Synthesis and Verification of Digital Circuits using Functional Simulation and Boolean Satisfiability

by

Stephen M. Plaza

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2008

Doctoral Committee:

Associate Professor Igor L. Markov, Co-Chair
Assistant Professor Valeria M. Bertacco, Co-Chair
Professor John P. Hayes
Professor Karem A. Sakallah
Associate Professor Dennis M. Sylvester

To my family, friends, and country

ACKNOWLEDGEMENTS

I would like to thank my advisers, Professor Igor Markov and Professor Valeria Bertacco, for inspiring me to consider various fields of research and providing feedback on my projects and papers. I also want to thank my defense committee for their comments and insights: Professor John Hayes, Professor Karem Sakallah, and Professor Dennis Sylvester. I would like to thank Professor David Kieras for enhancing my knowledge and appreciation for computer programming and providing invaluable advice.

Over the years, I have been fortunate to know and work with several wonderful students. I have collaborated extensively with Kai-hui Chang and Smita Krishnaswamy and have enjoyed numerous research discussions with them and have benefited from their insights. I would like to thank Ian Kountanis and Zaher Andraus for our many fun discussions on parallel SAT. I also appreciate the time spent collaborating with Kypros Constantinides and Jason Blome. Although I have not formally collaborated with Ilya Wagner, I have enjoyed numerous discussions with him during my doctoral studies. I also thank my office mates Jarrod Roy, Jin Hu, and Hector Garcia.

Without my family and friends I would never have come this far. I would like to thank Geoff Blake and Smita Krishnaswamy, who have been both good friends and colleagues and who have talked to me often when the stress at work was overwhelming. I also want to thank Geoff for his patience being my roommate for so many years. I am blessed to

also have several good friends outside of the department who have provided me a lot of support: Steve Kibit (Steve^2 representin'), Rob Denis, and Jen Pileri.

Most of all, I would like to thank my family who has been an emotional crutch for me. My mom, dad, and brother Mark have all been supportive of my decision to go for a PhD and have continuously encouraged me to strive for excellence.

PREFACE

The semiconductor industry has long relied on the steady trend of transistor scaling, that is, the shrinking of the dimensions of silicon transistor devices, as a way to improve the cost and performance of electronic devices. However, several design challenges have emerged as transistors have become smaller. For instance, wires are not scaling as fast as transistors, and delay associated with wires is becoming more significant. Moreover, in the design flow for integrated circuits, accurate modeling of wire-related delay is available only toward the end of the design process, when the physical placement of logic units is known. Consequently, one can only know whether timing performance objectives are satisfied, *i.e.,* if timing closure is achieved, after several design optimizations. Unless timing closure is achieved, time-consuming design-flow iterations are required. Given the challenges arising from increasingly complex designs, failing to quickly achieve timing closure threatens the ability of designers to produce high-performance chips that can match continually growing consumer demands.

In this dissertation, we introduce powerful constraint-guided synthesis optimizations that take into account upcoming timing closure challenges and eliminate expensive design iterations. In particular, we use logic simulation to approximate the behavior of increasingly complex designs leveraging a recently proposed concept, called *bit signatures*, which allows us to represent a large fraction of a complex circuit's behavior in a com-

pact data structure. By manipulating these signatures, we can efficiently discover a greater set of valid logic transformations than was previously possible and, as a result, enhance timing optimization. Based on the abstractions enabled through signatures, we propose a comprehensive suite of novel techniques: (1) a fast computation of circuit don't-cares that increases restructuring opportunities, (2) a verification methodology to prove the correctness of speculative optimizations that efficiently utilizes the computational power of modern multi-core systems, and (3) a physical synthesis strategy using signatures that re-implements sections of a critical path while minimizing perturbations to the existing placement. Our results indicate that logic simulation is effective in approximating the behavior of complex designs and enables a broader family of optimizations than previous synthesis approaches.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

LIST OF ABBREVIATIONS

2-SAT	2-SATisfiability (Satisfiability instance where each clause has at most two literals)
ABC	Software package for logic synthesis and verification from UC Berkeley
AI	Artificial Intelligence
AIG	And-Inverter Graph
AMD	Advanced Micro Devices
AT	Arrival Time
ATPG	Automatic Test Pattern Generation
BCP	Boolean Constraint Propagation
BDD	Binary Decision Diagram
C(b)	Care set of b
CAD	Computer Aided Design
CNF	Conjunctive Normal Form
CODC	Compatible Observability Don't Care
CPU	Central Processing Unit
D2M	Delay with 2 Moments
DC	Design Compiler (Synopsys synthesis tool) or Don't Care
DC(b)	Don't Care set of b
DPLL	Davis-Putnam-Logemann-Loveland (satisfiability algorithm)
EDA	Electronic Design Automation
FLUTE	A software package from Iowa State University implementing fast RSMT construction. It is based on lookup tables.
FPGA	Field-Programmable Gate Array
GB	GigaByte
GHz	GigaHertz
GSRC	Gigascale Systems Research Center
GTECH	Generic TECHnology library (Synopsys)
HPWL	Half-Perimeter WireLength
IC	Integrated Circuit
ITRS	International Technology Roadmap for Semiconductors

IWLS	International Workshop on Logic and Synthesis
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
MUX	MUltipleXer
NMF	Non-Monotonic Factor
ODC	Observability Don't Care
OFFSET(b)	Set of input combinations where $b = 0$
ONSET(b)	Set of input combinations where $b = 1$
OS	Operating System
PBD	Pairs of Bits to be Distinguished
RAR	Redundancy Addition and Removal
RSMT	Rectilinear Steiner Minimal Tree
RTL	Register Transfer Level
SAT	SATisfiability (problem) or SATisfiable (solution)
SDC	Satisfiability Don't Care
SMP	Symmetric MultiProcessing
SPFDs	Sets of Pairs of Functions to be Distinguished
SSF	Single-Stuck-at Fault
STA	Static Timing Analysis
UIP	Unique Implication Point
UNSAT	UNSATisfiable
U-SAT	Unique-SATisfiable problem (only one solution)
VSIDS	Variable State Independent Decaying Sum

Part I

Introduction and Background

CHAPTER I

Introduction: Overcoming Challenges in Nanometer Design

1.1 Trends in the Electronics and EDA Industries

The performance capabilities of computer chips continue to increase rapidly. This, in turn, is driving the technology evolution in many different application domains, such as gaming and scientific computing. A major impetus for this growth is consumer demand, which seeks the smallest, fastest, and coolest devices. Consumer demand guides performance objectives and pressures computer companies to meet time-to-market expectations. Failure to meet these expectations can result in the loss of competitive advantage. For example, in 2006 Sony postponed the release date of the PlayStation 3 console by six months due to technical problems, exposing Sony's gaming market share to competing consoles by Microsoft and Nintendo.

Many applications depend on the predictability of improvements to integrated circuits.

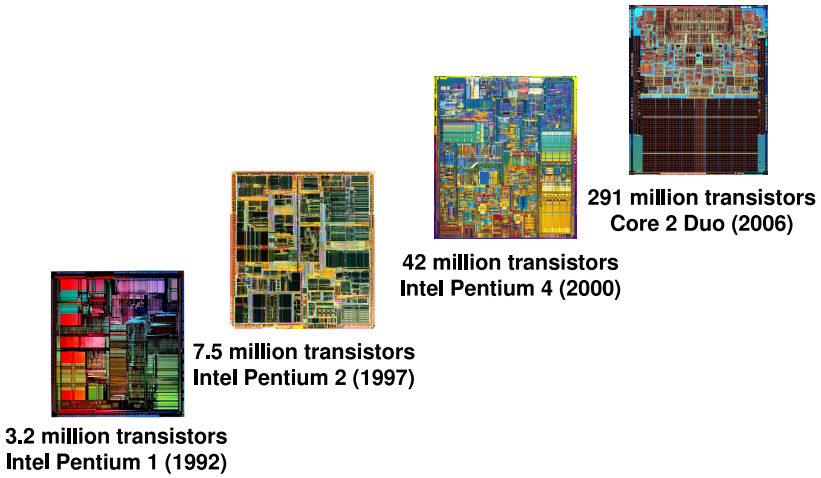

Figure 1.1: Transistors manufactured on a single chip over several generations of Intel CPUs.

As shown in Figure 1.1, the number of transistors on a chip has been steadily increasing during the past 40 years. As a result, the Core 2 Duo CPU has almost one hundred times more transistors than the Pentium CPU 14 years ago. These scaling (and performance) trends have been made possible by advances in device manufacturing, which have resulted in the fabrication of smaller transistors. Transistor sizes are determined by the minimum size of a geometrical feature (usually a rectangle) that can be manufactured at a given technology node. Figure 1.2 illustrates the decreasing transistor size, where the physical length L_{GATE} of the transistor's gate is currently at 50nm and expected to shrink to 15nm as manufacturing techniques continue to improve. This scaling trend was observed by Gordon Moore in 1965, when he projected that the number of transistors that fit in an integrated circuit would double every two years [64], corresponding to an exponential growth. With more transistors, entire systems that were once implemented across a computer board, now fit on a single chip. More recently, this trend has made it possible to pack multiple processors in the same chip, so-called multi-core processors. Multi-core processors are now

mainstream in the mass-market desktop computing domain, unlocking the performance wall that single core microprocessors had reached, and allowing multiple applications to run in parallel on the same desktop system.

