DESIGN AUTOMATION FOR CIRCUIT RELIABILITY AND ENERGY EFFICIENCY

BY

CHEN-HSUAN LIN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

ABSTRACT

This dissertation presents approaches to improve circuit reliability and energy efficiency from different angles, such as verification, logic synthesis, and functional unit design. A variety of algorithmic methods and heuristics are used in our approaches such as SAT solving, data mining, logic restructuring, and applied mathematics. Furthermore, the scalability of our approaches was taken into account while we developed our solutions.

Experimental results show that our approaches offer the following advantages: **1)** SAT-BAG can generate concise assertions that can always achieve 100% input space coverage. **2)** C-Mine-DCT, compared to a recent publication, can achieve compatible performance with an additional 8% energy saving and 54x speedup for bigger benchmarks on average. **3)** C-Mine-APR can achieve up to 13% more energy saving than C-Mine-DCT while confronting designs with more common cases. **4)** CSL can achieve 6.5% NBTI delay reduction with merely 2.5% area overhead on average. **5)** Our modulo functional units, compared to a previous approach, can achieve a 12.5% reduction in area and a 47.1% reduction in delay for a 32-bit mod-3 reducer. For modulo-15 and above, all of our modulo functional units have better area and delay than their previous counterparts.

To my parents and Judy, for their love and support.
To my feathered friends, for our invaluable shared memory.

ACKNOWLEDGMENTS

I would like to express my special thanks to my adviser Prof. Deming Chen, who always encourages me to be bold and brave. I would like to thank you for advising me on my research and also giving me the freedom to explore different career paths. My dissertation would not be possible without your advice and wisdom, and my career would not be the same without your profound influence. I also would like to especially thank Prof. Rob A. Rutenbar, who runs a MOOC course, "VLSI CAD: Logic to Layout", and is also my doctoral committee member. I would like to thank you for inviting me to build and TA this first EDA MOOC course. It's my pleasure to be part of this awesome and impactful course. I am also grateful to my doctoral committee members, Prof. Martin D. F. Wong and Prof. Wen-Mei Hwu. Their constructive comments and suggestions have been proven to be extremely helpful for this dissertation.

Meanwhile, I would like to thank Lingyi Liu, Lu Wan, Subhendu Roy, and Keith Campbell for their collaboration in our various research projects that are included in this dissertation. Furthermore, I would like to thank all my friends I met at UIUC, who have kept me company over the years.

Finally, I give my deepest gratitude to my parents and my brother who have been always supportive throughout my whole life. I also want to thank my girlfriend, Judy, for the great support in my PhD life. I cannot express my love and gratitude for them in words.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

As mobile and internet of things (IoT) devices become globally ubiquitous, consumer demands for performance, reliability, and energy efficiency are increasing drastically. With technology downscaling to nanometer range, circuit reliability and energy efficiency have become critical concerns for robust system designs [1, 2]. Unfortunately, reliability and energy efficiency are necessary design trade-offs. Therefore, this dissertation develops new reliability- and energy-centric design techniques and frameworks that can help hardware designers in reducing circuit design effort.

In this dissertation, we conduct several researches to tackle reliability and energy issues from different angles, such as verification and logic synthesis, and in different ways like assertions, better-than-worst-case (BTW) design methodologies, logic restructuring, and low-cost modulo functional units for self-checking arithmetic components. In addition to reliability improvement and energy efficiency, we also take scalability into account so that our methodologies can be applied to industry-strength designs.

Assertions are valuable and commonly applied to formal verification and simulation-based verification in IC design flow [3]. Unfortunately, assertion generation is a time-consuming process that depends heavily on human efforts [4]. Some dynamic methods based on simulation and static methods based on structure analysis are proposed to automate assertion generation process. However, dynamic methods [5–10] cannot guarantee the quality of assertions due to incomplete simulation, while static methods [11, 12] might have scalability limits. With the significant advances in Boolean satisfiability (SAT) solving, SAT solving becomes a promising technique to overcome these methods' weaknesses [13–15]. Therefore, we proposed a **S**AT-**B**ased Automatic **A**ssertion **G**eneration method (**SAT-BAG**), which can avoid the drawbacks of dynamic methods while circumventing scalability issues faced by static methods. We successfully formulate assertion generation to a SAT problem and use unit assumption to generate concise assertions. Furthermore, we consider input constraints and word-level features to generate meaningful and high-readability assertions.

The better-than-worst-case (BTW) design methodology [16, 17] is well-known for its potential to improve circuit energy efficiency, performance, and reliability. However, most existing methods [18] do not provide sufficiently scalable solutions. Thus, we proposed a **C**ommon case **Mining** method (**C-Mine**), which combines two scalable techniques, data mining [19] and SAT solving [20], to provide scale-up solutions for improving energy efficiency by optimizing common cases. Data mining can efficiently extract patterns from an enormous data set, and SAT solving is famous for its scalable verification. This work shows that the combination of these two techniques can result in the best performance in terms of energy saving and scalability.

Negative bias temperature instability (NBTI) has become a major reliability concern in nanoscale designs. Although several previous studies have been proposed to address the NBTI effect during logic synthesis, their performance is limited because of focusing on a certain logic synthesis stage. Additionally, their complicated algorithms are not scalable to large designs. To tackle this, we proposed a **C**oordinated and **S**calable **L**ogic synthesis approach (**CSL**), which integrates techniques at different logic synthesis stages, ranging from subject graph to technology mapping and mapped netlist, to achieve an effective NBTI reduction. To our best knowledge, this is the first work that considers and mitigates NBTI impact in subject graphs, the earlier stage of logic synthesis.

Modulo (residue) arithmetic is useful for creating a shadow datapath to check the computation of an arithmetic datapath and involves three key steps: reduction of the inputs to modulo shadow inputs, computation with those shadow values, and checking the outputs for consistency with the shadow outputs. The focus of this work is to develop new gate-level architectures and algorithms to reduce the cost of modulo shadow datapaths. We proposed low-cost architectures for all four key functional units in a shadow datapath: (1) a modulo reduction algorithm that generates architectures consisting entirely of full-adder standard cells; (2) minimum-area modulo adder and subtractor architectures; (3) an array-based modulo multiplier design; and (4) a modulo equality comparator that handles the residue encoding produced by the above. To demonstrate the applicability of our approach for reliability improvement, we also used these building blocks to create self-checking multiply-accumulate and linear algebra primitive datapaths.

The contributions and results of this dissertation are summarized as follows:

- SAT-BAG [21] is proposed to tackle reliability issues by automatically generating concise assertions with complete coverage. Experimental results on industry-strength designs, SpaceWire, Ethernet, and Floating Point, show that the generated assertions can always achieve 100% input space coverage. In addition, SAT-BAG can remove vacuous assertions by considering input constraints, and generate highly readable word-level assertions using the discovered word-level features.

- C-Mine-DCT [22] is proposed to improve the energy efficiency of BTW design by optimizing common cases. The experimental results show that, compared to a recent publication, C-Mine can achieve compatible performance with an additional 8% energy savings, and 54x speedup for bigger benchmarks on average.

- C-Mine-APR [23] is proposed to compensate for the weakness of C-Mine-DCT, the typical limitation of tree-based algorithms, and to approach the problem from a different angle. The experimental results show that C-Mine-APR can achieve up to 13% more energy saving than C-Mine-DCT while confronting designs with more common cases.

- CSL [24] is proposed to address the NBTI effect, the major reliability issue, from the logic synthesis perspective. Experimental results on industry-strength benchmarks show that CSL can achieve 6.5% NBTI delay reduction with merely 2.5% area overhead on average, while a previous work barely gets NBTI delay reduction when the circuits are optimized beforehand, the circuit sizes are large, and standard cell libraries are richer.

- We proposed new gate-level architectures [25][1] for Mersenne modulo functional units targeting shadow datapaths for reliability improvement. The experimental results show that compared to a previous state-of-the-art approach, a 12.5% reduction in area and a 47.1% reduction in delay for a 32-bit mod-3 reducer can be achieved; furthermore, our

[1]This work is a collaboration with Keith Campbell.

reducer costs, which are the major costs in shadow datapaths, do not increase with larger modulo bases, and for modulo-15 and above, all of our modulo functional units have better area and delay than their previous counterparts. Using these building blocks to create self-checking multiply-accumulate and linear algebra primitive datapaths, the cost-effective results can be achieved such as 32-bit datapath overheads of 6–10% for a 3–61× reliability improvement and overheads of 15–20% for a 121–2477× reliability improvement.

The rest of the dissertation is organized as follows: Chapter 2 presents SAT-BAG, our SAT-based assertion generation work. Chapter 3 presents our proposed C-Mine methodologies, which aim at common case optimization for low-power BTW designs from different angles. Chapter 4 presents CSL, our coordinated and scalable logic synthesis for reducing NBTI effect. Chapter 5 presents our low-cost Mersenne modulo functional units targeting shadow datapaths for reliability improvement. Chapter 6 draws conclusions and outlines future work.

CHAPTER 2

SAT-BAG: GENERATING CONCISE ASSERTIONS WITH COMPLETE COVERAGE

2.1 Introduction

Verification is the bottleneck in IC design flow and this obstacle becomes more serious as the complexity of design grows. Assertion-based verification [3] is widely used in industry to facilitate checking of desired properties. Assertions are applied to formal verification for property checking and to simulation-based verification for monitoring design behavior and locating bugs [3].

In the verification process, assertion generation is a tedious and time-consuming manual procedure. In order to generate high-quality assertions, this manual procedure would take several refining iterations and lots of manpower [4]. Therefore, methods to automatically generate assertions [5–10, 26, 27] are proposed to relieve this hardship. Assertions, automatically generated from a register transfer level (RTL) design, can be used in future regression tests to guarantee the quality of design, and also help designers detect unexpected behaviors. Assertions are typically generated for a *target* RTL signal, and are of the form $A \rightarrow B$, which includes all propositional and temporal formulas in a standard temporal logic like LTL [28].

Dynamic methods generate assertions based on dynamic simulation or execution paths of RTL designs [5–10]. Some of them also check the correctness of candidate assertions using formal techniques [5, 10]. These techniques examine simulation data and infer behavior, often using machine learning [7, 10] and pattern matching algorithms [8] for the same. The quality of assertions thus inferred heavily depends on the information provided by simulation data. The generated assertions are always biased towards the simulation data. This is contradictory to the intention of assertion generation process. If generated assertions have exactly the same behaviors as the simulation data from designers, no new bugs or design behaviors can be exposed or covered by the generated assertions.

Moreover, directed or even random simulation patterns cannot cover all behaviors of the design. Insufficient simulation may result in corner case or rare case scenarios not being covered. The generated assertions fail to cover the corner cases of the design. However, these *corner case assertions* are more valuable to the verification process than typical case scenarios. In addition, insufficient simulation will lead to poor guesses by the machine learning algorithms, thereby leading to overfitting, or many redundant propositions in the assertion [7, 10]. Candidate assertions generated from simulation data need extra time and effort to formally verify their correctness.

Static analysis of source code and models have been used in software for generating assertions [11, 12] and in deductive verification for generating invariants [29]. In hardware, the techniques in [26, 27] also generate assertions based on structural analysis of designs. Static analysis can generate accurate assertions without spending formal verification time and effort in checking them. It can also cover 100% of the design input space without reliance on "typical case scenarios".

The downside in static analysis is the limited scalability. In the context of assertion generation, however, scalability is not a critical issue, since assertions spanning more than $2-3$ modules are generally unreadable for the users. For the scaling required by the assertion generation problem, sophisticated, scalable Boolean SAT based algorithms can be applied very effectively [13–15].

In this work, we propose a **SAT**-**B**ased Automatic **A**ssertion **G**eneration method (**SAT-BAG**), which can avoid the drawbacks of dynamic methods while circumventing scalability issues faced by static methods. SAT-BAG converts the assertion generation problem into an ALL-SAT problem [20]. The ALL-SAT problem is to generate all satisfying assignments for a SAT problem. The design information is completely stored in a SAT instance instead of simulation data. With this SAT instance, SAT-BAG can generate concise satisfying assignments for target signals. These satisfying assignments, together with the target signal value, can be regarded as assertions.

When directly applying SAT solvers to generate satisfying assignments for target signal, SAT solvers give concrete value to every variable in the design. As a result, the generated assertions that involve one proposition for each variable are very verbose. However, not all assignments are necessary for determining the target variable's value, and there are a lot of *redundant* propositions in the generated assertion. In this work, we apply a SAT technique, *unit assumption* [13], to remove redundant propositions in assertions. Unit assumption is a SAT technique that allows users to make assumptions for variables during SAT solving and reports which assumptions contribute to the result. In our context, the propositions of an assertion can be viewed as assumptions for a target signal. We can examine which propositions contribute to the target, and include only those in our generated assertions.

Our SAT-BAG method stores the entire design as SAT instance and also considers design constraints on the input signals. Generating assertions without regard to the input constraints may lead to vacuous assertions, in which the antecedent conditions are never reached or satisfied in the design. Therefore, to avoid potentially generating assertions on unreachable states, SAT-BAG converts input constraints into SAT clauses that can filter out vacuous assertions generated for unreachable states, and also assist in further redundancy removal from the assertions. For example, a bus controller may specify the legal work mode of the bus, and our SAT-BAG method can then incorporate these constraints for shaping input signals in work mode.

To further optimize our proposed SAT-BAG method, we use *word-level features* to improve the readability and input space coverage of generated assertions. Although SAT-BAG can generate concise and high coverage assertions, the generated assertions are at bit-level and thus have low readability, and the number of bit-level assertions might be large. Therefore, our SAT-BAG method uses word-level features to elevate the assertions to word level. Take design $c = a[k:0] \geq b[k:0]$ as an example, SAT-BAG will generate one word-level assertion instead of many bit-level assertions for valid bit vector combinations of a and b. Word-level features will be discovered from RTL design and added into SAT instances to let SAT-BAG generate word-level assertions.

Our contributions in this work are as follows. We propose a SAT-BAG method with unit assumption technique to generate concise assertions for given target signals. Our SAT-BAG method is not biased toward any simulation data and is able to generate assertions to fully cover both typical case scenarios and corner case scenarios. Our method can also filter out vacuous assertions by taking into account input design constraints. We also discover word-level features from RTL design and apply these features to generate word-level assertions, which have higher readability and expressiveness.

2.2 Preliminaries

2.2.1 Assertion

Our assertions are of the form *antecedent* $\rightarrow$ *consequent*, where *antecedent* and *consequent* are propositional or temporal logic formulas. We use notation of SystemVerilog assertion [30] for expressing assertions in this work. An example assertion is $(in_0 == 0$ && $in_1 == 1)$ ##1 $(in_1 == 1) \rightarrow\ (t == 1)$. It reads as follows: If both $(in_0 = 0)$ and $(in_1 = 1)$ in the current cycle, and $(in_1 = 1)$ in the next cycle, then the target signal t should evaluate to true.

In this work, propositions can contain one of two features. A *bit-level feature* is a bit signal evaluating to true or false. A *word-level feature* is a first order formula in terms of word-level variables, vectors of bit signals, that evaluates to true or false. In this work, propositions in antecedents of assertions only consider *boundary input signals*, which are composed of primary inputs (PIs), and registers, also called pseudo primary inputs (PPIs). In our assertions, the consequent contains only one proposition – a target signal that evaluates to true/false (equals 1/0). *A word-level assertion* is an assertion having at least one word-level feature. An example of a bit-level assertion is: $(in_0 == 1$ && $in_1 == 0) \rightarrow\ (t == 0)$, while an example of a word-level assertion is: $(in_0[k:0] \geq in_1[k:0]) \rightarrow\ (t == 1)$.

Previous techniques generate assertions within a sliding time window [7,10]. We use a similar concept here. A parameter *window size* (w) is used to restrict the duration of time cycles for temporal assertions we want to generate. For the example at the beginning of this section, w is set to 2, which means temporal assertions involving boundary input signals across two cycles will be generated.

A *vacuous assertion* is an assertion in which the antecedent cannot be satisfied. Consider the truth table of a target variable's function in terms of features. A table entry is *covered by a given assertion* if the concrete value of the entry can satisfy the antecedent of the given assertion. The *input space* *coverage* of an assertion denotes the percentage of truth table entries covered by the assertion.

2.2.2 Boolean satisfiability problem

A Boolean satisfiability (SAT) problem is a decision problem [20]. Given a finite set of Boolean variables $V = \{v_0, \ldots, v_n\}$, a *literal* l is either a Boolean variable v_i or its negated form $\neg v_i$ or $\sim v_i$. A *clause* is a disjunction of literals. A *SAT instance* is a conjunction of clauses, also named as *conjunction normal form (CNF)*. An *assignment* over V assigns each Boolean variable v_i either true or false value. A SAT instance is **satisfiable** if there exists an assignment, named *satisfying assignment*, such that the SAT instance evaluates to true; otherwise it is **unsatisfiable**.

An *ALL-SAT* problem is a variation of the SAT problem. While the traditional SAT problem only concerns one satisfying assignment for a satisfiable SAT instance, an ALL-SAT solution enumerates all satisfying assignments of a SAT instance. All satisfying assignments can be achieved by solving multiple SAT problems repeatedly. Each time a new satisfying assignment ψ is derived, the blocking clause of ψ will be added into the SAT instance to avoid deriving the same ψ again. The blocking clause of ψ is a disjunction of negated values of ψ. This collecting process will continue until no more satisfying assignments can be derived.

2.2.3 Conversion of a circuit into SAT instance

An RTL design will be compiled to its corresponding circuit netlist. Given the circuit netlist, it can be converted to a SAT instance such that the functionality of design is completely preserved [31, 32]. The conversion, named Tseitin transformation [31], can be done in linear time. For instance, a two input AND gate, $c = AND(a, b)$, can be converted into the following three clauses:

$$(v_a \vee \neg v_c) \wedge (v_b \vee \neg v_c) \wedge (\neg v_a \vee \neg v_b \vee v_c)$$

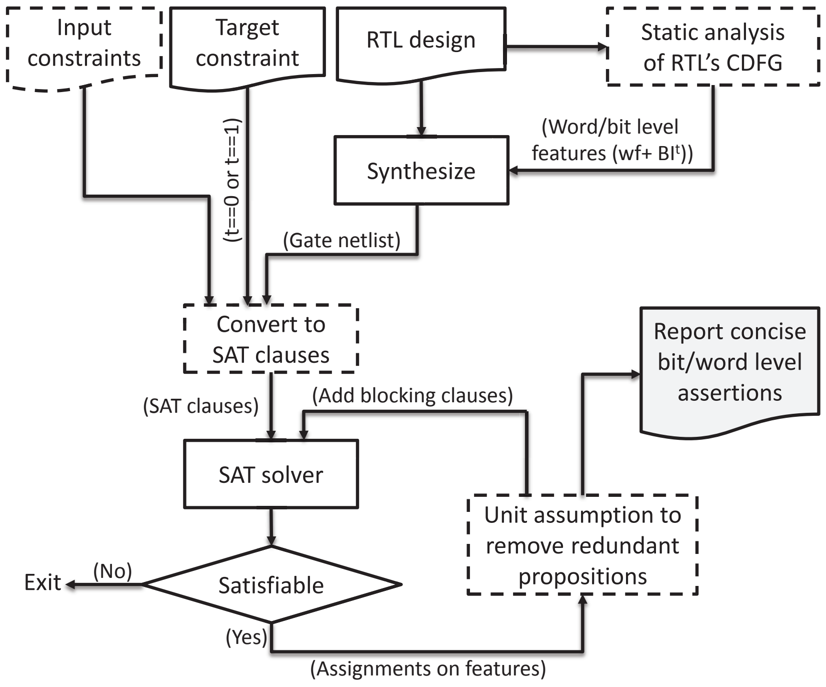

Input constraints
Target constraint
RTL design
Static analysis of RTL's CDFG
(t==0 or t==1)
(Word/bit level features (wf+ Bl^t))
Synthesize
(Gate netlist)
Convert to SAT clauses
Report concise bit/word level assertions
(SAT clauses)
(Add blocking clauses)
SAT solver
Unit assumption to remove redundant propositions
Exit
(No)
Satisfiable
(Yes)
(Assignments on features)

Figure 2.1: Flow chart of SAT-BAG.

2.3 SAT-based automatic assertion generation method

This section will use a running example in Fig. 2.3 to introduce how to construct a SAT instance for assertion generation, and how to efficiently generate assertions with unit assumption technique and enhance the quality of assertions by applying input constraints. In addition, discovering word-level features and generating word-level assertions are also discussed in this section.

The overall flowchart of SAT-BAG is shown in Fig. 2.1. Our contributions are shown in dotted boxes. First of all, an RTL design is converted into a gate netlist with bit- and word-level features, which are discovered from static analysis of RTL's CDFG. Next, the gate netlist is converted to a SAT instance based on window size. The constraints for a target signal are set as well. If needed, the input constraints of the design will be converted to SAT clauses and added to the SAT instance. With this SAT instance, we begin to repeatedly generate satisfying assignments. Each assignment can determine features' values, and redundant features can be removed by unit assumption. After that, concise bit- and word-level assertions will be collected and reported. The algorithm terminates after no more satisfying assignments can be generated.

2.3.1 Formulation of an All-SAT problem for assertion generation

Given a sequential circuit synthesized from an RTL design, the transition functions (T) can be extracted as Fig. 2.2(a). PIs and POs are primary inputs and outputs, while PPIs and PPOs are register signals, pseudo primary inputs and outputs. Next, we unroll the circuit based on the parameter w, which determines how many cycles we generate temporal assertions for. As shown in Fig. 2.2(b), if window size $w = 3$, three T's will be duplicated. The internal register signals in these functions are connected. Signals in each T are annotated with superscripts, -2, -1, and 0, according to the corresponding time cycle. The superscript 0 represents the current cycle while -1 and -2 represent the previous two cycles.

A set of boundary input signals of the unrolled circuit based on window size w is $BI = PIs^0 \cup \ldots \cup PIs^{-(w-1)} \cup PPIs^{-(w-1)}$, where PIs^i and $PPIs^i$ denote the sets of PI and PPI signals at cycle i, respectively. For a target signal t^0, BI^{t^0} is a subset of BI only collecting boundary input signals that are in the logic cone of t^0, as the shaded region in Fig. 2.2(b). Therefore, the feature set F for generating assertions for the target signal t^0 will be set to BI^{t^0} because t^0's value is obviously determined by the boundary input signals in its logic cone.

Definition 1. *Unit assumption [13] is a technique that allows users to assume specific conditions for each SAT solving through function interface $S.solve(assumps)$, denoted as $S^{assumps}$, where S is a SAT instance and **assumps** is the conjunction of literals like $a_0 \wedge a_1 \wedge \ldots \wedge a_n$. During each SAT solving, literals a_i are **temporarily** assigned to true. Therefore, if there exists an assignment that satisfies the SAT instance as well as assumptions, that satisfying assignment is returned. On the other hand, if the SAT problem is unsatisfiable under the given assumptions, the **subset** of those assumptions that collects assumptions contributing to the unsatisfiable result is returned. After each SAT solving, the unit assumptions are discharged.*

The unrolled circuit is converted to a SAT instance S as mentioned in Section II.C. To generate assertions for $t^0 = 1$, an assumption for the target signal $t^0 = 1$ would be added during solving S by function interface $S.solve(v_{t^0})$, where v_{t^0} is a Boolean variable representing signal t^0 in S. This assumed S is denoted as S^{t^0}. With S^{t^0} and F, ALL-SAT technique is applied to collect all satisfying assignments by repeatedly solving and blocking. Each satisfying assignment ψ for S^{t^0} can determine a unique vector of values for features in F which leads to $t^0 = 1$. Therefore, this vector of values and $t^0 = 1$ can be considered the antecedent and consequent of an assertion, respectively. With iterative collection of satisfying assignments, all assertions for $t^0 = 1$ can be generated.

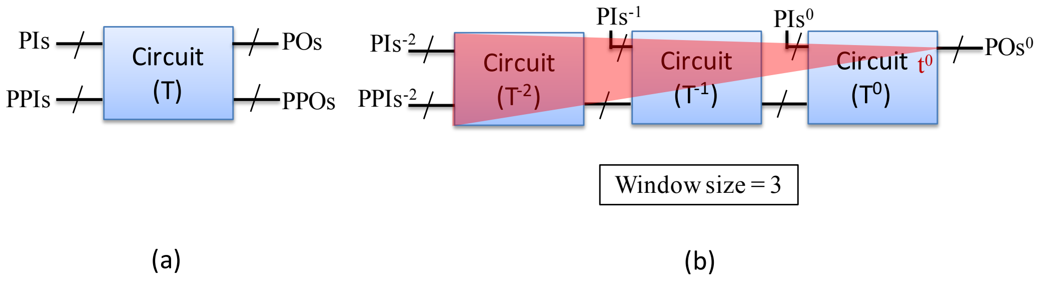


Figure 2.2: Unroll circuit based on parameter w and determine relevant boundary inputs by logic cone information.

Using the running example in Fig. 2.3 to demonstrate, we generate assertions for $o_2 = 0$ with window size $w = 1$. Here we omit cycle annotations of signals for better readability, since this is a single cycle assignment. The candidate feature set is $F = BI^{o_2} = \{a, b, ppi_1\}$. The design is converted to a SAT instance S and the corresponding Boolean variables in S representing signals a, b, ppi_1 and o_2 are v_a, v_b, v_{ppi_1} and v_{o_2}, respectively. We then collect all satisfying assignments for $S^{\neg o_2}$. Each assignment can decide a vector of values for $[v_a,\ v_b,\ v_{ppi_1}]$. In this case, we will collect 7 vectors for $[v_a,\ v_b,\ v_{ppi_1}] = \{000, 001, \ldots, 110\}$, and each vector can be viewed as an assertion. For example, the vector $[0, 1, 0]$ stands for a valid assertion $(a == 0 \ \&\& \ b == 1 \ \&\& \ ppi_1 == 0) \rightarrow o_2 == 0$.

Obviously, these 7 assertions, converted from above 7 vectors, are valid assertions but not concise assertions due to including redundant propositions. Therefore, the method proposed to generate concise assertions using unit assumption is introduced in the next section.

2.3.2 Generation of concise assertions using unit assumption

When SAT-BAG computes a vector for features in F, it will apply unit assumption to check whether all these features are necessary to determine

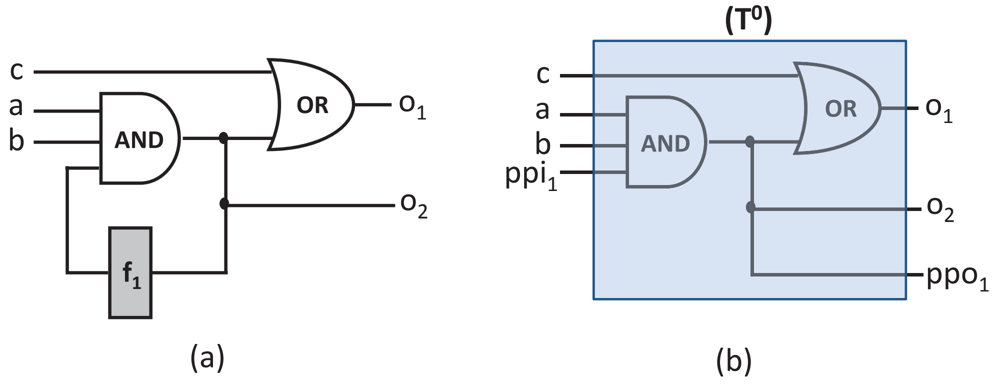


Figure 2.3: A running example.

the value of target signal and remove redundant features from the vector. In addition, unit assumption also can accelerate the process of collecting vectors by blocking several satisfying assignments at the same time.

For the same example, if SAT-BAG computes a vector $[0, 1, 0]$ for $[v_a,\ v_b,\ v_{ppi_1}]$, unit assumption is applied to check whether all features $a = 0$, $b = 1$, and $ppi_1 = 0$ are necessary for determining $o_2 = 0$. First of all, we examine the necessity of $ppi_1 = 0$. The original assumption for S is changed from $o_2 = 0$ to $o_2 = 1$ and additional assumptions $a = 0$, $b = 1$ are added to S as well. Now we solve S under new assumptions, which is $S.solve(v_{o_2} \wedge \neg v_a \wedge v_b)$, and marked as $S^{o_2 \wedge \neg a \wedge b}$.

If the result of $S^{o_2 \wedge \neg a \wedge b}$ is **satisfiable**, that means signal o_2 can be 1 without the feature $ppi_1 = 0$. Hence, the feature $ppi_1 = 0$ is critical to determine $o_2 = 0$ and cannot be removed from the vector. And the process will continue examining the remaining features. In this case, the result of $S^{o_2 \wedge \neg a \wedge b}$ is **unsatisfiable**, which means these two features $a = 0$ and $b = 1$ are enough to determine signal $o_2 = 0$ and the feature $ppi_1 = 0$ is redundant. Therefore, the vector can be optimized from $[0, 1, 0]$ to $[0, 1, X]$, where X is a don't-care term.

Furthermore, many redundant features can be caught simultaneously, because unit assumption will also report the **subset** of assumptions that really contributes to the unsatisfiable result. In this case, $\{v_{o_2}, \neg v_a\}$, a subset of the original assumptions $\{v_{o_2}, \neg v_a, v_b\}$, is reported, and we can know that only feature $a = 0$ contributes to this unsatisfiable result and another redundant feature $b = 1$ is found. Therefore, the vector can be further optimized from $[0, 1, X]$ to $[0, X, X]$. Now we generate a concise assertion $(a == 0) \rightarrow o_2 == 0$

instead of $(a == 0 \ \&\& \ b == 1 \ \&\& \ ppi_1 == 0) \rightarrow o_2 == 0$. In addition, by blocking the optimized vector $[0, X, X]$, we can avoid generating vectors that are already covered by $[0, X, X]$. Therefore, the process of collecting vectors can be improved as well.

In other words, the original vectors can be regarded as minterms and the optimized vectors as cubes. Each time we will try to collect cubes instead of minterms. This not only can help SAT-BAG generate concise assertions but also can enhance the collecting process by blocking cubes instead of minterms.

2.3.3 Elimination of vacuous assertions and refinement using input constraints

There might exist constraints on boundary inputs of modules, called *input constraints*. These constraints should be taken into account while generating assertions. On one hand, the inputs of modules or PIs can be driven by other modules. Obviously these inputs are not real free variables, *i.e.* some combinations of the inputs are invalid. There is research [14,33] on generating this kind of constraint on the connections between modules. On the other hand, register signals of modules, PPIs, are also not free variables, due to the presence of some unreachable states. Unreachable states are states for which there exist no execution traces from the initial states of a given finite state machine; thus, unreachable states cannot be considered valid state values on register signals. Reachability analysis [15,34] explores reachable states and this information can be regarded as a constraint on register signals.

For the same example in Fig. 2.3, if the initial state of flip-flop f_1 is set to 0, the input ppi_1 will always be 0 during all clocks. Therefore, the input constraint $ppi_1 = 0$ should be considered while generating assertions. With this constraint, the assertion $(a == 1 \ \&\& \ b == 1 \ \&\& \ ppi_1 == 1) \rightarrow o_2 == 1$, though valid, will become a vacuous assertion because ppi_1 can never be 1. In addition, the assertion $(c == 0 \ \&\& \ ppi_1 == 0) \rightarrow o_1 == 0$ can be further refined to $(c == 0) \rightarrow o_1 == 0$.

