two inductive invariant candidates $(\overline{C} + D)$ and $(\overline{C} + \overline{D})$, and both would be true.

6.4 Inductive Invariant Candidates Computation

Rarity simulation is efficient in filtering invalid invariants. However, it cannot prove their validity. Thus, we have a tradeoff. If rarity simulation runs for very long time, it might discover all invalid invariants. However, keep it running after that is simply a waste of time and resources. On the other hand, when rarity simulation runs only for a small amount of time, it could miss many of invalid invariants which would cause two problems. First, it would end up with many invalid invariants that would impair from having an efficient filtering algorithm. In other words, the filtering algorithm will work on a possibly highly contaminated set which have lots of valid invariants. Accordingly, filtering will have a considerable chance to filter out some of valid invariants and keep many invalid invariants. Hence, even if the model checker is able to quickly invalidate them (in the ultimate optimistic case), we would have very low quality solution.

Secondly, validating all the inductive invariant candidates by a model checker would be computationally expensive. Thus, as a tradeoff, we propose an algorithm to reduce the time to disprove many (if not all) invalid candidates as depict in Fig. 6.2. In this algorithm; rarity simulation has "*normal*" and "*aggressive*" modes. Each mode has its own max timeout t_{normal} and $t_{aggressive}$ respectively, and $t_{aggressive} >> t_{normal}$. The normal mode is executed first and it is kept running as long as it discovers more invalid candidates (lines $2-5$). When it saturates, it switches to the aggressive mode (lines $6-7$). In the aggressive mode, the max. timeout is much higher than the normal mode. Whenever it finds an invalid invariant, it switches back to the normal mode again(line 8). It decides to quit if and only if it could not find an invalid invariant in two consequent *normal* and *aggressive* runs.

6.5 Proving Inductive invariant candidate

In order to prove one survived candidate from rarity simulation, a new circuit for this inductive invariant candidate (which is modeled as a property p) would be construct such that its output will be satisfied if and only if there is a counter example to p. This could be

```
1: repeat
2:     set $T_{timeout} = t_{normal}$
3:     repeat
4:         Rarity_Sim()
5:     until invalid invariant saturation
6:     set $T_{timeout} = t_{aggressive}$
7:     Rarity_Sim()
8: until invalid invariant saturation
```

Figure 6.2: Iterative Rarity Simulation Saturation Algorithm.

accomplished by building circuit with output $\bar{p}$. Newly constructed circuit is checked with a PDR-based model checker [58]. In case we have multiple invariant candidates we will present the conventional approach to implement that and our efficient way.

6.5.1 Conventional approach

the reasoning in this approach [61], in order to benefit from the experience acquired from proved inductive invariant and to minimize the effort required to prove all inductive invariant candidates, invariant candidates counter examples are built in the same circuit and they proven monolithically. In other word, learnt inductive invariants from a valid (and proved) inductive invariants is applied in proving of all subsequent inductive invariants. circuit of the whole counter examples may be combinationally (and not sequentially) optimized to reduce no. of literals.

6.5.2 Our approach

As the heaviest part of our algorithm is the model checker. So, in order to have an efficient algorithm, we need to reduce the run time of this major stage. We have proposed an efficient heuristic in order to achieve that. This heuristic is based on the observation that generated invaraints are structurally simple (not functionally) as it consists of 2 literals. This would leverage the capability of combinationnal and sequential synthesis to large reduces them and hence, help the model checker to prove them latter. In other words, for our with In order to reduce the complexity of the generated inductive invariant candidates, and reduce the burden on the model checker. We have proposed to use logic synthesis (combinational and sequential optimization) as a mandatory step before running the model checking. That is, we used logic optimization to reduce the complexity of our circuit combinational and sequentially, by building the inductive invariant candidate as an output and try to optimize the part of the original circuit contributes to this output only. Our experiments shows that running PDR on each candidate individually reduces the overall computation time compared to run it in a monolithic approach. This is because the shared logic between different properties limit the synthesis process to highly optimize the constructed property circuit and hence add a high burden on the model checker.