1.2 Challenges in High-Performance Integrated Circuit Design

Ensuring continued performance improvements has become more challenging as transistors reach the nanometer scale. First, the complexity of integrated circuits has already exceeded the capability of designers to optimize[1] and verify their functionality. In design processes, verifying design correctness is a major component that affects time-to-market. Also, buggy designs released to the consumer market can significantly impact revenue as evidenced by Intel's floating-point division bug [100] and, more recently, by a bug in AMD's quad-core Phenom processor [99]. Second, the miniaturization of transistors to the atomic scale poses several challenges in terms of the variability in the manufacturing process, which leads to unpredictable performance. Third, the scaling of wire interconnect is not as pronounced as that of transistors. As transistors get faster and smaller, the width of wires decreases at a slower rate, and the per-unit resistance of wires may increase. Therefore, the advantages of having shorter wires are mitigated by the increase in time that it takes to propagate a signal. Consequently, an increasing percentage of chip area is necessary for wires, and the maximum clock frequency is primarily determined by wire lengths, rather than transistor switching time. Figure 1.3 illustrates the prevalence of interconnect on multiple metal layers on a chip. In this figure, two metal layers are shown with several wires and interlayer connectors called *vias*. This interconnect overshadows

[1] In this dissertation, *optimize* is often used to mean performing operations that improve some performance characteristic.

the polysilicon gates, which are a component of MOSFETs (transistors).

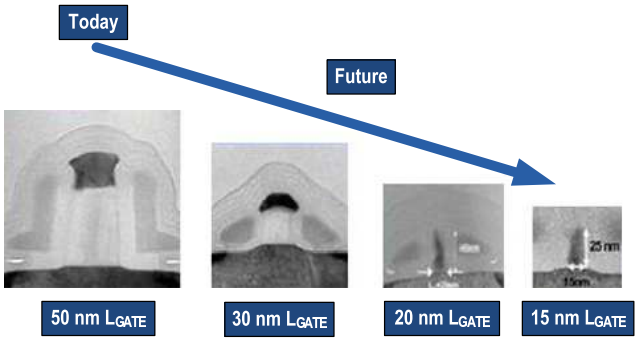

Figure 1.2: Transistor scaling projected at future technology nodes.

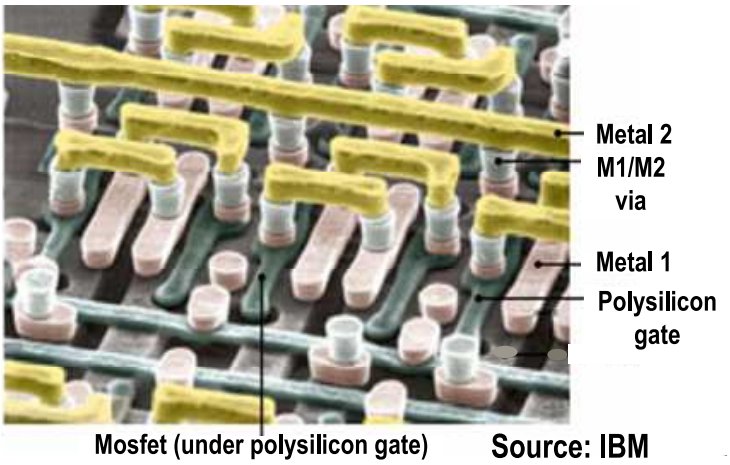

Figure 1.3: Major components of multilayer interconnect: single-layer wire segments and inter-layer connectors (vias).

Traditionally, a computer chip design entails a series of steps from high-level conceptualization to final chip fabrication. It is this design flow (shown in Figure 1.4) that must be able to address technology scaling. Starting from the top left of Figure 1.4, a design team specifies the desired functionality of the chip. The design team identifies the chip's

major components and designs each of them at a high level; this functionality may be expressed in a hardware description language, such as *SystemC*. Numerous optimizations are facilitated by the design team through the use of automated software tools. Eventually, the design description is translated into a register transfer level (RTL) description (top-right corner of Figure 1.4), which specifies a design in more detail. Through a process called *logic synthesis*, an RTL description is translated into a gate-level netlist as shown at the bottom right of the figure. In order to simplify transistor-level layout, multiple gates are mapped to pre-designed *standard cells*. This process is called *technology mapping*. At this point, an area estimate can be made based on the number of transistors required for the design. Also, one can estimate the fastest possible clock frequency for the chip based on transistor switching, since the number of cells that occur between the design's inputs and outputs is known. After a netlist is mapped to a set of cells, placement is performed. During *placement*, a physical location is given to each cell such that they do not overlap, and then wires connecting cells are routed. Finally, both wires and transistors are represented by polygons, and the resulting design description is sent for fabrication to obtain the final product, shown at the end of the flow in Figure 1.4.

Functional and physical verification are needed throughout the design flow. After performing RTL and netlist-level optimization, the design's outputs are checked against the expected behavior of the design. Physical verification ensures that *design rules* (such as maintaining minimum spacing between wires), as well as electrical and thermal constraints, are satisfied. Furthermore, at the end of the design flow before fabrication, the performance characteristics of the design are checked against desired goals and objectives. This process of meeting performance objectives is known as achieving design clo-

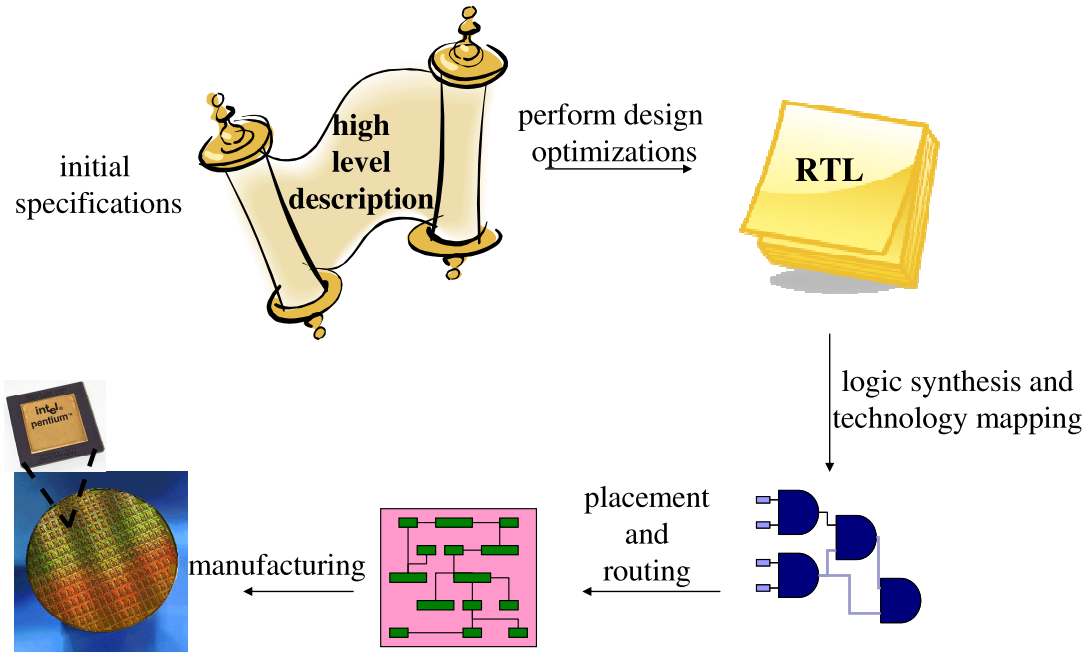

Figure 1.4: Typical integrated circuit design flow. The design flow starts from an initial design specification. Several optimization steps are performed, and then a final chip is manufactured.

sure. The process of ensuring that the circuit timing (delay) constraints are met is known as achieving *timing closure*.

Physical information about the design, such as cell locations and wire routes, is known only at the end of the flow. As previously noted, delay associated with wires is becoming more prominent, hence accurate timing estimates are known only when wire lengths are determined after the routing phase. However, most functional design optimizations are traditionally performed early at the RTL, as well as during logic synthesis and technology mapping. After several of these optimization steps, placement and routing might produce a modified design that no longer achieves timing closure. Hence, the inability to gather accurate timing and performance estimations early in the design flow leads to less flexibility in performing design optimizations. For instance, optimizing for a specific performance

metric after placement and routing, such as timing, can negatively affect the quality of other performance metrics. To avoid this tension between various design goals, late design flow optimizations are normally limited. Hence more heavyweight optimizations may necessitate the re-iteration of earlier steps in the design flow. In some cases, the number of design iterations required to achieve timing closure is prohibitive. Multiple iterations increase the turn-around-time, development costs, and time-to-market, while also resulting in a design that might fail to meet original expectations.

1.3 Bridging the Gap between Logic and Physical Optimizations

This dissertation develops powerful and efficient logic transformations that are applied late in the design flow to achieve timing closure. The transformations overcome the limitations of current methodology by 1) extracting and exploiting more circuit flexibility to improve performance late in the design flow and 2) minimizing the negative impact to other performance metrics. The goal is to eliminate costly design iterations and enable efficient use of multi-core processors, to overcome increased design complexity and scaling challenges.