However, previous work does not take into account input constraints. Their assertions might be vacuous and might have redundant propositions. Therefore, in this work, SAT-BAG will take input constraints into consideration by converting these constraints into SAT clauses [35]. For the same example,

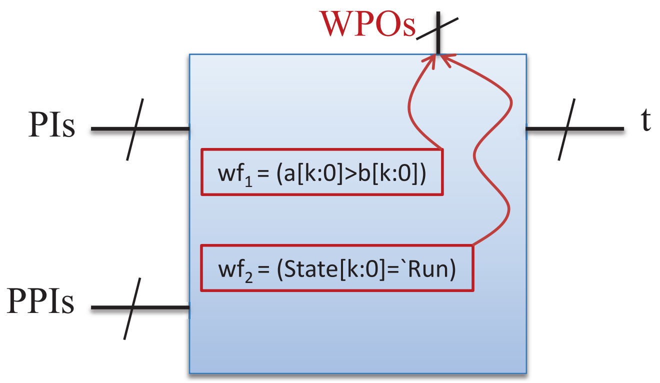


Figure 2.4: Use boundary signals wf_i to display the evaluated values of word-level features.

the constraint $ppi_1 = 0$ can be converted to a SAT clause $(\neg v_{ppi_1})$. With these clauses, SAT-BAG can generate more meaningful assertions and more accurate assertions.

2.3.4 Word-level feature discovery and application

The evaluated values of word-level variables are either true or false; therefore, these values can be displayed by bit signals wf_i as shown in Fig. 2.4. The signals wf_i are added to cycle copies (Ts) like primary outputs; thus, they can be considered features by SAT-BAG for generating assertions as well. The candidate feature set will be updated to $F = BI^{\triangle} \cup WF$, where $WF = \bigcup WPOs^i$. Furthermore, as to SAT-BAG, word-level features in WF will have higher priority over bit-level features in $BI^{\triangle}$ to be chosen as propositions in assertions.

To discover word-level features, an RTL design will be compiled into the control data flow graph CDFG [36]. There are three typical nodes in the CDFG: branch, assignment and merge nodes. A branch node represents a branch statement in RTL; an assignment node represents an assignment statement in RTL; a merge node represents an end of branch. Word-level features can be found in branch nodes (ex: $if\ (a \geq b)$) or assignment nodes (ex: $assign\ c = (a! = b)$). We will unroll the design CDFG based on the window size and trace target signal's fanin cone backward to the boundary inputs, PIs and PPIs. Word-level features in terms of boundary input signals can be collected along the paths of fanin cone.

2.4 Experimental results

We implemented our proposed method in ABC [37] with a built-in MiniSAT solver [13]. We present results on SpaceWire (SWR), Ethernet IP core (ETHNET) and floating point unit (FPU) designs from OpenCores [38] and IWLS2005 suite [39] for our experiments. All experiments were run on a Linux machine with Intel Core 2 Quad 2.66GHZ CPU and 8GB RAM.

The experimental results are shown in two subsections. Section 2.4.1 compares of generating bit-level assertions with SAT-BAG and Goldmine [10], one of the state-of-the-art dynamic works. Section 2.4.2 shows how optimization techniques, input constraints and word-level features can benefit SAT-BAG.

2.4.1 Comparison between SAT-BAG and the previous work

Table 2.1 compares the performance of SAT-BAG with Goldmine [10] for generating bit-level assertions. We set the window size w to 1 or 2 to get temporal assertions.

From this experiment, we observe that SAT-BAG generates high quality assertions that have higher input space coverage, and does not miss any corner case assertions. This is because all test cases can reach 100% input space coverage. In case `StarIdle`, SAT-BAG has lower average coverage per assertions than Goldmine because it generates more additional assertions that are difficult to be hit by simulation. With these assertions, SAT-BAG can reach 100% input space coverage while Goldmine can reach only 53.18%. Due to insufficient simulation, Goldmine also fails to find assertions in some cases. Furthermore, while Goldmine spends formal verification time and effort verifying "guesses" by data mining, SAT-BAG does not need this step.

To compare the quality of assertions generated by SAT-BAG and Goldmine, the redundant proposition distributions of Goldmine's assertions for `WB_ACK_O` and `expa_ff` are shown in Figs. 2.5 and 2.6. The distribution is built by counting redundant propositions in assertions of Goldmine. These redundant propositions are propositions that are not used in the assertions of SAT-BAG, which are more concise.

Although Goldmine generates more assertions than SAT-BAG on cases `WB_ACK_O`: (Goldmine 9 vs. SAT-BAG 8) and `expa_ff`: (Goldmine 119 vs. SAT-

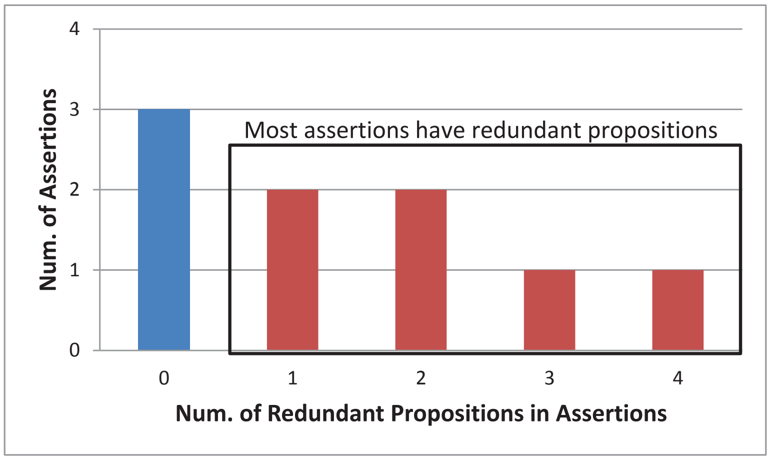


Figure 2.5: Distribution for redundant propositions of assertions generated by Goldmine for WB_ACK_O. Most assertions have redundant propositions

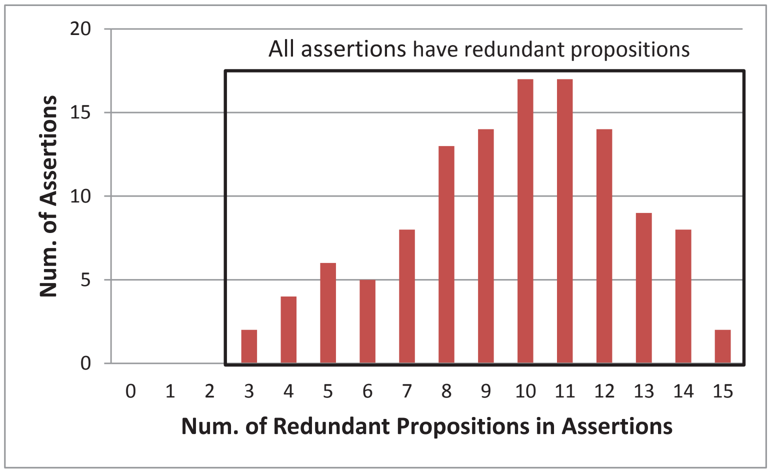


Figure 2.6: Distribution for redundant propositions of assertions generated by Goldmine for expa_ff. All assertions have redundant propositions

BAG 9), some of Goldmine's assertions have redundant propositions in antecedents. This means more than one assertion will convey the same amount of information and have same input space coverage. Hence, more assertions generated by Goldmine does not imply more information is gained.

2.4.2 Improvements of SAT-BAG

This section will demonstrate the benefits of applying input constraints and word-level features to SAT-BAG. Table 2.2 shows the results of SAT-BAG with and without input constraints. We use SpaceWire design, which has

Table 2.1: Comparison on Assertion Generation of SAT-BAG and the Previous Work, Goldmine [10].

Circuit (output)	SAT-BAG				Goldmine [10]			
	Asrt. #.	Avg. Cov. %	Tot. Cov. %	Time	Asrt. #.	Avg. Cov. %	Tot. Cov. %	Time
`SWR-active_o`	18	13.29	100	0.01	5	9.38	46.88	18.79
`SWR-err_sqc`	18	16.54	100	0.01	0	–	–	–
`SWR-RST_tx_o`	10	24.22	100	0.01	2	18.75	37.50	3.68
`ETHNET-StartIdle`	994	0.50	100	0.33	5	10.64	53.18	12.63
`ETHNET-ByteCntMax`	17	47.06	100	0.01	0	–	–	–
`ETHNET-ExcessiveDefer`	16	46.88	100	0.01	0	–	–	–
`ETHNET-WB_ACK_O`	8	17.58	100	0.10	9	11.11	100	7.86
`FPU-expa_ff`	9	44.49	100	0.01	119	0.39	46.81	18.49
`FPU-infa_f_r`	24	47.92	100	0.01	0	–	–	–
`FPU-exp_in_80`	9	44.49	100	0.01	9	11.11	100	3.67

* We list designs and target signal names, the number of assertions (Asrt. #.), the average input space coverage per assertion (Avg. Cov.), and the total input space coverage (Tot. Cov.) for SAT-BAG and Goldmine.
* Avg. Cov. for SAT-BAG might not equal to 100% divided by the number of assertions due to coverage overlap between assertions.

a one-hot encoding state machine, to show the effect of providing input constraints to SAT-BAG. The constraint for a one-hot encoded k bit state variable, $s[k-1:0]$, can be presented as: $\forall x, (k-1) \geq x \geq 0, s[x] = 1 \rightarrow \forall y, (k-1) \geq y \geq 0, y \neq x, s[y] = 0$. This constraint can be converted to SAT clauses by enumerating invalid values of the state variable. Therefore, SAT-BAG will avoid generating assertions from the invalid state values.

We observe that SAT-BAG can further filter out vacuous assertions and redundant propositions due to the usage of additional input constraints. Due to the input constraints, both the number of assertions and propositions decrease, and the average input space coverage increases. We do not show the total input space coverage here because SAT-BAG with constraints also achieves 100% input space coverage. Therefore, if we can provide input constraints to SAT-BAG, the generated assertions will be more meaningful and concise.

Table 2.2: Comparison on Assertion Generation of SAT-BAG With and Without Input Constraints.

Circuit (output)	w/o constraints			w/i constraints		
	Asrt. Num.	Avg. prop.	Avg. cov. %	Asrt. Num.	Avg. prop.	Avg. cov. %
`SWR-active_o`	18	4.56	13.29	15	2.80	17.50
`SWR-err_sqc`	18	3.33	16.54	15	2.07	25.42
`SWR-RST_tx_o`	10	3.00	24.22	7	1.29	44.64

* Avg. prop. denotes the average number of propositions in antecedents of assertions.

Table 2.3 shows the enhancement of SAT-BAG using word-level features. The first column shows designs and target signals. The remaining columns show the same information as previous tables for SAT-BAG using two different kinds of features. SAT-BAG-bit and SAT-BAG-word represent assertion generations using bit-level features and word-level features.

In this experiment, SAT-BAG-word generates few assertions but still achieves 100% input space coverage, which means it can generate more concise assertions, which have higher input space coverage, but also catch corner case assertions. Take `StartIdle` as an instance: the number of assertions can be reduced from 994 to 6 and the average number of propositions in antecedents decreases from 8.8 to 2.3. Furthermore, the average input space coverage increases drastically from 0.5% to 26.39% and the total input space coverage is still 100%. All these improvements result from using word-level features, which can cover many bit-level features.

Table 2.3: Comparison on Bit- and Word-Level Assertion Generation of SAT-BAG.

Circuit (output)	SAT-BAG-bit				SAT-BAG-word			
	Asrt. #	Avg. prop.	Avg. cov. %	Tot. Cov. %	Asrt. #.	Avg. prop.	Avg. cov. %	Tot. Cov. %
`SWR-active_o`	18	4.56	13.29	100	12	3.08	14.01	100
`SWR-err_sqc`	18	3.33	16.54	100	7	2.14	17.97	100
`SWR-RST_tx_o`	10	3.00	24.22	100	4	1.50	25.00	100
`ETHNET-StartIdle`	994	8.80	0.50	100	6	2.33	26.39	100
`ETHNET-ByteCntMax`	17	1.88	47.06	100	2	1.00	50.00	100
`ETHNET-ExcessiveDefer`	16	1.88	46.88	100	3	1.33	49.99	100
`ETHNET-WB_ACK_O`	8	2.88	17.58	100	5	2.22	22.81	100
`FPU-expa_ff`	9	1.78	44.49	100	2	1.00	50.00	100
`FPU-infa_f_r`	24	1.92	47.92	100	2	1.00	50.00	100
`FPU-exp_in_80`	9	1.78	44.49	100	3	1.33	49.87	100

Word-level assertions have high input space coverage and high readability. Some examples of bit-level assertions and word-level assertions are listed in Table 2.4 to explain these two attributes. For example, the word-level assertion of `active_o` has better readability than bit one, because it can use a word-level feature, $(state == Started)$, where $Started$ equals to $6'b001000$, to represent bit level features, $state[0] == 0$ && $state[1] == 0$ && $state[2] == 0$ && $state[3] == 1$ && $state[4] == 0$ && $state[5] == 0$. Furthermore, take `StartIdle` as an another instance; the two word-level features, $(NibCnt[6:0] \geq IPGT)$ and $(NibCnt[6:0] \geq IPGR2)$, in the word-level assertion obviously can cover a considerable number of bit-level features. Therefore, this word-level assertion not only has high input space coverage but also hits corner cases in the design. Of course, it has better readability than bit-level assertions as well.

2.5 Conclusion

SAT-BAG method was proposed to generate concise and high coverage assertions by using unit assumption technique as well as by considering input constraints and word-level features.

Table 2.4: Examples for Showing Difference between Bit- and Word-Level Assertions.

`active_o`	Word	(lnk_dis == 0 && lnk_start == 1 && **(state == Started)**) → (active_o == 1)
	Bit	(lnk_dis == 0 && lnk_start == 1 && **state[0] == 0** && . . . && **state[3] == 1** && . . .) → (active_o == 1)
`StartIdle`	Word	(StateIPG == 1 && **(NibCnt[6:0] ≥ IPGT)** && **(NibCnt[6:0] ≥ IPGR2)**) → (StartIdle == 1)
	Bit	(StateIPG == 1 && **NibCnt[6] == 1** && . . . && **IPGT[6] == 0** && . . . && **IPGT2[6] == 1** && . . .) → (StartIdle == 1)
`exp_in_80`	Word	(exp_in[7] == 1 && **(| exp_in[6:0] == 0)**)**→** (exp_in_80 == 1)
	Bit	(exp_in[7] == 1 && **exp_in[0] == 0** && . . . && **exp_in[6] == 0**) → (exp_in_80 == 1)

CHAPTER 3

C-MINE: DATA MINING OF LOGIC COMMON CASES FOR IMPROVED TIMING ERROR RESILIENCE WITH ENERGY EFFICIENCY

3.1 Introduction

As design complexity and environmental uncertainty grow, the better-than-worst-case (BTW) design methodology [16, 17] shows its strengths to improve circuit energy efficiency, performance, and reliability. In contrast to traditional design methodology, which selects a conservative guard band—higher Vdd or slower clock—for a circuit to guarantee 100% timing correctness, BTW design can operate the circuit more aggressively by removing the guard band and complementing it with an error detection and recovery mechanism such as Razor logic [40, 41] or Error-Detection Sequential (EDS) circuit [42]. With timing error correction, circuits can be more energy-efficient by overscaling Vdd [1, 16, 43] and more reliable because of the built-in robustness that can protect the circuits from dynamic voltage droop, aging, wearout of transistors, etc. [44].

Despite the promise of BTW design, its performance crucially depends on how often error correction intervenes, whose penalty includes flushing wrong results and re-running the same input patterns. In other words, the less frequency of triggering correction mechanism, the higher performance BTW design can gain. As Vdd scales down, timing errors start happening, and the error frequency relies on the *path delay distribution* of the design. If the majority of path delays are long and close to the guard band, the correction mechanism will be triggered too frequently, thus deteriorating the performance of BTW designs.

From the logic synthesis perspective, several works aim at reshaping path delay distribution for the benefit of BTW designs, which calls for improving the *timing error resilience* of designs. Reference [45] applies retiming technique to redefine the boundaries of combinational logic for better operation. Reference

[43] is a power-aware slack redistribution method to shift the slack of frequently exercised timing paths by gate sizing. DynaTune [46] improves commonly exercised paths by assigning lower V_t to critical gates. Blueshift [47] takes advantage of adaptive body biasing. Reference [48] reorders the inputs of gates for path delay balancing. CCP [18] is a very recent work that resynthesizes the circuit to reduce the delay of common case inputs, thus improving the overall energy efficiency by 15%.

Although CCP [18] can optimize the delay paths of common case inputs, its performance is potentially limited by the scalability issue. Under timing window constraints, CCP partitions the original circuit into sub-circuits and resynthesizes each sub-circuit based on probability analysis and BDD-based time characteristic function (TCF) analysis. However, the granularity of partitioning influences the performance of CCP drastically. Huge partition size is unmanageable for BDD-based TCF, while small partition size results in a considerable number of sub-circuits, i.e., tasks. Furthermore, CCP misses the whole picture of a design due to the partitioning. For example, CCP does not consider the variety of slacks of each sub-circuit's inputs; thus, the original circuit delay paths might be distorted during the partitioning and synthesis process. In the chapter, we propose new approaches using data mining and Boolean satisfiability (SAT) solving to overcome these limitations.

With more than a decade of intensive research, data mining [19,49] and SAT solving [20] become mature and popular for their scalable problem-solving capabilities. With great capability of model prediction and pattern matching, data mining has been used in design automation fields such as assertion generation [10] and hotspot detection [50]. In addition, the power of SAT solving has been demonstrated by its successful applications to many problems that suffer from scalability issues, such as model checking [51], reachability analysis [15], and assume-guarantee reasoning [14]. Therefore, we will take advantage of these two techniques to provide highly scalable solutions for BTW synthesis in this study.

We propose a **C**ommon case **Mining** method (**C-Mine**), which applies data mining and SAT solving to optimize the delay paths of common case inputs, the majority of path delay distribution. C-Mine has two versions, C-Mine-DCT [22] and C-Mine-APR [23]. They adopt different mining techniques to mine *common case cubes*, which represent *common case input patterns that have long delays*, from simulation data. C-Mine-DCT adopts *decision tree* *learning* with tree-pruning techniques proposed to avoid tree size explosion. Unfortunately, C-Mine-DCT might miss some critical common case cubes due to the nature of tree-based algorithms. Therefore, we proposed C-Mine-APR to approach the problem from a different angle. C-Mine-APR employs *Apriori-based frequent itemset mining* with database reduction techniques proposed to accelerate the process. However, without having exhaustive simulation, these cubes generated from the miners of C-Mine-DCT and C-Mine-APR should be regarded as "candidates." Therefore, we exploit SAT solving to verify these candidate cubes and also apply *unit assumption* [13] to further enlarge the obtained cubes. Finally, the timing error resilience of the BTW design can be improved based on these common case cubes.

Our contributions are summarized as follows. We propose C-Mine-DCT and C-Mine-APR methods, which successfully combine different data mining techniques and SAT solving to resynthesize the BTW design for better timing error resilience and energy efficiency. To improve their performance, tree-pruning and database reduction techniques are proposed to accelerate the mining processes, and a unit assumption technique is proposed to refine the results of SAT-based cube verification. Experimental results show that, compared to CCP [18], C-Mine-DCT can achieve compatible performance with an additional 8% energy saving and 54x speedup for bigger benchmarks on average. Furthermore, since C-Mine-APR has an exceptional ability of handling designs having more common cases, it can achieve up to 13% more energy saving than C-Mine-DCT while confronting these kinds of designs.

3.2 Background

This section briefs the reader on the error correction mechanism of BTW design used in this study and also illustrates the importance of path delay distribution for BTW design performance. Finally, the usage of common case cubes for improving the delay distribution is introduced.

3.2.1 Error correction mechanism of BTW design

Razor logic [40, 41] is the BTW design methodology used in this study to detect and recover the timing errors caused by dynamic voltage scaling [16].

Razor logic is a circuit-level transformation that augments each flip-flop in the design by adding a shadow latch to validate the captured values of the flip-flop.

Fig. 3.1 illustrates the concept of Razor flip-flop. The shadow latch, which is controlled by a delayed clock, provides a second sample of all pipeline circuit computations. Under normal operation where $L1$ combinational logic stage meets the setup time of main flip-flop, both the main flip-flop and the shadow latch will latch the same value; thus, no timing error is detected. However, in the case that $L1$ logic stage cannot finish its computation in time because of Vdd overscaling, the main flip-flop data will latch an incorrect value, which is different from the correct late-arriving value lathed by the shadow latch; thus, a timing error is detected.

For correcting timing errors, Razor logic proposed two approaches, which have different design complexity and cycle penalty, to recover pipeline state. One simple method is based on global clock gating to directly restore the main flip-flop to the correct value in the shadow latch, but this method is slow and cannot be applied to complex designs; while the other method is based on counterflow pipeline techniques [52] and is very scalable with multiple cycle penalty. More details can be found in [40, 41].

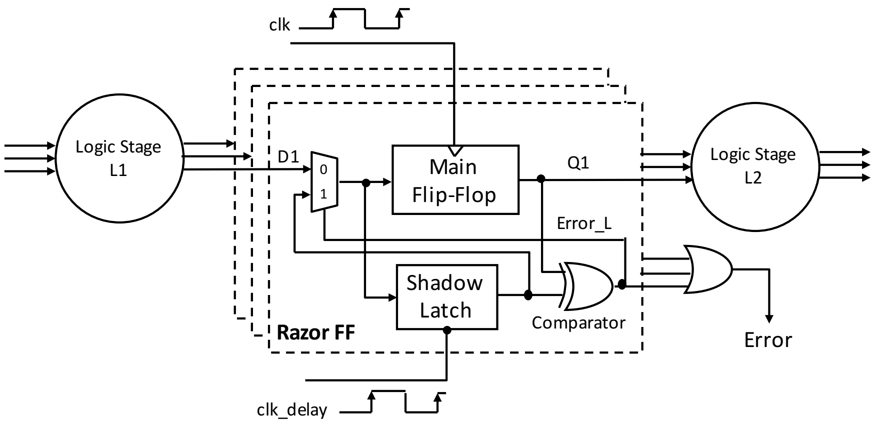


Figure 3.1: Error correction mechanism of Razor flip-flop [40, 41].

3.2.2 Importance of correctness probability curve for BTW design performance

Scaling down Vdd can save power immediately, but the overall energy efficiency also depends on throughput [18]. Equation (3.1) defines a BTW design's *throughput* (TR) as a function of operating *frequency* f, the *correctness probability* (P_s) of all primary outputs, POs, to be stabilized by the cycle time $1/f$, and the *error correction penalty* r, which includes flushing wrong results and replaying the input patterns [40, 41]. Equation (3.2) defines the overall energy cost as the product of the expected *run time* (inverse of TR) and the *power consumption* at Vdd. Therefore, derived from these two equations, overscaling Vdd can save energy only if the throughput can be maintained at a certain level.

$$TR = P_s \times f + (1 - P_s) \times \frac{f}{r} \tag{3.1}$$

$$E = P(Vdd) \times \frac{1}{TR} \tag{3.2}$$

In BTW design optimization, the correctness probability (P_s) plays a critical role in maintaining the throughput. In traditional design methodology, Vdd and clock are chosen conservatively to guarantee $P_s = 100\%$, known as a guard band. However, with Vdd overscaling, P_s drops as the Vdd scales down, thus compromising the throughput due to error correction penalty. To keep the throughput high, we need to maintain a high correctness probability P_s even if the circuit slows down.

Fig. 3.2, for example, shows the correctness probability curves of two different implementations D_1 and D_2 of the same design. With conservative guard band, both D_1 and D_2 can operate at $t_{clk} = 500$ ps with no timing errors — $P_s = 100\%$. However, there is a significant P_s difference between D_1 and D_2 if operating at $t_{clk} = 390$ ps. Considering a scenario that Vdd overscaling slows down the signal propagation by 110 ps (i.e., both curves shift right by 110 ps), under the same clock setting, D_2 will have a considerable drop of P_s from 100% to 50%, while D_1 can still maintain its P_s at 96.2%. Obviously, the implementation D_1 is more suitable than D_2 for BTW design operation, and the purpose of this chapter is to resynthesize design for providing this BTW-specific feature.

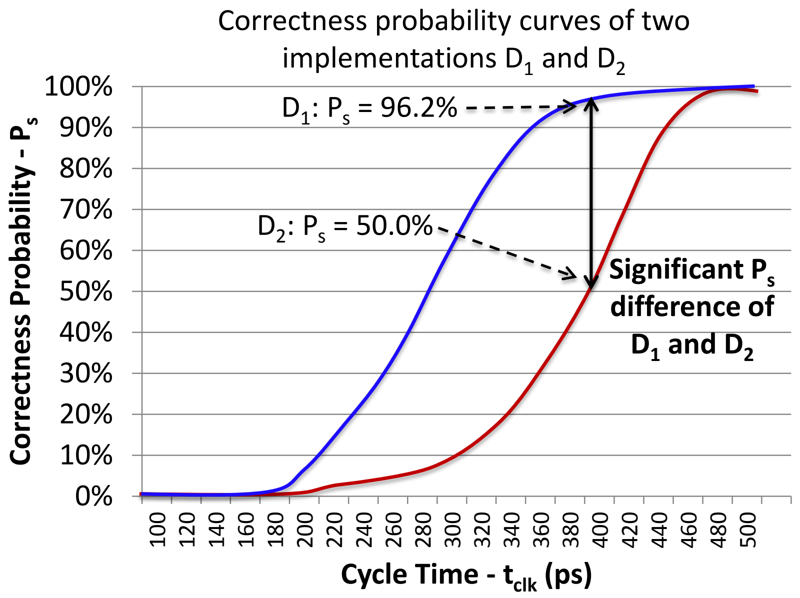


Figure 3.2: Correctness probability curves of two different implementations for the same design.

3.2.3 Correctness probability curve reshaping

The correctness probability curve can be improved by appending *shortcut logic* into the original design [18]. Shortcut logic is a small design built from common case cubes to provide shortcuts for frequent input patterns that have long delays, thus improving their delays. The overall concept is to merge on-set and off-set common case cubes with the corresponding POs using OR and AND gates, respectively. This curve-reshaping strategy has been proved in [18] to keep the functionality of the design intact after the appending. However, appending shortcut logic into design would introduce additional area cost; therefore, in this work, we will couple this strategy with redundancy removal techniques to minimize area overhead.

For example, there is a 4-input circuit shown in Fig. 3.3, and we assume it has a frequent input pattern $(a, b, c, d) = (1, 1, 1, 1)$, which frequently triggers a long delay path with a delay of 3. The common case cube[1] to collect this frequent pattern is *abcd*, and its corresponding shortcut logic (marked in blue) is merged into the original circuit to improve the delay of this pattern from 3 to 2.3.

[1]*abcd* should be a minterm, but we call it a cube for consistency.

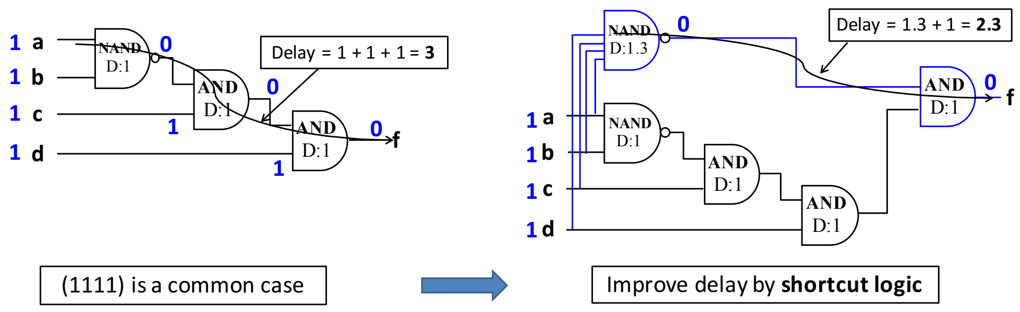


Figure 3.3: An example demonstrates the delay of a common case improved by shortcut logic.

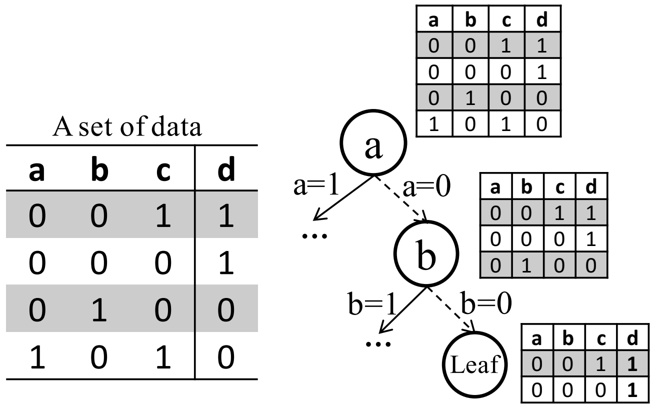
Therefore, in this work, we focus on identifying common case cubes from a design and then apply these cubes to reshape the correctness probability curve of design, improving its timing error resilience.

3.3 Preliminaries

This section briefly introduces the techniques of data mining and SAT solving used in this chapter.

3.3.1 Data mining technique

Data mining techniques [19, 49] can be roughly classified into two categories: *constraint-based* and *non-constraint-based* ones. Usually, users have a good sense of the mining direction that might lead to the "data" they are interested in. Therefore, constraint-based mining is often referred to as the mining approach that considers such user intuition and expectations to determine the direction of mining, e.g., frequent pattern mining and classification, while non-constraint-based ones do not, e.g., clustering and outlier detection. The following paragraphs will introduce the two constraint-based mining techniques used in this work, *decision tree learning* and *frequent itemset mining*, as well as the related terminology and algorithm.



a	b	c	d
0	0	1	1
0	0	0	1
0	1	0	0
1	0	1	0

a	b	c	d
0	0	1	1
0	0	0	1
0	1	0	0

a	b	c	d
0	0	1	**1**
0	0	0	**1**

Figure 3.4: An example of decision tree building on a set of data.

3.3.1.1 Decision tree learning

Decision tree learning is a widely used technique in data mining to train a predictive model from a set of data. This model can predict the value of a target variable based on several input variables. A decision tree is a flowchart-like tree structure, where each *internal node* denotes a split on one of input variables, each *branch* represents a decision of the input variable, and each *leaf node* (or *terminal node*) represents a predicted value of the target variable. The path from the root to the leaf represents a possible implication between the target variable and the input variables with decided values.

During tree building, the source set will split into subsets based on the decision of input variables, and this splitting process will terminate when the subset at a node has the same value of the target variable. Fig. 3.4 shows an example of decision tree building on three input variables a, b, and c and a target output variable d. A path from the root to the leaf stands for a prediction that is $((a, b) = (0, 0)) \rightarrow (d = 1)$.

3.3.1.2 Gini index

Gini index, in this work, is a criterion that measures the impurity of D, a set of Boolean values 0 and 1, defined as

$$Gini(D) = 1 - (\frac{|D_0|}{|D|})^2 - (\frac{|D_1|}{|D|})^2$$

where D_0 (resp. D_1) denotes a subset of D that collects all 0 (resp. 1) values. For example, the Gini index of a set $D : \{0, 0, 1, 1, 1\}$ is $1 - (\frac{2}{5})^2 - (\frac{3}{5})^2 = 0.48$. Please note that the Gini index of a set equaling 0 means the set is a pure set, which has the same values.