In PDR, instead of taking a monolithic approach, in which we PDR on all of inductive

```
1: Construct property monitor circuit
2: Sequential Sweeping [110]
3: loop
4:     AIG balancing [111]
5:     AIG rewriting [111]
6: end loop
7: PDR()
```

Figure 6.3: PDR Prove Algorithm.

invariant candidates survived from filtering algorithm. We take the opposite approach and prove each individual inductive invariant candidate. The "common wisdom" suggest to do all of them once and get benefit of learnt clauses. But, in other hand, model checker have to deal with the whole circuit and this add a big heavy task on it. So, we take it one by one, and get benefit of the simplicity form of our invariant and run very quick (but efficient) combinational and sequential optimization, which shows that it is helpful to reduce the size of the reachable state of the circuit (for this invariant). Fig. 6.3 depicts the algorithm for Prove invariant candidates.

6.6 Low-Power Synthesis Algorithm

The sequential version of our algorithm is depicted in Fig. 6.4. The algorithm start with generating the inductive invariants candidates based on the heuristics described in the pre-

```
1: Generate inductive invariant candidates
2: Rarity Simulation Filtering
3: Select invariant
4: PDR Prove
5: Write back
```

Figure 6.4: Sequential algorithm for Power-Aware Synthesis.

vious section as shown in line 1. After that Rarity simulation is run rapidly to eliminate as many as possible of the invalid invariants (line no. 2). The proved inductive invariants, will be grouped and selected according to a greedy approach describe in next section. The filtered inductive invariant candidates will be passed to PDR-based model checker to prove them.

6.7 Greedy Selection of Inductive Invariants

Figure 6.5 depicts the greedy algorithm for candidates selection. S is the set of selected invariants. It is initially empty (line 1). Survived invariants from rarity simualtion are grouped according to controlling variable A (line 2). Shared nodes in each group is computed and amount of power saving of blocking the group is calculated(line $3-4$). The Algorithm starts greedy to select a group with max. power saving (line $5-8$) and remove the shared node and conflict node from other remaining groups. The process continue until all groups are empty.

```
1: $S = \phi$
2: group survived candidates
3: compute node shared between each group
4: calculate power saving for each group
5: **repeat**
6:     $S = S \vee$ group with Max. Power Saving
7:     remove conflict and shared node from other groups
8: **until** all groups are empty
9: return S
```

Figure 6.5: Greedy selection of candidates algorithm.

6.8 Results

We have evaluated our algorithm on the largest circuits from public domain ISCAS89 and ITC99 benchmarks. The benchmarks are preprocessed by ABC [104] as follows: the designs were (a) attened , (b) structurally hashed , (c) sequentially optimized by register sweeping [110], (d) combinationally optimized with AIG rewriting and balancing [111]. The power is computed by random simulation tool available in ABC [112]. The proposed algorithm has been developed with C/C++ inside ABC and the performance was evaluated on 6 dedicated 2.66 GHz Intel Xeon cores, running a 64-bit Linux distribution. We have compiled our program with g++ under -O3 option.

The experiments are designed to answer the following questions:

Table 6.1: Circuit Statistics

ckt	IN	OT	FF	AIG	LV	PWR
s9234	36	39	211	1947	34	477.62
s13207	31	121	669	2721	34	445.88
s15850	14	87	597	3553	45	504.62
s38417	28	106	1636	9219	31	2855.09
s38584	12	278	1452	12394	36	5694.93
b17_opt	37	97	1414	27645	71	1706.49
b18_opt	37	23	3270	80668	140	2382.71
b19	24	30	6642	163520	138	3728.6

- How much can our algorithm reduce power?
- How other synthesis objectives are affected (i.e., area overhead)?
- How our approach is efficient compared to other previous work?

Table 6.1 reports the statistics of our benchmark circuits. In Table 1, input (IN), output (OT), no. of ip ops (FF), size of AIG graph (AIG) [113], no. of levels (LV) and original power dissipation (PWR).

Table 6.2 shows the quality of solution and performance of our proposed method compared to BDD-based ODC method proposed in [71], and state-of-art SAT-based STC method proposed in [34]. Acronyms used in the table is as follows, power reduction percentage (PR%), power reduction in optimized circuit (OPR%) area increase percentage due to added logic (AI%), no of flip flop added (FFA), number of generated inductive invariant candidates (IIC), and time (T). Our approach shows a power reduction with 11.96% in b18_opt and up to 37.1 % in s38417, with an average of 23.2% over all circuits. The area increase is 1.5 % in b19 and up to 19.8% in s9234, with 9.5% on average. The number of FF added due to gating was as low as 0 in s13207 and at most 6 FFs in s38417 with average 2 FFs. Run time is 31 seconds in s9234 and up to 324 in s38584 with average 154.75 seconds. Compared to BDD-based ODC method, our approach achieve higher power saving with 37.1% and 32.4% power reduction in s38417 and s38584 respectively compared to 24% and 9.8% for BDD-based ODC method. Also the area overhead is still small 10.5% and 9.1% compared to 4.9% and 6% in [71]. Our algorithm has better time performance 215s and 324s compared to 3986s and 3391s. We have to note that the average power savings for BDD-based method is 19.4% for small and medium size circuits (their experiments done on MNCN benchmarks) meanwhile we have an average power saving 23.2%. Moreover, our method scales for larger circuits. In addition, their average area increase is 9.3% while we get an average of 9.5%. For the SAT-based STC method, they run their experiments without pre-optimizing circuits first. We report the power percentage for our algorithm before pre-optimization in PR%. On average our approach shows a 30.7% compared to 25.23% power reduction in SAT-based STC method. For pre-optimized circuits, our approach provides higher power saving in all circuits except s38584.