To enable these transformations, our work leverages the principle of *abstraction* by temporarily discarding all aspects of circuit behavior not observed during a fast bit-parallel simulation. Under this abstraction, we can handle complex designs, pinpoint potential logic transformations that may lead to improvements in the design, and assess the quality of a wide range of transformations. In the following section, we discuss our abstraction technique and its components in more detail.

1.4 Using Simulation-based Abstractions for Circuit Optimizations

Key to our approach is the use of logic simulation to approximate the behavior of each node in a circuit through information known as a *bit signature* [50]. The functionality of a node in a circuit is defined by its truth table that specifies the node's output for all input combinations. A signature is a partial truth table selected by a (usually small) subset of the possible input combinations. Such a partial truth table can be viewed as an abstracted representation that can be exponentially smaller than a complete truth table, yet can accurately guide optimizations as we show throughout this dissertation. Because of the efficiency of logic simulation, approximating circuit behavior scales linearly with the number of nodes in the circuit, and consequently it can tackle large circuits. While such signatures have already been used in the literature, these pre-existing techniques suffer from a number of limitations.

Summary of related work. The effectiveness of logic simulation has been demonstrated in terms of its ability to distinguish different nodes in a circuit [50, 59]. Consequently, signatures can be used in both logic optimization and verification. With respect to verification, the correctness of a design can be ascertained up to the abstraction by comparing its output signature to the corresponding output of a functionally correct design, also known as a *golden model*. Design optimizations are also enabled by signatures because equivalent nodes in a circuit can be merged to simplify the design [59]. Furthermore, the signature representation is amenable to simple transformations [18], that can generate new signatures and that, in turn, can be mapped to logic optimizations on the actual design.

Key aspects and limitations of signature-based abstractions. When signatures are used, optimization and verification are correct only with respect to the abstraction. A for-

mal proof mechanism is often required to verify the correctness of the abstraction. Formal proof engines, such as SAT solvers, invariably have exponential worst-case runtimes. This lack of scalability is particularly problematic as design complexity grows. Since formal proof mechanisms are typically based on hard-to-parallelize algorithms, it is difficult to efficiently utilize the resources offered by recent multi-core CPUs.

Generating high-quality signatures is paramount so as to avoid incorrect characterizations and to minimize the number of invocations of expensive proof mechanisms. The quality of a signature rests in its ability to capture both typical-case behaviors and important corner-behaviors, while being occasionally refined through formal techniques and additional simulation patterns. In [59], signatures are refined to improve their distinguishing capabilities in finding equivalent nodes in a design. Despite the efficiency of generating signatures, ensuring their high *quality* in very large designs with complex hierarchical components is a major challenge. In this scenario, if a signature abstraction is desired for a component far from the primary inputs of a design, the limited *controllability* of this component undermines the quality of the signature generated and its ability to expose interesting behavior in that component. Furthermore, previous works do not consider a node's *downstream* logic information when characterizing its behavior with a signature, and therefore fail to exploit logic flexibilities present in large designs.

Finally, a general methodology *for performing design optimizations with signatures* has not yet been developed. As we show in this dissertation, signatures simplify the search for logic transformations and thus facilitate powerful gate-level optimizations that involve both logic and physical design aspects. Such optimizations are known as physical synthesis. However, conventional strategies for synthesis are inadequate in exploiting the runtime

savings and optimization potential of signature-based synthesis. This dissertation presents the first generalized solution achieving this goal.

1.5 Components of Our Simulation-based Framework

To enable complex transformations that can be applied late in the design flow to achieve timing closure, we introduce a series of improvements to signature-based abstractions that overcome previous limitations. The new elements of our simulation-based framework, developed throughout the dissertation, not only result in better optimizations, but improve the quality of verification efforts. We now outline our major contributions:

- A high-coverage verification engine for stimulating a component deep within a hierarchical design while satisfying constraints. Our strategy relies on a simulation engine for high performance, while improving the fidelity of signatures and verification coverage.

- An efficient linear-time *don't-care* analysis to extract potential flexibility in synthesizing each node of a circuit and to enhance the corresponding signatures.

- A technique to improve the efficiency of the formal proof mechanism in verifying the equivalence between a design and its abstraction enhanced by don't-cares.

- A strategy to improve the efficiency of verifying abstractions by exploiting parallel computing resources such as the increasingly prevalent multi-core systems. Harnessing these parallel resources is one mechanism to partially counteract the increasing costs of verification and to enable deployment of our signature-based optimizations on more complex designs.

- A goal-driven synthesis strategy that quickly evaluates different logic implementations leveraging signatures.

- A constraint-guided synthesis algorithm using signatures to improve physical performance metrics, such as timing.

1.6 Organization of the Dissertation

In this dissertation, we introduce several algorithmic components that enhance our signature-based abstraction. We then leverage these components in logic and physical synthesis to enable powerful optimizations, where traditional techniques perform poorly. Throughout the chapters of this dissertation, we gradually extend the scope and power of signature-based optimizations. In Part II, we propose techniques that enhance signatures by generating better simulation vectors that activate parts of the circuit in a design and by encoding information on logic flexibility in the signatures. In Part III, we develop verification strategies that mitigate the runtime costs of verifying the correctness of the abstraction. In Part IV, we utilize our enhanced signatures and verification strategies to enable design optimizations by manipulating these signatures. The rest of the dissertation is structured as follows:

- For the remainder of Part I, we provide background material necessary to navigate through this dissertation.

  - Chapter II covers background in logic synthesis, verification, and logic simulation. We outline recently-discovered synergies between these tasks and explain how this dissertation builds upon these synergies.

  - Chapter III describes the evolution of the design flow to address timing closure and to survey previous work in late design flow optimization.

- In Part II, we introduce strategies to improve the quality and strength of signatures.

  - Chapter IV introduces the notion of entropy for identifying parts of a design that experience low simulation coverage. We then develop a strategy for improving simulation of these regions using a SAT solver. This approach is useful for stimulating internal components in complex hierarchies. In particular, it helps in exposing bugs in corner-case behaviors.

  - Chapter V introduces a technique for the efficient extraction of global circuit don't-cares based on a linear-time analysis and encodes them in signatures.

- Part III introduces strategies to counteract the complexity of verifying signature-based abstractions in increasingly large designs.

  - Chapter VI describes an incremental approach to verify a given abstraction up to the derived don't-cares and to refine it by generating additional signatures.

  - Chapter VII introduces techniques to address the growing complexity of formal verification by exploiting the increasing availability of multi-core systems which can execute several threads simultaneously. We develop a parallel-SAT solving methodology that consists of a priority-based scheduler for handling multiple problem instances of varying complexity in parallel and a lightweight strategy for handling single instances of high complexity.

- Part IV introduces techniques for performing logic manipulations using signatures.

  - Chapter VIII describes how signatures can be exploited to enable powerful synthesis transformations. In particular, we show how node merging up to don't-cares can greatly simplify a circuit. Then, we introduce a new general approach for performing logic synthesis using signatures.
  - Chapter IX proposes a path-based resynthesis algorithm that finds and shortens critical paths with wire bypasses. We apply our path-based resynthesis after placement, when better timing estimates are available, and we report significant improvements, indicating that current design flows still leave many optimization opportunities unexplored.

- The dissertation is concluded in Chapter X with a summary of contributions and an outline of future research directions.

CHAPTER II

Synergies between Synthesis, Verification, and Functional Simulation

In the traditional design flow for integrated circuits, logic synthesis and verification play a critical role in ensuring that integrated circuit parts released to the market are functionally correct and achieve the specified performance objectives. Logic synthesis generates circuit netlists and transforms them to improve area and delay characteristics. These transformations are carried forward by software used by circuit designers. To ensure the correctness of these transformations, along with custom-made optimizations, verification is typically performed over multiple steps in the design flow, where the actual behavior of the circuit is verified against the desired behavior. Traditionally, synthesis and verification are considered separate and independent tasks; however, recent research [59, 63, 95] has exposed a number of common traits and synergies between synthesis and verification. Functional verification often involves algorithms whose worst-case runtime complexity grows exponentially with design size. However, the design size can be reduced through synthesis optimizations, typically reducing the verification effort. Logic simulation has also been employed to improve verification, and more recently, to enable netlist simplifications [95].

A major contribution of this dissertation is its in-depth exploration of synergies be-

tween synthesis and verification, as well as the gains that can be derived by integrating the two tasks through simulation techniques. Our goal is to improve the quality of results and the scalability of both practices, which are continually challenged by increasing design complexity. In particular, we introduce speculative transformations that require verification, a major departure from traditional correct-by-synthesis techniques, typically employed today. In the remainder of this chapter, we discuss previous work in verification, synthesis and logic simulation, focusing on strategies to improve their scalability.