3.3.1.3 Frequent itemset

A set of items is referred to as an *itemset*. An itemset that contains k items is a *k-itemset* (e.g., an itemset $\{a, b\}$ is a 2-itemset). The occurrence frequency of an itemset I is the number of transactions (patterns) that contain I. This frequency is also known as the *support* or *support count* of I. I is a *frequent itemset* if its support is large than or equal to a user-prespecified *minimum support threshold*, *min_sup*. The set of frequent k-itemsets is commonly denoted by L_k.

3.3.1.4 Apriori algorithm

Apriori [19,49,53] is a frequent itemset mining algorithm. It adopts a level-wise search, where k-itemsets are used to explore $(k + 1)$-itemsets. The algorithm is briefly described as follows: At first, the set of frequent 1-itemsets, L_1, is generated by scanning the database to accumulate the count for each item and collecting those items that satisfy *min_sup*. Next, to find the set of frequent 2-itemsets, L_2, a set of "candidate" frequent 2-itemsets, denoted by C_2, is generated by *joining* L_1 with itself, and L_2 is then determined by collecting those candidate 2-itemsets in C_2 that satisfy *min_sup*. This process will continue until no more frequent k-itemsets can be found.

However, the generation of each L_k requires one full scan of the database. Therefore, to improve the efficiency of the level-wise search for frequent itemsets, *Apriori property* is proposed to confine the search space.

Definition 2. ***Apriori property*** *[19]: All nonempty subsets of a frequent itemset must also be frequent.*

For example, if $\{a, b\}$ is a frequent itemset, both $\{a\}$ and $\{b\}$ should be a frequent itemset. The property is used as follows. Any $(k - 1)$-itemset that is not frequent cannot be a subset of a frequent k-itemset. Thus, if any $(k - 1)$-subset of a candidate k-itemset is not in L_{k-1}, then this candidate cannot be frequent either and so can be removed from C_k directly. This subset test can be done efficiently by maintaining a hash tree of all frequent itemsets.

Furthermore, many techniques are also proposed to accelerate Apriori algorithm such as hash-based technique, transaction reduction, partitioning, and sampling, of which details can be found in [19, 49].

3.3.2 Boolean satisfiability problem and technique

A Boolean satisfiability (SAT) problem is a decision problem [20]. Given a finite set of Boolean variables $V = \{v_0, \ldots, v_n\}$, a *literal* l is either a Boolean variable v_i or its negated form $\neg v_i$ (or $\sim v_i$). A *clause* is a disjunction of literals. A *SAT instance* is a conjunction of clauses, also named as *conjunction normal form (CNF)*. An *assignment* over V assigns each Boolean variable v_i either true or false value. A SAT instance is **satisfiable** if there exists a *satisfying assignment* such that the SAT instance evaluates to true. Otherwise, it is **unsatisfiable**.

Unit assumption is a SAT technique of MiniSAT [13], which allows users to assume specific conditions for each SAT solving call through an interface $S.solve(assumps)$, where S is a SAT instance, and $assumps$ is a conjunction of literals like $a_0 \wedge a_1 \wedge \ldots \wedge a_n$. For each SAT solving call, literals a_i are *temporarily* assigned to true and will be discharged after the solving. The return data of this technique are as follows:

- **Satisfiable**: Return an assignment that can satisfy the SAT instance as well as the assumptions.
- **Unsatisfiable**: Return a **subset** of those assumptions that contribute to the unsatisfiable result.

3.3.3 Conversion of a circuit into SAT instance

Given a circuit netlist, its functionality can be totally represented by a SAT instance using Tseitin transformation [31, 32]. This transformation can be done in linear time. For instance, a two input AND gate, $c = AND(a, b)$, can be converted into three clauses, $(v_a \vee \neg v_c) \wedge (v_b \vee \neg v_c) \wedge (\neg v_a \vee \neg v_b \vee v_c)$, whose satisfying assignments comprise the valid functionality of the AND gate.

3.4 C-Mine algorithm

This section introduces the algorithms of C-Mine-DCT and C-Mine-APR, both of which contain two main phases: (1) candidate mining of common case cubes and (2) SAT-based cube verification and enlargement. After collecting common case cubes for timing-critical POs, the corresponding shortcut logic will be generated and merged into the original design for a better correctness probability curve.

3.4.1 The flowcharts of C-Mine-DCT and C-Mine-APR

The overall flowcharts of C-Mine-DCT and C-Mine-APR are shown in Fig. 3.5 and Fig. 3.6, respectively. Before running C-Mine, we simulate the design and collect the delay information of each simulated input pattern. In addition, timing-critical POs, whose probability curves need to be improved, are identified by static timing analysis during this preprocess.

Both C-Mine-DCT and C-Mine-APR consist of two phases: candidate mining and verification. They have individual mining processes but share the same verification process. For each timing-critical PO, its common case cubes are generated by C-Mine-DCT and C-Mine-APR as follows.

In the mining phase, C-Mine-DCT applies decision tree learning (Sec. 3.3.1.1) to generate possible common case cubes from the simulation data. To avoid tree size explosion, we proposed early tree pruning techniques to reduce unnecessary and non-promising space searching. The mining process of C-Mine-DCT will repeat until reaching tree-level limits or no more promising nodes to split. C-Mine-APR employs Apriori (Sec. 3.3.1.4), a frequent itemset mining algorithm, to generate possible cubes from the simulation data. To avoid the "curse of dimensionality" [19], we proposed early database shrinking techniques to reduce the size of database beforehand and also skip the small-size ones. The mining process of C-Mine-APR will repeat until no more frequent itemsets are found. Please note that all cubes generated by the mining phases of C-Mine-DCT and C-Mine-APR are just *candidates* because they are not collected after exhaustive simulation. Additionally, the accuracy of common case mining can be improved by using directed simulation if the workload is provided.

In the verification phase, at first, we apply a SAT-based covering check procedure to filter out candidate cubes that are already covered by existing common case cubes, thus avoiding unnecessary cube verification and reducing the area of shortcut logic. Next, a SAT-based verification is applied to verify the candidates. If candidate cubes are proved to be true, further cube enlargement can be obtained by applying unit assumption; if false, the corresponding counterexamples are generated. Although C-Mine-DCT and C-Mine-APR share the same verification process, the feedbacks provided for their individual mining phases are different. For C-Mine-DCT, the counterexamples in the proof of false cubes are provided back to its mining phase to refine the tree, thus enhancing the hit rate of mining, while for C-Mine-APR, the information of true cubes is provided to reduce the size of candidate frequent itemsets, C_i, of Apriori algorithm, thus accelerating the mining process without sacrificing the performance. Finally, the common case cubes of timing-critical POs are collected for constructing shortcut logic.

C-Mine-DCT and C-Mine-APR terminate when no more qualified candidate cubes are found. The algorithms are detailed in the following subsections.

3.4.2 C-Mine-DCT: Decision-tree-based candidate mining of common case cubes

C-Mine-DCT adopts a *promise-driven tree growing* strategy to generate candidate common case cubes; that is, it prunes some tree branches early and only allows the decision tree to split on promising nodes. *Promising nodes* are decisions (directions) that have a higher probability of finding good candidates. Therefore, we proposed effective criteria to evaluate the *promise value* of a node. Additionally, C-Mine-DCT traverses the tree by breadth-first search (BFS) with a tree-level constraint, which is less than five. This promise-driven strategy can guide the searching to target at big common case cubes that can improve more delay paths using less area cost in shortcut logic. With the tree pruning techniques and level-constraint BFS, we can not only find high-quality cubes but also prevent tree size explosion.

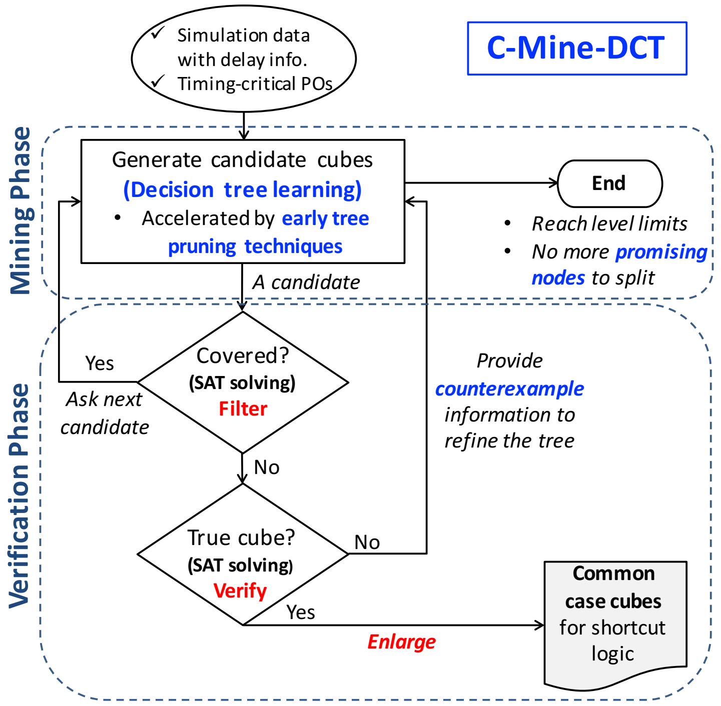

Simulation data with delay info.
Timing-critical POs
C-Mine-DCT
Mining Phase
Generate candidate cubes
(Decision tree learning)
Accelerated by early tree pruning techniques
End
Reach level limits
No more promising nodes to split
A candidate
Verification Phase
Covered? (SAT solving) Filter
Yes
Ask next candidate
No
Provide counterexample information to refine the tree
True cube? (SAT solving) Verify
No
Yes
Enlarge
Common case cubes for shortcut logic

Figure 3.5: The flowchart of C-Mine-DCT, and its main contributions are marked in blue and red.

Fig. 3.7 is a running example of tree growing by splitting on an input a, which will partition a set of simulation data of a 3-input design into two subsets. Based on this splitting relationship, we call the original set F a *parent set* and its two subsets, $F_{a=0}$ and $F_{a=1}$, *child sets*. In the simulation database (set F in the figure), each entry consists of an input pattern, the value of the target output (f), and delay information (d_f). The tree-pruning techniques and candidate evaluation used in C-Mine-DCT will be introduced by this running example in later subsections.

3.4.2.1 Avoidance of unnecessary searching

Since the tree search space increases exponentially with the number of internal nodes (i.e., decisions), the avoidance of irrelevant decisions is an intuitive and effective way to reduce the search space; therefore, we apply the fanin cone information of the target output to identify *related inputs*, which are those in the fanin cone, and only grow the tree by making decisions on these inputs.

Furthermore, to avoid duplicate searching, we set an order constraint such that the tree nodes should split in ascending order such as $x_1 \prec x_2 \prec x_3 \prec \ldots$,

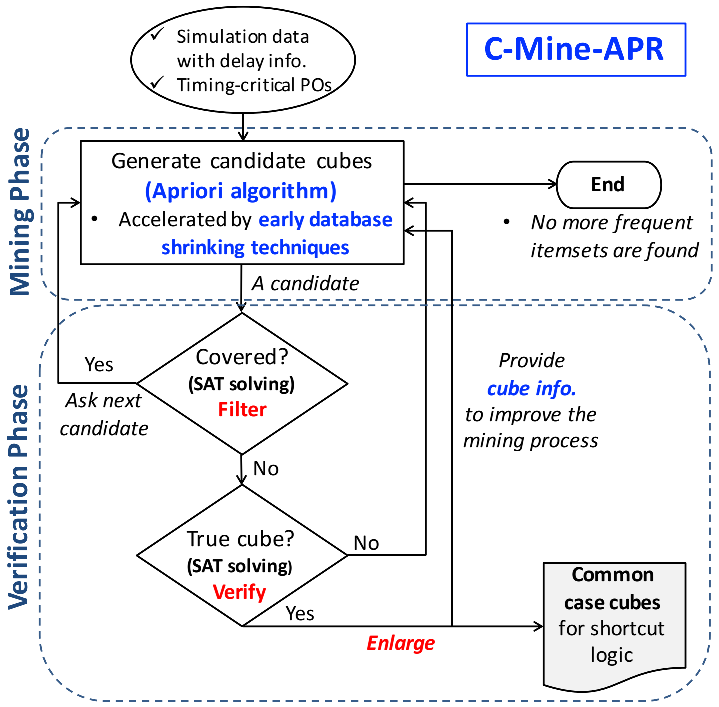

C-Mine-APR
✓ Simulation data with delay info.
✓ Timing-critical POs
Mining Phase
Generate candidate cubes
(Apriori algorithm)
• Accelerated by early database shrinking techniques
End
• No more frequent itemsets are found
A candidate
Verification Phase
Yes
Ask next candidate
Covered? (SAT solving) Filter
No
True cube? (SAT solving) Verify
No
Yes
Provide cube info. to improve the mining process
Enlarge
Common case cubes for shortcut logic

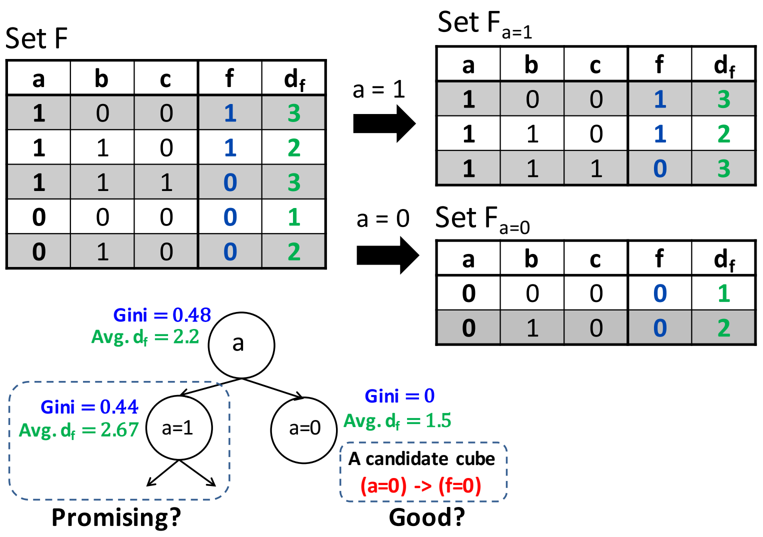
Figure 3.6: The flowchart of C-Mine-APR, and its main contributions are marked in blue and red.

where x_i is the ith decision variable. Keeping a specific order can shrink the searching space drastically. For example, without the order constraint, two splitting sequences $\{a = 1,\ c = 0\}$ and $\{c = 0,\ a = 1\}$ in Fig. 3.7 will be traversed; however, both of them will generate the same candidate cube, $((a, c) = (1, 0)) \rightarrow (f = 1)$. Therefore, we can know that only one searching direction, which is $\{a = 1, c = 0\}$ in our setting, is needed.

Although this order constraint can save enormous searching time, it might impair the quality of candidates by inserting redundant input decisions, such as making decisions on non-controlling values of gates. Fortunately, this situation can be recovered by the cube enlargement mechanism in our SAT-based verification.

3.4.2.2 Promise of splitting nodes

The promise of nodes is evaluated by the *Gini index* and *timing information* of their corresponding sets. Splitting on promising nodes can increase the probability of finding high-quality candidates as well as accelerate the tree growing process. The physical meanings of evaluation are as follows:



Set F

a	b	c	f	d_f
1	0	0	1	3
1	1	0	1	2
1	1	1	0	3
0	0	0	0	1
0	1	0	0	2

Set $F_{a=1}$

a	b	c	f	d_f
1	0	0	1	3
1	1	0	1	2
1	1	1	0	3

Set $F_{a=0}$

a	b	c	f	d_f
0	0	0	0	1
0	1	0	0	2

Figure 3.7: A running example for demonstrating promise-driven decision tree growing for candidate common case cubes.

- Gini Index: If the *Gini index* of a child set is **lower** than its parent's, it means this decision leads to a purer set, that is, the target output is converging to the same Boolean value, 0 or 1. Thus, splitting on this node will have a higher probability of reaching the candidates.

- Timing Information: If the *average delay* of a child set is **higher** than its parent's, it means this decision will lead to a subspace that contains common case inputs having longer delays.

With these two evaluating criteria, C-Mine-DCT follows a positive strategy that a decision node will be **discarded** only if (1) its Gini index is higher than its parent's and (2) its average delay is lower than its parent's at the same time. Otherwise, the tree will keep splitting on this node.

Take the same example in Fig. 3.7 to show. The original set F has split into two child sets $F_{a=0}$ and $F_{a=1}$ based on the decisions made on input variable a. Now we are going to determine the next splitting nodes. Obviously, we just need to examine the promise value of decision $a = 1$ because the other decision $a = 0$ is already a candidate. In this case, the decision $a = 1$ is promising because its Gini index (0.44) is lower than its parent's (0.48) and its average delay (2.67) is higher than its parent's (2.2). Therefore, the tree will keep splitting on this promising node $a = 1$.

3.4.2.3 Quality of candidates

To save total runtime, we only pass high-quality candidates to the SAT-based verification phase. The criterion C-Mine-DCT uses is defined as below.

- Timing Information: If the *average delay* of the set at the node that finds a candidate is **higher** than the *average delay* of the **whole** simulation data set, this candidate will be considered high-quality and passed to the next verification phase.

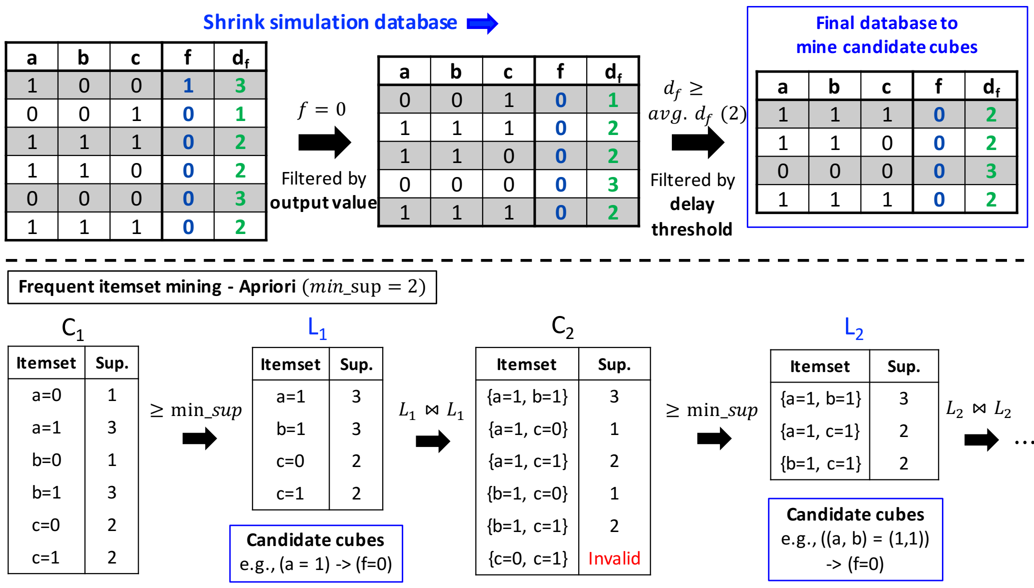
In the same case of Fig. 3.7, although $(a = 0) \to (f = 0)$ is a candidate, it will be discarded because of its low quality, which is evaluated by the fact that the average delay (1.5) of set $F_{a=0}$ is lower than the average delay (2.2) of the whole data set F.

3.4.3 C-Mine-APR: Apriori-based candidate mining of common case cubes

C-Mine-APR is an *Apriori-based* approach to generate candidate common case cubes. We formulated the common case cube generation problem into a *frequent itemset mining* problem, where items are *input variables assigned to* 0 *or* 1, and itemsets are *cubes*. For example, an input assignment $a = 0$ is seen as an item, and an itemset $\{a = 0, b = 1\}$ represents a cube $\bar{a}b$.

However, a high-dimensional database usually leads to a well-known problem in data mining: "curse of dimensionality" [19], that is, searching over high-dimensional spaces is time-consuming. Therefore, we applied the domain knowledge of common case cubes to reduce the size of simulation database in advance, thus improving the computational efficiency of mining as well as the quality of cubes.

Fig. 3.8 is a running example of frequent itemset mining of common case cubes for a target output $f = 0$, and the simulation data of a 3-input design like Fig. 3.7 is stored in a database. The proposed database shrinking techniques and candidate generation procedure used in C-Mine-APR will be introduced by this running example in later subsections.


Shrink simulation database
$f = 0$
Filtered by output value
$d_f \geq avg.\ d_f$ (2)
Filtered by delay threshold
Final database to mine candidate cubes
Frequent itemset mining - Apriori ($min\_sup = 2$)
C_1
$\geq$ min_sup
L_1
$L_1 \bowtie L_1$
C_2
$\geq$ min_sup
L_2
$L_2 \bowtie L_2$
…
Candidate cubes e.g., (a = 1) -> (f=0)
Candidate cubes e.g., ((a, b) = (1,1)) -> (f=0)

a	b	c	f	d_f
1	0	0	1	3
0	0	1	0	1
1	1	1	0	2
1	1	0	0	2
0	0	0	0	3
1	1	1	0	2

a	b	c	f	d_f
0	0	1	0	1
1	1	1	0	2
1	1	0	0	2
0	0	0	0	3
1	1	1	0	2

a	b	c	f	d_f
1	1	1	0	2
1	1	0	0	2
0	0	0	0	3
1	1	1	0	2

Itemset	Sup.
a=0	1
a=1	3
b=0	1
b=1	3
c=0	2
c=1	2

Itemset	Sup.
a=1	3
b=1	3
c=0	2
c=1	2

Itemset	Sup.
{a=1, b=1}	3
{a=1, c=0}	1
{a=1, c=1}	2
{b=1, c=0}	1
{b=1, c=1}	2
{c=0, c=1}	Invalid

Itemset	Sup.
{a=1, b=1}	3
{a=1, c=1}	2
{b=1, c=1}	2

Figure 3.8: A running example for demonstrating Apriori-based mining for candidate common case cubes.

3.4.3.1 Database reduction

To avoid the curse of dimensionality, we apply the domain knowledge of circuits and common case cubes to reduce the size of the database in advance. This database reduction can not only lead to more efficient searching but also enhance the quality of candidate cubes. Given a target output and its value (i.e., $f = 0$ in Fig. 3.8), the proposed reduction techniques are applied as follows.

To reduce the columns of the database, which are input assignments, the fanin cone information of the target output is employed to filter out *unrelated inputs*, which are those not in the fanin cone. Removing these unrelated input assignments (columns) from the database will not affect the correctness of common case cube mining for the target output, because they do not determine the value of target output.

To reduce the rows of the database, which are simulation patterns and results, the value of target output and its timing information are both employed to filter out redundant rows. Based on the target output value, $f = val$, we can remove the rows (i.e., simulation patterns) whose output values are not equal to val, because they will not be involved in the cube mining of $f = val$. Furthermore, we filter out rows whose delays are below a user-specified *delay* *threshold*, D, because the cubes we want to mine are the ones that cover the input patterns triggering long delay paths.

To avoid unnecessary mining, the mining process of common case cubes for $f = val$ will **terminate** if the remaining rows of the reduced database are below a threshold, which is set to 10% of the original rows in this work. Please note that common case cubes should satisfy two conditions: triggering long delay paths and occurring frequently. Therefore, if the size of the reduced database is too small, it implies that not too many common case cubes exist for $f = val$, and this case should be skipped.

Take the example in Fig. 3.8 to demonstrate the reduction process. We assume all inputs are in the fanin cone of the target output f; therefore, no columns of the database are removed at this time. To generate common case cubes for the target output $f = 0$, we can filter out a row of $f = 1$. Then, a delay threshold, which is set to the average delay (2), is applied to filter another row of $d_f = 1$. The final reduced database is obtained by removing these two rows. Finally, the mining process of common cases cubes for $f = 0$ will begin since the final database has sufficient rows remaining.

3.4.3.2 Frequent itemset mining of candidate cubes

Having the reduced database, our Apriori-based frequent itemset mining approach will begin to mine candidate common case cubes for the target output $f = val$. Before running the proposed approach, users need to specify a minimum support threshold, *min_sup*. The threshold *min_sup* is used to filter out the itemsets (cubes) of **low frequency**, thus controlling the quality of generated common case cubes and the speed of the approach. We suggest that the value of *min_sup* is set between 40% to 60% of the total row count of the database based on the design.

Use the same example in Fig. 3.8 to illustrate the algorithm. Given the reduced database and *min_sup* = 2, at the first iteration, each item (i.e., input assignment) is a member of the set of candidate 1-itemsets, C_1, and the occurrence count of each item is obtained by simply scanning the whole database once. The set of frequent 1-itemsets, L_1, can then be determined by collecting the candidate 1-itemsets that satisfy *min_sup* = 2. To find the set of frequent 2-itemsets, L_2, the algorithm uses the join $L_1 \bowtie L_1$ to generate a set of candidate 2-itemset, C_2, where although $\{c = 0, c = 1\}$ is a candidate

2-itemset, it will be removed from C_2 because it is not a valid cube. Then L_2 can be collected from C_2 according to min_sup. The algorithm will iterate until L_k is an empty set.

Itemsets in each L_i are candidate common case cubes for $f = 0$. The physical reason behind this is that these itemsets are frequent because of belonging to L_i, and they also cover long delay paths because they are mined from the reduced database that only stores simulation patterns with long delays. Additionally, the itemsets representing invalid cubes in C_i (e.g., $\{c = 0, c = 1\} \in C_2$) will be ignored during the process to avoid unnecessary search and verification.

3.4.4 SAT-based cube verification and enlargement

In the verification phase, SAT solving plays an important role in verifying and enlarging candidate cubes passed from the mining phase. In addition, SAT solving also conducts a covering check to filter out candidates that have been covered by already-collected common case cubes to save runtime. Furthermore, different feedback is provided to improve the mining phases of C-Mine-DCT and C-Mine-APR accordingly.

Initially, two copies of the SAT instance of the circuit, S_{cov} and S_{ver}, are prepared for covering check and verification/optimization, respectively. The technique of converting the circuit into the SAT instance is detailed in Sec. 3.3.3. The usage of these two SAT instances is introduced next.

3.4.4.1 Cube covering check

Before verifying a candidate, C-Mine will check if this candidate has been covered by existing common case cubes. This action can prevent collecting cubes that improve the same delay paths, thus saving unnecessary verification time as well as area cost of shortcut logic.

Here S_{cov} works like a list to register what common case cubes have been collected so far. This registering action can be accomplished by using *blocking clauses*, which are clauses that can forbid SAT instances to generate specific satisfying assignments. Therefore, each time a common case cube is collected, we will add a blocking clause of this cube into S_{cov} to avoid collecting duplicate cubes in the future.

For example, assume a common case cube $((a, b) = (0, 1)) \rightarrow (f = 1)$ is found, it will be registered by adding a blocking clause, $(v_a \lor \neg v_b \lor \neg v_f)$, into S_{cov}. Next time, if we obtain a candidate cube like $((a, b, c) = (0, 1, 0)) \rightarrow (f = 1)$, which is obviously covered by the previously found cube, it will be filtered out because of the **unsatisfiable** result of SAT solving: $S_{cov}.solve(\neg v_a \land v_b \land \neg v_c \land v_f)$.

3.4.4.2 Cube verification and enlargement

Candidate cubes might be false, while true cubes could be enlarged by removing redundant input decisions. Therefore, we apply *unit assumption*, a SAT technique, to verify and enlarge cube candidates **simultaneously**.

Take the same example in Fig. 3.3 to demonstrate the verification process. Assume $((a, b, c, d) = (1, 1, 1, 1)) \rightarrow (f = 0)$[2] is a candidate cube we obtain, and it will be verified by solving the SAT instance S_{ver} with a specific unit assumption, $S_{ver}.solve(v_a \land v_b \land v_c \land v_d \land v_f)$. The physical meaning behind this solving is to ask an opposite question, *Does there exist a satisfying assignment such that $((a, b, c, d) = (1, 1, 1, 1)) \rightarrow (f = 1)$?* Two possible results of candidate cube verification are explained as follows:

- **Satisfiable**: The candidate cube is a **false** cube because there exists a satisfying assignment, which is a counterexample to disprove the candidate cube.

- **Unsatisfiable**: The candidate cube is a **true** common case cube because $(a, b, c, d) = (1, 1, 1, 1)$ always implies $(f = 0)$. Furthermore, unit assumption technique can report a **subset** of assumptions that mainly contribute to this unsatisfiable result. In this case, a subset $\{v_a, v_b, v_f\}$ will be reported, which means $(a, b) = (1, 1)$ is sufficient to determine the value of f to 0.

In sum, the candidate cube $((a, b, c, d) = (1, 1, 1, 1)) \rightarrow (f = 0)$ is verified (true) as well as **enlarged** to a bigger cube $((a, b) = (1, 1)) \rightarrow (f = 0)$. Compared to the original cube, this enlarged cube not only contributes a shorter delay but also costs less area in the shortcut logic, which can been seen in Fig. 3.9.

[2]This candidate cube should be a minterm, but we call it a cube for consistency.

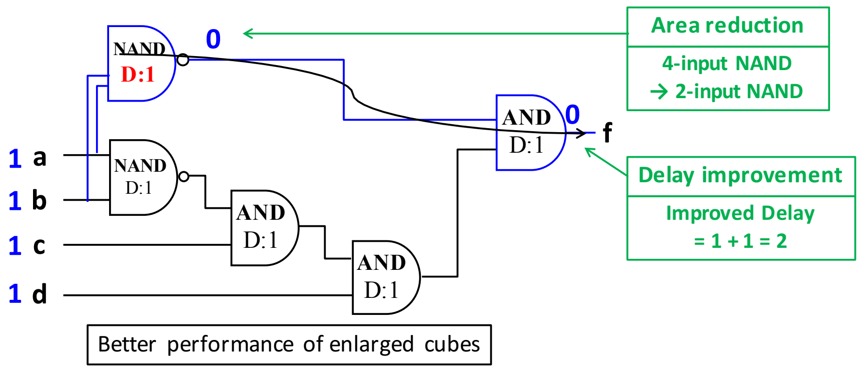


Figure 3.9: Enlarged common case cubes contribute towards better shortcut logic in terms of delay and area cost.

3.4.4.3 Feedback for improving the mining phase

After verifying candidate cubes, some useful feedback can be provided to improve the mining phases of C-Mine-DCT and C-Mine-APR accordingly. C-Mine-DCT can use the counterexamples from the proof of **false cubes** to refine the decision tree incrementally, while C-Mine-APR can apply the information from **true cubes** to reduce the search space of candidate cubes, thus accelerating its mining phase. The usage of feedback is introduced with the examples in Fig. 3.10 as follows.

The decision tree-based mining of C-Mine-DCT can be improved by considering the counterexamples from the proof of false cubes. In the previous example of Fig. 3.7, $(a = 0) \rightarrow (f = 0)$ is identified as a candidate cube based on the algorithm of C-Mine-DCT. Assume this candidate is proved as a false cube, and its corresponding counterexample will also be generated by the SAT-based verification. To refine the tree incrementally, shown in Fig. 3.10(a), this counterexample will be added to the simulation data set at the node that find the candidate, so that this node is no longer a leaf node, and the tree can keep splitting on this node if it is promising. This feedback loop can help C-Mine-DCT to increase the hit rate of its tree-based candidate mining.