Table 6.2: Comparison between our proposed method, BDD-based ODC method, and state-of-art SAT-based STC methods

ckt	Our approach						BDD ODC [71]			SAT STC [34]			
	OPR%	PR%	AI%	FFA	IIC	T	PR%	AI%	T	PR%	AI%	IIC	T
s9234	12.0	15.6	19.8	3	3728	31	-	-	-	7.01	1.31	34281	12.27
s13207	26.4	34.2	17.1	0	17218	42	-	-	-	12.1	4.08	173738	9.21
s15850	12.2	20.5	12.2	2	42302	279	-	-	-	3.35	0.97	57538	42.65
s38417	37.1	41.9	10.5	6	1954586	215	24	4.9	3986	4.1	3.77	452724	110.85
s38584	32.4	36.5	9.1	1	1044546	324	9.8	6	3391	64.56	4.39	148418	55.44
b17_opt	21.5	31.3	3.3	1	721962	40	-	-	-	-	-	-	-
b18_opt	11.96	32.4	2.7	2	935471	70	-	-	-	-	-	-	-
b19	31.9	33.1	1.5	1	1190160	237	-	-	-	-	-	-	-

6.9 Conclusion

We have proposed an efficient scalable SAT-based low-power synthesis. Fast discovery of useful inductive invariants is used to clock-gate critical signals that can significantly reduce power consumption. We have proposed two heuristics to reduce the number of invariants that need to be searched. In addition, optimizations to different stages of our algorithm, in rarity simulation and model-checking, are applied to improve the performance and quality of our results. The results showed that an average of 23.2% power reduction can be achieved with a small area overhead, all in short execution times.

Chapter 7

Future Work

In this chapter, we present different possibilities to improve the work proposed in the dissertation. First, we introduce different possible extensions to our package to handle very large BDDs. Secondly, we propose the extension of this package to handle POBDDs. In addition, we will propose to investigate another research direction, in which we will propose to filter Potential Inductive invariants. Furthermore, we suggest to investigate the use of variant techniques(i.e., BDD-based/SAT-based techniques) to our timing-aware synthesis algorithm. Finally, we propose to extend our SAT-based low-power approach to STC-based clock-gating techniques.

7.1 Extending Hopscotch Hashing Technique to Very Large Monolithic BDDs

Roomy package [114] is a c/c++ library allows to use the secondary storage as the main working memory of computation instead of RAM. In order to reduces the latency and improve the limited bandwidth of off-shelf hard disk, Roomy uses many disks in parallel. its data structures are transparently distributed across many disks. However, it provides a small options of data structures (arrays, unordered lists and hash tables)

Kunkle *etal*. [84] propose to use Roomy package for very large BDDs. Although they have succeeded to build BDDs, for very large problem, for the first time. However, their implementation have some major problems. First, they use shared quasi BDDs (SQBDD) [81], which is known to have large memory overhead (i.e., SQBDDs have order of 2 or 3 higher than ROBDDs). This overhead is not acceptable for very large BDDs. Secondly, They don't use shared BDDs. Finally, their BDD construction algorithms are not efficient in term of computation time.

There are different directions to improve our BDD package such that be able to accommodate very large BDDs using parallel-disk computation. First, the Hopscotch hashing could be integrate with Roomy package. Secondly, primitive atomic synchronizations could be used with Hopscotch hashing to improve its scalability.

7.2 Extending Hopscotch Hashing Technique to Partition-BDDs

partitioned-ROBDDs (POBDD) are another version of ROBDDs [115] . Similar to ROBDDs, They are canonical. However, they have two important advantages over monolithic ROBDDs. First, their size is usually compact than monolithic ROBDDs and even Free BDDs. Secondly, only one partition need to be manipulated which further increases space and time processing efficiency.