2.1 Scalable Verification

Verifying the functional correctness of a design is a critical aspect of the design flow. Typically, comprehensive verification methodologies [96, 97] are employed and require a team of specialized verification engineers to construct test cases that exercise the functionality of the circuit. The output of this circuit is usually compared against an ideal *golden model*. To reduce the demands on the verification engineer in exposing interesting design behavior through test cases, input stimuli can be automatically refined or modified, leading to improvement in the verification coverage. For example, at the instruction level, Markov models can be used [83] to produce instruction sequences that effectively stimulate certain parts of the design. However, explicit monitors are necessary to guide this refinement, thus still requiring detailed understanding of the design. At the gate-level, simulation can also be refined [59] to help distinguish nodes, but this is primarily useful for equivalence checking. The goal of all these procedures is to generate test cases that can expose corner-case behavior in the circuit. In Chapter IV, we discuss how simulation coverage is improved automatically, without requiring any detailed understanding of the design.

Because generating exhaustive test cases is infeasible and releasing a buggy design

is undesirable, formal verification techniques can be used to achieve higher verification coverage. However, the limited scalability of formal techniques is a major bottleneck in handling increasingly complex designs. Therefore, a combination of intelligently chosen test suites and formal techniques on small components is often adopted to maximize verification coverage.

One prominent formal proof mechanism particularly relevant to this work is equivalence checking. In equivalence checking, the output response of a design is compared against a golden model for all legal input combinations. If the response is always the same, the designs are said to be equivalent. Often, binary decision diagrams (BDDs) can be used to check the equivalence between two combinational circuits. A BDD [14] is a data structure that can often efficiently represent a circuit in a canonical way, so that checking equivalence means building this canonical form for both designs. However, the number of nodes in a BDD can be exponential with respect to the number of inputs, thus limiting the scalability of the approach. Satisfiability-based equivalence checking techniques have been developed [13] as an alternative to BDDs. Despite having exponential worst-case runtime, SAT-based techniques typically have lower memory requirements and successfully extend to larger designs. Below we provide the background on satisfiability necessary to navigate this dissertation. Then we examine previous attempts to scale the performance of SAT solvers by exploiting multiple processing units concurrently.

2.1.1 Satisfiability

The SAT problem entails choosing an assignment V for a set of variables that satisfies a Boolean equation or discovering that no such assignment exists [76]. The Boolean equation is expressed in conjunctive normal form (CNF), $F = (a + b' + c)(d' + e)....$ — a

conjunction of *clauses*, where a clause is a disjunction of literals. A literal is a Boolean variable or its complement. For instance, $(a+b'+c)$ and $(d'+e)$ are clauses, and a, b', c, d', e are literals.

A Framework for Solving SAT

A common approach to solving SAT is based on the branch-and-backtrack DPLL algorithm [24]. Several innovations, such as non-chronological backtracking, conflict-driven learning, and decision heuristics greatly improve upon this approach [65, 80, 88]. The essential procedural components of a SAT solver are outlined in the pseudo-code of Figure 2.1.

```
search {
  while(true) {
   propagate();
   if(conflict) {
    analyze_conflict();
    if(top_level_conflict) return UNSAT;
    backtrack();
   }
   else if(satisfied) return SAT;
   else decide();
  }
}
```

Figure 2.1: Pseudo-code of the search procedure used in DPLL-SAT. The procedure terminates when it either finds a satisfying assignment or proves that no such solution exists.

The `search()` function explores the decision tree until a satisfying assignment is found or the entire solution space is traversed without finding any such assignment. The `decide()` function selects the next variable for which a value is chosen. Many methods exist for selecting this "decision" variable, such as the VSIDS (Variable State Independent Decaying Sum) algorithm developed in Chaff [65]. VSIDS involves associating an

activity counter with each literal. Whenever a new learnt clause is generated (see below) from conflict analysis, the counter of each literal in that clause is incremented while all other variables undergo a small decrease. The `propagate()` function performs Boolean Constant Propagation (BCP), *i.e.*, it identifies the clauses that are still unsatisfied and for which only one literal is still unassigned, and then assigns the literal to the only value that can satisfy the clause. If the decision assignment implies a contradiction or conflict, the `analyze_conflict()` function produces a *learnt* clause, which records the cause of the conflict to prevent the same conflicting sets of assignments. The `backtrack()` function undoes the earlier assignments that contributed to the conflict. Periodically, the `search()` function is restarted: all current assignments are undone, and the search process starts anew using a random factor in restarting the decision process so that different parts of the search space are explored. Extensive empirical data shows the effectiveness of restarts in boosting the performance of SAT solvers by minimizing the exploration in computation-intensive search paths [9, 65].

Learning

In this part, we consider two types of learning performed to reduce SAT search space: preprocessing and conflict-driven learning.

Preprocessing. The goal of preprocessing a SAT instance is to simplify the instance by adding implied constraints that reduce propagation costs: by eliminating variables, by adding symmetry breaking clauses, or by removing clauses that are subsumed by others. Preprocessing has led to improved runtime in solving several instances, although the computational effort sometimes outweighs the benefit. A recent preprocessor, SatELite [28], achieves significant simplifications through the efficient implementations of variable

elimination and subsumption.

Conflict-driven learning. Dynamic learning is important to prevent repeated exploration in similar parts of the search space. When a set of assignments results in a conflict, the conflict analysis procedure in SAT determines the cause by analyzing a *conflict graph*. In Figure 2.2, we show an example of a SAT instance and a set of assignments that result in a conflicting assignment for variable v. Each decision ($f = 1$, $g = 1$, and $a = 1$) is depicted by a different leaf node, and *implications* of these decisions, are shown as internal nodes in the graph. An implication occurs when a set of variable decisions forces an unassigned variable to be assigned 0 or 1. A *decision level* is associated with each node (nodes at the same level are denoted by the same color in Figure 2.2), which is the set of variable assignments implied by a decision. For instance, the second decision level consists of the second decision ($g = 1$), and the implications $k = 0$ and $m = 0$.

A learnt clause can be derived in a conflict graph from a cut that crosses every path from the leaf decision values to the conflict exactly once. The nodes to the left of the cut are on the *reason side* and those on the right are on the *conflict side*. `Cut 1` in the figure shows the *1-UIP* (first unique implication point) cut, *i.e.*, the cut closest to the conflict side. In this cut, the reason side contains one node of the last decision level (node a) that dominates all other nodes at the same decision level and on the same (reason) side. The assignment $e = 1, f = 1, k = 0, m = 1$ is determined to be in conflict and hence $(e' + f' + k + m)$ can be added to the original CNF to prevent this assignment in the future. `Cut 2` indicates the 2-UIP cut where the reason side contains one node in the previous decision level (level 2) that dominates every other node in that decision level. Here, the 2-UIP learnt clause $(e' + f' + g')$ can be added.

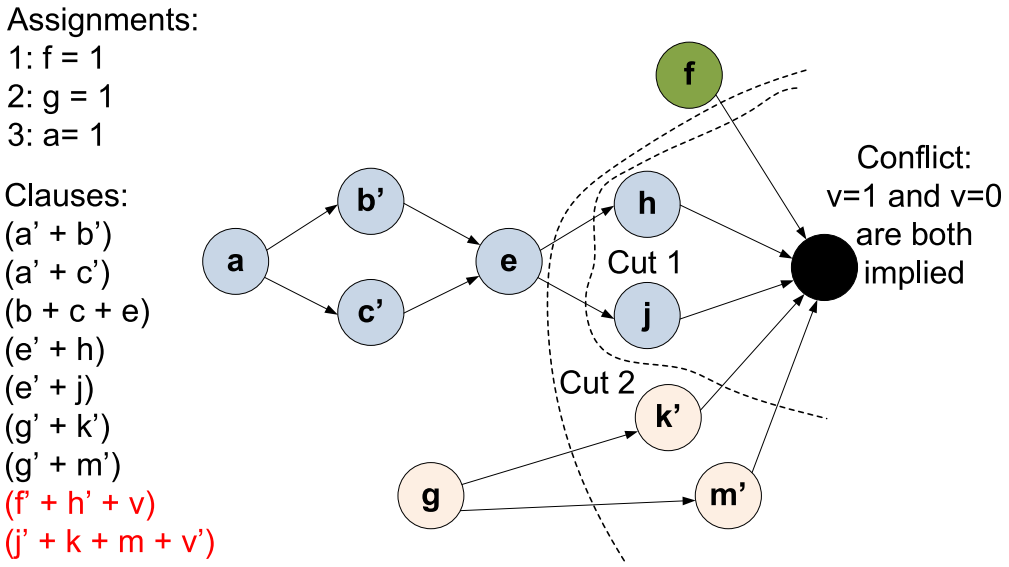

Figure 2.2: An example conflict graph that is the result of the last two clauses in the list conflicting with the current assignment. We show two potential learnt clauses that can be derived from the illustrated cuts. The dotted line closest to the conflict represents the 1-UIP cut, and the other is the 2-UIP cut.

A learning strategy commonly employed adds only the 1-UIP learnt clause for every conflict. Despite the possibility of using smaller learnt clauses that technically prune larger parts of the search space, it has been empirically shown in [90] that 1-UIP learning is most effective. In [27], 1-UIP was shown to be more effective at pruning the search space because the corresponding backtracking more often satisfied learnt clauses generated by other UIP cuts.