The Apriori-based frequent itemset mining of C-Mine-APR can be accelerated by considering the information from true cubes. In the previous example of Fig. 3.8, we can know that $(a = 1) \rightarrow (f = 0)$ is a candidate cube because the itemset $\{a = 1\}$ is in L_1. Assume this candidate is verified as a true cube, which will not only be collected for building shortcut logic but also be

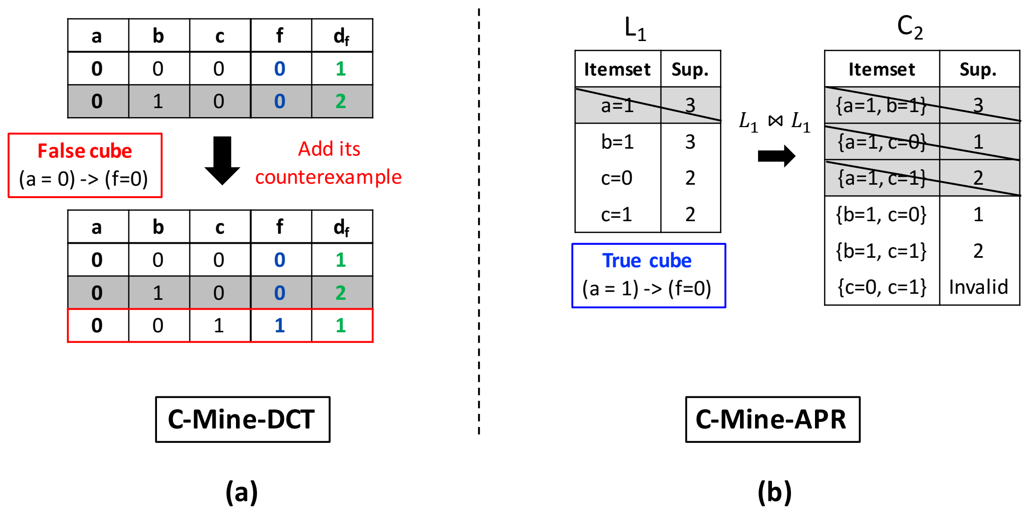


Figure 3.10: Different feedback is provided to improve the mining phases of (a) C-Mine-DCT and (b) C-Mine-APR accordingly.

used to avoid unnecessary candidate search in the mining phase. As shown in Fig. 3.10(b), the itemset $\{a = 1\}$ can be removed from L_1 because it has been verified (true). This removal of itemset $\{a = 1\}$ from L_1 will reduce the size of C_2, which is generated by joining L_1 and itself, effectively. The idea is that removing itemsets at the early stage can shrink the search space rapidly and also avoid collecting duplicate cubes. In the example, three candidates in C_2, $\{a = 1, b = 1\}$, $\{a = 1, c = 0\}$, and $\{a = 1, c = 1\}$, have already been covered by the true cube $(a = 1) \to (f = 0)$. Therefore, ignoring these candidates will not compromise the results of mining. This feedback loop can help C-Mine-APR to speed up its candidate mining drastically as well as enhance the quality of candidate cubes.

3.5 Comparison and discussion

This section compares the features of C-Mine-DCT and C-Mine-APR and also discusses their possible corner cases.

3.5.1 Comparison of C-Mine-DCT and C-Mine-APR

Although both C-Mine-DCT and C-Mine-APR are composed of data mining and SAT solving techniques, they have distinct characteristics because of choosing different data mining methods, decision tree learning and frequent itemset mining.

At first, we proposed C-Mine-DCT (i.e., C-Mine [22]), which adopts decision tree learning as its mining method, to overcome the limitations of previous works. C-Mine-DCT can provide sufficiently scalable solution and also consider a design as a whole without partitioning it. The remarkable feature of C-Mine-DCT is its fast speed due to the proposed promise-driven techniques and tree-level constraint. The tree level constraint can not only avoid tree size explosion but also restrict the search space for finding bigger cubes, which has been shown to offer better performance in terms of delay and area cost in Fig. 3.9. However, this level constraint might cause C-Mine-DCT to miss some critical common case cubes that are **deep** in the tree. Therefore, we proposed another version of C-Mine, C-Mine-APR, which employs a different data mining method, Apriori-based frequent itemset mining, to tackle this issue.

Unlike C-Mine-DCT, C-Mine-APR does not have the level constraint; therefore, it might achieve better performance because of having the chance to find more **fine-grained** common case cubes. As can be expected, runtime will be a trade-off, but fortunately, due to the scalable nature of the algorithm, C-Mine-APR is still competitive compared to the previous works. Furthermore, another feature of C-Mine-APR is its great ability to handle designs that have many common cases with long delays. This is because C-Mine-APR reduces the simulation database beforehand with a *delay threshold*, the average delay in this work, and also applies *minimum support threshold*, *min_sup*, to focus on high-frequency common case cubes during the mining process. All these phenomena can be observed in the experimental results.

3.5.2 Corner case discussion

Since heuristic strategies are used in the cube mining of C-Mine-DCT and C-Mine-APR algorithms, it is worth discussing some possible corner cases that the algorithms might not handle efficiently. Additionally, not all designs are suitable for BTW optimization using common case cubes, and we will also briefly discuss these designs at the end of the subsection.

C-Mine-DCT and C-Mine-APR have different corner cases. For C-Mine-DCT, a possible corner case is when its tree-based mining splits on too many redundant inputs, which can be removed in the cube enlargement process. Please note that input decisions might become redundant after other input decisions are added to the cube candidates. However, within the preset tree level and order constraints, including too many redundant input decisions would compromise the quality of cube candidates or even prevent the miner from finding any ideal candidates. Fortunately, we believe this corner case should rarely happen because our proposed tree-based mining only considers related inputs (i.e., inputs in the fanin cone) and splits first on the inputs that reduce the Gini index most. Nevertheless, even if cube candidates include redundant input decisions, those redundant inputs can still be optimized out in the SAT-based verification.

For C-Mine-APR, its possible corner case happens when the size of the database to mine is too enormous. Although we have applied the domain knowledge proposed in Sec. 3.4.3.1 to reduce the database efficiently, this case could still happen under certain conditions. For example, if the target output f has uneven distribution of values, one of the reduced databases for $f = 0$ or $f = 1$ will not shrink much. Fortunately, we did not see this corner case happen frequently in our benchmarks. However, if this corner case indeed happens, we can tackle this issue by partitioning or sampling [19] the database at the sacrifice of mining quality.

Last, we discuss the general corner cases for C-Mine algorithms. Both algorithms conduct BTW optimization by finding big common case cubes, which can optimize more long delay paths with less area in the shortcut logic. However, not all designs have these kinds of common case cubes, such as cryptographic hardware, which is designed to hide valuable and confidential information; therefore, it should be difficult to mine and retrieve any "informative" data from these cryptographic designs, and we can expect that our C-Mine algorithms would have limited performance on these designs.

3.6 Experimental results

The proposed algorithms C-Mine-DCT and C-Mine-APR are implemented in C/C++ on ABC [37], which is a logic synthesis and verification platform.

MiniSAT [13] is the SAT solver performing SAT solving in this work. We present results on the designs of medium and large area size from ITC'99 and MCNC benchmarks. All experiments were run on a Linux machine with AMD Opteron 6276 16-Core 2.3GHz CPU and 128GB RAM.

The comparison on timing error resilience and energy efficiency improved by C-Mine-DCT, C-Mine-APR, and CCP [18], one of the state-of-the-art works, is shown in Tables 3.1 - 3.3. The correctness probability curves are calculated by the timing simulation data generated from Cadence NCSim v5.7 accompanied with Verilog Program Interface (VPI) to record the timestamps of stabilization of POs. In addition, the relationship between Vdd and power is characterized with HSPICE [18].

To save runtime and area, all methods are only applied to POs whose delays are within $80\% - 100\%$ of the longest path delay according to static timing analysis. Furthermore, to reduce the area overhead caused by appending redundant shortcut logic, CCP applied a lightweight logic optimization command, *fraig_sweep*, of ABC at the end of method. This command preserves the design structure and only merges the functionally equivalent gates. Its mechanisms are to group functionally equivalent gates, replace the gates at high logic levels (i.e., near POs) with the ones at low logic levels (i.e., near primary inputs, PIs) in the same group, and finally sweep out dangling gates in the design. Please note that shortcut logic would be reserved because it is mainly constructed by gates at low logic levels. Therefore, in order to have fair comparison between the methods, C-Mine-DCT and C-Mine-APR also adopted this command to offset the area overhead.

3.6.1 Timing error resilience improvement

Table 3.1 shows the correctness probabilities (P_s) improved by C-Mine-DCT, C-Mine-APR, and CCP over medium benchmarks. Columns 1-4 list the basic information of the benchmarks and the overscaling Vdd used. The correctness probability (P_s), area, and runtime of original design (Ori.), CCP, C-Mine-DCT, and C-Mine-APR are listed in Columns 5-15. The difference between the P_s of original designs and the P_s improved by CCP, C-Mine-DCT, and C-Mine-APR are listed in Columns 16-18. Their area overhead costs are

Table 3.1: Comparison of Improvement on Timing Error Resilience by C-Mine-DCT, C-Mine-APR, and CCP [18] Over Medium Benchmarks.

Circuit	PI	PO	Vdd	Ori.		CCP			C-Mine-DCT			C-Mine-APR			P_s Difference			Area Overhead			Speedup		
				P_s	Area	P_s	Area	Time	P_s	Area	Time	P_s	Area	Time	CCP	DCT	APR	CCP	DCT	APR	CCP	DCT	APR
`alu4`	14	8	0.87	55.8%	2216	62.9%	2268	6.3	70.4%	2091	1.2	74.5%	2125	4.7	7.1%	14.6%	18.7%	2.3%	−5.6%	−4.1%	1	5.3	1.3
`apex2`	39	3	0.85	76.9%	598	78.3%	626	2.2	88.1%	555	0.3	90.9%	540	0.5	1.4%	11.2%	14.0%	4.7%	−7.2%	−9.7%	1	7.3	4.4
`apex4`	9	18	0.89	61.7%	6456	86.8%	6695	18.8	80.0%	6498	1.9	77.4%	5852	3.2	25.1%	18.3%	15.7%	3.7%	0.7%	−9.4%	1	9.9	5.9
`dalu`	75	16	0.90	67.9%	2937	81.7%	2984	14.3	86.4%	2849	14.9	80.0%	2649	6.7	13.8%	18.5%	12.1%	1.6%	−3.0%	−9.8%	1	1.0	2.1
`ex1010`	10	10	0.91	67.7%	7083	83.5%	7391	27.4	80.9%	6728	4.3	79.1%	6726	7.7	15.8%	13.2%	11.4%	4.3%	−5.0%	−5.0%	1	6.4	3.6
`ex5p`	8	63	0.92	65.9%	923	92.4%	954	3.8	75.1%	919	0.3	84.6%	945	0.3	26.5%	9.2%	18.7%	3.4%	−0.4%	2.4%	1	12.7	12.7
`misex2`	14	14	0.86	73.8%	2448	84.3%	2568	12.4	76.5%	2331	1.0	76.5%	2421	1.7	10.5%	2.7%	2.7%	4.9%	−4.8%	−1.1%	1	12.4	7.3
`pdc`	16	40	0.73	41.5%	1667	71.9%	1695	9.3	77.1%	1612	1.9	88.5%	1742	6.6	30.4%	35.6%	47.0%	1.7%	−3.3%	4.5%	1	4.9	1.4
`seq`	41	35	0.73	12.9%	5376	44.2%	5496	19.0	74.9%	4727	2.0	85.0%	4682	6.3	31.3%	62.0%	72.1%	2.2%	−12.1%	−12.9%	1	9.5	3.0
`spla`	16	46	0.78	47.5%	1576	59.9%	1593	10.4	53.5%	1564	0.5	72.8%	1614	2.2	12.4%	6.0%	25.3%	1.1%	−0.8%	2.4%	1	20.8	4.7
Ave.															17.4%	**19.1%**	**23.8%**	3.0%	−4.2%	−4.3%	1	**9.0**	**4.6**

Note: Original Vdd is 1 volt. Area unit: μm^2 Time unit: second

Table 3.2: Comparison of Improvement on Timing Error Resilience by C-Mine-DCT, C-Mine-APR, and CCP [18] Over Large Benchmarks.

Circuit	PI	PO	Vdd	Ori.		CCP			C-Mine-DCT			C-Mine-APR			P_s Difference			Area Overhead			Speedup		
				P_s	Area	P_s	Area	Time	P_s	Area	Time	P_s	Area	Time	CCP	DCT	APR	CCP	DCT	APR	CCP	DCT	APR
`des`	256	245	0.96	81.3%	14814	97.9%	15287	252.9	94.2%	13266	3.4	94.1%	13627	14.7	16.6%	12.9%	12.8%	3.2%	−10.4%	−8.0%	1	74.4	17.2
`b15_opt_C`	484	519	0.79	88.5%	18145	91.8%	18677	17722.0	91.1%	18592	210.5	91.1%	17490	193.7	3.3%	2.6%	2.6%	2.9%	2.5%	−3.6%	1	84.2	91.5
`clma`	416	115	0.63	85.4%	18979	86.8%	19245	286.7	87.2%	19036	15.7	92.0%	19031	106.6	1.4%	1.8%	6.6%	1.4%	0.3%	0.3%	1	18.3	2.7
`clmb`	415	33	0.64	77.4%	45811	77.6%	46615	3225.3	83.2%	46598	79.8	82.3%	45915	261.6	0.2%	5.8%	4.9%	1.8%	1.7%	0.2%	1	40.4	12.3
Ave.															5.4%	**5.8%**	**6.7%**	2.3%	−1.5%	−2.8%	1	**54.3**	**30.9**

Note: Original Vdd is 1 volt. Area unit: μm^2 Time unit: second

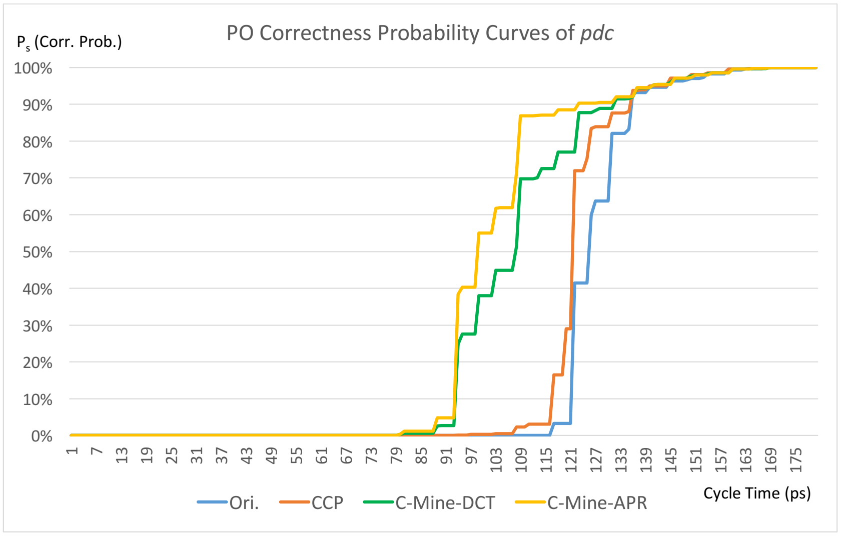


Figure 3.11: Correctness probability curves of `pdc` reshaped by C-Mine-DCT, C-Mine-APR, and CCP [18].

listed in Columns 19-21. The last two columns list the normalized runtime speedup of C-Mine-DCT and C-Mine-APR compared to CCP.

The results in Table 3.1 show that, on average, C-Mine-DCT and C-Mine-APR can improve correctness probabilities (P_s) by 19.1% and 23.8%, respectively, while CCP only achieves 17.4%. For example, the correctness probability curves of `pdc` reshaped by different methods are shown in Fig. 3.11, and we can see that C-Mine-APR and C-Mine-DCT effectively reshape the curves to provide wider plateaus of P_s than CCP. This wide plateau of P_s will result in better energy efficiency, which is presented in the next subsection.

Additionally, thanks to the scalability of data mining and SAT solving, the results in Table 3.1 also show that C-Mine-DCT and C-Mine-APR can achieve average 9x and 4.6x speedup over CCP, respectively. In addition, our proposed SAT-based verification can verify and enlarge cubes at the same time; thus, the shortcut logic can be constructed from big cubes such that area overhead can be controlled. Interestingly, compared to original designs, C-Mine-DCT and C-Mine-APR actually achieved 4.2% and 4.3% area reduction, respectively, while CCP had 3% area overhead. Such area reduction is caused by the fact that merging big redundant cubes into the design could result in more redundant gates to be optimized and removed. Please refer to a demonstrating example in Sec. 3.8.

Table 3.3: Comparison of Throughput and Energy Efficiency of C-Mine-DCT, C-Mine-APR, and CCP [18] Over Medium Benchmarks.

Circuit	N. Power				N. Throughput				N. Energy Cons.			
	Tra.	CCP	DCT	APR	Tra.	CCP	DCT	APR	Tra.	CCP	DCT	APR
`alu4`	1	0.77	0.71	0.73	1	0.70	0.76	0.80	1	1.10	0.94	0.91
`apex2`	1	0.76	0.67	0.65	1	0.83	0.90	0.93	1	0.92	0.74	0.70
`apex4`	1	0.82	0.80	0.72	1	0.89	0.84	0.82	1	0.92	0.95	0.88
`dalu`	1	0.82	0.79	0.73	1	0.85	0.89	0.84	1	0.96	0.88	0.87
`ex1010`	1	0.86	0.79	0.79	1	0.87	0.85	0.83	1	1.00	0.93	0.94
`ex5p`	1	0.87	0.84	0.87	1	0.94	0.80	0.88	1	0.93	1.05	0.99
`misex2`	1	0.78	0.70	0.73	1	0.87	0.81	0.81	1	0.89	0.87	0.90
`pdc`	1	0.54	0.52	0.56	1	0.78	0.82	0.91	1	0.70	0.63	0.61
`seq`	1	0.54	0.47	0.46	1	0.55	0.80	0.88	1	0.98	0.59	0.53
`spla`	1	0.61	0.60	0.62	1	0.68	0.63	0.78	1	0.91	0.96	0.80
Ave.	1	0.74	**0.69**	**0.69**	1	0.80	**0.81**	**0.85**	1	0.93	**0.85**	**0.81**

Note: N.: Normalized. Tra.: Traditional design Cons.: Consumption

Furthermore, to demonstrate the scalability of C-Mine-DCT and C-Mine-APR, we compared them and CCP over large benchmarks in Table 3.2. The results show that C-Mine-DCT and C-Mine-APR can achieve average 54x and 31x speedup over CCP, respectively, with compatible correctness probability improvement. However, for all the methods, the improvement of P_s of some benchmarks seems limited. One possible explanation is that a huge number of PIs in a design might prevent C-Mine-DCT and C-Mine-APR from finding effective common case cubes, while the performance of CCP could be restricted by the scalability issue of BDD-based TCF and the narrow scope of circuit partition.

3.6.2 Energy efficiency

Table 3.3 shows the energy efficiency improved by C-Mine-DCT, C-Mine-APR, and CCP over medium benchmarks. To evaluate the energy saving, we set a penalty $r = 5$ in Eq. (3.1), which has been commonly used in previous studies [18, 40, 46, 47]. Then energy efficiency is calculated by

Eq. (3.2). Table 3.3 lists the data of C-Mine-DCT, C-Mine-APR, and CCP, which are normalized by the data of traditional design methodology without Vdd overscaling. Columns 3-5 list the normalized power. The normalized throughput and energy consumption of CCP, C-Mine-DCT, and C-Mine-APR are listed in Columns 7-9 and Columns 11-13, respectively.

Results in Table 3.3 demonstrate that compared to CCP, C-Mine-DCT and C-Mine-APR not only can achieve similar or better levels of throughput, $TR = 0.81$ and $TR = 0.85$, but also achieve additional 8% $(0.93 - 0.85)$ and 12% $(0.93 - 0.81)$ of energy saving, respectively. Having a complete picture of the design, C-Mine-DCT and C-Mine-APR can generate high-quality common case cubes to reshape the curve of correctness probability more efficiently.

However, C-Mine-APR seems to outperform C-Mine-DCT in the energy efficiency by just 4% $(0.85 - 0.81)$ on average. To further demonstrate the strength of C-Mine-APR, which is the ability to handle designs that have more common cases, we applied five different signal probabilities (SP) from 0.1 to 0.9 at primary inputs to imitate the effects of using different workloads. The signal probability here means the probability of signal to be logic 1. Intuitively, the workloads of $SP = 0.1$ and $SP = 0.9$ will have more common case input patterns than the other settings such as $SP = 0.5$, which is the default setting at PIs for our previous experiments. For example, the input pattern $(111 \cdots 11)$ in the workload of $SP = 0.9$ will be a possible frequent pattern (common case) based on probability. Fig. 3.12 shows the additional energy saving C-Mine-APR can achieve compared to C-Mine-DCT in the different workloads. We can see that C-Mine-APR achieves 13% and 7% more energy saving than C-Mine-DCT in the workloads of $SP = 0.1$ and $SP = 0.9$, respectively, while only 4% is obtained in the workload of $SP = 0.5$.

From the above experiments, we show that both C-Mine-DCT and C-Mine-APR can achieve better performance than CCP in terms of runtime and energy saving. C-Mine-DCT has better runtime performance, while C-Mine-APR has remarkable handling of designs with more common cases.

3.7 Conclusion

The C-Mine-DCT and C-Mine-APR methods were proposed to provide scale-up and comprehensive solutions for improving the timing error resilience of

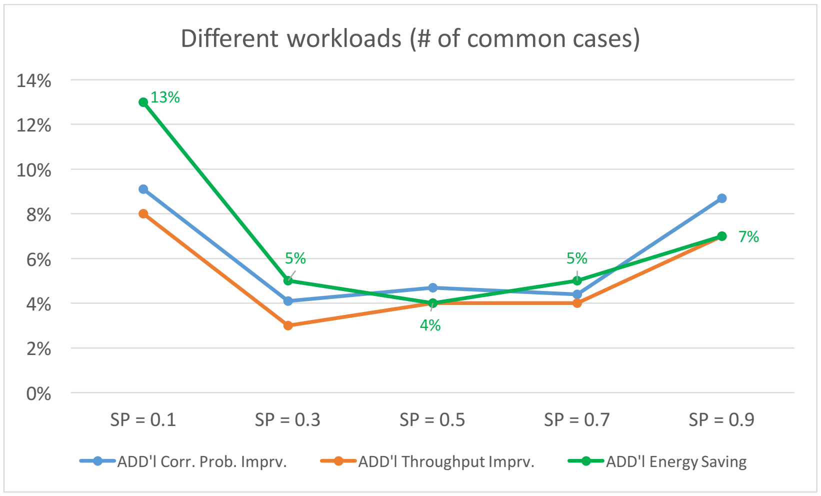


Figure 3.12: Additional energy saving achieved by C-Mine-APR compared to C-Mine-DCT over medium benchmarks with different workloads.

BTW designs, so that greater efficiency in energy can be achieved. We applied data mining and SAT solving to generate common case cubes, which can construct shortcut logic for reshaping the correctness probability curve of a design. Experimental results demonstrated significantly better scalability of C-Mine-DCT and C-Mine-APR with equivalent or better performance and energy results compared to a recent BTW synthesis solution. Furthermore, our new proposed C-Mine-APR can even achieve up to 13% more energy saving than C-Mine-DCT on the designs with more common cases.

3.8 Area reduction through redundant logic and command *fraig_sweep*

In this section, we use an example to demonstrate how appending shortcut logic into the design and applying an ABC [37] command, *fraig_sweep*, can generate additional area reduction. Fig. 3.13 shows a 4-input design, which is composed of three AND gates and one OR gate. The area of each two-input gate is 3 units, and the total area of this design is 12 units.

Assuming that C-Mine finds a common case cube, $((a, b, c) = (1, 1, 1)) \rightarrow (o = 1)$, a new design improved by the corresponding shortcut logic (blue part) is shown in Fig. 3.14, and its area increases from 12 to 19 units. Fortunately, this area overhead can be reduced by applying the command, *fraig_sweep*.

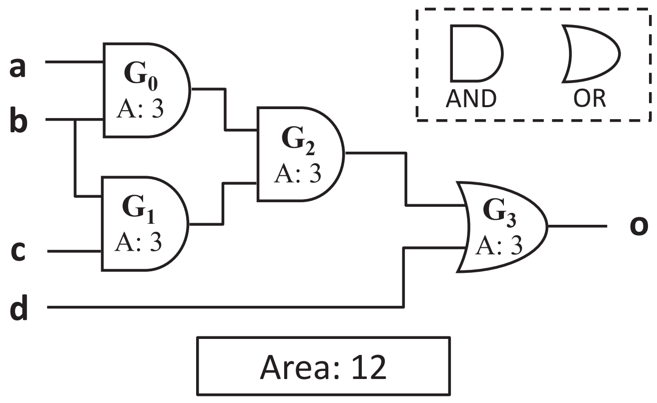


Figure 3.13: A demonstrating example.

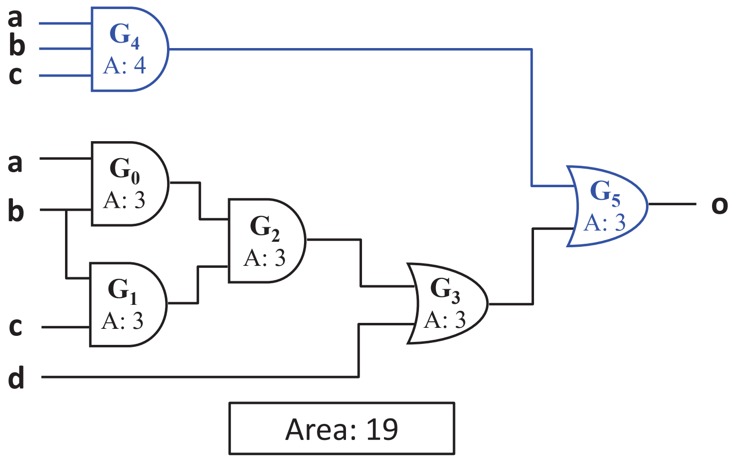


Figure 3.14: Append shortcut logic into the design.

The command, *fraig_sweep*, is a lightweight logic optimization command, which would preserve the design structure by only merging the functionally equivalent gates. Its main steps are summarized below:

1. Group functionally equivalent gates.
2. Replace the gates at high logic levels (i.e., near POs) with the ones at low logic levels (i.e., near PIs) in the same group.
3. Sweep out dangling gates caused by the previous replacement process.

To reduce the area overhead of the improved design in Fig. 3.14, *fraig_sweep* will group gates G_4 and G_2 together because they are functionally equivalent, which can be seen in Fig. 3.15.

Next, since the logic level of gate G_4 is smaller than that of gate G_2, G_2 will be replaced by G_4, and its original fanout will be reconnected to G_4, shown

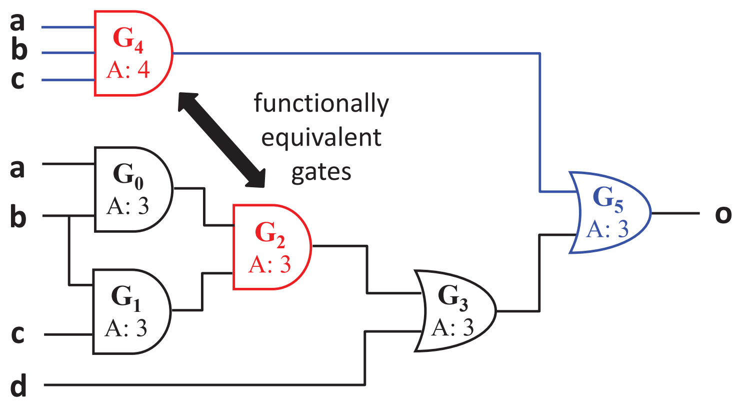


Figure 3.15: Group functionally equivalent gates.

in Fig. 3.16; therefore, the area of design will be reduced to 10 units after sweeping out dangling gates G_0, G_1, and G_2. Finally, the optimized design in Fig. 3.17 can not only preserve the shortcut logic but also have additional 16.7% area reduction, compared to the original design in Fig. 3.13.

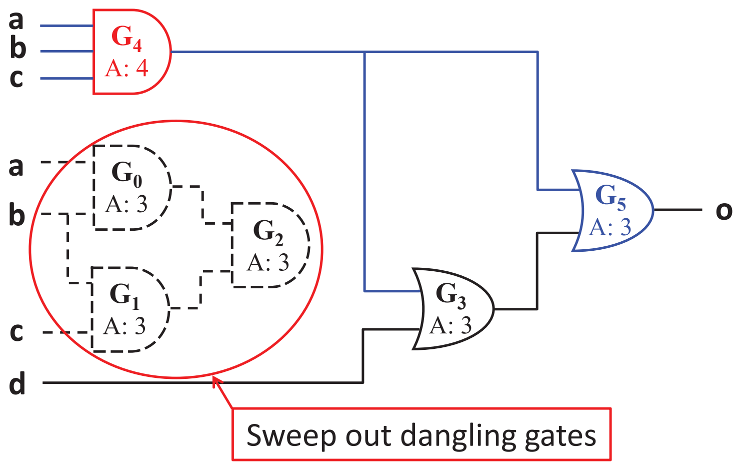


Figure 3.16: Merge equivalent gates and sweep out dangling gates.

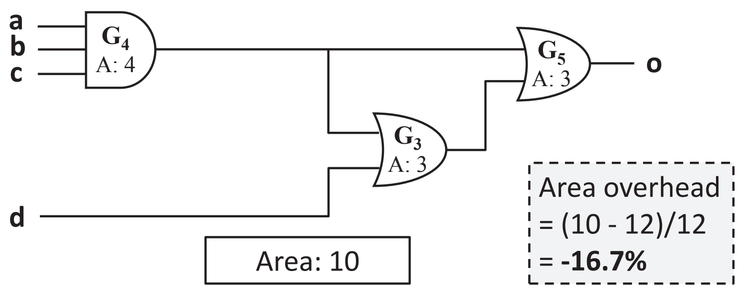


Figure 3.17: Area reduction after appending shortcut logic into the design and applying the command, *fraig_sweep*.

CHAPTER 4

CSL: COORDINATED AND SCALABLE LOGIC SYNTHESIS TECHNIQUES FOR EFFECTIVE NBTI REDUCTION

4.1 Introduction

With technology downscaling to nanometer range, circuit reliability has become a critical challenge for robust system designs [2]. Reliability degradation results from factors such as soft errors, manufacturing variability, temperature effects, and aging. As the trend moves to nanoscale devices, aging, which causes significant loss on circuit performance and lifetime, is becoming relatively dominant in reliability concerns. *Hot carrier injection* (HCI) [54] and *negative bias temperature instability* (NBTI) [55, 56] are two major aging phenomena, which can lead to permanent degradation of transistors, thus hurting the reliability of nanoscale circuits. Among these aging phenomena, NBTI has become particularly prominent and has received considerable attention.

NBTI is an aging phenomenon that increases the threshold voltage (V_{th}) of PMOS transistors over a long period of time, thus slowing logic gates and preventing circuits from meeting the timing requirements. NBTI occurs when PMOS transistors are under negative gate-to-source bias (stress phase: $V_{gs} = -V_{dd}$). During the stress phase, interface traps along the silicon-oxide interface take place due to the dissociation of $Si - H$ bonds. For instance, over a period of ten years, these traps can increase the V_{th} of PMOS in 65 nm technology by up to 50 mV [57], resulting in the delay degradation of circuits. Although some of interface traps can be annealed by relaxing the stress condition ($V_{gs} = 0$), this recovering process is incomplete. Therefore, the NBTI-induced delay degradation crucially depends on the amount of time during which PMOS transistors are under the stress phase. The signal probability SP (the probability of signal to be logic 1) is an effective metric to estimate the NBTI-induced aging degradation of PMOS, which causes the delay of logic gates increase over a period of time [58].