We propose to get the advantage of our package combined with Roomy and POBDD to construct very large circuits. We propose to extend the resultant package from the previous section to handle POBDD. In other words, we will propose techniques to handle different variable order for different branches in POBDD. For example, we would change UT and CT and hashing mechanism to handle this change in BDD structure. In addition, The resizing and GC would also be modified.

7.3 Selection of Potential Inductive Invariants

For sequential circuits, static implications are relations that hold in all states (reachable and unreachable states). However, Inductive Invariants are relations that hold in all reachable states, but they are not necessary hold in unreachable states. By adding inductive invariants to SAT formula (Eq. 2.8), we obtain a much tighter over-approximation of the reachable state space because any illegal states that violate one or both of these invariants is discarded from the search space. In other words, inductive invariants are used to prune the search space, and hence improve SAT solver performance.

Inductive invariants are often proven using induction [116] and/or properties strengthening technique [117]. In property strengthening, properties ϕ, δ and ζ are assumed together (given that δ and ζ also passed the base case) in the first time frame and then verified one by one in the second. If the SAT solver returns inconclusive for a property during validation process, its respective assumption is removed from the first frame. The process is repeated until all properties being assumed are in fact true invariants. Both induction and property strengthening are used in attempt to eliminate the spurious initial state (The initial state now has to satisfy all three properties together, making it closer to the reachable state space).

Checking the validity of all potential inductive invariants (PII) with induction is an exhaustive task. This due to the large number of PII. We will propose a technique to filter the PII before we check their validity with SAT solver.

7.4 Bi-Decomposition for Logic Synthesis

Recent work [63] proposed SAT-based solutions for bi-decomposition synthesis problem. The use of SAT not only makes the computation of bi-decomposition feasible for large circuits, but also serves for automatically selecting and optimizing variable partitions. SAT-based OR, AND and XOR bi-decompositions under known and unknown partition of variables were proposed in [63].

The relative inefficiency of the existing SAT-based models [63] prevent their use on very large industrial circuits. We propose to use static implication combined with PII to help SAT solver to successfully bi-decompose large circuits.

7.5 SAT-Based STC Clock-Gating for Logic Synthesis

Our framework could be used in generation and inductive invariant candidates computation in STC clock-gating. Lin *et al.* [34] proposed use of interpolation to improve clock-gating. New improvements have proposed to improve IC3 model checker with interpolation [118] . Both technique could be investigated with our low-power approach to enable efficient SAT-based STC clock-gating.

Bibliography

[1] M Kochte, S Kundu, K Miyase, S Wen, and H Wunderlich. Efficient bdd-based fault simulation in presence of unknown values. In *Test Symposium (ATS), 2011 20th Asian*, pages 383–388. IEEE, 2011.

[2] C Kao. Bdd-based synthesis for mixed cmos/ptl logic. *International Journal of Circuit Theory and Applications*, 39(9):923–932, 2011.

[3] D Tille, S Eggersgluss, R Krenz-Baath, J Schloeffel, and R Drechsler. Improving cnf representations in sat-based atpg for industrial circuits using bdds. In *Test Symposium (ETS), 2010 15th IEEE European*, pages 176–181. IEEE, 2010.

[4] A Mishra and A Chandra. Equ-iitg: A multi-format formal equivalence checker. In *Energy, Automation, and Signal (ICEAS), 2011 International Conference on*, pages 1–6. IEEE, 2011.

[5] J Hennessy and D Patterson. *Computer architecture: a quantitative approach.* Morgan Kaufmann, 2011.

[6] J Sanders and E Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[7] S Khatri and K Gulati. *Hardware Acceleration of EDA Algorithms: Custom ICs, FPGAs and GPUs*. Springer, 2010.

[8] S Minato. *Binary Decision Diagrams and Applications for VLSI CAD*. Springer, first edition, 1995.

[9] M Ishihat, T Sato, , and S Minato. *Compiling Bayesian Networks for Parameter Learning Based on Shared BDDs*, volume 7106 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2011.

[10] L Xing. An efficient binary-decision-diagram-based approach for network reliability and sensitivity analysis. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 38(1):105–115, 2008.

[11] G Hachtel and F Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, 2006.

[12] R Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[13] K Brace, R Rudell, and R Bryant. Efficient implementation of a bdd package. In *Proceedings of the 27th ACM/IEEE design automation conference*, pages 40–45. ACM, 1991.