2.1.2 Previous Parallel SAT Approaches

To boost the performance of SAT solvers on increasingly prevalent parallel architectures, parallel SAT solving strategies have explored coarse-grain and fine-grain parallelization. Fine-grain parallelization strategies target Boolean Constraint Propagation (BCP) — the runtime bottleneck for most SAT solvers. In BCP, each variable assignment is checked

against all relevant clauses, and any implications are propagated. BCP can be parallelized by dividing the clause database among n different solvers so that BCP computation time of each solver is approximately $\frac{1}{n}$ the original. Coarse-grain parallelization strategies typically involve assigning a SAT solver to different parts of the search space.

Fine-grain parallelization. The performance of fine-grain parallelization depends on the partitioning of clauses among the solvers, where an ideal partition ensures an even distribution of BCP costs while minimizing the implications that need to be communicated between each solver. This strategy also requires low-latency inter-solver communication to minimize contention for system locks on general microprocessors, which can exacerbate runtime performance. Therefore, fine-grain parallelization has been examined on specialized architectures [91] that can minimize communication bottlenecks. In [2, 92], significant parallelization was achieved by mapping a SAT instance to an FPGA and allowing BCP to evaluate several clauses simultaneously. However, the flexibility and scalability of this approach is limited, since each instance needs to be compiled to the specific FPGA architecture (a non-trivial task), and conflict-driven learning is difficult to implement effectively in hardware because it requires dynamic data structures.

Coarse-grain parallelization. The runtime of an individual problem can also be improved with parallel computation by using a solver portfolio [38], where multiple SAT heuristics are executed in parallel and the fastest heuristic determines the runtime for the problem. A solver portfolio also offers a way of countering the variability that backtrack-style SAT solvers experience on many practical SAT instances [39]. Because one heuristic may perform better than another on certain types of problems, one can reduce the risk of choosing the wrong heuristic by running both. Although parallelization here consists of

running multiple versions of the same problem simultaneously, if the runtime difference between these heuristics is significant, a solver portfolio can yield runtime improvements.

However, using a portfolio solver does not guarantee high resource utilization as each heuristic may perform similarly on any given instance or one heuristic may dominate the others. The primary limitation of solver portfolios is that there is no good mechanism to coordinate the efforts of these heuristics and the randomness inherent to them. Other approaches consider analyzing different parts of the search space in parallel [22, 55, 66, 89]. If the parts of the search space are disjoint, the solution to the problem can be determined through the processing of these parts in isolation. However, in practice, the similarities that often exist between different parts of the search space mean that redundant analysis is performed across the different parts. To counter this, the authors in [55] develop an approach to explore disjoint parts of the search space by relying on the shared memory of multi-core systems to transfer learned information between them. The approach considers dividing the problem instance using different variable assignments called *guiding paths*, as originally described in [89]. One major limitation of this type of search space partitioning is that poor partitions can produce complex sub-problems with widely varying structure and complexity.

The benefits of learning between solvers working on different parts of the search space in parallel suggest potential super-linear improvement. However, the improvements achieved by current strategies seem more consistent with the inherent variability of solving many real-world SAT problems and the effect of randomization on reducing this variability. Through clever randomization strategies, sequential solvers can often avoid complex parts of the search space and outperform their parallel counterparts.

2.2 Scalable Logic Synthesis

Due to the development of powerful multi-level synthesis algorithms [74], scalable logic synthesis tools have been able to optimize increasingly large designs since the early 1990s. During logic optimization, different multi-level optimization strategies are interleaved and executed several times, including fast extraction (finding different decompositions of a node based on algebraic transformations) and node simplification (exploiting circuit don't-cares). These techniques are correct by construction, so that the resulting netlist is functionally equivalent to the original assuming no implementation errors are present in the synthesis tool. We outline several key aspects of improving the quality and scalability of synthesis below.

2.2.1 Don't Care Analysis

To enhance synthesis, circuit flexibility in terms of *don't-cares* can be exploited. Figure 2.3 provides examples of satisfiability don't-cares (SDCs) and observability don't-cares (ODCs). An SDC occurs when certain input combinations do not arise due to limited controllability. For example, the combination of $x = 1$ and $y = 0$ cannot occur for the circuit shown in Figure 2.3a. SDCs are implicitly handled when using SAT in validating the netlist because SDC input combinations cannot occur for any satisfying assignment. ODCs occur when the value of an internal node does not affect the outputs of the circuit because of its limited observability [26]. In Figure 2.3b, when $a = 0$ and $b = 0$, the output value of F is a don't-care.

Figure 2.4 shows a strategy for identifying ODCs for a node a. First, the circuit D is copied, and a is inverted in the copy D^*. Then an XOR-based *miter* [13] is constructed between the outputs of the two circuits. A miter is a single output function typically imple-

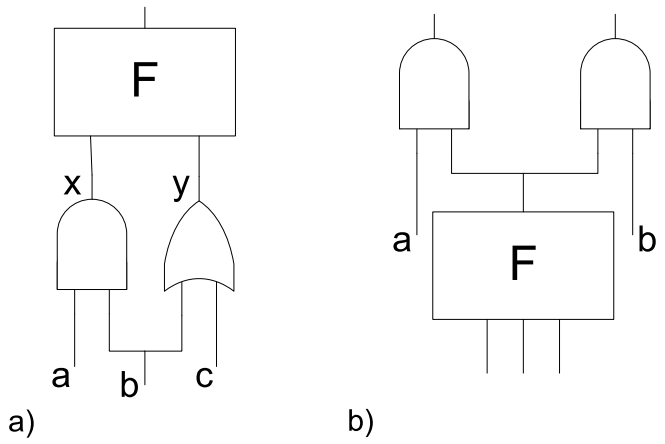

Figure 2.3: Satisfiability don't-cares (SDCs) and observability don't-cares (ODCs). a) An example of an SDC. b) An example of an ODC.

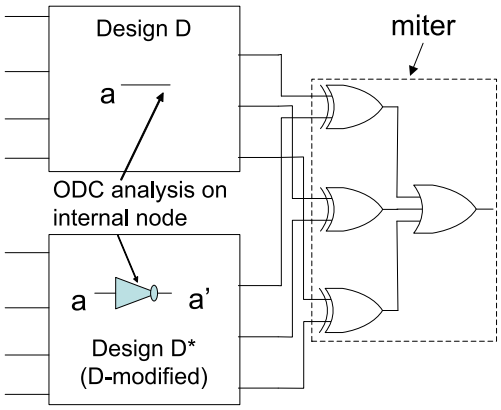

Figure 2.4: ODCs are identified for an internal node a in a netlist by creating a modified copy of the netlist where a is inverted and then constructing a miter for each corresponding output. The set of inputs for which the miter evaluates to 1 corresponds to the care-set of that node.

mented with XOR gates that compares the outputs of two circuits; functional equivalence is proven if the output is a constant 0. The care set, denoted as $C(a)$, can be derived as follows:

$$C(a) = \bigcup_{i:D(X_i)\neq D^*(X_i)} X_i \tag{2.1}$$

where X_i is an input vector.

A SAT solver can derive C by adding successive constraints called *blocking clauses* to invalidate previous assignments satisfying the miter. The ODC of a is therefore:

$$ODC(a) = \bigcup_i X_i - C(a) \tag{2.2}$$

that is, the difference between the set of all input vectors and the care set.

This approach can be computationally expensive and scales poorly, particularly when the XORs are far from a. In [61], this pitfall is managed by examining only small windows of logic surrounding each node being optimized. The don't-cares extracted are used to reduce the circuit's literal counts. In [95], a very efficient methodology is developed to merge nodes using local don't-cares through simulation and SAT. The authors limit its complexity by considering only a few levels of downstream logic for each node. However, these techniques fail to efficiently discover don't-cares resulting from logic beyond the portion considered, a limitation that is directly addressed in this dissertation. Another strategy to derive don't-cares efficiently entails algorithms for computing compatibility ODCs (CODCs) [71, 72]. However, CODCs are only a subset of ODCs, and fail to expose certain types of don't-cares; specifically, CODCs only enable optimizations of a node which do not affect other node's don't-cares.

2.2.2 Logic Rewriting

Performing scalable logic optimization requires efficient netlist manipulation, typically involving only a small set of gate primitives. Given a set of Boolean expressions that describe a circuit, the goal of synthesis optimization is to minimize the number of literals in the expressions along with the number of logic levels. Several drawbacks of

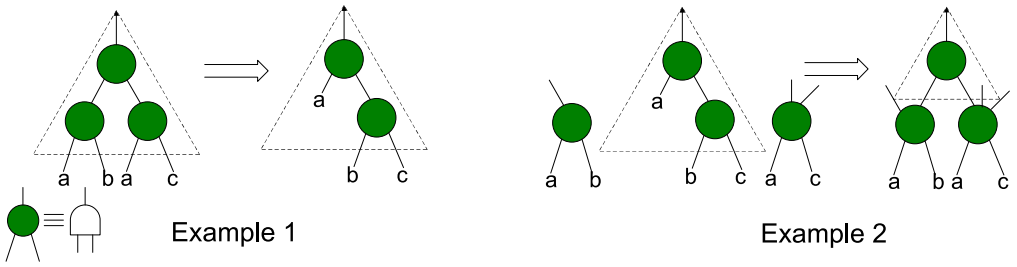

Figure 2.5: Two examples of AIG rewriting. In the first example, rewriting results in a subgraph with less nodes than the original. Through structural hashing, external nodes are reused to reduce the size of the subgraph, as shown in the second example.

these techniques are discussed in [62], including limited scalability. To this end, an efficient synthesis strategy called *rewriting* was introduced [62]. Logic rewriting is performed over a netlist representation called an And-Inverter Graph (AIG) [47], where each node represents an AND gate, while complemented (dotted) edges represent inverters. In logic rewriting, the quality of different functionally-equivalent implementations for a small logic block in a circuit is assessed. For example, in Figure 2.5 the transformation on the left leads to an area reduction. Moreover, by using a technique called *structural hashing* [47], nodes in other parts of the circuit can be reused. For instance, in the example on the right, there is a global reduction in area by reusing gate outputs already available in other parts of the circuit. In [63], logic rewriting resulted in circuit simplification and was used to improve the efficiency of combinational equivalence checking.