From the logic synthesis perspective, previous works mitigate NBTI effect by taking account of signal probability during synthesis, and these works can be classified into two major groups: considering NBTI either during or after technology mapping (TechMap). During-TechMap: [58] matched the standard cells with the most suitable gate size based on signal probability for reducing NBTI effects. [59] proposed a commercial tool flow to balance the circuit timing with respect to specific NBTI-aware guardbands for improving lifetime. After-TechMap: [60, 61] used logic restructuring and pin reordering techniques with considering signal probabilities to mitigate NBTI-induced delay degradation. [62–64] applied gate sizing techniques with variable V_{th} to decrease NBTI impact and achieve timing closure. However, some of them might have scalability issues because of applying complicated algorithms designed only for a certain synthesis stage. For example, the complexity of restructuring algorithms proposed in [60] might be up to $O(n^3)$, where n is the number of gates in a circuit. Furthermore, their performance is constrained by the results of corresponding technology mapping.

Technology mapping [65, 66] based on tree- or directed acyclic graph-covering has a known issue of suffering from structural bias. In other words, the structure of the resultant mapped netlist depends heavily on the given subject graph, which is a multi-level network of simple gates for representing the Boolean function of the circuit. Although some researches [67–69] targeted mitigating structural bias heuristically, they cannot avoid this issue completely. Therefore, based on this fact, our proposed work is inspired by two ideas: *Can the mapped netlist have better NBTI tolerance if the given subject graph is NBTI-friendly?* and further *How to generate an NBTI-friendly subject graph?*

Unlike previous works, which attacked the NBTI effect at certain later stages of logic synthesis and were limited by complicated algorithms, we propose a **C**oordinated and **S**calable **L**ogic synthesis approach (**CSL**), which integrates techniques at different stages to achieve an effective NBTI reduction. Furthermore, the proposed techniques are designed to deal with large-scale benchmarks. We observe that considering the NBTI effect in the early stages of the design flow can have a better chance to boost the results with less overhead. At the first stage, *subject graph*, we propose an algorithm to restructure the subject graph into NBTI-friendly one iteratively. At the second stage, *technology mapping*, we search for the best matching gates that result in better NBTI tolerance with minimum area overhead from standard cell libraries. This stage also prevents the performance gain at the previous stage from being eliminated. At the last stage, *mapped netlist*, we propose a scalable pin reordering techniques, *smart pin*, to tweak the structure of transistor connections for further reducing NBTI effect with negligible runtime overhead. In sum, NBTI-aware logic restructuring, NBTI-aware technology mapping, and NBTI-aware pin reordering work together to construct a comprehensive and robust NBTI-aware logic synthesis approach.

The contributions of this work are three-fold. To our best knowledge, this is the first NBTI study that (1) considers the NBTI effect at the subject graph stage, (2) deals with large-scale benchmarks even with around a million of gates, and (3) coordinates techniques across several stages to build a comprehensive logic synthesis approach for NBTI reduction. Experimental results show that on average CSL can achieve 6.5% NBTI delay reduction with 2.5% area overhead among the industry-strength benchmarks from ISPD'12 contest [70] without worrying about the size of circuits and the composition of standard cell libraries.

4.2 Preliminaries

This section introduces the background of this work, including subject graph, NBTI modeling, and transistor stacking effect in the PMOS network.

4.2.1 Subject graph

The subject graph, the input to the technology mapping stage, used in this work is in *And-Inverter Graph* (AIG) format, which has been shown an efficient data structure for manipulating large Boolean networks in logic synthesis and formal verification [37, 67, 71]. An AIG is a multi-level Boolean network composed of two-input ANDs and INVs. The data structure of AIG is a *directed acyclic graph*, in which nodes with no incoming edge are primary inputs (PIs) while those with two incoming edges are two-input AND gates. The edges in AIGs represent wires. Inverters are represented by bubbles on the edges. All primitive gates have their corresponding forms in AIGs as

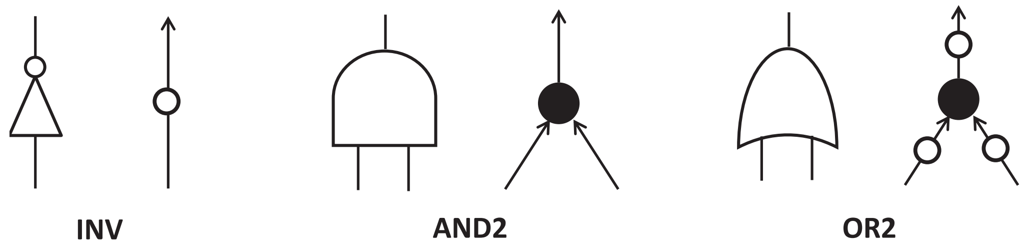


Figure 4.1: The corresponding AIGs of primitive gates: INV, AND2, and OR2.

shown in Fig. 4.1; therefore, arbitrary Boolean networks can be represented by AIGs.

4.2.2 NBTI modeling

This section briefly introduces the NBTI modeling [58, 62, 72] used in this work. The model is used to estimate the delay degradation of each gate in the standard cell library, as a function of the signal probabilities of gate inputs and intrinsic gate delay. An *NBTI-stress factor* of a gate input i, denoted by $\gamma_{nbti}(i)$, represents the probability of the PMOS transistor at input i being stressed. $\gamma_{nbti}(i)$ impacts the *rise delay* of the timing arcs from the gate input i to the gate output. The idea of NBTI modeling is to estimate the corresponding increase in V_{th} for different NBTI-stress factors at the end of a time period. Then, the final V_{th} is plugged into HSpice simulation to get the NBTI-affected rise delay. Finally, a piecewise linear model, similar to that in [62, 72], is developed (within 1% difference to the HSpice simulation data) for adjusting the output rise delay caused by an input i based on $\gamma_{nbti}(i)$ during timing analysis. The details of the modeling can be found in [58, 62]. Fig. 4.2 shows the rise delay characterization of an inverter with input i based on $\gamma_{nbti}(i)$. We can see that the maximum increase of rise delay of the inverter is about 25%.

Next, let us discuss the relationship between signal probability and $\gamma_{nbti}(i)$ of a gate input i. The $\gamma_{nbti}(i)$ can be derived from the signal probability of input i and that of other inputs. Take an example of and-or-inverter (AOI12) gate in Fig. 4.3(a), whose output function is $o = \overline{a + bc}$. Let SP_a, SP_b, and SP_c denote the signal probabilities of inputs a, b, and c, respectively. The γ_{nbti} of b and c, $\gamma_{nbti}(b)$ and $\gamma_{nbti}(c)$, are simply their probabilities of being

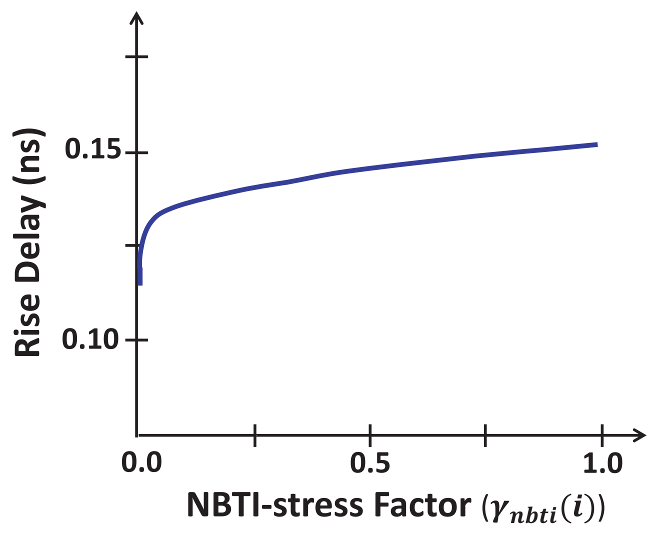


Figure 4.2: Rise delay vs. NBTI-stress factor in an INV [62].

logic 0, while $\gamma_{nbti}(a)$ is the probability that a is equal to logic 0 as well as at least one of b and c is equal to 0, where the PMOS transistor at a is stressed. The γ_{nbti} for inputs, a, b, and c, of the AOI12 gate are shown in Eq. 4.1. Therefore, the NBTI-aware timing model can be built by characterizing all gates in the standard cell library accordingly.

$$\begin{aligned}
\gamma_{nbti}(b) &= (1 - SP_b) \\
\gamma_{nbti}(c) &= (1 - SP_c) \\
\gamma_{nbti}(a) &= (1 - SP_a)(1 - SP_b \cdot SP_c)
\end{aligned} \tag{4.1}$$

4.2.3 PMOS transistor stacking effect

Two transistors connected in series are called stacking. For a CMOS logic gate, if its pull-up (PMOS) network has stacking transistors, we say that these PMOS transistors have the *stacking effect* [58, 60]. The stacking effect causes the NBTI effect of lower PMOS transistors, which are closer to the output signal, to be **milder** than that of upper PMOS transistors, which are closer to the power supply (V_{dd}). This is because the lower PMOS transistors are under stress (connected to V_{dd}) only when their upper PMOS transistors are "on" simultaneously. In other words, a lower PMOS transistor can be protected from NBTI-induced aging by its upper ones. Take a three-input

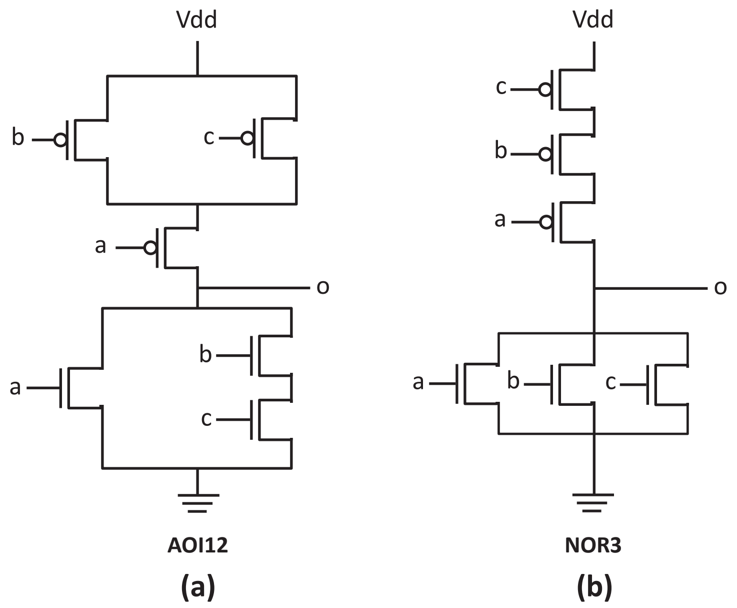


Figure 4.3: The transistor schematics of gates AOI12 and NOR3.

NOR gate in Fig. 4.3(b) as an example. The PMOS transistor of input a can be protected by the PMOS transistors of inputs b and c. By leveraging the stacking effect of pull-up transistors, the previous work [60] proposed a pin reordering method to reduce the NBTI effect. However, it only dealt with NOR gates. In this work, we also propose a new pin reordering method, which considers all kinds of gates that leverage the stacking effect in the library. The details are presented in Sec. 4.3.3.

4.3 NBTI-aware logic synthesis

This section introduces CSL, a coordinated and scalable NBTI-aware logic synthesis approach, which consists of three techniques: *NBTI-aware subject graph restructuring*, *NBTI-aware technology mapping*, and *NBTI-aware smart pin reordering* for the three stages mentioned, respectively. To make the approach scalable, we restrict both time and space complexity of the proposed techniques.

To begin with, we perform parallel simulations to accelerate the calculation of the signal probabilities of gates in the subject graph and mapped netlist. Please note that CSL is a general approach that can accept both purely random simulation and directed simulation; therefore, if a workload is given, designers can apply the same directed simulation patterns through the whole process to optimize the NBTI behavior of designs specifically.

Additionally, the major overhead of this NBTI effect reduction approach is area, which is a common trade-off in logic synthesis: the shorter delay of a circuit, the bigger area of the circuit. Therefore, our objective is not only to reduce the longest NBTI-affected delay of circuits, but also to control the area overhead in an acceptable range. The techniques in the approach for mitigating NBTI effect at different stages are discussed separately in the following subsections.

4.3.1 Subject graph restructuring

Our purpose at this stage is to provide an NBTI-friendly subject graph to CSL's technology mapping stage as well as to control the area overhead of mapped netlist in advance. CSL is the first work that considers and reduces NBTI effect in the subject graph.

4.3.1.1 NBTI-aware static timing analysis for AIG

An AIG, the format of subject graph used in this work, is a multi-level Boolean network of simple nodes to represent the functionality of a circuit. The delay of an AIG is usually measured by performing *static timing analysis (STA)* with a *unit-delay model*, in which both the rise and fall delays of gates are set to one. However, to generate NBTI-friendly AIGs, *NBTI-aware STA for AIG* is needed. Therefore, we propose an *NBTI-aware delay model* based on the NBTI modeling in Sec. 4.2.2 to estimate the NBTI-degraded delay of AIG nodes. Although the NBTI modeling in Sec. 4.2.2 is designed for standard cells, we found that its concepts can be still applied to AIG nodes, so that the NBTI delay paths in AIGs can be identified.

The usage of the NBTI modeling for AIG nodes is similar to that for standard cells. That is, the **rise delays** of simple nodes (AND gates) of AIGs should reflect NBTI degradation by considering inputs' NBTI-stress factors. As aforementioned, the structure of the mapped netlist depends strongly on the subject graph. Therefore, the intuition behind the prediction is that long NBTI delay paths in an AIG might have higher probabilities to be mapped than the long NBTI ones in the final mapped netlist. Although this predictive model might not be completely accurate, it still provides useful guidance to generate NBTI-friendly subject graphs for technology mappers, which can be observed in our experimental results.

4.3.1.2 NBTI balance

With the NBTI-aware STA for AIGs, we propose an AIG restructuring procedure, named *NBTI balance*, to reduce the NBTI effect. The proposed idea is illustrated in Fig. 4.4, and NBTI balance consists of three main steps as follows:

Step. 1: Identify the NBTI-critical POs based on the parameter *threshold*.

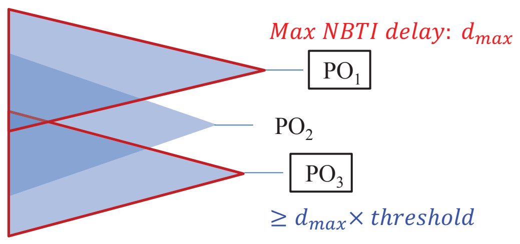
Step. 2: Extract and remove the fanin cones of these NBTI-critical POs.

Step. 3: Add the optimized fanin cones (with better NBTI delays) back for these NBTI-critical POs and minimize the area overhead.

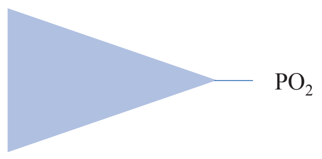
In Step. 1, we identify *NBTI-critical primary outputs (POs)* in an AIG as follows. After finding the maximal NBTI delay of the circuit, d_{max}, the POs whose NBTI delays are larger than or equal to $d_{max} \times threshold$ are considered NBTI-critical POs as shown in triangles with red bold line in Fig. 4.4(a). The parameter *threshold* is a user-defined parameter within an interval $[0, 1]$, which is used to determine how many POs to be re-synthesized in the next step. Optimizing a group of POs together not only saves runtime but also helps reduce area overhead down the road. The parameter setting is shown in the experimental section.

In Step. 2, as shown in Fig. 4.4(b), *NBTI-critical cones*, the fanin cones that belong to the NBTI-critical POs, are extracted and removed from the AIG to form a subcircuit. While extracting NBTI-critical cones, the functionalities of non-NBTI-critical POs are preserved by duplicating the sharing logic between critical and non-critical cones if needed. Please note that most of the duplicate logic can be shared again when the optimized NBTI-critical cones are added back in the next step.

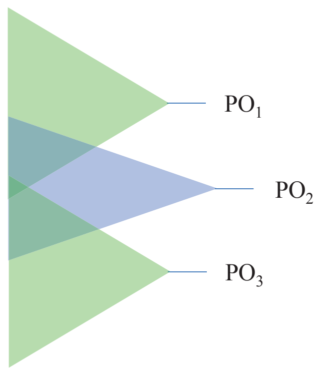
In Step. 3, the subcircuit of NBTI-critical cones is re-synthesized using a *resyn2* script in ABC [37]. This script can optimize both timing and area of the subcircuit. The physical meaning behind this operation is to destroy long NBTI delay paths of critical POs in the AIG. In the end, the optimized cones



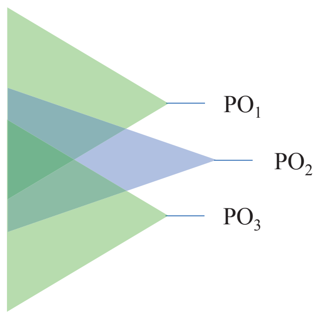
(a) Identify NBTI-critical POs



(b) Extract and remove NBTI-critical cones



(c) Add back optimized NBTI-critical cones



(d) Minimize area overhead using structural hashing

Figure 4.4: The illustration of NBTI Balance procedure.

with respect to the NBTI-critical POs are added back to the AIG to maintain the original functionality as shown in Fig. 4.4(c). Additionally, the *structural hashing* mechanism in ABC, which can increase the number of sharing nodes by merging functionally equivalent ones, is also adopted to control the area overhead during the adding back operation as shown in Fig. 4.4(d).

Next, let us discuss the efficiency of this procedure. The time complexities of Step 1 and Step 2 are linear to the number of nodes in the AIG. For Step 1, the delay is calculated from the PIs to the POs in the breadth-first search (BFS) manner. For Step 2, the logic cones of NBTI-critical POs are also extracted in BFS manner from the POs to the PIs. In Step 3, the *resyn2* script can optimize a circuit of a million of gates in few seconds, and *structural hashing* mechanism is also efficient by using hash tables. Therefore, the proposed NBTI balance procedure is efficient and thus scalable to large-scale benchmarks. This can be seen in the experimental results.

4.3.1.3 Complete NBTI-aware subject graph restructuring

Although NBTI balance generates an NBTI-friendly AIG for the next stage, this procedure cannot be complete without appropriate termination conditions. In other words, we have to determine the number of iterations of NBTI balance for maximizing the reduction of NBTI effect with **acceptable area overhead**. Therefore, we add two termination conditions that complete our restructuring technique. The pseudo code of the proposed ***NBTI-aware subject graph restructuring*** is shown in Algorithm 1, where the two termination conditions, $imprvIter < minImprv$ and $imprvTotal > targetImprv$, are involved. The physical meanings of these two conditions are explained as follows.

- ($imprvIter < minImprv$) : Terminate the iteration when the improvement of NBTI delay of **one iteration** is **less than *minImprv***, which is set to 0.1% in this work. This condition indicates that the NBTI balance has reached its limit and no more significant improvement of NBTI delay would be expected.

- ($imprvTotal > targetImprv$) : Terminate the iteration when the **accumulated** improvement of NBTI delay (compared to the original NBTI delay) is **larger than *targetImprv***. According to our experiments, the average delay degradation under NBTI effect among the benchmark set we used is around 9%; therefore, the $targetImprv$ is set to 10% in this work to avoid over-optimization of NBTI delay. The value of $targetImprv$ should be set according to the observation of NBTI degradation of benchmark set.

In sum, the NBTI balance procedure will be iterated until the improvement is tiny or the target improvement has been achieved.

Area is traded for NBTI delay improvement in this work. However, large area overhead is not acceptable. Therefore, for maintaining the area overhead within a range, the total improvement of NBTI delay is “restricted” to $targetImprv$ as mentioned. As a result, if the total improvement of NBTI delay is larger than $targetImprv$, we undo the last iteration and explore the solution space of last iteration by **increasing threshold parameter**, such that **fewer** NBTI-critical POs are selected for optimization in this iteration. The objective is to achieve an NBTI delay improvement less than but close

Algorithm 1 NBTI-aware subject graph restructuring.

```
function nbtiBalanceAig($oriAig$, $thld$)
    // initialization
    $oriDelay \leftarrow$ nbtiSTA($oriAig$)
    $currAig \leftarrow nbtiBalAig \leftarrow oriAig$
    repeat
        // save the results of previous iteration
        $currAig \leftarrow nbtiBalAig$
        $currDelay \leftarrow$ nbtiSTA($currAig$)
        // detect NBTI critical POs and optimize their delays
        $nbtiBalAig \leftarrow \boxed{\text{nbtiBalance(currAig, thld)}}$
        $newDelay \leftarrow$ nbtiSTA($nbtiBalAig$)
        // examine termination conditions
        $imprvIter \leftarrow$ compImprv($currDelay$, $newDelay$)
        $imprvTotal \leftarrow$ compImprv($oriDelay$, $newDelay$)
    until $\boxed{imprvIter < 0.1\% \textbf{ or } imprvTotal > 10\%}$
    // control $imprvTotal$ not exceed 10% by exploring the
        solution space of this iteration with different $thld$'s
    if $imprvTotal > 10\%$ then
        $nbtiBalAig \leftarrow$ explDiffThld($oriDelay$, $currAig$, $thld$)
    end if
    return $nbtiBalAig$
end function
```

to *targetImprv*. This idea about solution space exploration with various threshold parameters is detailed in the pseudo code of Algorithm 2.

4.3.2 Technology mapping

Given an NBTI-friendly subject graph (AIG), we propose an NBTI-aware technology mapping technique, which not only preserves the NBTI reduction gains from the previous stage, but also alleviates NBTI impact from the aspect of technology mapping. Additionally, the mapping selection step in it also considers the succeeding stage, smart pin reordering, ahead for maximally reducing the NBTI effect.

Algorithm 2 Explore the solution space of a specific iteration with various threshold parameters.

```
1:  function EXPLDIFFTHLD($oriDelay$, $currAig$, $startThld$)
2:      $thld \leftarrow startThld$
3:      $nbtiBalAig \leftarrow currAig$
4:      repeat
5:          // use a bigger $thld$ to balance the tradeoff
6:          $thld \leftarrow thld + 0.01$
7:          // have explored all possible $thld$'s, discard
                the results of this iteration
8:          if $thld > 1$ then
9:              return $currAig$
10:         end if
11:         $nbtiBalAig \leftarrow$ NBTIBALANCE($currAig$, $thld$)
12:         $newDelay \leftarrow$ NBTISTA($nbtiBalAig$)
13:         $imprvTotal \leftarrow$ COMPIMPRV($oriDelay$, $newDelay$)
14:     until $imprvTotal \leq 10\%$
15:     return $nbtiBalAig$
16: end function
```

The technology mapper adopted in the work is based on a cut-based Boolean matching method [37,67]. The mapper consists of five major steps: **(1)** *Compute k-feasible cuts.* A feasible cut of a node n in the AIG is a set of nodes C_n in the fanin cone of n such that any path from a PI to n passes through C_n. A k-feasible cut means the size of the cut must be less than or equal to k. A k-feasible cut is *redundant* if there exits a node in the cut whose value can be completely determined by the other nodes in it. For example, in Fig. 4.5, the set $\{a, b, c\}$ is a 3-feasible cut of node n, while the set $\{a, b, c, e\}$ is a redundant 4-feasible cut of node n because the value of node e can be determined by nodes b and c in the same cut. The redundant k-feasible cuts will not be considered during cut enumeration. The parameter k is heuristically set to 5 for the tradeoff of efficiency and effectiveness in the work. **(2)** *Compute the truth tables of cuts.* The local function of a node in terms of its cut is computed symbolically. With considering 5-feasible cut, the truth table (function) of each cut can be stored in a 32-bit integer, thus accelerating the symbolic function computation as well as the matching

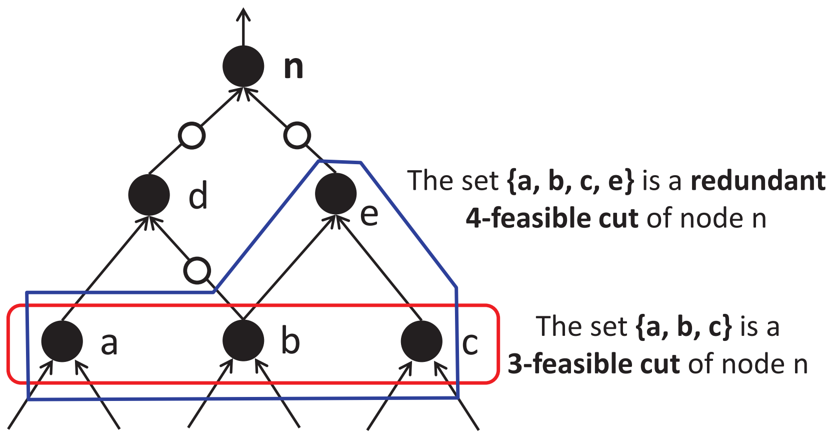

n
d
e
a
b
c
The set **{a, b, c, e}** is a **redundant 4-feasible cut** of node n
The set **{a, b, c}** is a **3-feasible cut** of node n

Figure 4.5: The redundant and irredundant feasible cuts of node n.

process in the next step. **(3)** *Perform Boolean matching.* Each node in the AIG might have more than one 5-feasible cut. For each cut, a matching gate, if existing, is selected from the library. **(4)** *Compute the best arrival time of each node.* The best arrival time of each node is computed and selected from all its matchings of cuts in a topological order. **(5)** *Select the best cover.* In a reverse topological order, which is from the POs to the PIs, the best matching gates are chosen using a delay-oriented method with the area constraint until all nodes in the AIG are covered.

To reduce NBTI effect at this stage, the arrival time computation of matches should reflect NBTI-induced degradation accordingly. Given a matched gate and a cut, γ_{nbti} for the inputs are computed from their signal probabilities based on the NBTI modeling mentioned in Sec. 4.2.2. Since only the rise delay would be affected by NBTI, two out of four timing arcs, **input rise to output rise** and **input fall to output rise**, are adjusted with considering γ_{nbti}.

Furthermore, to break a tie during the best cover selection, NOR or Or-And-Inverter (OAI) gates will be chosen with **high priorities** to benefit our next stage. The beneficial effects will be discussed in the next subsection. Our technology mapping stage produces an NBTI-tolerant mapped netlist to the next stage.

4.3.3 Smart pin reordering

Given an NBTI-tolerant mapped netlist, we propose a scalable pin reordering technique, named *smart pin*, to tweak the structure of the netlists for more NBTI effect reduction. Based on the discussion of stacking effect in Sec. 4.2.3, it is intuitive to assign inputs to the PMOS transistor stack of a gate in descending order of input signal probabilities from the top to the bottom of pull-up network. Although this assigning order can result in the smallest NBTI degradation for the gate, it might not lead to the smallest signal arrival time at the gate's output. This is because the inputs with higher signal probabilities (smaller probabilities of being logic 0) might have larger signal arrival time. Therefore, to minimize the NBTI-degraded delay of the overall circuit, both signal probabilities and signal arrival time of inputs should be considered simultaneously for pin reordering technique. The previous work [60] considered the stacking effect in NOR gates only, and only considered input arrival time to search a pin ordering that leads to the best timing **exhaustively**. Therefore, it might not be applicable for richer standard cell libraries, which have other gates with PMOS transistor stacking or gates with many pins.

The two major characteristics of the proposed smart pin reordering technique are applicability and scalability. For applicability, in addition to leveraging stacking effect of NOR gates, we also explored and leveraged the stacking effect in OAI gates, as marked in rectangles in Fig. 4.6, for NBTI reduction. For scalability, the smart pin reordering technique heuristically determines pin assignment by slack, which is defined as the difference between the required time and arrival time of the gate's output signal for the timing path, rather than exhaustively searching like the previous work.

The scalable heuristic is described as follows. First, given the required time of the POs, the slack of each gate in the mapped netlist is computed using an NBTI-aware STA with considering signal probabilities of the gate inputs. Next, pins of PMOS stack(s) in NOR and OAI gates are reassigned based on the slack information in a topological order from the PIs to the POs. Specifically, the input with the smallest slack is assigned to the lowest PMOS transistor, and the other inputs are dealt with in the same way. The physical meaning behind this strategy is that a timing path through an input with a small slack is tight and more **fragile** to NBTI-induced delay degradation;

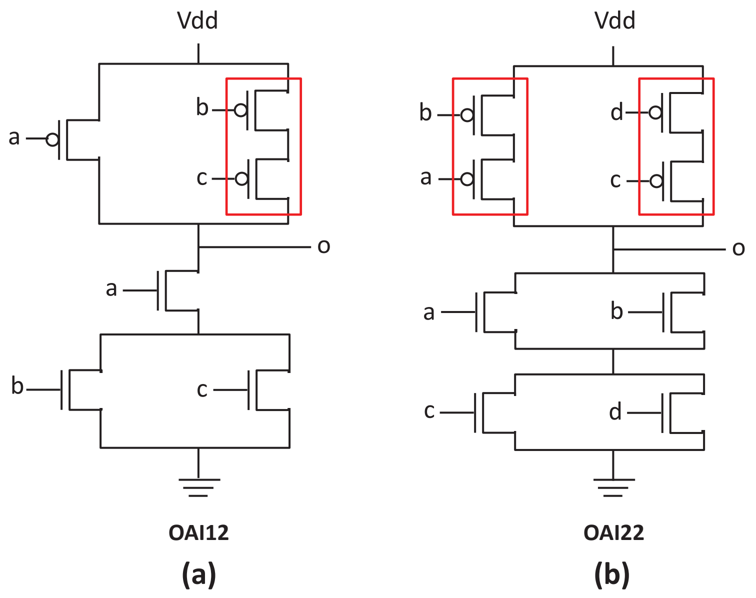


Figure 4.6: The PMOS transistor stacking effect exists in the schematics of gates OAI12 and OAI22.

therefore, assigning this input to the lower position of PMOS stack can protect the tight timing path against NBTI effect in the future.

In addition to the applicability and scalability of the smart pin technique, in this coordinated approach, CSL's technology mapping stage chooses NOR and OAI gates with higher priorities when a selection ends in a tie, thus inducing more flexibility and increasing the effectiveness of this smart pin reordering technique.

4.4 Experimental results

The CSL for NBTI reduction was implemented in C/C++ in ABC [37], which is a state-of-the-art logic synthesis and verification platform. The benchmarks are industry-strength designs from ISPD'12 contest [70] and benchmark statistics is listed in Table 4.1. The standard cell library used in the experiments is a subset of library *mcnc.genlib* [73], which contains INV, NAND2, NAND3, NAND4, NOR2, NOR3, NOR4, AOI12, AOI22, OAI12, and OAI22. The technology process used is 32nm Predictive Technology Model (PTM) [74], and NBTI effect is considered at the end of a 5-year

Table 4.1: The Statistics of Benchmarks from ISPD'12 Contest [70]

Circuit	PI #	PO #	Comb. cell #	Seq. cell #	Total cell #
`pci_bridge32`	160	201	29844	3359	33203
`DMA`	683	276	23109	2192	25301
`des_perf`	234	140	102427	8802	111229
`vga_lcd`	85	99	147812	17079	164891
`b19`	22	25	212674	6594	219268
`leon3mp`	254	79	540352	108839	649191
`netcard`	1836	10	860949	97831	958780

period. All experiments were run on a Linux machine with AMD Opteron 6276 16-Core 2.3GHz CPU and 128GB RAM.