[14] R Brayton J Sanghavi, R Ranjan and A Sangiovanni-Vincentelli. High performance bdd package by exploiting memory hierarchy. In *Proceedings of the 33rd annual Design Automation Conference*, pages 635–640. ACM, 1996.

[15] R Bryant B Yang, Y Chen and D O'hallaron. Space-and time-efficient bdd construction via working set control. In *Design Automation Conference 1998. Proceedings of the ASP-DAC'98. Asia and South Pacific*, pages 423–432. IEEE, 1998.

[16] N Shavit M Herlihy and M Tzafrir. Hopscotch hashing. *Distributed Computing*, pages 350–364, 2008.

[17] R Pagh and F Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[18] S Morishita H Yoshida and M Fujita. Demonstration of hardware-accelerated formal verification. In *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pages 380–383. IEEE, 2009.

[19] A Biere, A Cimatti, E Clarke, M Fujita, and Y Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 317–320. ACM, 1999.

[20] K Torii, K Nakamura, K Takagi, and N Takagi. Backward multiple time-frame expansion for accelerating sequential sat. In *Proc. SASIMI*, 2012.

[21] O Grumberg A Yadgar and A Schuster. Hybrid bdd and all-sat method for model checking. *Languages: From Formal to Natural*, pages 228–244, 2009.

[22] T Matsunaga, S Kimura, and Y Matsunaga. Power and delay aware synthesis of multi-operand adders targeting lut-based fpgas. *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*, pages 217–222, 2011.

[23] W Shiue. Power/area/delay aware fsm synthesis and optimization. *Microelectronics journal*, 36(2):147–162, 2005.

[24] V Sklyarov and I Skliarova. Synthesis of parallel hierarchical finite state machines. *Electrical Engineering (ICEE), 2013 21st Iranian Conference on*, pages 1–8, 2013.

[25] S Roy, M Choudhury, R Puri, and D Pan. Towards optimal performance-area trade-off in adders by synthesis of parallel prefix structures. *Proceedings of the 50th Annual Design Automation Conference*, page 48, 2013.

[26] R Uma and P Dhavachelvan. Performance enhancement through optimization in fpga synthesis: Constraint specific approach. In *Computer Networks & Communications (NetCom)*, pages 195–204. Springer, 2013.

[27] L Amarú, P Gaillardon, and G De Micheli. Bds-maj: a bdd-based logic synthesis tool exploiting majority logic decomposition. *Proceedings of the 50th Annual Design Automation Conference*, page 47, 2013.

[28] P Kaushik, S Gulhane, and A Khan. A survey on power management techniques. *International Journal of Scientific & Engineering Research*, 3(8):1–5, 2012.

[29] A Beloglazov, R Buyya, Y Lee, and A Zomaya. A taxonomy and survey of energy-efficient data centers and cloud computing systems. *Advances in Computers*, 82(2):47–111, 2011.

[30] P Stanley-Marbell, M Hsiao, and U Kremer. A hardware architecture for dynamic performance and energy adaptation. In *Power-Aware Computer Systems*, pages 33–52. Springer, 2003.

[31] V Venkatachalam and M Franz. Power reduction techniques for microprocessor systems. *ACM Computing Surveys (CSUR)*, 37(3):195–237, 2005.

[32] P Babighian, L Benini, and E Macii. A scalable odc-based algorithm for rtl insertion of gated clocks. *Design automation and test in Europe*, pages 500–505, 2004.

[33] R Wiener, G Kamhi, and M Vardi. Intelligate: scalable dynamic invariant learning for power reduction. *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, pages 52–61, 2009.

[34] T Lin and C Huang. Using sat-based craig interpolation to enlarge clock gating functions. *Proceedings of the 48th Design Automation Conference*, pages 621–626, 2011.

[35] C Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38(4):985–999, 1959.

[36] F Towhidi, A Lashkari, and R Hosseini. Binary decision diagram (bdd). *Future Computer and Communication, 2009. ICFCC 2009. International Conference on*, pages 496 –499, april 2009.

[37] N Ishiura S Minato and S Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*, pages 52–57. IEEE, 1990.

[38] R Arora and M Hsiao. Enhancing sat-based bounded model checking using sequential logic implications. In *VLSI Design, 2004. Proceedings. 17th International Conference on*, pages 784–787. IEEE, 2004.