2.2.3 Physically-aware Synthesis

Logic synthesis can be guided by metrics other than literal-count reduction. Although detailed wire information is unavailable during logic synthesis, rough delay estimates can be made by placing gates before synthesis optimization. Instead of reducing literals, one

favors delay-improving transformations [21]. However, delay estimation is becoming increasingly inaccurate before detailed placement and routing, as the actual interconnect routes become more significant with every technology node. This continuing trend suggests the need to integrate new synthesis algorithms after placement and routing, rather than optimize beforehand with inaccurate estimates, which can have undesirable consequences for other performance metrics.

2.3 Logic Simulation and Bit Signatures

Logic simulation involves evaluating the design on many different input vectors. For each simulation vector, the circuit output response can be determined by a linear topological-order traversal through the circuit. In our work, we exploit a type of information known as a *signature* [50], that can be associated with each node of the circuit and is computed through simulation.

A given node F in a Boolean network can be characterized by its signature S_F for K-input vectors $X_1 \cdots X_K$.

Definition 2.3.1 $S_F = \{F(X_1), \ldots, F(X_K)\}$ *where* $F(X_i) \in \{0,1\}$ *indicates the output of* F *for input vector* X_i.

Vectors X_i can be generated at random and used in *bit-parallel* simulation [1] to compute a signature for each node in the circuit. For a network with N nodes, the time complexity of generating signatures for K input vectors for the whole network is $O(NK)$. Nodes can be distinguished by the following implication: $S_F \neq S_G \Rightarrow F \neq G$. Therefore, equivalent signatures can be used to efficiently identify potential node equivalences in a circuit by deriving a hash index for each signature [50]. Since $S_F = S_G$ does not imply

that $F = G$, this potential equivalence must be verified, *e.g.*, using SAT. In [59], simulation was used to merge circuit nodes while incrementally building a mitered circuit. The resulting mitered circuit is much smaller and is typically easier to formally verify since the corresponding SAT problem has fewer clauses, and it is thus often easier to solve.

2.4 Summary

Improving algorithms for logic synthesis and verification is an important, difficult and multi-faceted challenge. Effective and scalable solutions may significantly impact electronic design automation and consumer electronics industries. In developing such solutions, we leverage logic simulation used in conjunction with synthesis and verification to improve scalability. While some research has shown the benefits of using simulation to boost the performance of combinational equivalence checking, using signatures to guide synthesis optimizations has only been considered in a few, limited forms. In our work, we develop novel techniques to exploit signatures for synthesis optimization and to improve verification coverage.

CHAPTER III

Challenges to Achieving Design Closure

Achieving timing closure is becoming more difficult because of the increasing significance of interconnect delay. When gate delay was the primary component of chip delay, logic synthesis tools could accurately estimate and improve delay by reducing the maximum number of logic levels in a circuit. However, the resistance and capacitance of wires have increased, preventing interconnect delay from scaling as well as gate delay. Figure 3.1 shows a plot from the 2005 report of the International Technology Roadmap for Semiconductors (ITRS) indicating that gate delay is foreseen to decrease for future technology nodes faster than local interconnect and buffered global interconnect delay.

Because of the increasing significance of interconnect delay, designs that are optimized by traditional logic synthesis strategies often contain delay violations that are discovered only after accurate interconnect information is available toward the end of the design flow. Therefore, optimization is typically confined to physical transformations, as opposed to logic or high-level design optimizations. Performing higher-level optimizations is often problematic at the late design stages because they could perturb cell placement and cause modifications that would violate other performance metrics and constraints. Furthermore, there are less opportunities to perform logic optimizations with new timing information late in the design flow, because the design has already been optimized thoroughly based

on earlier incomplete and inaccurate information. Therefore, more restrictive physical synthesis techniques are used after placement, including interconnect buffering [56], gate sizing [45], and cell relocation [3], which can all improve circuit delay without significantly affecting other constraints. When timing violations cannot be fixed with these localized techniques, a new design iteration is required, and the work of previous optimization stages must be redone. In many cases, several time consuming iterations are required, and even then, the finished product may not achieve all the desired objectives.

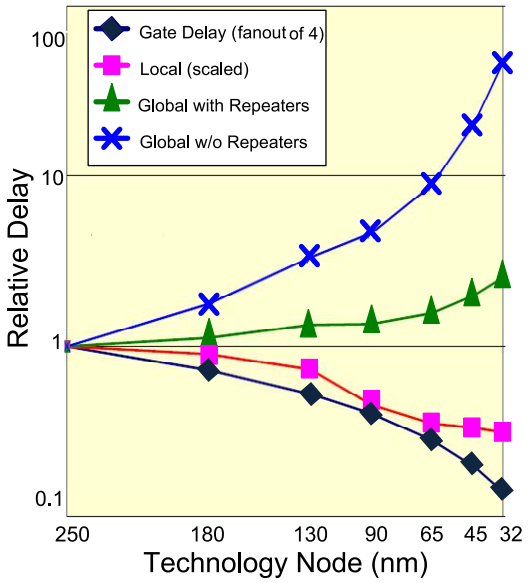

Figure 3.1: Delay trends from ITRS 2005. As we approach the 32nm technology node, global and local interconnect delay become more significant compared to gate delay.

Achieving timing closure without undergoing many design iterations is a pervasive problem studied in electronic design automation, and the leading cause of growing times to market in the semiconductor industry. For this reason, we devote much effort to this issue in the dissertation. In this chapter, we highlight previous efforts dedicated to improving timing closure, as well as their shortcomings. We first describe ideas to improve the quality

of physical synthesis; then we discuss how the design flow has been transforming over time from several discrete steps into a more integrated strategy that can better address interconnect scaling challenges. Finally, we conclude by outlining our strategy to tackle the timing closure challenge while overcoming the limitations of previous work.

3.1 Physical Synthesis

Physical synthesis is the optimization of physical characteristics, such as delay, using logic transformations, buffer insertion, wire tapering, gate sizing, and logic replication. Static timing analysis (STA) is used to estimate the delay guiding the physical synthesis optimizations. The accuracy of timing analysis is dependent on the delay model considered and the wire information available. For instance, the Elmore delay model [30] is a well-established approximation for delay based on cell locations. The following equation gives the Elmore delay for a wire segment of length L with wire resistance r and capacitance c per unit length:

$$ED = rc\frac{L^2}{2} \tag{3.1}$$

Notice that delay increases quadratically as a function of wire length. The Elmore delay model is commonly used, but tends to overestimate delay for long nets. Other delay models, such as D2M [6], are more accurate because they use multiple delay moments, but are also more complex.

Physical optimizations can produce a placement where cell locations overlap. Overlapping cells can be eliminated through placement *legalization*. Effective legalization strategies must produce a placement with no cell overlaps, while not significantly disrupting the

bulk of the already *legal* placement. Even small changes in the placement can substantially alter circuit timing. Therefore, after legalization, incremental STA is performed to assess the quality of the optimization. Evidently, an optimization that produces large changes in the netlist is undesirable because the legalization procedure could be more disruptive to the pre-existing placement than the benefit brought by the optimization.

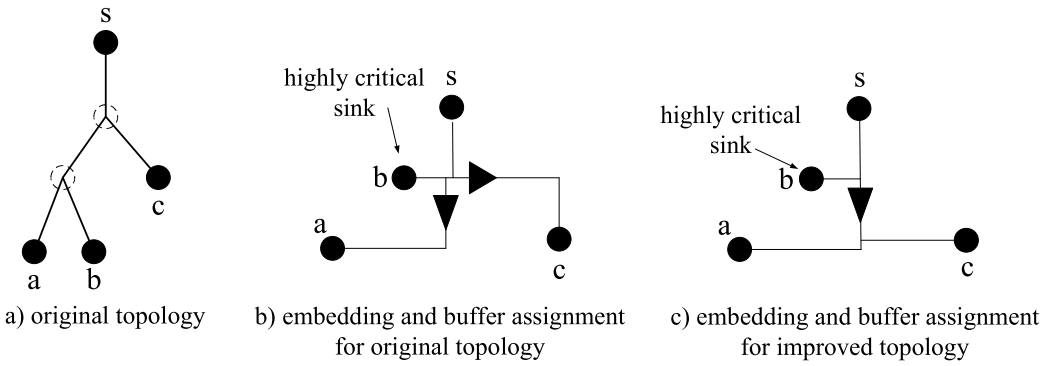

Figure 3.2: Topology construction and buffer assignment [43]. Part a) shows the initial topology and part b) shows an embedding and buffer assignment for that topology that accounts for the time criticality of *b*. In part c), a better topology is considered whose embedding and buffer assignment improves the delay for *b*.