The previous work [60], which is the most related to CSL, combines logic restructuring and pin reordering based on functional symmetry detection and transistor stacking effect to mitigate NBTI-induced delay degradation. Given a mapped netlist, it identifies functional symmetries using the concept of *supergates (SG)* [75], where a supergate is a group of connected gates that logically equals a big AND/OR gate. Having these SGs detected, [60] can swap wires inside supergates to improve NBTI delay without altering the functionality of netlist. The NBTI delay of the netlist will be improved iteratively until no further improvement is obtained. To extract SGs in a netlist, [60] first treats all POs as SG roots and assigns non-controlling values to them. Then backward implication is applied to each gate in a reverse topological order to determine the values of all inputs until **(1)** no more implication can be made or **(2)** the current gate is not fanout-free. The gates where backward implication stops are treated as new SG roots. The same backward implication is applied to those new SG roots with non-controlling values recursively until no more SGs can be detected. To compare [60] and CSL, we reimplemented [60] on the same platform ABC, and the results were verified by the equivalent checking commands of ABC to guarantee the functional correctness of logic restructuring and pin reordering.

Table 4.2 shows the comparison of [60] and CSL on NBTI reduction over industry-strength benchmarks. Since CSL includes stages that optimize and remap the original benchmarks, to fairly compare the performance of [60] and CSL, the original benchmarks were first optimized and remapped by

Table 4.2: The Comparison of [60] and CSL on NBTI Reduction Over Industry-strength Benchmarks.

Circuit	Non-NBTI Optimization				[60]		CSL			[60] Performance	CSL Performance	
	Area	Nominal Delay	NBTI Delay	Aging (%)	NBTI Delay	Time (s)	Area	NBTI Delay	Time (s)	NBTI Delay Imprv. (%)	Area Overhead (%)	NBTI Delay Imprv. (%)
`pci_bridge32`	44000	20.5	22.26	8.59%	22.23	16.46	44195	19.61	3.35	0.13%	0.44%	11.90%
`DMA`	45490	18.2	19.87	9.18%	19.87	20.54	48528	18.73	4.13	0.00%	6.68%	5.74%
`des_perf`	169322	15.9	17.45	9.75%	17.39	219.51	176750	17.01	21.61	0.40%	4.39%	2.52%
`vga_lcd`	266398	15.8	17.23	9.05%	–	OOT	284537	15.64	29.51	–	6.81%	9.23%
`b19`	441778	61.3	66.60	8.65%	66.57	352.14	442631	63.38	103.14	0.05%	0.19%	4.83%
`leon3mp`	1250676	39.7	43.57	9.75%	–	OOT	1249915	40.17	2796.92	–	−0.06%	7.80%
`netcard`	1532384	29.6	32.44	9.59%	–	OOT	1517186	31.24	1503.62	–	−0.99%	3.70%
Ave.				9.22%						0.15%	2.49%	**6.53**%

[1] Area size is normalized by INV's size and reported by ABC [37].
[2] OOT denotes "Out Of Time" and the time limit is 5 hours.

ABC to eliminate redundant logic. These new optimized benchmarks are our baseline results. The *threshold* parameter of CSL is set to 0.97 empirically. Because of lack of real workloads, the signal probabilities are calculated using purely random simulation for both [60] and CSL. The NBTI delay is obtained by an NBTI-aware STA using the NBTI modeling in Sec. 4.2.2. The aging degradation is the percentage difference between the nominal delay and NBTI delay. Columns 1-5 list the basic information and NBTI-induced degradation of the baseline benchmarks. Columns 6-7 and Columns 8-10 list the results of [60] and CSL, respectively. The runtime is in seconds. Compared to the baseline, the NBTI delay improvements of both methods are listed in Columns 11 and 13, and the area overhead of CSL is listed in Column 12. There is no area overhead in [60], since it only swaps wires and does not introduce additional gates.

4.4.1 The performance of [60]

As shown in Table 4.2, we observed that the performance of [60] on large-scale benchmarks is limited based on our reimplementation. After investigating the algorithm and benchmarks, the possible explanations for this phenomenon follow.

First, as we mentioned, the performance of after-TechMap works might be constrained by the results of technology mapping. Therefore, after benchmarks are optimized and remapped (e.g., by ABC in this work) for delay, there might be little space left for further delay or NBTI delay improvement, thus affecting the performance of these works. However, this effect is more dramatic in [60] because it only performs wire swapping without inducing any additional gates.

Second, the size of supergates does matter to the performance due to the complexity of algorithm. The statistical information of supergates detected by [60] for each benchmark is listed in Table 4.3. As shown in Table 4.2, [60] cannot finish `vga_lcd`, `leon3mp`, and `netcard` within the specified time limit because these benchmarks have big SGs whose sizes are 219, 225, and 877, respectively. In Table 4.3, we can know the number of wires in a SG is essentially proportional to its size. However, the number of valid combinations of wire swappings is indeed exponential to the number of wires. Therefore, when the size of a SG is enormous, [60] spends much time in exploring many

Table 4.3: The Statistics of Supergates Detected by [60] in Benchmarks

Circuit	Supergate #	Max Size of SG	Wire # in Max-size SG
`pci_bridge32`	2021	11	37
`DMA`	1269	12	41
`des_perf`	6915	7	13
`vga_lcd`	1167	219	730
`b19`	15313	21	50
`leon3mp`	12545	225	735
`netcard`	59196	877	2669

[1] Supergate size is measured by the number of primitive gates in it.
[2] The number of wires in a SG includes inputs to SG and its internal wires.

possible combinations of wire swappings to find the one that reduces the NBTI delay of the SG most. Although [60] has proposed some heuristics to prune the exploring space, the number of wire swappings to try is still intractable in big SGs. Furthermore, large-scale benchmarks usually have more big SGs than small-scale ones; therefore, we could infer that large-scale benchmarks are not friendly to [60].

Third, the NBTI delay reduction is limited because not many SGs are located on the NBTI longest path of circuits. Please note that the standard cell library used in the work is richer than that used in [60]. Our library has two additional primitive gate types, OAI and AOI, which do not have non-controlling values and cannot perform backward implication. Additionally, large-scale benchmarks usually have many non-fanout-free gates because of logic sharing. Therefore, for `pci_bridge32`, `DMA`, `des_perf` and `b19`, [60] cannot find enough SGs on the longest path to improve NBTI delay due to keeping restarting SG expansion on new SG roots, while encountering OAI, AOI, and non-fanout-free gates. This scattered SG structure compromises the performance of [60] drastically. Take `DMA` as an example, we found that there is only one SG of size 2 on the NBTI longest path. Worse, this SG, consisting of only a NAND2 and an INV, has no valid wire swappings that can improve the NBTI delay. Therefore, no NBTI improvement can be made in `DMA`.

Table 4.4: The Effectiveness of NBTI-aware Subject Graph Restructuring of CSL.

Circuit	CSL (AIG restr.)		CSL (AIG restr.) Performance	
	Area	NBTI Delay	Area Overhead (%)	NBTI Delay Imprv. (%)
`pci_bridge32`	44166	19.86	0.38%	10.78%
`DMA`	48494	19.16	6.60%	3.57%
`des_perf`	180731	17.04	6.74%	2.35%
`vga_lcd`	284397	15.64	6.76%	9.23%
`b19`	442878	63.81	0.25%	4.19%
`leon3mp`	1254236	40.89	0.28%	6.15%
`netcard`	1617043	31.85	5.52%	1.82%
Ave.			3.79%	**5.44**%

* Area size is normalized by INV's size and reported by ABC [37].

4.4.2 The performance of CSL

According to Table 4.2, CSL can achieve 6.53% NBTI delay improvement with merely 2.49% area overhead on average. Thanks to NBTI-aware subject graph restructuring technique, NBTI-friendly graphs can be generated by considering NBTI effect and controlling area overhead as early as possible. Given the NBTI-friendly graphs, our technology mapping technique can have more flexibility in choosing the matching gates that can reduce NBTI delay most as well as not sacrificing area too much. For example, CSL can mitigate the NBTI effect of `b19` and `pci_bridge32` with insignificant area overhead (< 1 %). This contribution is even more significant for large benchmarks like `netcard` and `leon3mp`, for which CSL can improve NBTI delay without any area overhead or even a little area reduction. Interestingly, for `pci_bridge32` and `vga_lcd`, the improved NBTI delay is slightly better than the original nominal delay, possibly due to the restructuring at the subject graph stage. Additionally, the runtimes of benchmarks also demonstrate the scalability of CSL. All the benchmarks can finish in several seconds to an hour. Please note that the major part of runtime is spent on technology mapping, which is also a necessary effort for non-NBTI-aware approaches. We could find that CSL has a great ability to handle large-scale benchmarks as well as having no constraints on the libraries used.

To demonstrate the benefit of considering NBTI effect at an earlier stage, we conducted another experiment that only applies NBTI-aware subject graph restructuring technique in CSL without using NBTI-aware technology mapping and smart pin reordering techniques. Results in Table 4.4 demonstrate that NBTI-aware subject graph restructuring alone can achieve 5.44% NBTI delay improvement on average, which is the major contribution of the complete CSL. These results support the idea of early consideration of NBTI effect. Although the techniques of the rest stages, NBTI-aware technology mapping and NBTI-aware smart pin reordering, seemingly provide 1.09% (6.53% − 5.44% = 1.09%) improvement in NBTI delay, they help reduce area overhead from 3.79% to 2.49%. Therefore, the coordination among the techniques at different stages can result in the best performance of CSL.

4.5 Conclusion

This work proposes a coordinated and scalable logic synthesis approach, CSL, to address NBTI effect, which is a major cause of aging and reliability issues in nanometer IC designs. It consists of NBTI-aware subject graph restructuring, technology mapping, and smart pin reordering techniques at different stages to form a coordinated NBTI-aware logic synthesis approach. Experimental results demonstrated the capability and scalability of CSL to mitigate the NBTI effect with acceptable area overhead and runtime.

CHAPTER 5

COST-EFFECTIVE ERROR DETECTION THROUGH MERSENNE MODULO SHADOW DATAPATHS

5.1 Introduction

As technology scales into deep-submicron nodes and more transistors are packed into a chip, reliability problems ranging from transistor wear-out to soft errors to electromigration are getting worse. For each reliability problem, there are logic and physical-design level techniques proposed to address the problem. For example, flip-flop hardening [76, 77] is proposed to address soft errors in flip-flops and Razor logic [78, 79] is proposed to detect timing errors (such as those caused by transistor wear-out). But these low-level solutions are limited in that they only protect some of the hardware components (e.g. flip-flop hardening only protects flip-flops) and/or only apply to certain classes of errors (e.g. flip-flop hardening only protects against soft errors).

Each of these solutions adds some complexity to the design process and impacts quality of results (QoR). Building a comprehensive solution with these low-level techniques would involve combining multiple solutions which can further compound the design complexity and QoR cost. Clearly there is a need for a *holistic* solution to reliability problems. Ideally, such a solution would protect *all* of the hardware against essentially *all* possible reliability problems.

With the end of Dennard scaling, technology scaling has also resulted in a proliferation of hardware accelerators to meet high image and video processing demands despite limited scaling in microprocessor performance and efficiency. While microprocessor area is usually control-logic dominated, hardware accelerators are *datapath-dominated* with the majority of the area dedicated to computation in application-specific, computation-intensive, and complex datapaths.

For arithmetic-oriented datapaths (i.e., involving the elementary addition, subtraction, and multiplication operations) modulo shadow datapaths [80] are a useful technique for protecting arithmetic datapaths using lightweight (e.g. 2-bit) shadow logic that redundantly performs the same computation as the main datapath, but with modulo residues. Modulo shadow datapaths have been shown to detect stuck-at faults, timing errors, and soft errors [80].

As with any reliability improvement technique, the cost of modulo shadow datapaths limits its applicability. Furthermore, increasing the width of the shadow datapath is also desirable, as it reduces the chance of false error detection negatives (i.e., the output was wrong, but it happened to have the correct residue value) and improves coverage, but increasing the shadow datapath width also increases costs.

Existing approaches to modulo logic rely on logic synthesis tools to produce optimal QoR. Furthermore, some approaches [81, 82] only use the FPGA platform for QoR evaluation, which does not accurately model the effects of ASIC technology mapping.

In this chapter, which is an extension of our conference publication [25], we develop our own gate-level design techniques to create minimum area and minimum delay modulo functional units for any given modulo residue bitwidth, dramatically improving QoR over the state-of-the-art approach. To demonstrate the applicability of our approach for reliability, we use these building blocks to create self-checking multiply-accumulate and linear algebra primitive datapaths. We observe cost-effective results, with 32-bit datapath overheads of 6–10% for a 3–61× improvement in reliability, and overheads of 15–20% for a 121–2477× improvement in reliability. Additionally, we propose an area overhead estimation method for self-checking arithmetic component implementation.

Our contribution highlights are:

1. A modulo reduction algorithm which generates architectures consisting entirely of full-adder standard cells that efficiently reduce large numbers of bits;

2. minimum-area modulo adder and subtractor architectures;

3. an array-based modulo multiplier design;

4. QoR comparisons showing the cost-effectiveness of our functional units compared to the previous state-of-the-art in area and delay;
5. low-cost designs for self-checking multiply accumulator and linear algebra primitives; and
6. an area overhead estimation method.

The rest of this chapter is organized as follows: Section 5.2 discusses related work, Section 5.3 provides some background knowledge used in subsequent sections, Section 5.4 describes the gate-level architecture of our modulo functional units, Section 5.5 provides QoR and overhead comparisons for individual functional units and for our example self-checking datapaths, and we conclude in Section 5.6.

5.2 Related work

5.2.1 Fault detection

As mentioned in Section 5.1, there are many alternative approaches to error detection. The classical approach is modular redundancy [83, 84], duplicating the entire hardware module and comparing the outputs for discrepancies. Such an approach has at least a $2\times$ area cost, which can be prohibitively expensive and negates the benefits of Moore's law scaling.

Razor logic [78, 79], an approach involving creating a shadow latch for each flip-flop in a design, has been proposed to address timing errors, but also imposes timing constraints on a design. Flip-flop hardening techniques [76, 77] have been proposed to address soft errors in flip-flops, but such techniques do not protect combinational logic. Logic parity [85] is another technique for protecting flip-flops by adding a parity flip-flop for flip-flop clusters with parity prediction and checking logic. Such parity techniques are practically limited to protecting only the flip-flops in a design using the aforementioned clustering technique [85] due to the high overheads (e.g., around 30% area overhead for a 32-bit adder [86]) associated with parity prediction across functional units.

While Razor logic, flip-flop hardening, and parity are limited to certain kinds of faults and certain parts of a datapath, modulo shadow datapaths have none of these limitations. Modulo shadow datapaths [80] holistically protect the entire datapath from input to output, including all of the combinational logic. Modulo shadow datapaths is a general purpose error detection technique with essentially no assumptions about fault behavior.[1]

Finally, Algorithm-Based Fault Tolerance (ABFT) [87, 88] is an algorithm-level technique for protecting linear vector and matrix computations by predicting and checking the sums of groups of output elements. ABFT can involve expensive extra memory accesses for checksum computation and storage and may require the duplication of vectors for certain computations.

5.2.2 Modular arithmetic

For small modulo bases, a lookup table based approach has been used for basic functional units [80,89] with explicit don't cares inserted to provide hints to the logic synthesis engine for inputs combinations that should never occur. A reducer is built with a tree of such lookup-table based modulo adders [89]. Such an approach is impractical for larger bases due to exponential scaling.

Piestrak et al. propose a design for a modulo-3 reducer consisting of full-adder (FA) cells and interleaved inverters [89, 90] which exploits the fact that for a given bit $b \in \{0, 1\}$, $2b = -b = 3 - b = 2 + (1 - b) \pmod 3$. In other words, bits of weight 2 can be inverted and treated as a bit of weight 1 with a constant offset (which can be lumped together at the end) so that all bits have the same weight of 1 and can be passed through stages of FAs. While this design may appear superficially similar to our reducer design in Fig. 5.1(b), our design uses a more general strategy inspired by Wallace trees that does not require separate inverters. Furthermore, our strategy generalizes to any Mersenne base while their design trick is limited to modulo-3 arithmetic.

Wei and Shimizu [91] proposed a signed-digit architecture which can handle any modulo base of the form $2^n \pm 1$. The authors measured the area cost of their approach with rough gate count numbers and do not provide ASIC area with real units. Area cost would be a significant problem for the hardware

[1]Modulo shadow datapaths are limited in their applicability to non-arithmetic logic (e.g. bitwise operations). Campbell et al. demonstrated in [80] that these limitations can be worked around by considering non-arithmetic components to be modulo shadow datapath barriers and insuring that inputs to those components are checked directly or indirectly, while protecting the non-arithmetic components themselves with another complementary technique.

cost-effectiveness of their approach considering the complexity of their signed-digit adders which must use 2 bits for each binary digit instead of the single bits used in our approach.

For cryptography applications, there are also a number of hardware accelerator designs for accelerating modulo exponentiation of large (e.g. 256-bit) numbers, which is performed with a series of modulo multiplies [92]. These designs use application-specific algorithms (e.g. Montgomery multiplication [93]) that make them very specialized for big-integer modulo exponentiation, and thus unsuitable for reliability applications.

5.3 Background

5.3.1 Modulo arithmetic

Modulo-b arithmetic (also called residue arithmetic) is arithmetic defined in a finite field with b possible values, where each possible value corresponds to a remainder when an integer is divided by b, which we refer to as the *modulo base* (using Euclidean division so that remainders are always positive). Addition, subtraction, and multiplication are defined with "wraparound" arithmetic where the result is immediately divided by b and the remainder taken as the result.

For example, in modulo-3 space the possible values are $\{0, 1, 2\}$ and $2+2 = 1$ since in integer space $(2 + 2) \bmod 3 = 1$ where $a \bmod b$ is the remainder after dividing a by b.

Since equivalent lightweight computations can be performed in modulo-b space as in integer space, modulo arithmetic can be used as a way to independently check integer computation. This works because we have defined a homomorphism from integer arithmetic to modulo arithmetic. In other words, given integers $\{x, y, z\}$ and corresponding modulo variables $\{x', y', z'\} = \{x, y, z\} \bmod b$ we observe the following properties:

$$x + y = z \implies x' + y' = z' \pmod{b} \tag{5.1}$$

$$x - y = z \implies x' - y' = z' \pmod{b} \tag{5.2}$$

$$xy = z \implies x'y' = z' \pmod{b} \tag{5.3}$$

where (mod b) next to an equation indicates that the arithmetic is performed in modulo-b space. Thus for Eqs. (5.1), (5.2), and (5.3), z' can be independently computed two ways: by mapping z to modulo space or by mapping x' and y' to modulo space and performing the "shadow computation" in each equation.

Note that this "shadow computation" property holds for arbitrarily complex integer arithmetic involving addition, subtraction, and multiplication. For example, $x^2 - 4xy + 2y^2 = z \implies x'^2 - 4x'y' + 2y'^2 = z' \pmod{b}$. Exploiting the ability of homomorphisms such as this integer to modulo mapping to scale to arbitrarily complex expressions is the key to implementing cost-effective error detection.

5.3.2 Mersenne numbers

For positive integers n we define the Mersenne numbers by $M(n) = 2^n - 1$. The use of $M(n)$ as a modulo base has the following useful property for $n \geq 2$:

$$2^n = 1 \pmod{M(n)} \tag{5.4}$$

5.3.3 Binary representations

Our encodings for modulo residues are based on the standard binary representation for integers, where bits have weights with successive powers of two. In other words the integer value of a particular sequence $\{b_{n-1}, b_{n-2}, ..., b_0\}$ of bits is defined as:

$$v = \sum_{i=0}^{n-1} 2^i b_i \tag{5.5}$$

A standard Mersenne number residue r with base $M(n)$ will be in the range $0 \leq r \leq M(n) - 1 = 2^n - 2$. Thus n bits are sufficient to encode a residue with base $M(n)$, and the most significant bit (MSB), b_{n-1}, will have weight 2^{n-1}. If a carry bit is generated from adding two MSB bits, it will have weight 2^n which is equivalent to 1 by application of Eq. (5.4).

5.3.4 Normalization

There is one special encoding possible for an $M(n)$ residue encoded with n bits, the value where all bits $b_i = 1$. This encoding has the integer value $2^n - 1 = M(n)$ by Eq. (5.5). Since this residue is the same as the modulo base, it is equivalent to zero. We call this special encoding for zero the *denormalized* encoding of zero, write it as -0, and call encodings that allow it *non-normalized* encodings.

5.3.5 Fault model

Our fault model considers all gate outputs and assumes a uniform probability of a bit flip occurring at a given output at a given cycle. This fault model is referred to as a *single event transient* (SET) fault model which is a generalization of flip-flop bit-flips to all gates. Formally speaking, we let X be the set of all gate output bit flip events in a given cycle and say $\forall(x \in X)\ P(x) = p$, for some uniform bit-flip probability p. By assuming that $p \ll 1$ (i.e. that a fault is a rare event) we can neglect the probability of two faults occurring at the same cycle (e.g. $P(x_i \cap x_j) \ll p$ for $x_i, x_j \in X$) and approximate the probability of *any* fault occurring in a given cycle as:

$$P(F) = P\left(\bigcup_{x \in X} x\right) \approx pn \tag{5.6}$$

where $n = |X|$.

5.3.6 Reliability model

Reliability is a measure of how well a design tolerates faults and prevents them from leading to undetected errors. We can define reliability formally as $P(W)$, the probability that a design will produce the wrong output in a given cycle, where values closer to zero are better. Building on our fault model from Section 5.3.5, we can model reliability as follows:

$$P(W) = P(F)P(W|F) \tag{5.7}$$

Here $P(W|F)$ represents the average probability that a single fault will lead to a design failure. For designs without an error detection mechanism, this is equivalent to the fraction of all possible faults that are not *masked*. (Masked faults temporarily change the internal behavior of the design but not the outputs.) Let $Y \subseteq X$ be the subset of fault events that result in design failure and let $m = |Y|$. Then:

$$P(W|F) = \frac{m}{n} \tag{5.8}$$

$$\therefore\ P(W) \approx pm \tag{5.9}$$

Assuming that different designs using the same standard cell library have the same p value, we can measure the relative reliability of those two designs as the ratio of their m values. In other words, for two designs A and B:

$$\frac{P(W_A)}{P(W_B)} \approx \frac{pm_A}{pm_B} = \frac{m_A}{m_B} \tag{5.10}$$

We call this value the *reliability improvement* of design B relative to design A. Values greater than one indicate an improvement in reliability (i.e. reduction in failure rate), while fractions indicate a regression in reliability (i.e. an increase in failure rate).

5.3.7 Error recovery

Our methodology enables self-checking designs which raise an error signal when an output check fails. To use an error signal to enable error recovery, such self-checking designs must be integrated with a higher-level error recovery strategy. Examples of this strategy include:

- **Restart the accelerator.** This is the simplest option which is a cost-effective strategy when the faults are very rare events (e.g. once a day) and accelerator inputs and outputs are saved in dedicated memories.
- **Flush the pipeline.** In an accelerator design with a long computation pipeline, it is possible to trigger a pipeline flush followed by a restart of the pipeline wind-up, similar to a microprocessor pipeline flush after a branch misprediction. This is essentially a fine-grained execution restart.

- **Rollback to a checkpoint.** This is the most general strategy, which involves the use of a regular checkpointing mechanism that saves sufficient information about the state of the accelerator to enable a rollback to that state. Multiple redundant checkpoints may be needed if the probability of *checkpoint corruption* (checkpointing an erroneous computation result before the error is detected) is high enough to significantly impede reliability improvement.

5.4 Modulo functional units architecture

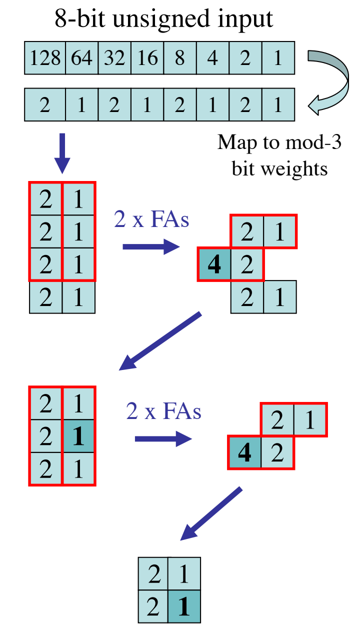
The following subsections discuss our gate-level architectures for our modulo $M(n)$ integer reducers, adders, multipliers, negators, and zero comparator functional units. All of these functional units work with *non-normalized* n-bit encodings (see subsection 5.3.4 for definition) for residues modulo $M(n)$. Furthermore, we provide illustrated examples with specific values of n for explanatory purposes, but these architectures generalize in a straightforward manner (except where noted otherwise) to any $n \geq 2$ and any input bitwidth $w \geq 2n$.

Algorithm 3 Partial modulo reduction

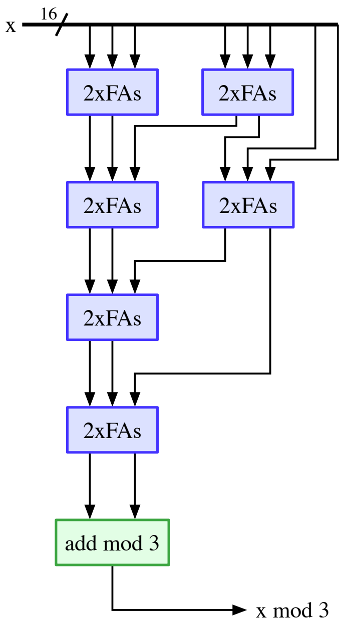
```
procedure REDUCE($A$)                          ▷ $A$ is a $m \times n$ matrix of bits
    while $|A| \geq 3$ do                      ▷ $|A|$ is the number of rows in $A$
        $G \leftarrow$ row triplets selected from $A$.
        $L \leftarrow$ leftover rows, $|L| < 3$.
        $G' \leftarrow G$ passed through $n \times$ FA blocks.
        $A \leftarrow \{G', L\}$.
    end while
    return $A$                                 ▷ The result is a $2 \times n$ matrix of bits
end procedure
```

5.4.1 Reducer

Our reducer functional units compute $y = a \bmod M(n)$, where a is a w-bit wide datapath value, and y is a n-bit residue value.



(a) 8-bit mod-3 reduction strategy



(b) 16-bit mod-3 reducer

Figure 5.1: Wallace-tree like reduction strategy and 16 bit modulo-3 reducer. In (a), each square represents a bit, and the number in the square is the weight of that bit. In (b), each $2 \times$ FA box represents a pair of full adders, one taking three bits of weight 1 as input and one taking three bits of weight 2. Each wire (except the top input bundle) bundles two bits of weights 1 and 2.

5.4.1.1 Reduction strategy

In order to perform this reduction to a residue, our unique approach is a Wallace-tree like reduction strategy shown in Fig. 5.1(a). Our reducer starts with a standard bit sequence representing an integer with bits having weights with successive powers of two. Using the standard homomorphism from integer arithmetic to modulo arithmetic (see subsection 5.3.1), we can reduce the weights in Eq. (5.5) to residues as follows:

$$y = \left(\sum_{i=0}^{w-1} 2^i a_i\right) \bmod M(n) \tag{5.11}$$

$$= \left(\sum_{i=0}^{w-1} (2^i \bmod M(n)) a_n\right) \bmod M(n) \tag{5.12}$$

$$= \left(\sum_{i=0}^{w-1} (2^{i \bmod n}) a_n\right) \bmod M(n) \tag{5.13}$$

where the last equivalence follows from Eq. (5.4).

In other words, the weights on the input bits shown in Fig. 5.1(a) are reduced to a repeating cycle of successive powers of two, drawn as a 4×2 matrix in Fig. 5.1(a) for $n = 2$ and $M(n) = 3$. We now feed these bits to full adder (FA) gates. A full adder takes three bits of weight w as input and produces two bits as output: one of weight w and another of weight $2w$. A FA is a transistor-level optimized cell in a standard cell library that reduces the number of bits by 1 (3 inputs less 2 outputs), and as we will see shortly, performs arithmetic amenable to a modulo context. Thus FAs are ideal technology mapping targets for cost-effective modulo arithmetic.

In the 4×2 matrix in Fig. 5.1(a), we can select two groups of 3 bits with the same weight (highlighted in red) and pass them through full adders. The result is two bits of weight 2, one bit of weight 1, and one bit of weight 4. But $4 = 1 \pmod 3$ (an example of Eq. (5.4)), so the output of the FAs is equivalent to two bits of weight 1 as well as 2. Since we also have two bits left over from the input, we now have a 3×2 matrix of bits (lower left corner of Fig. 5.1(a)). We repeat this process, selecting groups of 3 bits and putting them through FAs until no groups of 3 bits remain. This process is formalized in Algorithm 3.

Intuitively, it is desirable to perform reductions with entirely full adders since each FA gate is doing useful work reducing the number of bits by 1. Half adders (HA) take two bits of the same weight, w, as input and produce two bits of weights w and $2w$ and thus do not by themselves reduce the number of bits.

5.4.1.2 Architecture

Fig. 5.1(b) provides a block diagram for our Mersenne modulo reducer gate-level architecture for $n = 2 \implies M(n) = 3$ and input width $w = 16$. Each wire (except the top input bundle) corresponds to a bundle for a bit matrix row in Fig 5.1(a) if it were to be expanded from an 8-bit input to a 16-bit input. Three wires representing three rows are connected to corresponding bundles of FAs (the $2 \times$ FA blocks) which generate two rows (wires) of output. Note that, perhaps counterintuitively, the two output wires of each $2 \times$ FA block in Fig. 5.1(b) represent bundles with all of the different possible bit weights (i.e. a row of a bit matrix in Fig 5.1(a)), **not** a bundle of sum bits and a bundle of carry bits.

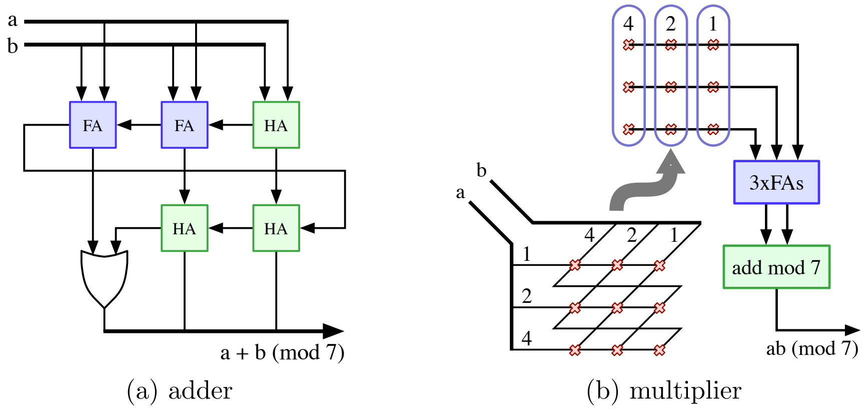
Using the reduction strategy in Algorithm 3, we iteratively process all of the groups of three wires from the previous stage in parallel by passing them through $2 \times$ FA blocks until only two wires remain. Note that while Fig. 5.1(b) provides an example for $n = 2 \implies M(n) = 3$ and input width $w = 16$, along with our other functional units in this section, this example generalizes to any $n \geq 2$ and any $w \geq 2n$.[2] For the final reduction stage, we use a binary modulo adder, which we discuss next.