[39] V Vimjam and M Hsiao. Increasing the deducibility in cnf instances for efficient sat-based bounded model checking. In *High-Level Design Validation and Test Workshop, 2005. Tenth IEEE International*, pages 184–191. IEEE, 2005.

[40] L Zhang, M Prasad, and M Hsiao. Incremental deductive & inductive reasoning for sat-based bounded model checking. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 502–509. IEEE Computer Society, 2004.

[41] E Rudnick J Zhao and J Patel. Static logic implication with application to redundancy identification. In *VLSI Test Symposium, 1997., 15th IEEE*, pages 288–293. IEEE, 1997.

[42] S Misra. Efficient graph techniques for partial scan pattern debug and bounded model checkers. Master's thesis, Virginia Polytechnic Institute and State University, 2012.

[43] R Arora M Syal and M Hsiao. Extended forward implications and dual recurrence relations to identify sequentially untestable faults. In *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, pages 453–460. IEEE, 2005.

[44] O Shtrichman. Tuning sat checkers for bounded model checking. In *Computer Aided Verification*, pages 480–494. Springer, 2000.

[45] O Shtrichman. Pruning techniques for the sat-based bounded model checking problem. In *Correct Hardware Design and Verification Methods*, pages 58–70. Springer, 2001.

[46] A Kuehlmann. Dynamic transition relation simplification for bounded property checking. In *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, pages 50–57. IEEE, 2004.

[47] Q Zhu, N Kitchen, A Kuehlmann, and A Sangiovanni-Vincentelli. Sat sweeping with local observability don't-cares. In *Proceedings of the 43rd annual Design Automation Conference*, pages 229–234. ACM, 2006.

[48] M Case, V Kravets, A Mishchenko, and R Brayton. Merging nodes under sequential observability. In *Proceedings of the 45th annual Design Automation Conference*, pages 540–545. ACM, 2008.

[49] A Bradley. Incremental, inductive model checking. *Temporal Representation and Reasoning (TIME), 2013 20th International Symposium on*, pages 5–6, 2013.

[50] Z Hassan, A Bradley, and F Somenzi. Incremental, inductive ctl model checking. *Computer Aided Verification*, pages 532–547, 2012.

[51] H Sipma, T Uribe, and Z Manna. Deductive model checking. *Computer Aided Verification*, pages 208–219, 1996.

[52] M Ganai, C Wang, and W Li. Efficient state space exploration: interleaving stateless and state-based model checking. *Computer-Aided Design (ICCAD), 2010 IEEE/ACM International Conference on*, pages 786–793, 2010.

[53] M Elbayoumi, M Hsiao, and M ElNainay. Selecting critical implications with set-covering formulation for sat-based bounded model checking. *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pages 390–395, 2013.

[54] M Elbayoumi, M Hsiao, and M ElNainay. Set-cover-based critical implications selection to improve sat-based bounded model checking. In *Proceedings of the 23rd ACM international conference on Great lakes symposium on VLSI*, pages 331–332. ACM, 2013.

[55] M Case, A Mishchenko, and R Brayton. Cut-based inductive invariant computation. *International Workshop on Logic and Synthesis*, 2008.

[56] H Jin and F Somenzi. An incremental algorithm to check satisfiability for bounded model checking. *Electronic Notes in Theoretical Computer Science*, 119(2):51–65, 2005.

[57] K McMillan. Applying sat methods in unbounded symbolic model checking. In *Computer Aided Verification*, pages 303–323. Springer, 2002.

[58] N Een, A Mishchenko, and R Brayton. Efficient implementation of property directed reachability. *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 125–134, 2011.

[59] H Savoj, A Mishchenko, and R Brayton. Sequential equivalence checking for clock-gated circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 33(2):305–317, 2014.

[60] Hardware model checking competition 2014. http://fmv.jku.at/hwmcc14cav/.

[61] A Bradley. Sat-based model checking without unrolling. *Verification, Model Checking, and Abstract Interpretation*, pages 70–87, 2011.

[62] F Dresig D Bochmann and B Steinbach. A new decomposition method for multilevel circuit design. In *Design Automation. EDAC., Proceedings of the European Conference on*, pages 374–377. IEEE, 1991.

[63] J Jiang R Lee and W Hung. Bi-decomposing large boolean functions via interpolation and satisfiability solving. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 636–641. IEEE, 2008.

[64] J Jiang H Lin and R Lee. To sat or not to sat: Ashenhurst decomposition in a large scale. In *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*, pages 32–37. IEEE, 2008.