The primary optimization strategies used in physical synthesis include reducing capacitive load to improve the delay of long wires. Unbuffered long wires experience quadratic increase in delay as a function of wirelength, as shown in the delay model of Equation 3.1. However, by optimally placing buffers along a wire, the delay can be made linear [67]. Moreover, buffers have been successfully deployed to improve delay for cells that have large fanout. For example, given a source signal that fans out to several nodes, optimal buffering involves both 1) constructing an interconnect topology that maintains the connections between the source and the fanout and 2) assigning buffers to the topology so that timing is optimized.

In Figure 3.2, we show an example of a netlist portion where a source signal *s* fans out to signals *a*, *b*, and *c*. In part a), we show the original topology connecting the source signal with its fanout. The topology indicates how fanout nodes are partitioned in the interconnect. For instance *a* and *b* are grouped together in part a). Finding a *fanout embedding* for this topology corresponds to determining how the interconnect is routed given the topology. One of the problems related to buffering is that of determining the actual buffer locations once the topology is fixed (*buffer assignment*). In [82], the authors develop an efficient algorithm for placing buffers into a given interconnect topology to optimize delay. Their approach would take into account the required arrival time by each of the fanout nodes in placing the buffers. If fanout *b* is timing critical, their solution applied to the topology of Figure 3.2a would produce the buffer assignment shown in Figure 3.2b where there is a short unbuffered path to fanout *b*. More recently, the authors of [43] considered approaches for finding an optimal topology in addition to optimal fanout embedding and buffer assignment. For instance, if fanout *b* is timing critical, a better topology can be constructed as shown in Figure 3.2c, where the capacitive load of signal *s* is reduced and the arrival time at fanout *b* is still earlier than *a* and *c*.

Fanin embedding, studied in [42], is the process of finding optimal cell locations for a one output subcircuit also for the purpose of improving delay. This is conceptually similar to the fanout embedding described above, where a routed interconnect tree is derived from a given topology. In fanin embedding, a fanin tree consists of a single output signal, input signals, and internal nodes representing the subcircuit's logic cells. The topology of the subcircuit is determined by the logic implementation of the output. The goal of fanin embedding is to find a placement that optimizes delay, while ensuring that no cell overlaps

occur. However, because of logic reconvergence, most subcircuits are not trees in practice. To address this issue, the authors of [42] explore a procedure that uses replication to convert any subcircuit into a tree. Unlike the work in [43] that considers fanout embedding on different topologies, fanin embedding has limited range of solutions because the topology is fixed by the logic implementation.

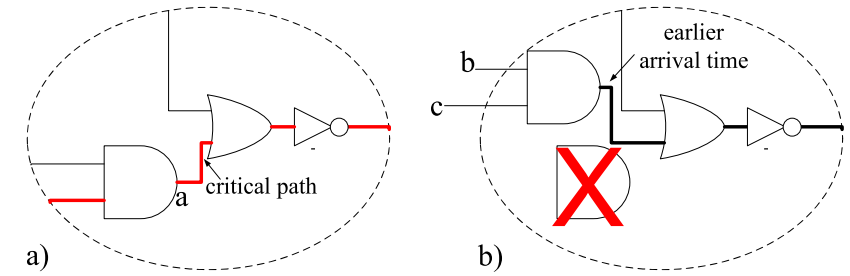

Figure 3.3: Logic restructuring. The routing of signal a with late arrival time shown in part a) can be optimized by connecting a to a substitute signal with earlier arrival time as in part b). In this example, the output of the gate $AND(b,c)$ is a resynthesis of a.

So far, in this section we have analyzed physical synthesis strategies that involve buffering, finding optimal cell locations, and cell replication. We now discuss logic restructuring techniques. Logic restructuring are netlist transformations that preserve the functionality of the circuit. For example, in [18, 53, 78] logic restructuring is performed by using only simple signal substitutions, thus missing many other restructuring opportunities. The authors in [84] develop an approach to identify restructuring opportunities involving more types of logic transformations, but global don't-cares are not exploited. For example, consider Figure 3.3, where signal a can be replaced by using the output of the $AND(b,c)$ gate, producing a small change to the netlist that hopefully has minimal impact on the existing placement. The work in [18] uses simulation to quickly identify potential substitutions, but does not explore the full range of potential transformations; for

instance observability don't-cares are not taken into consideration. In [53], redundancy addition and removal (RAR) techniques are proposed to improve circuit timing. Our empirical analysis of RAR in Chapter IX shows that these techniques leave significant room for improvement because the transformations they consider are somewhat limited.

Examining a broader set of transformations is challenging for current synthesis strategies. In a post-placement environment, a netlist has already been mapped to a standard-cell library and restructuring must be confined to a small section of logic, so that placement is not greatly disrupted. If this small section were restructured using rewriting [62] (as explained in Section 2.2.2), logic sharing would be difficult with AIG-based structural hashing [47] since the netlist may be mapped to something considerably different from an AIG. The solution we propose in Chapter IX overcomes this limitation by integrating logic sharing on an arbitrarily mapped netlist with aggressive restructuring.

3.2 Advances in Integrated Circuit Design

Figure 3.4a illustrates the traditional division of design stages outlined in Chapter I. In this flow each stage is invoked only once, and design closure is expected to be achieved at the end of the flow. In practice, extensive physical verification is required to assess whether design goals are achieved and whether all constraints are satisfied. Often, design closure fails in several ways. For example, logic synthesis fails if the total area of the synthesized netlist is greater than specified. The result is a re-iteration of synthesis, perhaps with a modified objective function. In a more dramatic scenario, the design might need to be re-optimized at a higher level of abstraction. As another example, if routing resources are insufficient for a wire, an iteration of the placement phase could reduce congestion.

To avoid costly iterations of several of the design stages, this traditional design flow

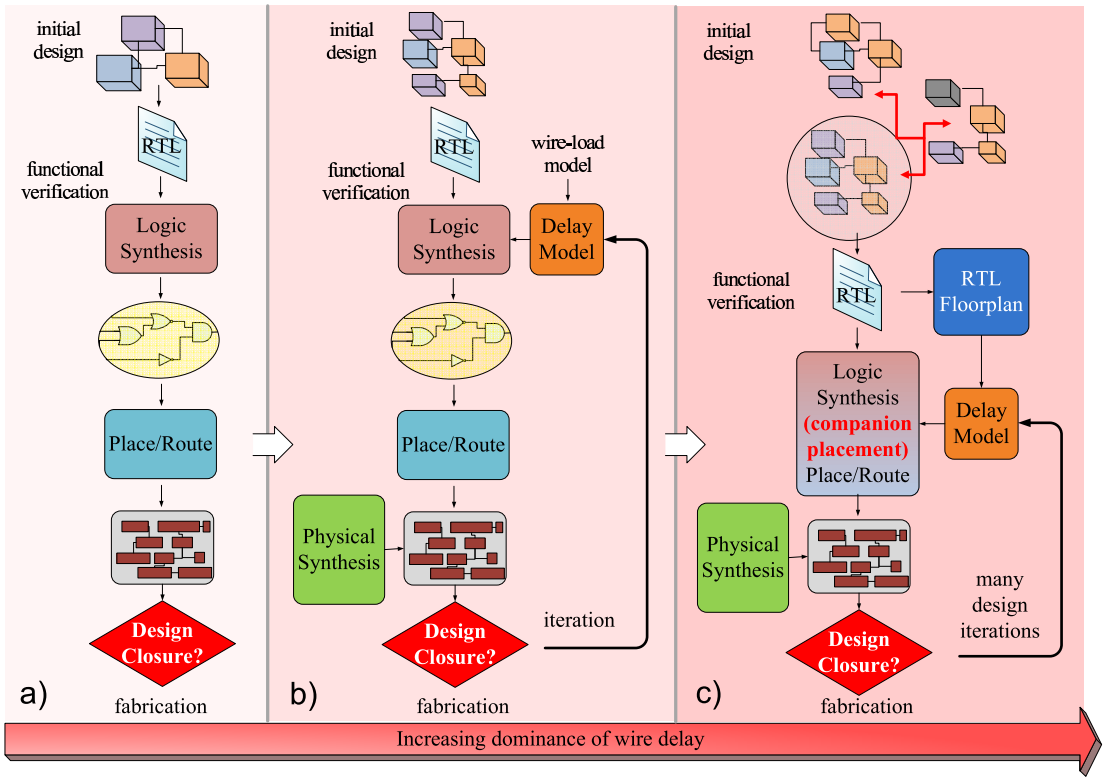

Figure 3.4: Evolution of the digital design flow to address design closure challenges due to the increasing dominance of wire delay. a) Design flow with several discrete steps. b) Improved design flow using physical synthesis and refined timing estimates to achieve timing closure more reliably. c) Modern design flow where logic and physical optimization stages are integrated to leverage better timing estimates earlier in the flow.

comprising several discrete optimization stages has evolved into a more integrated strategy where the boundary between synthesis and place-and-route are blurred. The goal of design integration is to perform optimizations that are simultaneously aware of many objectives and constraints, such as performing logic optimizations during synthesis to lower wire congestion in the placed design. The evolution of the design flow is described in the following paragraphs and is shown in Figure 3.4.