5.4.2 Adder

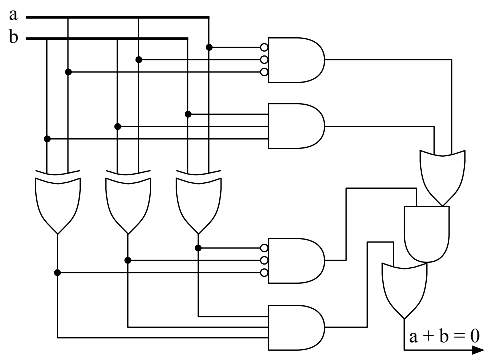
Our gate-level modulo binary adder architecture is shown in Fig. 5.2(a) for $n = 3 \implies M = 7$. The first stage is a standard ripple-carry adder. The final carry produced by the first stage has weight $2^n = 1 \pmod{M}$ by Eq. (5.4), so it wraps around as a carry-in to the second stage. We guarantee under all possible adder input combinations that this carry circulation will stop before or at the most significant bit in the second stage. In other words, at most one of the input bits to the MSB adder gate in the second stage is a 1. We call an adder gate with this input constraint a *quarter adder* (QA) and implement it with a 2-input OR gate.

Theorem 1. *At most one of the inputs to the quarter adder gate in our modulo binary adder is* 1.

[2]For w values that are not a multiple of n, we can pad the input with conceptual constant zero bits until $w \bmod n = 0$. Each of those zero bits will be connected to a different full-adder, so we can recover from most of the padding overhead by optimizing those full adders with one constant zero bit to half adders.



(a) adder (b) multiplier



(c) zero comparator

Figure 5.2: Modulo-7 adder, multiplier, and zero comparator. In (b) bits are annotated with their weights. Each X represents a 2-input AND gate with inputs connect on the left, and outputs connected on the right.

Proof. We prove this guarantee by contradiction. Suppose both inputs to the QA are 1. Then both inputs to each half adder in the second stage must be 1. Then all inputs to each full adder in the first stage must be 1. Then both outputs of the half adder in the first stage must be 1, which is impossible. □

5.4.3 Multiplier

Refer to Fig. 5.2(b) for our modulo binary multiplier architecture for $n = 3 \implies M = 7$. The multiplier is like an array multiplier with a twist: each combination of input bits is combined with a 2-input AND gate, but bit weights wrap around modulo M, resulting in a $n \times n$ matrix of partial product bits as shown in the upper part of Fig. 5.2(b), which also corresponds to the lower-left corner. For example, the product of the two bits of weight $2^2 = 4$ will have weight $2^4 = 16 = 2^1 = 2 \pmod 7$ by Eq. (5.4).

Since the output is a square matrix of bits, we can then apply our reduction techniques from subsection 5.4.1 to reduce them. Fig. 5.2(b) elaborates on the reduction for a 3×3 matrix of 9 bits.

5.4.4 Negation and subtraction

Negation with our encodings of Mersenne modulo numbers is quite simple: just pass each bit through a NOT gate. Mathematically, this works because:

$$-a = M - a = (2^n - 1) - \sum_{i=0}^{n-1} 2^i a_i \tag{5.14}$$

$$= \sum_{i=0}^{n-1} 2^i - \sum_{i=0}^{n-1} 2^i a_i = \sum_{i=0}^{n-1} 2^i (1 - a_i) \pmod M \tag{5.15}$$

These NOT gates can be integrated into gates in upstream or downstream functional units to effectively eliminate their overhead (e.g., flip-flops with inverted outputs or NAND gates instead of AND gates in a multiplier array). Subtraction is implemented as a composition of negation and addition, i.e. $a - b = a + (-b)$.

5.4.5 Zero comparator

We created a custom architecture for a zero comparator which takes $2n$ bits as input and compares the sum of the $2 \times n$ matrix of bits with zero, illustrated in Fig. 5.2(c) for $n = 3 \implies M = 7$. We take $2n$ bits as input due to the extra cost of reducing $2n$ bits to n bits (see Subsection 5.5.4).

Theorem 2. *Our zero comparator architecture illustrated in Fig. 5.2(c) is correct.*

Proof. We start with some special cases: the inputs $(-0 + -0)$ (all ones) and $(0 + 0)$ (all zeroes) produce the correct output by inspection. For the remaining cases the only way to get a sum of zero is if a and b are bitwise complements of each other. Again, we see by inspection that the logic will output a 1 for this case. If a and b are not bitwise complements, the only way for the logic to output a 1 is if $a = b = \pm 0$, the special cases we already discussed. $\square$

5.5 Cost-effectiveness evaluations

5.5.1 Cost evaluation methodology

To evaluate the area and delay of our approach, we implemented our gate-level designs with a 45 nm ARM standard cell library. Our focus is on minimum area to minimize cost, so we selected the smallest ($1\times$) standard cell for each gate type for the modulo functional units. The longer delay of a modulo shadow datapath simply increases error detection latency by a few cycles, so this tradeoff is acceptable in return for reduced area cost. We compare with other techniques compiled with the logic synthesis tool Synopsys Design Compiler 2016.03-SP5-5 and also map modulo functional units from those designs to $1\times$ standard cells to enable meaningful comparisons.

5.5.2 Effectiveness evaluation methodology

Our reliability improvement evaluation is based on gate-level error injection. Our reliability evaluation of each design starts with 10,000 simulated fault injection experiments with each main datapath and shadow datapath width variation of that design as well as baseline unprotected designs with no shadow datapath. In each fault injection experiment, a random gate output is selected to have its bit flipped while a random test vector is selected for input to the design. The test vectors are selected using a decaying exponential probability distribution such that each possible number of leading zeroes is equally probable. This distribution provides a more realistic input distribution that represents the full dynamic range of the largest and smallest input values rather than just the largest input values as a uniform distribution would. Test inputs that are so large that they result in an overflow in the main datapath are discarded and replaced with another random vector. Our shadow datapath technique is capable of detecting overflows, but the main focus of this chapter is detecting single event transient errors. Thus we set these overflow cases aside to avoid polluting our error injection data.

Table 5.1: Functional-Unit Level Results.

		[80]		Ours		Difference	
		Area	Delay	Area	Delay	Area	Delay
Modulo-3	adder	8.3	0.09	12.8	0.13	53.7%	42.9%
	subtractor	8.3	0.09	14.0	0.15	68.9%	60.4%
	multiplier	4.5	0.04	17.8	0.16	299.3%	292.7%
	32-bit reducer	177.8	0.73	155.6	0.39	−12.5%	−47.1%
Modulo-7	adder	55.9	0.32	21.1	0.21	−62.3%	−35.8%
	subtractor	59.7	0.33	23.0	0.22	−61.6%	−32.7%
	multiplier	30.0	0.21	47.8	0.30	59.2%	42.3%
	32-bit reducer	493.2	1.27	153.6	0.61	−68.8%	−52.0%
Modulo-15	adder	188.0	0.46	29.3	0.27	−84.4%	−41.6%
	subtractor	192.8	0.53	31.9	0.29	−83.5%	−46.0%
	multiplier	133.4	0.51	90.4	0.42	−32.2%	−16.8%
	32-bit reducer	687.6	1.55	151.7	0.53	−77.9%	−66.1%

Thus in each injection experiment we apply the input test vector and flip the bit of the selected gate output at the cycle when the input reaches that gate. We observe whether the fault is masked (i.e. whether the output is correct) and whether the fault is detected (i.e. whether the error output is asserted). Faults that are both unmasked and undetected are counted as design failures. Now that we have a failure rate metric, we apply Eq. (5.10) to compute the relative reliability of each of our design variations over the corresponding baseline. We approximate the expected number of all possible faults that lead to design failure, m, through this sampling procedure as follows:

$$m \approx \text{injection-sites} \times \frac{\text{undetected-unmasked-faults}}{\text{faults-injected}} \tag{5.16}$$

5.5.3 Modulo functional units

Our first set of comparisons looks at the functional-unit level and compares our designs for modulo adders, subtractors, reducers, and multipliers to equivalent designs from [80]. We implement a subtractor with a negation of one input followed by an adder. In [80], the adder, subtractor, and multiplier are implemented with lookup tables, while a reducer is implemented as a tree of modulo adders.

Table 5.1 shows the results of our comparisons. Area is measured in µm^2 while delay is measured in ns. We observe that our reducer designs, which tend to be the dominant part of shadow datapath costs, provide lower area and delay than those of [80]. Even for the simplest modulo-3 reducer, we achieve a 12.5% reduction in area and a 47.1% reduction in delay. Furthermore, this reducer cost is essentially fixed as the modulo base scales because the number of full adders required is the same as the number of bits reduced $(w - n)$. Longer delays also increase the need for pipeline flip-flops which in turn impacts area cost. Our other observation is that as we scale to larger Mersenne bases, even the adders and subtractors and eventually the multiplier become less costly than [80]. This is expected due to the exponential scaling nature of lookup tables in [80].

5.5.4 Self-checking multiply accumulator

To evaluate the cost-effectiveness of our approach in a functional-unit level application, we consider a self-checking multiply-accumulator (MAC) illustrated in Fig. 5.3(a). The shadow datapath is built from components introduced in Section 5.4: a full reducer to n bits (Fig. 5.1(b)), a partial reducer to $2n$ bits (omitting the final binary adder in Fig. 5.1(b)), a modulo multiplier matrix from Fig. 5.2(b) (with NAND gates to negate the output), a negation inverter (subsection 5.4.4), and a zero comparator (Fig. 5.2(c)). Note that the reducers are summation reducers, so they function as adders.

Under error free conditions, the shadow datapath will compute $-(a \bmod M)(b \bmod M)$ and $-(c \bmod M)$, add it to $ab + c$ from the output, reduce the result modulo M, and get a result of 0. Computation errors in either the main or shadow datapath will generate a nonzero result (provided aliasing does not occur, which in our experience is unlikely for single bit errors).

A key strategy in this design is the avoidance of reduction beyond $2n$ bits (except for multiplier inputs) as reduction beyond $2n$ bits involves the use of half adders which do not directly provide bit reduction while the main reduction process is mapped entirely to full adders. This strategy is similar to the carry save technique used in standard binary integer arithmetic design.

We evaluated our MAC architecture by synthesizing the multiply accumulate main datapath with Design Compiler targeting minimum delay while generating 1× gate-level designs for the shadow datapath with gate-level architectural templates and Algorithm 3. QoR and reliability results for different width datapaths and modulo widths (n) are shown in Figs. 5.3(b), 5.3(c), and 5.3(d). We observe $12 - 18\%$ area overhead for a 32-bit self-checking MAC with reliability improvements of 4–142×. We observe error signal delays of about 1.5× the delay of the main datapath.

As mentioned at the start of this section, the longer delay of a modulo shadow datapath simply increases error detection latency by a few cycles and does not affect the performance of the main datapath. For example, in [80], the authors used a pipelining strategy to run the shadow datapath 2 cycles behind the main datapath without affecting performance. We adopted a similar pipelining strategy to implement a pipelined version of the self-checking MAC, whose architecture and QoR results are shown in Fig. 5.4. The main datapath has one pipeline stage while the shadow datapath has three, for an error signal delay of 2 cycles. We observe a $1 - 2\%$ increase in area overhead for a 32-bit pipelined design due to additional flip-flips, while the delay (clock) overhead decreases from 100% to $0 - 20\%$. Since our error injection methodology involves injection into *all* gates, not just flip-flops, the reliability improvements for the pipelined version of the MAC are very similar to the combinational version as flip-flops represent a small fraction of the gates in these designs.

It is important to note that our delay overhead evaluation represents a near worst-case analysis: we tell Design Compiler to synthesize our main datapath to run at the maximum possible clock frequency and the critical path of the main datapath goes through a single multiplier and adder. Such a high clock frequency is a difficult target to achieve in a complex design even without considerations for error detection logic. Nevertheless, for designs with very high clock frequencies, there are strategies for eliminating this clock overhead in the shadow datapath. Examples include:

1. Retime the flip-flops in the shadow datapath for better balancing of the pipeline stages. This is the best solution for small clock-period overheads where there is likely a solution involving overloaded stages offloading their excess delay to underloaded stages.

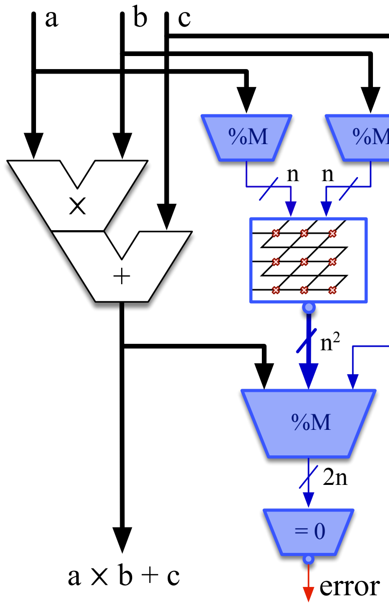
2. Perform minimum-area gate upsizing on the shadow datapath to satisfy timing constraints. As our shadow datapath consists entirely of 1× sized gates, there is plenty of room for upsizing.

3. Add additional pipeline stages to the shadow datapath. This solution will work in the worst cases where the other methods are insufficient to address timing on their own.

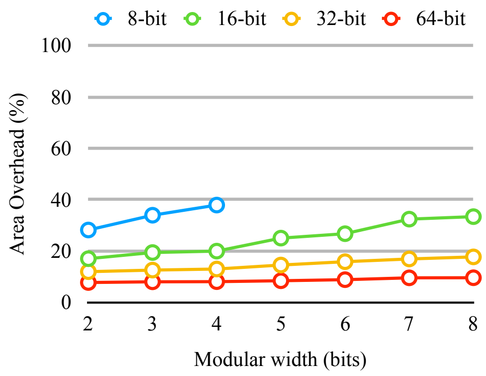
5.5.5 Self-checking linear algebra primitives

Expanding on our MAC design in Section 5.5.4, we consider a variety of key functional units in vector and matrix computation. The designs we selected are: **(1)** vector-scalar product, **(2)** vector inner-product, **(3)** vector outer-product, **(4)** matrix-vector product, and **(5)** matrix product. We selected these functional units as representative examples because such operations tend to be the computationally intensive parts of complex datapaths in accelerators. As mentioned in Section 5.1, accelerators with complex datapaths are becoming increasingly prevalent due to microprocessor designs reaching the limits of traditional CMOS scaling. Furthermore, these functional units can be used directly in large vector and matrix computations by breaking the computation into appropriately sized tiles that correspond directly to the input of our primitives. Matrix and vector based complex computation applications include machine learning, data mining, artificial neural networks, image and video processing, and digital signal processing.

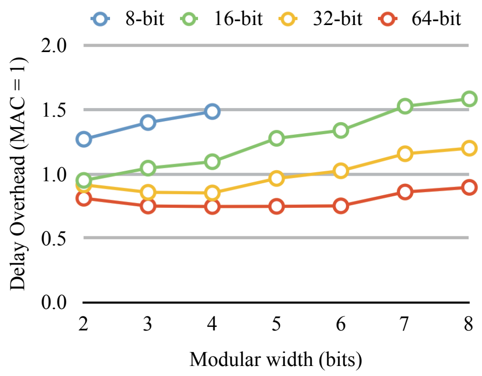
All designs were implemented with the same pipelining strategy as the pipelined multiply accumulator discussed in Section 5.5.4. The cost (QoR overhead) and benefit (reliability improvement) of each variation of each of our five compound functional unit designs are charted in Figs. 5.5–5.9. For a 32-bit main datapath and 2-bit shadow datapath, we observe area costs of 6–10% with a reliability of 3–61×. Looking at the 5-bit shadow variation, the area cost increases to 9–14%, but the reliability benefit jumps to 28–901×. Finally, in the widest 8-bit shadow variation, we observe area costs of 15–20% with reliability benefits of 121–2477×. Overall, we observe a slow, graceful increase in area cost as the shadow datapath width scales while the improvement in reliability shows an exponential trend (note the logarithmic scales on the reliability metric). The exponential trend in reliability is expected as



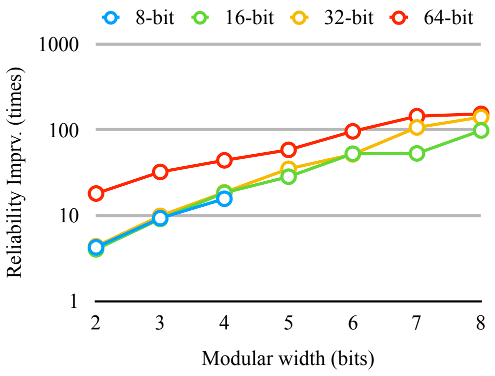
(a) Architecture



(b) Area overhead

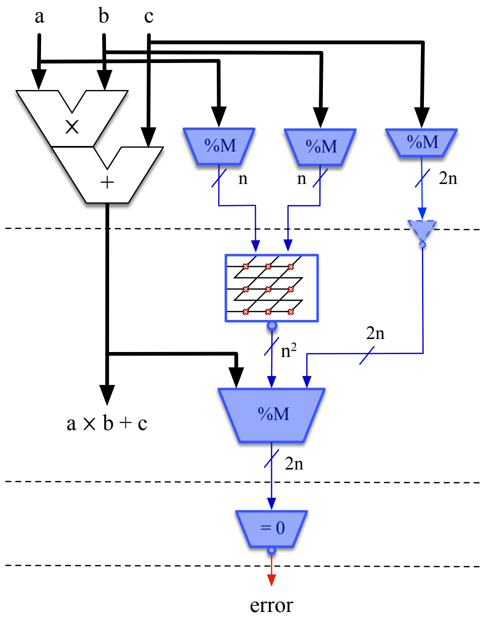


(c) Error delay

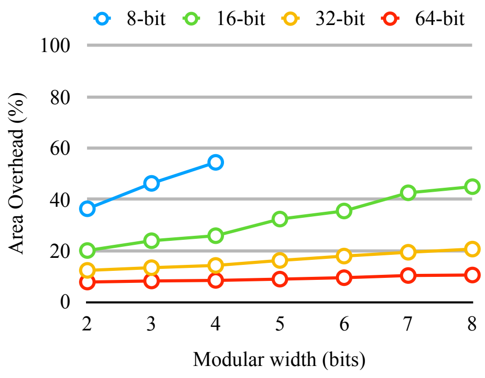


(d) Reliability improvement

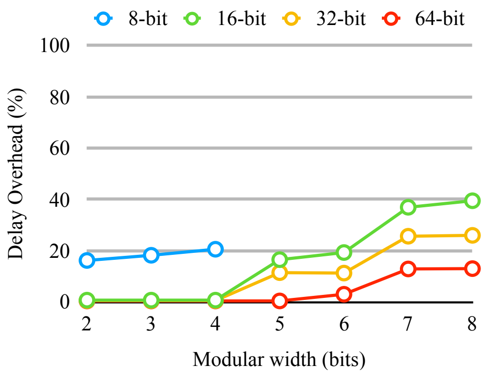
Figure 5.3: Self-checking multiply accumulator architecture and overhead evaluation. $M = 2^n - 1$.



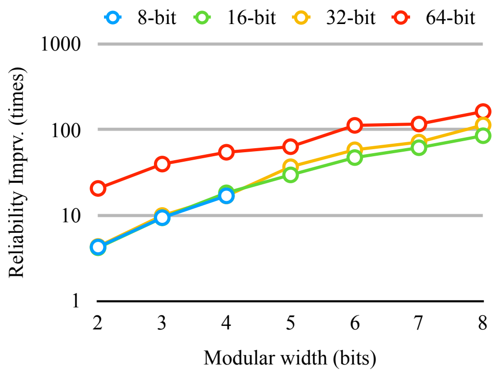
(a) Architecture



(b) Area overhead



(c) Clock overhead



(d) Reliability improvement

Figure 5.4: Self-checking pipelined multiply accumulator architecture and overhead evaluation. $M = 2^n - 1$. Dotted lines represent the boundaries of pipeline stages. We implemented the inverter at the output of the reducer for input c with an inverting flip-flop.

the probability of aliasing (the output is wrong, but happens to have the correct modulo checksum) is proportional to the modular base $M(n) = 2^n - 1$. These results demonstrate that our technique scales well to wide shadow datapaths to produce very cost-effective solutions for designs requiring a multiple order-of-magnitude boost in reliability.

It is important to note that these results represent the worst-case needle-in-a-haystack fault scenario: A *single* bit-flip occurring in a *single* cycle at *any* gate output in the design with test vectors representing the *full dynamic range* of possible input. Faults that affect multiple locations or multiple cycles will have a greater chance of being detected. Fault models that consider a subset of gates (such as only flip-flops as is often found in the literature) create artificial advantages for techniques that only consider that subset of gates. Flip-flops in particular represent a small fraction of the area occupied by the combinational logic in the same design, so the assumption that faults only significantly affect flip-flops is questionable. As mentioned in Section 5.1, our approach is to avoid assumptions and design general-purpose error detection solutions.

We also measure the clock period overhead of each variation of our designs. For many variations, particularly those with a wide main datapath and a narrow shadow datapath, we observe no overhead. However, we do observe clock period overheads of up to 60% in some cases. As discussed in Section 5.5.4, the clock frequencies of these designs are aggressive and there are many strategies for eliminated clock overhead if the shadow datapath does turn out to be the critical path.

Finally, looking at the 64-bit reliability results, we observe some interesting anomalies: the trend for reliability improvement diverges significantly from the trend for 8-bit, 16-bit, and 32-bit reliability improvement. Note that reliability improvement is largely a function of two factors: the shadow datapath width which determines the probability of aliasing (the output is wrong, but it happens to have the correct modulo checksum) and the design of the main datapath. A shadow datapath design of a given width is largely unchanged from the 32-bit to the 64-bit main datapath, which leaves the design of the main datapath as implemented by our logic synthesis tool, Design Compiler. As Design Compiler is a proprietary tool, we have limited information about its behavior, but we suspect the reason for the divergence is a transition from

Table 5.2: Self-Checking Component Area Overhead.

Design	Red. #	Mul. #	R/M	Area O/H. (32-bit)		Area O/H. (64-bit)	
	(shadow)	(main)	Ratio	Mod-3	Mod-7	Mod-3	Mod-7
Matrix mul.	12	8	1.50	6.2%	7.1%	3.7%	4.0%
Vector outer product	15	9	1.67	7.4%	8.4%	4.2%	4.6%
Matrix vector mul.	11	6	1.83	7.4%	8.3%	4.7%	5.0%
Vector inner product	7	3	2.33	9.2%	10.2%	5.5%	5.8%
Scalar vector mul.	7	3	2.33	10.0%	11.1%	5.5%	5.9%
MAC (pipelined)	4	1	4.00	12.3%	13.4%	7.8%	8.2%

* Red.: reducer. Mul.: multiplier. R./M: the ratio of Red. # over Mul. #. O/H.: overhead.

one strategy for 8-bit, 16-bit, and 32-bit multiplier synthesis to a different strategy for 64-bit multiplier synthesis.

5.5.6 Early protection overhead estimation

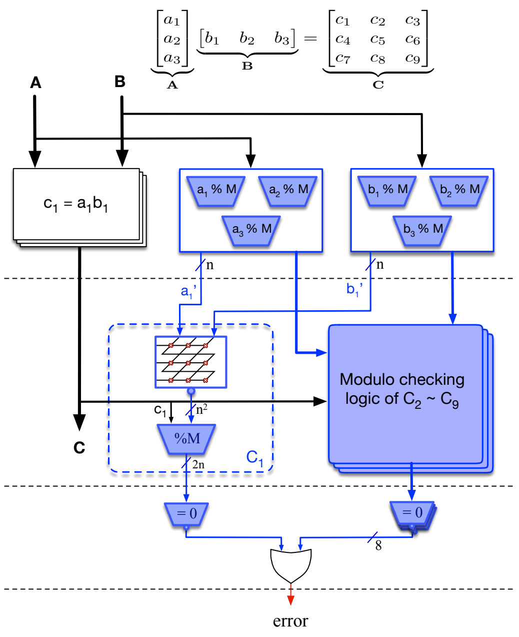
For early-stage design modeling, we propose an area overhead estimation indicator, *reducer to multiplier ratio (R/M ratio)*, which is the ratio of the reducer count in the shadow datapath to the multiplier count in the main datapath. The R/M ratio is an analytical way to approximate area overhead assuming that reducers dominate shadow datapath cost while multipliers dominate main datapath cost. We approximate reducer count as the number of inputs and outputs; therefore, the approximation is more accurate for narrow shadow datapaths, that is, $n^2 \ll d$, where n is modular width and d is original datapath width.

We take our self-checking pipelined MAC and linear algebra primitives with different combinations of 32/64-bit main datapaths and mod-3/mod-7 shadow datapaths to compare their R/M ratios and actual area overheads in Table 5.2. Except for the MAC, whose size is small and whose area overhead can be easily caused by other components, e.g., flip-flops, we observe that the proposed R/M ratio is a good indicator to estimate the area overhead of self-checking arithmetic components. We find that the R/M ratio is a useful heuristic for designers to estimate area overhead at the early stages of self-checking design development.

5.6 Conclusions

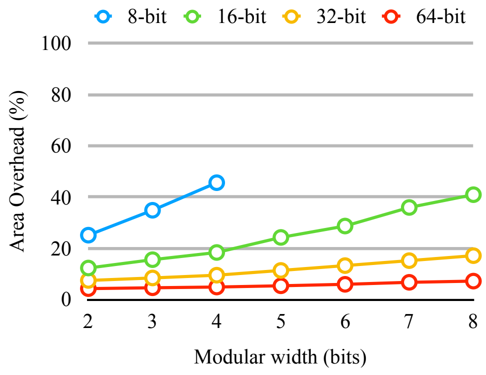
We introduced new gate-level architectures for Mersenne modulo functional units targeting shadow datapaths for reliability, including a modulo reduction algorithm that maps entirely to full adders and new adder and multiplier designs based on integer counterparts with a wraparound twist. We compared our functional units to the previous state-of-the-art approach, observing a 12.5% reduction in area and a 47.1% reduction in delay for a 32-bit mod-3 reducer. We also observed that our reducer costs, which tend to dominate shadow datapath costs, do not increase with larger modulo bases, and that for modulo-15 and above, all of our modulo functional units have better area and delay than their previous counterparts. To demonstrate the applicability of our approach for reliability, we used these building blocks to create self-checking multiply-accumulate and linear algebra primitive datapaths. We observed cost-effective results, with 32-bit datapath overheads of 6–10% for a 3–61× improvement in reliability, and overheads of 15–20% for a 121–2477× improvement in reliability. Additionally, we proposed an area overhead estimation method for self-checking arithmetic component implementation.

Future directions for this research include: **(1)** extending support for modulo bases beyond Mersenne numbers; **(2)** support for fixed-point arithmetic; **(3)** gate-level automation through integration into a logic synthesis engine; and **(4)** full automation of shadow datapath generation with a high-level synthesis approach like [80].

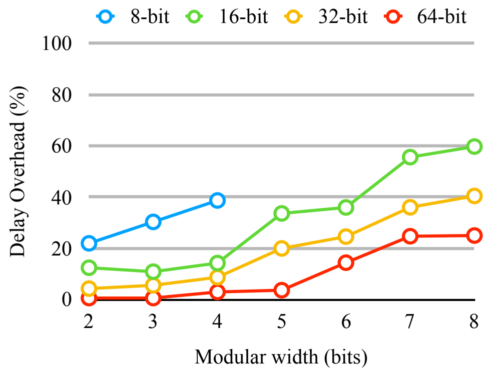


$\underbrace{\begin{bmatrix}a_1\\a_2\\a_3\end{bmatrix}}_{\mathbf{A}}\underbrace{\begin{bmatrix}b_1 & b_2 & b_3\end{bmatrix}}_{\mathbf{B}}=\underbrace{\begin{bmatrix}c_1 & c_2 & c_3\\c_4 & c_5 & c_6\\c_7 & c_8 & c_9\end{bmatrix}}_{\mathbf{C}}$
A
B
$c_1 = a_1b_1$
a_1 % M
a_2 % M
a_3 % M
b_1 % M
b_2 % M
b_3 % M
n
a_1'
b_1'
Modulo checking logic of C_2 ~ C_9
C_1
n^2
%M
2n
C_1
C
= 0
8
error

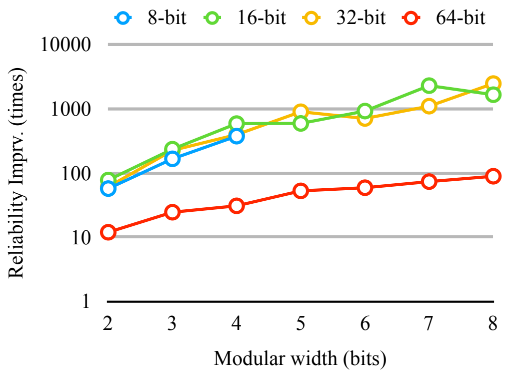
(a) Architecture



(b) Area overhead

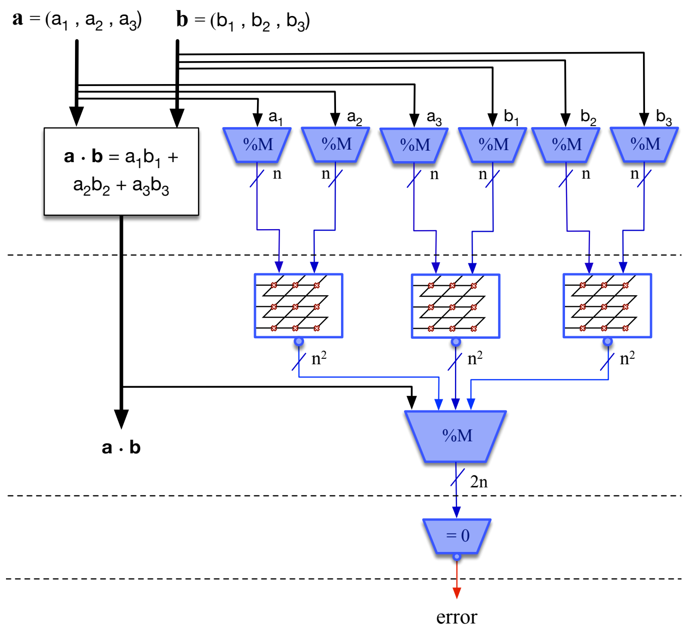


(c) Clock overhead

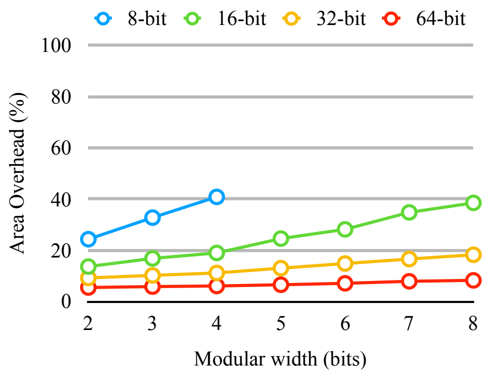


(d) Reliability improvement

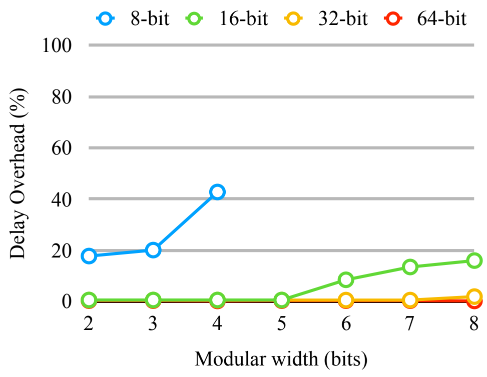
Figure 5.5: Self-checking pipelined vector outer product and overhead evaluation. $M = 2^n - 1$. Dotted lines represent the boundaries of pipeline stages.