[65] C Yang and M Ciesielski. Bds: A bdd-based logic optimization system. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 21(7):866–876, 2002.

[66] M Choudhury and K Mohanram. Bi-decomposition of large boolean functions using blocking edge graphs. *Proceedings of the International Conference on Computer-Aided Design*, pages 586–591, 2010.

[67] J Cortadella. Timing-driven logic bi-decomposition. *IEEE Transaction on Computer aided design and integrated circuits and Systems*, 2003.

[68] V Kravets and K Sakallah. Resynthesis of multi-level circuits under tight constraints using symbolic optimization. *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 687–693, 2002.

[69] R Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974.

[70] Y Hsu and S Wang. Retiming-based logic synthesis for low-power. *Proceedings of the 2002 international symposium on Low power electronics and design*, pages 275–278, 2002.

[71] Q Dinh, D Chen, and M Wong. Bdd-based circuit restructuring for reducing dynamic power. *Computer Design (ICCD), 2010 IEEE International Conference on*, pages 548–554, 2010.

[72] S Chatterjee, A Mishchenko, R Brayton, and A Kuehlmann. On resolution proofs for combinational equivalence. *Proceedings of the 44th annual Design Automation Conference*, pages 600–605, 2007.

[73] M Elbayoumi, M Hsiao, and M ElNainay. A novel concurrent cache-friendly binary decision diagram construction for multi-core platforms. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1427–1430. EDA Consortium, 2013.

[74] S Akers. Binary decision diagrams. *Computers, IEEE Transactions on*, C-27(6):509 –516, june 1978.

[75] S Tani, K Hamaguchi, and S Yajima. The complexity of the optimal variable ordering problems of shared binary decision diagrams. *Algorithms and Computation*, pages 389–398, 1993.

[76] K Yasuoka H. Ochi and S Yajima. Breadth-first manipulation of very large binary-decision diagrams. In *Computer-Aided Design, 1993. ICCAD-93. Digest of Technical Papers., 1993 IEEE/ACM International Conference on*, pages 48–55. IEEE, 1993.

[77] P Ashar and M Cheong. Efficient breadth-first manipulation of binary decision diagrams. In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pages 622–627. IEEE Computer Society Press, 1994.

[78] R Brayton R Ranjan, J Sanghavi and A Sangiovanni-Vincentelli. Binary decision diagrams on network of workstations. In *Computer Design: VLSI in Computers and Processors, 1996. ICCD'96. Proceedings., 1996 IEEE International Conference on*, pages 358–364. IEEE, 1996.

[79] T Stornetta and F Brewer. Implementation of an efficient parallel bdd package. In *Proceedings of the 33rd annual Design Automation Conference*, pages 641–644. ACM, 1996.

[80] K Milvang-Jensen and A Hu. Bddnow: A parallel bdd package. In *Formal Methods in Computer-Aided Design*, pages 501–507. Springer, 1998.

[81] B Yang and D O'hallaron. Parallel breadth-first bdd construction. In *ACM SIGPLAN Notices*, volume 32, pages 145–156. ACM, 1997.

[82] F Bianchi, F Corno, M Rebaudengo, M Reorda, and R Ansaloni. Boolean function manipulation on a parallel system using bdds. In *High-Performance Computing and Networking*, pages 916–928. Springer, 1997.

[83] S Minato. Streaming bdd manipulation. *Computers, IEEE Transactions on*, 51(5):474–485, 2002.

[84] V Slavici D Kunkle and G Cooperman. Parallel disk-based computation for large, monolithic binary decision diagrams. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, pages 63–72. ACM, 2010.

[85] F Somenzi. Cudd: Cu decision diagram package release 2.3.0. *University of Colorado at Boulder*, 1998.

[86] A Biere. *ABCD package.* 1997.

[87] D Knuth. Fundamental algorithms, volume iii of the art of computer programming, 1975.

[88] F Somenzi. Cudd 2.4.2 package. In *http://vlsi.colorado.edu/ fabio/CUDD/ (last visited April 16, 2012).*

[89] C Leiserson, R Rivest, C Stein, and T Cormen. *Introduction to algorithms.* The MIT press, 2001.

[90] G Blelloch, H Simhadri, and K Tangwongsan. Parallel and i/o efficient set covering algorithms. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*, pages 82–90. ACM, 2012.

[91] F Shahrokhi. *Survey of approximation algorithms for set cover problem.* PhD thesis, UNIVERSITY OF NORTH TEXAS, 2009.