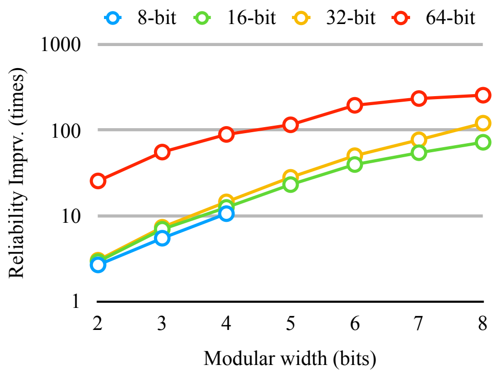
(a) Architecture



(b) Area overhead

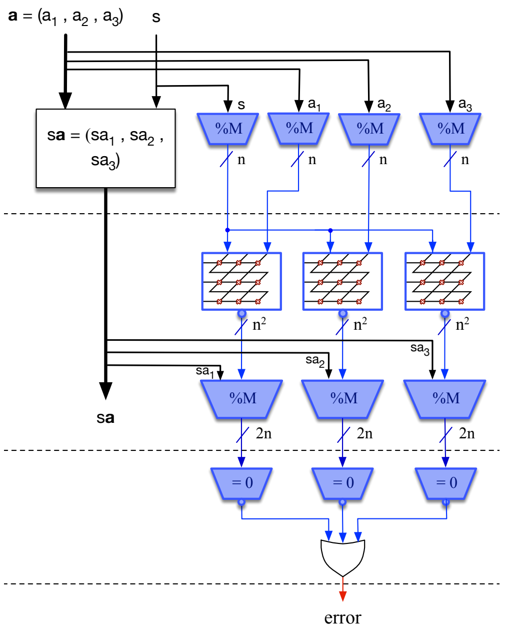


(c) Clock overhead

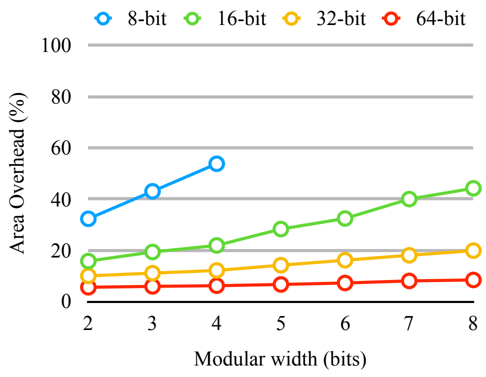


(d) Reliability improvement

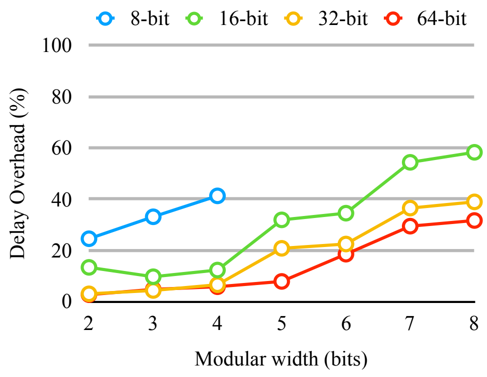
Figure 5.6: Self-checking pipelined vector inner product and overhead evaluation. $M = 2^n - 1$. Dotted lines represent the boundaries of pipeline stages.



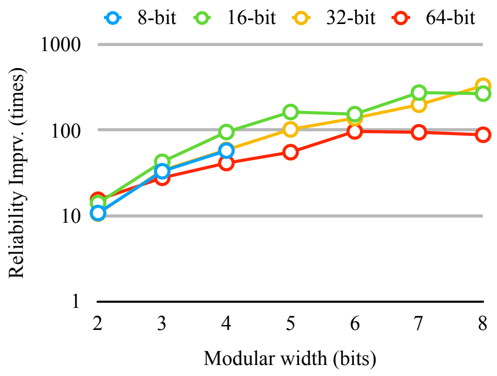
(a) Architecture



(b) Area overhead

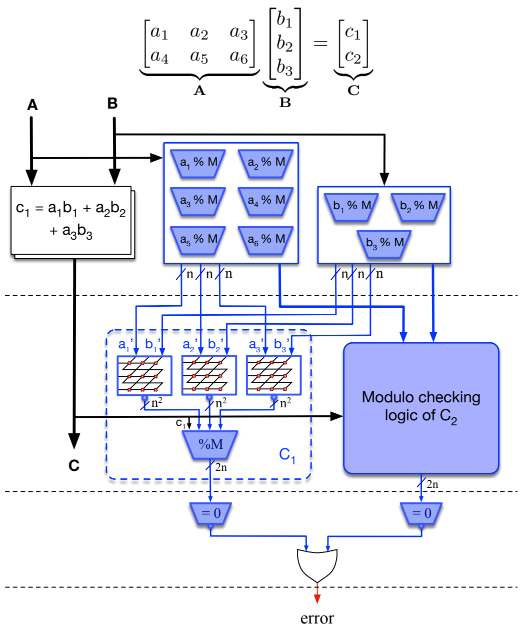


(c) Clock overhead

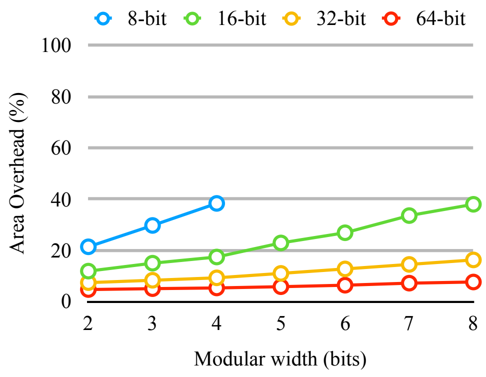


(d) Reliability improvement

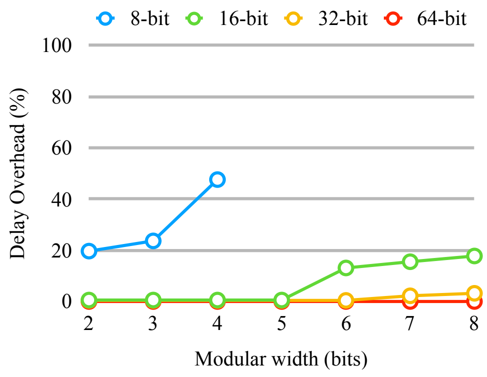
Figure 5.7: Self-checking pipelined scalar vector multiplication and overhead evaluation. $M = 2^n - 1$. Dotted lines represent the boundaries of pipeline stages.



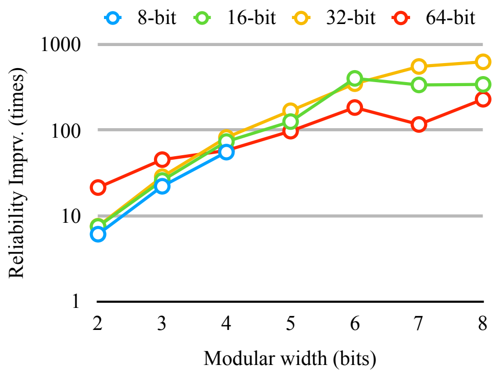
(a) Architecture



(b) Area overhead

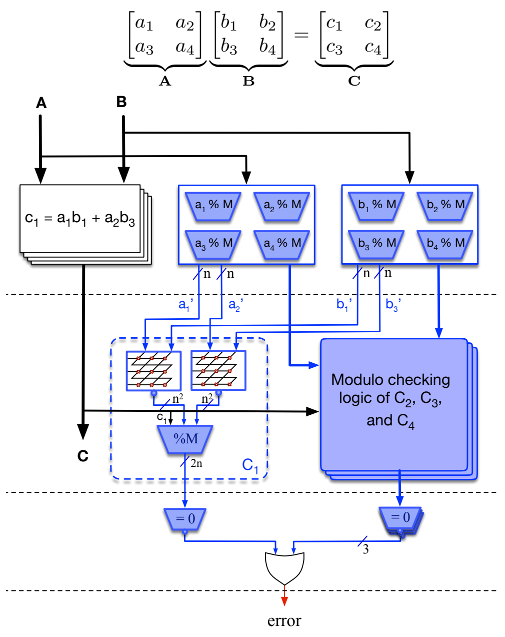


(c) Clock overhead

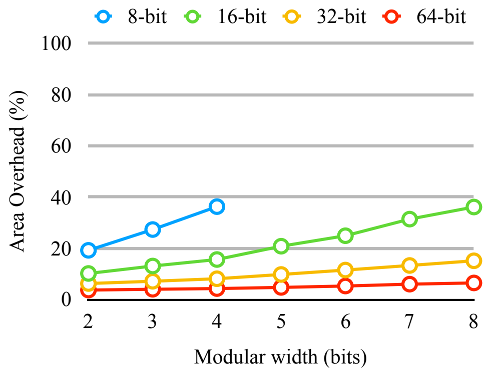


(d) Reliability improvement

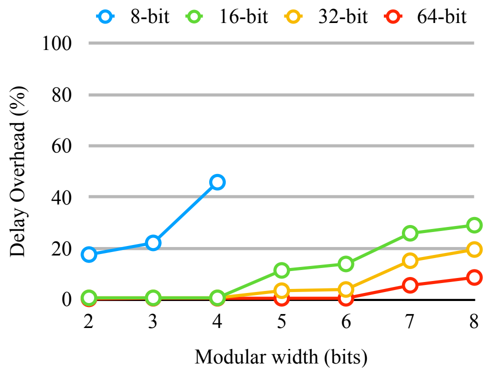
Figure 5.8: Self-checking pipelined matrix vector multiplication and overhead evaluation. $M = 2^n - 1$. Dotted lines represent the boundaries of pipeline stages.


$\underbrace{\begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} b_1 & b_2 \\ b_3 & b_4 \end{bmatrix}}_{\mathbf{B}} = \underbrace{\begin{bmatrix} c_1 & c_2 \\ c_3 & c_4 \end{bmatrix}}_{\mathbf{C}}$
A
B
$c_1 = a_1b_1 + a_2b_3$
a_1 % M
a_2 % M
a_3 % M
a_4 % M
b_1 % M
b_2 % M
b_3 % M
b_4 % M
n
a_1'
a_2'
b_1'
b_3'
Modulo checking logic of C_2, C_3, and C_4
n^2
c_1
%M
2n
C_1
C
= 0
3
error

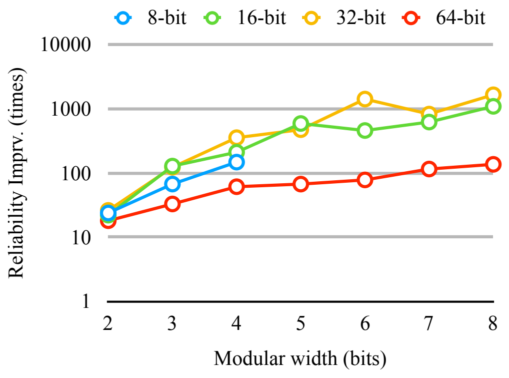
(a) Architecture



(b) Area overhead



(c) Clock overhead



(d) Reliability improvement

Figure 5.9: Self-checking pipelined matrix multiplication and overhead evaluation. $M = 2^n - 1$. Dotted lines represent the boundaries of pipeline stages.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this dissertation, we present several approaches to improve circuit reliability and energy efficiency from different angles. The approaches include a variety of algorithmic methods, heuristics, and design techniques. Furthermore, we take into account the scalability of our approaches for the applicability to industry-strength designs, which has been shown in the experimental results. The highlights of this dissertation are as follows:

1) Applying SAT solving to tackle reliability issues by automatically generating concise assertions with complete coverage.

2) Applying data mining and SAT solving to improve the energy efficiency of BTW design by optimizing common cases.

3) Proposing a coordinated and scalable methodology including early-stage logic restructuring, NBTI-aware technology mapping, and smart pin swapping technique to alleviate NBTI effect.

4) Proposing a new low-cost gate-level architectures for Mersenne modulo functional units targeting shadow datapaths for reliability improvement.

In addition to the angles and methods we have explored, we would like to propose some possible future directions of circuit reliability and energy efficiency. As the complexity of hardware increases, development effort becomes enormous, thus dramatically increasing time-to-market and design costs. High-level synthesis (HLS) has emerged as a promising way to deal with this complexity and accelerate design automation by raising the abstraction levels. Higher abstraction levels not only provide designers with a global picture of the design but also allow designers to explore the design space more efficiently and rapidly.

Compared to the layers of verification, logic synthesis, and functional unit design we have explored, HLS has richer information about the design behavior such as computation expressions and control flow, thus enabling further optimizations, which are not feasible at the RTL or gate level. Therefore, tackling reliability and energy issues from the angle of HLS is worthwhile to explore.

An interesting idea is to automatically generate assertions through HLS for reliability improvement. In traditional ways, assertions are written by designers manually, but these designer-written assertions might not be sufficient enough to catch all bugs and errors. Therefore, we believe that an HLS-based assertion generation technique can serve as a supplementary methodology to provide additional fault coverage, since HLS has more comprehensive information of design behavior. One usage of assertions is to be synthesized as checkers in hardware for detecting errors. However, these extra checkers might incur unacceptable performance degradation and area overhead. Therefore, another possible aspect to explore is to develop light weight checkers, which can maintain fault coverage at a certain level but do not sacrifice performance and area too much.

REFERENCES

[1] R. A. Abdallah and N. R. Shanbhag, "Minimum-energy operation via error resiliency," *Embedded Systems Letters, IEEE*, vol. 2, no. 4, pp. 115–118, 2010.

[2] J. Henkel, L. Bauer, N. Dutt, P. Gupta, S. Nassif, M. Shafique, M. Tahoori, and N. Wehn, "Reliable on-chip systems in the nano-era: Lessons learnt and future trends," in *Proceedings of the 50th Annual Design Automation Conference.* ACM, 2013, p. 99.

[3] Mentor Graphics Inc., "Assertion-based verification," 2012.

[4] H. Foster, D. Lacey, and A. Krolnik, *Assertion-Based Design*, 2nd ed. Norwell, MA, USA: Kluwer Academic Publishers, 2003.

[5] "BugScope(TM): A full-chip assertion synthesis," *URL http://www.nextopsoftware.com/BugScope-assertion-synthesis.html.*

[6] X. Cheng and M. S. Hsiao, "Simulation-directed invariant mining for software verification," in *Proceedings of the conference on Design, automation and test in Europe.* ACM, 2008, pp. 682–687.

[7] P.-H. Chang and L.-C. Wang, "Automatic assertion extraction via sequential data mining of simulation traces," in *Proceedings of the 2010 Asia and South Pacific Design Automation Conference.* IEEE Press, 2010, pp. 607–612.

[8] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty, "IODINE: a tool to automatically infer dynamic invariants for hardware designs," in *Proceedings of the 42nd Annual Design Automation Conference.* ACM, 2005, pp. 775–778.

[9] W. Li, A. Forin, and S. A. Seshia, "Scalable specification mining for verification and diagnosis," in *Proceedings of the 47th Design Automation Conference.* ACM, 2010, pp. 755–760.

[10] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson, "Goldmine: Automatic assertion generation using data mining and static analysis," in *Proceedings of the Conference on Design, Automation and Test in Europe.* European Design and Automation Association, 2010, pp. 626–629.

[11] S. Bensalem, Y. Lakhnech, and H. Saidi, "Powerful techniques for the automatic generation of invariants," in *Computer Aided Verification.* Springer, 1996, pp. 323–335.

[12] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, "Non-linear loop invariant generation using Gröbner bases," *ACM SIGPLAN Notices*, vol. 39, no. 1, pp. 318–329, 2004.

[13] N. Eén and N. Sörensson, "An extensible sat-solver," in *Theory and Applications of Satisfiability Testing.* Springer, 2003, pp. 502–518.

[14] A. Gupta, K. L. McMillan, and Z. Fu, "Automated assumption generation for compositional verification," in *Computer Aided Verification.* Springer, 2007, pp. 420–432.

[15] A. Gupta, Z. Yang, P. Ashar, and A. Gupta, "Sat-based image computation with application in reachability analysis," in *Formal Methods in Computer-Aided Design.* Springer, 2000, pp. 391–408.

[16] T. Austin, V. Bertacco, D. Blaauw, and T. Mudge, "Opportunities and challenges for better than worst-case design," in *Proceedings of the 2005 Asia and South Pacific Design Automation Conference.* ACM, 2005, pp. 2–7.

[17] T. M. Austin and V. Bertacco, "Deployment of better than worst-case design: Solutions and needs," in *23rd International Conference on Computer Design (ICCD 2005), 2-5 October 2005, San Jose, CA, USA*, 2005, pp. 550–555.

[18] L. Wan and D. Chen, "CCP: Common case promotion for improved timing error resilience with energy efficiency," in *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design.* ACM, 2012, pp. 135–140.

[19] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques.* Elsevier, 2011.

[20] A. Biere, M. Heule, and H. van Maaren, *Handbook of Satisfiability.* IOS Press, 2009, vol. 185.

[21] C.-H. Lin, L. Liu, and S. Vasudevan, "Generating concise assertions with complete coverage," in *Proceedings of the 23rd ACM International Conference on Great Lakes Symposium on VLSI*. ACM, 2013, pp. 185–190.

[22] C.-H. Lin, L. Wan, and D. Chen, "C-Mine: Data mining of logic common cases for low power synthesis of better-than-worst-case designs," in *Proceedings of the 51st Annual Design Automation Conference*. ACM, 2014, pp. 1–6.

[23] C.-H. Lin, L. Wan, and D. Chen, "C-mine: Data mining of logic common cases for improved timing error resilience with energy efficiency," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2017.

[24] C.-H. Lin, S. Roy, C.-Y. Wang, D. Z. Pan, and D. Chen, "CSL: Coordinated and scalable logic synthesis techniques for effective NBTI reduction," in *Computer Design (ICCD), 2015 33rd IEEE International Conference on*. IEEE, 2015, pp. 236–243.

[25] K. Campbell, C.-H. Lin, and D. Chen, "Low-cost hardware architectures for mersenne modulo functional units," in *Design Automation Conference (ASP-DAC), 2018 23rd Asia and South Pacific*. IEEE, 2018.

[26] L.-C. Wang, M. S. Abadir, and N. Krishnamurthy, "Automatic generation of assertions for formal verification of powerpc microprocessor arrays using symbolic trajectory evaluation," in *Proceedings of the 35th annual Design Automation Conference*. ACM, 1998, pp. 534–537.

[27] A. Hekmatpour and A. Salehi, "Block-based schema-driven assertion generation for functional verification," in *Test Symposium, 2005. Proceedings. 14th Asian*. IEEE, 2005, pp. 34–39.

[28] A. Pnueli, "The temporal logic of programs," in *Foundations of Computer Science, 1977., 18th Annual Symposium on*. IEEE, 1977, pp. 46–57.

[29] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar, "A technique for invariant generation," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 113–127, 2001.

[30] S. Vijayaraghavan and M. Ramanathan, *A Practical Guide for SystemVerilog Assertions*. Springer Science & Business Media, 2005.

[31] G. S. Tseitin, "On the complexity of derivation in propositional calculus," in *Studies in Constrained Mathematics and Mathematical Logic*, 1968.

[32] D. A. Plaisted and S. Greenbaum, "A structure-preserving clause form translation," *Journal of Symbolic Computation*, vol. 2, no. 3, pp. 293–304, 1986.

[33] M. T. Oliveira and A. J. Hu, “High-level specification and automatic generation of IP interface monitors,” in *Proceedings of the 39th Annual Design Automation Conference.* ACM, 2002, pp. 129–134.

[34] K. Ravi and F. Somenzi, “High-density reachability analysis,” in *Proceedings of the 1995 IEEE/ACM International Conference on Computer-aided Design.* IEEE Computer Society, 1995, pp. 154–158.

[35] C. Barrett, D. Dill, and A. Stump, “Checking satisfiability of first-order formulas by incremental translation to sat,” in *Computer Aided Verification.* Springer, 2002, pp. 681–710.

[36] C. Wang, H. Kim, and A. Gupta, “Hybrid CEGAR: combining variable hiding and predicate abstraction,” in *Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on.* IEEE, 2007, pp. 310–317.

[37] A. Mishchenko et al., “ABC: A system for sequential synthesis and verification,” *URL http://www. eecs. berkeley. edu/˜ alanmi/abc*, 2011.

[38] “OpenCores,” *URL http://www.opencores.org.*

[39] C. Albrecht, “IWLS 2005 benchmarks,” *URL http://iwls.org/iwls2005/benchmarks.html*, 2015.

[40] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner et al., “Razor: A low-power pipeline based on circuit-level timing speculation,” in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on.* IEEE, 2003, pp. 7–18.

[41] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, and K. Flautner, “Razor: circuit-level correction of timing errors for low-power operation,” *IEEE Micro*, vol. 24, no. 6, pp. 10–20, 2004.

[42] K. Bowman, J. Tschanz, C. Wilkerson, S.-L. Lu, T. Karnik, V. De, and S. Borkar, “Circuit techniques for dynamic variation tolerance,” in *Proceedings of the 46th Annual Design Automation Conference.* ACM, 2009, pp. 4–7.

[43] A. B. Kahng, S. Kang, R. Kumar, and J. Sartori, “Slack redistribution for graceful degradation under voltage overscaling,” in *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific.* IEEE, 2010, pp. 825–831.

[44] J. Tschanz, K. Bowman, C. Wilkerson, S.-L. Lu, and T. Karnik, “Resilient circuits: enabling energy-efficient performance and reliability,” in *Proceedings of the 2009 International Conference on Computer-Aided Design.* ACM, 2009, pp. 71–73.

[45] S. G. Ramasubramanian, S. Venkataramani, A. Parandhaman, and A. Raghunathan, “Relax-and-retime: A methodology for energy-efficient recovery based design,” in *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE.* IEEE, 2013, pp. 1–6.

[46] L. Wan and D. Chen, “DynaTune: circuit-level optimization for timing speculation considering dynamic path behavior,” in *Proceedings of the 2009 International Conference on Computer-Aided Design.* ACM, 2009, pp. 172–179.

[47] B. Greskamp, L. Wan, U. R. Karpuzcu, J. J. Cook, J. Torrellas, D. Chen, and C. Zilles, “Blueshift: Designing processors for timing speculation from the ground up.” in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on.* IEEE, 2009, pp. 213–224.

[48] Y. Liu, R. Ye, F. Yuan, R. Kumar, and Q. Xu, “On logic synthesis for timing speculation,” in *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on.* IEEE, 2012, pp. 591–596.

[49] O. Maimon and L. Rokach, *Data Mining and Knowledge Discovery Handbook*, 2nd ed. Springer Publishing Company, Incorporated, 2010.

[50] D. Ding, X. Wu, J. Ghosh, and D. Z. Pan, “Machine learning based lithographic hotspot detection with critical-feature extraction and classification,” in *IC Design and Technology, 2009. ICICDT’09. IEEE International Conference on.* IEEE, 2009, pp. 219–222.

[51] K. L. McMillan, “Interpolation and sat-based model checking,” in *Computer Aided Verification.* Springer, 2003, pp. 1–13.

[52] C. E. Molnar, R. F. Sproull, and I. E. Sutherland, “Counterflow pipeline processor architecture,” Mountain View, CA, USA, Tech. Rep., 1994.

[53] R. Agrawal, R. Srikant et al., “Fast algorithms for mining association rules,” in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215, 1994, pp. 487–499.

[54] E. Takeda and N. Suzuki, “An empirical model for device degradation due to hot-carrier injection,” *IEEE Electron Device Letters*, vol. 4, no. 4, pp. 111–113, 1983.

[55] D. K. Schroder and J. A. Babcock, “Negative bias temperature instability: Road to cross in deep submicron silicon semiconductor manufacturing,” *Journal of applied Physics*, vol. 94, no. 1, pp. 1–18, 2003.

[56] V. Reddy, A. T. Krishnan, A. Marshall, J. Rodriguez, S. Natarajan, T. Rost, and S. Krishnan, “Impact of negative bias temperature instability on digital circuit reliability,” *Microelectronics Reliability*, vol. 45, no. 1, pp. 31–38, 2005.

[57] W. Wang, S. Yang, S. Bhardwaj, R. Vattikonda, S. Vrudhula, F. Liu, and Y. Cao, “The impact of NBTI on the performance of combinational and sequential circuits,” in *Proceedings of the 44th Annual Design Automation Conference.* ACM, 2007, pp. 364–369.

[58] S. V. Kumar, C. H. Kim, and S. S. Sapatnekar, “NBTI-aware synthesis of digital circuits,” in *Proceedings of the 44th Annual Design Automation Conference*, ser. DAC ’07, 2007, pp. 370–375.

[59] M. Ebrahimi, F. Oboril, S. Kiamehr, and M. B. Tahoori, “Aging-aware logic synthesis,” in *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on.* IEEE, 2013, pp. 61–68.

[60] K.-C. Wu and D. Marculescu, “Joint logic restructuring and pin reordering against NBTI-induced performance degradation,” in *Proceedings of the Conference on Design, Automation and Test in Europe.* European Design and Automation Association, 2009, pp. 75–80.

[61] K.-C. Wu and D. Marculescu, “Aging-aware timing analysis and optimization considering path sensitization,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011.* IEEE, 2011, pp. 1–6.

[62] S. Roy and D. Z. Pan, “Reliability aware gate sizing combating NBTI and oxide breakdown,” in *VLSI Design and 2014 13th International Conference on Embedded Systems, 2014 27th International Conference on.* IEEE, 2014, pp. 38–43.

[63] R. Vattikonda, W. Wang, and Y. Cao, “Modeling and minimization of PMOS NBTI effect for robust nanometer design,” in *Proceedings of the 43rd annual Design Automation Conference.* ACM, 2006, pp. 1047–1052.

[64] X. Yang and K. Saluja, “Combating NBTI degradation via gate sizing,” in *Quality Electronic Design, 2007. ISQED’07. 8th International Symposium on.* IEEE, 2007, pp. 47–52.

[65] K. Keutzer, “DAGON: technology binding and local optimization by dag matching,” in *Design Automation, 1987. 24th Conference on.* IEEE, 1987, pp. 341–347.

[66] Y. Kukimoto, R. K. Brayton, and P. Sawkar, “Delay-optimal technology mapping by dag covering,” in *Proceedings of the 35th annual Design Automation Conference.* ACM, 1998, pp. 348–351.

[67] S. Chatterjee, A. Mishchenko, R. K. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping," in *2005 International Conference on Computer-Aided Design, ICCAD 2005, San Jose, CA, USA, November 6-10, 2005*, 2005, pp. 519–526.

[68] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, "Logic decomposition during technology mapping," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 8, pp. 813–834, 1997.

[69] L. Stok, M. A. Iyer, and A. J. Sullivan, "Wavefront technology mapping," in *Proceedings of the conference on Design, automation and test in Europe.* ACM, 1999, p. 108.

[70] M. M. Ozdal, C. Amin, A. Ayupov, S. Burns, G. Wilke, and C. Zhuo, "The ISPD-2012 discrete cell sizing contest and benchmark suite," in *Proceedings of the 2012 ACM international symposium on International Symposium on Physical Design.* ACM, 2012, pp. 161–164.

[71] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 12, pp. 1377–1394, 2002.

[72] A. Chakraborty and D. Z. Pan, "Skew management of NBTI impacted gated clock trees," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 6, pp. 918–927, 2013.

[73] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," 1992. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1992/2010.html

[74] Device Group at Arizona State University, "Predictive technology model," *URL http://ptm.asu.edu/.*

[75] C.-W. Chang, C.-K. Cheng, P. Suaris, and M. Marek-Sadowska, "Fast post-placement rewiring using easily detectable functional symmetries," in *Proceedings of the 37th Annual Design Automation Conference.* ACM, 2000, pp. 286–289.

[76] L. H.-H. Kelin, L. Klas, B. Mounaim, R. Prasanthi, I. R. Linscott, U. S. Inan, and M. Subhasish, "Leap: Layout design through error-aware transistor positioning for soft-error resilient sequential cell design," in *Reliability Physics Symposium (IRPS), 2010 IEEE International.* IEEE, 2010, pp. 203–212.

[77] K. Lilja, M. Bounasser, S.-J. Wen, R. Wong, J. Holst, N. Gaspard, S. Jagannathan, D. Loveless, and B. Bhuva, "Single-event performance and layout optimization of flip-flops in a 28-nm bulk technology," *IEEE Transactions on Nuclear Science*, vol. 60, no. 4, pp. 2782–2788, 2013.

[78] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner et al., "Razor: A low-power pipeline based on circuit-level timing speculation," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on.* IEEE, 2003, pp. 7–18.

[79] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, and K. Flautner, "Razor: circuit-level correction of timing errors for low-power operation," *IEEE Micro*, vol. 24, no. 6, pp. 10–20, 2004.

[80] K. A. Campbell, P. Vissa, D. Z. Pan, and D. Chen, "High-level synthesis of error detecting cores through low-cost modulo-3 shadow datapaths," in *Proceedings of the 52Nd Annual Design Automation Conference*, ser. DAC '15. ACM, 2015, pp. 161:1–161:6.

[81] S. Wei, "Residue checker using optimal signed-digit adder tree for error detection of arithmetic circuits," in *TENCON 2014-2014 IEEE Region 10 Conference.* IEEE, 2014, pp. 1–6.

[82] S. Wei, "Computation of modular multiplicative inverses using residue signed-digit additions," in *SoC Design Conference (ISOCC), 2016 International.* IEEE, 2016, pp. 85–86.

[83] J. Von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automata Studies*, vol. 34, pp. 43–98, 1956.

[84] J. Tryon, "Quadded logic," *Redundancy Techniques for Computing Systems*, pp. 205–228, 1962.

[85] E. Cheng, S. Mirkhani, L. G. Szafaryn, C.-Y. Cher, H. Cho, K. Skadron, M. R. Stan, K. Lilja, J. A. Abraham, P. Bose et al., "CLEAR: Cross-layer exploration for architecting resilience - combining hardware and software techniques to tolerate soft errors in processor cores," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE.* IEEE, 2016, pp. 1–6.

[86] M. Nicolaidis, "Carry checking/parity prediction adders and ALUs," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 11, no. 1, pp. 121–128, Feb. 2003.

[87] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, "Algorithm-based fault tolerance applied to high performance computing," *Journal of Parallel and Distributed Computing*, vol. 69, no. 4, pp. 410–416, 2009.

[88] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518–528, June 1984.

[89] S. J. Piestrak, F. Pedron, and O. Sentieys, "VLSI implementation and complexity comparison of residue generators modulo 3," in *Signal Processing Conference (EUSIPCO 1998), 9th European.* IEEE, 1998, pp. 1–4.

[90] S. J. Piestrak, "Design of residue generators and multioperand modular adders using carry-save adders," *IEEE Trans. Computers*, vol. 43, no. 1, pp. 68–77, 1994.

[91] S. Wei and K. Shimizu, "Residue checker with signed-digit arithmetic for error detection of arithmetic circuits," *Journal of Circuits, Systems, and Computers*, vol. 12, no. 01, pp. 41–53, 2003.

[92] N. Nedjah and L. M. Mourelle, "Three hardware architectures for the binary modular exponentiation: sequential, parallel, and systolic," *IEEE Trans. Circuits and Systems I: Regular Papers*, vol. 53, no. 3, pp. 627–633, March 2006.

[93] H. S. Warren, *Hacker's Delight.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.