[92] C Lund and M Yannakakis. On the hardness of approximating minimization problems. *Journal of the ACM (JACM)*, 41(5):960–981, 1994.

[93] VT HokieOne (SGI UV). http://www.arc.vt.edu/resources/hpc/hokieone.php.

[94] BMC Benchmarks. http://www.cs.ubc.ca/ hoos/SATLIB/Benchmarks/SAT/BMC/description.html

[95] SATLIB. http://www.cs.ubc.ca/ hoos/SATLIB/.

[96] Minisat 2.0. http://minisat.se/.

[97] M Elbayoumi, M Choudhury, V Kravets, A Sullivan, M Hsiao, and M Elnainay. Tacue: A timing-aware cuts enumeration algorithm for parallel synthesis. *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, pages 1–6, 2014.

[98] D Baneres, J Cortadella, and M Kishinevsky. Dominator-based partitioning for delay optimization. *Proceedings of the 16th ACM Great Lakes symposium on VLSI*, pages 67–72, 2006.

[99] S Chatterjee, A Mishchenko, and R Brayton. Factor cuts. *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 143–150, 2006.

[100] O Martinello Jr, F Marques, R Ribas, and A Reis. Kl-cuts: a new approach for logic synthesis targeting multiple output blocks. *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 777–782, 2010.

[101] R Glantz and H Meyerhenke. Finding all convex cuts of a plane graph in cubic time. *Springer, Algorithm and Computation*, 2013.

[102] E Sentovich, K Singh, L Lavagno, C Moon, R Murgai, A Saldanha, H Savoj, P Stephan, R Brayton, and A Sangiovanni-Vincentelli. Sis: A system for sequential circuit synthesis. *University of California, Berkeley*, 94720:4, 1992.

[103] L Stok, D Kung, D Brand, A Drumm, A Sullivan, L Reddy, N Hieter, D Geiger, H Chao, and P Osler. Booledozer: logic synthesis for asics. *IBM Journal of Research and Development*, 40(4):407–430, 1996.

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

[105] M Elbayoumi, M Hsiao, and M Elnainay. Novel sat-based invariant-directed low-power synthesis. In *16th International Symposium on Quality Electronic Design*. IEEE, 2015.

[106] L Benini and G De Micheli. Automatic synthesis of low-power gated-clock finite-state machines. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 15(6):630–643, 1996.

[107] A Hurst. Automatic synthesis of clock gating logic with controlled netlist perturbation. *Proceedings of the 45th annual Design Automation Conference*, pages 654–657, 2008.

[108] N Een, A Mishchenko, and N Sörensson. Applying logic synthesis for speeding up sat. *Theory and Applications of Satisfiability Testing–SAT 2007*, pages 272–286, 2007.

[109] K Tinmaung, D Howland, and R Tessier. Power-aware fpga logic synthesis using binary decision diagrams. *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pages 148–155, 2007.

[110] A Mishchenko, M Case, R Brayton, and S Jang. Scalable and scalably-verifiable sequential synthesis. *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*, pages 234–241, 2008.

[111] A Mishchenko, S Chatterjee, and R Brayton. Dag-aware aig rewriting a fresh look at combinational logic synthesis. *Proceedings of the 43rd annual Design Automation Conference*, pages 532–535, 2006.

[112] S Jang, K Chung, A Mishchenko, and R Brayton. A power optimization toolbox for logic synthesis and mapping. *ERL Technical Report, EECS Dept., UC Berkeley*, 2009.

[113] Aiger file format. http://fmv.jku.at/aiger/.

[114] D Kunkle. Roomy: a system for space limited computations, 2010.

[115] M Fujita A Narayan, J Jain and A Sangiovanni-Vincentelli. Partitioned robddsa compact, canonical and efficiently manipulable representation for boolean functions. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 547–554. IEEE Computer Society, 1997.

[116] F Lu and K Cheng. Sechecker: A sequential equivalence checking framework based on¡ formula formulatype=. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(6):733–746, 2009.

[117] V Vimjam and M Hsiao. Explicit safety property strengthening in sat-based induction. In *VLSI Design, 2007. Held jointly with 6th International Conference on Embedded Systems., 20th International Conference on*, pages 63–68. IEEE, 2007.

[118] G Cabodi, M Palena, and P Pasini. Interpolation with guided refinement: revisiting incrementality in sat-based unbounded model checking. *Formal Methods in Computer-Aided Design (FMCAD 2014)*, page 43, 2014.