Figure 3.4b illustrates a design flow incorporating physical synthesis and iterative feed-

back to achieve timing closure. Initially, post-placement optimizations (physical synthesis) were sufficient to manage increasing wire delay by improving delay of global interconnects through buffer insertion. However, as wire delay has become more and more prominent, *wire-load* models have been incorporated in logic synthesis to target gates with large capacitive loads. Even though these models only consider a gate's input capacitance, inaccurate estimates and optimizations could be easily corrected at later stages through gate sizing [45]. Eventually these approximate models became inadequate too, as wire capacitance and resistance became even more significant, resulting in additional design iterations and requiring better delay models.

To address the challenges posed by increased wire delay, an even more integrated design flow is currently in use (shown in Figure 3.4c), where more physical information is used at earlier stages of the design flow. For example, to improve wire delay, the authors in [40] incorporate wirelength estimation in logic synthesis, so that the netlist generated would likely have fewer wire detours. In [21, 41, 68], incremental placement is coupled with logic synthesis to assess the quality of each synthesis transformation in terms of its impact on placement (this approach is known as *companion placement*). The companion placement can even be generated by attempting to place a netlist mapped to a simple gate library before synthesis optimization. Another strategy integrates physical information in early design stages to control the dominance of global interconnect (as forecasted in Figure 3.1). This solution generates an RTL floorplan before logic optimization, so that it can estimate timing for about the 10% of wires mostly constituting global interconnect [104].

3.3 Limitations of Current Industry Solutions

Even with better integration in the design flow, physical synthesis remains essential to avoid delay violations and only grows in importance with every technology node. Physical optimizations are limited because there is less opportunity for logic transformations at late design stages, reducing opportunities for improvement. Furthermore, post-placement optimizations must minimize the region affecting placement to contain hard-to-predict impact to delay due to legalization.

There are several additional limitations to the current methodology [103, 104] that are expected to exacerbate in future technology nodes:

1. Timing-driven transformations may fail to improve delay, but still negatively impact area. This may occur due to inaccurate timing estimates. In addition, incorporating timing models in traditional synthesis and technology-mapping increases computational effort.

2. Maintaining a companion placement during logic synthesis incurs computational overhead and may be inadequate for future technologies where even more accurate delay estimates are needed. First, generating placement for an un-optimized netlist with more logic cells than those in the final layout unduly stresses placement algorithms. Second, the accuracy is still limited because the companion placement estimates cell locations approximately and cannot determine actual wire routes and parasitic effects that can significantly affect delay.

3. The impact of poor optimization on shorter interconnect is becoming more profound, and using common physical synthesis strategies, such as buffering, may be

insufficient. In [73], it was observed that future technologies will require buffers at much smaller wire lengths. It was estimated that at the 45-nm node, 35% of cells in a large synthesizable block would be buffers. This fraction is projected to increase to 70% at the 32-nm node. Even if timing is maintained using buffering, the consequences for area utilization and power consumption will be severe. Current methodologies using higher-level estimates before physical synthesis will be unable to account for the increasing relevance of shorter interconnects.

To achieve better design flows, intuition suggests to incorporate synthesis optimization after placement because at that point the synthesis process can utilize more accurate timing information. This allows for more powerful timing optimizations since more detailed estimates are available, while minimizing negative impact to other performance metrics. However, traditional post-placement synthesis optimization is inadequate because it only considers a small subset of possible transformations, and fails to fully exploit the full range of possibilities (for instance, due to don't-cares).

3.4 Contributions of the Dissertation

In this chapter, we have described the evolution of the design flow in the last few decades to address the increasing dominance of interconnect. It is becoming more difficult to provide accurate timing information to logic synthesis, and current physical synthesis strategies are becoming inadequate at generating the delay improvement necessary to reduce costly design iterations. Our work overcomes the restrictiveness of current physical synthesis methodologies in improving interconnect delay. In summary, the major contributions of this dissertation to achieve this goal are:

- A post-placement resynthesis strategy that tightly integrates accurate physical constraints to improve critical path delay by constructing subcircuit topologies using static timing analysis. Our strategy greatly exceeds the optimization capabilities of traditional logic synthesis techniques, and, at the same time, minimizes perturbations to the placement.

- A novel metric for identifying sections of critical path in a netlist that are most amenable to logic resynthesis.

- A comprehensive simulation-based framework that uses signatures to identify post-placement optimizations in complex designs. The components of our framework include:

  – A solution that identifies *automatically* areas of a circuit inadequately sensitized under random simulation and relies on a SAT-based technique to generate new simulation vectors to target these areas. This improves the quality of signatures, which enables more efficient identification of resynthesis opportunities.

  – A novel parallel SAT solver infrastructure that produces better utilization of multi-core systems and therefore faster verification of synthesis optimizations identified with signatures.

  – A powerful synthesis approach that uses signature-based abstractions to quickly identify functionally equivalent logic structures up to global circuit don't-cares increasing the number of post-placement resynthesis opportunities.

Part II

Improving the Quality of Functional Simulation

The effectiveness of bit signatures in enabling powerful physical optimizations is the centerpiece of this dissertation. This effectiveness depends on the ability of signatures to 1) distinguish circuit nodes that are functionally different and 2) identify potential logic transformations that elude traditional logic synthesis techniques. In Chapter IV, we introduce a strategy that improves the quality of the input stimuli for simulation so that it produces high-quality signatures, *i.e.*, signatures that better distinguish functionally different nodes with just a few simulation cycles. Then in Chapter V, we propose an algorithm to incorporate circuit don't-cares in signatures, so that they can be leveraged to expose additional optimization opportunities.

CHAPTER IV

High-coverage Functional Simulation

As mentioned in Section 1.4, the advantage of using a bit signature to identify potential logic optimizations lies in its ability to characterize a circuit node's functionality with only a partial truth table. To generate a signature, input vectors are applied to the inputs of the circuit. The input vectors chosen determine the *quality* of the signature. High-quality signatures are defined as those that require few input vectors to generate, where the signatures' values distinguish functionally different nodes. However, the usefulness of signatures in guiding powerful design optimizations also depends on the efficiency of generating these high-quality signatures. If prohibitive computation efforts are required, the benefits of using signatures are negated.

Quickly generating high-quality input vectors is the challenge that this chapter addresses. Traditionally, input vectors for verification are created by performing a mixture of random simulation and *guided* simulation strategies. Guided simulation involves choosing input vectors either manually or through an automated mechanism to achieve some simulation coverage goal. The advantages of random simulation are 1) the speed at which new input vectors can be generated and 2) the ability of these input vectors to often expose scenarios that guided simulation fails to capture. However, random simulation does not always produce high-quality signatures. In fact, two nodes may have similar functionality

(and truth tables), which often requires a large number of random input vectors to distinguish them. To reduce the number of input vectors required to generate high-quality signatures requires either manually deriving input vectors using the expertise of the designer or automatically deriving input vectors using a proof engine like SAT. For instance, a SAT solver can be used to derive an input vector that distinguishes two circuit nodes [59]. However, using either manual or automatically-guided simulation to distinguish nodes is often time consuming compared to random simulation.

In complex designs, random simulation struggles to expose interesting behavior, as many components of a circuit are deep in the circuit's structure and thus difficult to *control* from the primary inputs. In other words, applying different input vectors often does not correspond to differences in the internal nodes of the component under analysis resulting in signatures that do not distinguish functionally different nodes. The use of formal methods can improve the signature quality of a single node; however, many of the logic transformations described in later chapters require good signatures for many internal nodes, and making numerous invocations of formal engines undermines the computational efficiency of using signatures as an abstraction.

In this chapter, we introduce an approach that produces high-quality input vectors by leveraging the benefits of both random and guided simulation. This approach, called *Toggle*, involves 1) identifying components in a design that are not controlled adequately by random input vectors and 2) targeting each component with guided simulation leveraging a SAT solver. Our guided simulation overcomes the limitations of previous methodology by generating input vectors that target several circuit nodes simultaneously, rather than one at a time. Furthermore, our approach is applicable to verifying the functional correctness

of a design, as we show a correlation between improving the quality of signatures and improving the verification coverage in a design.

4.1 Improving Verification Coverage through Automated Constrained-Random Simulation

Constrained random simulation can be used to expose interesting behavior in a design [96]. Constrained random simulation involves the addition of constraints that limits and controls the input combinations that are sent to the design. However, there are major challenges involved in constrained-random simulation that we address in this chapter. First, generating constraints suggests that the design team has a thorough knowledge of internal aspects of the design. Second, generating specific input vectors that satisfy given constraints requires the use of formal methods. This second challenge is partially addressed by [86], where constraints are modeled as BDDs and simulation vectors are obtained through a random walk of the BDD. However, the approach in [86] still requires complex constraint specifications, and it is limited in the constraint complexity that it can handle by its dependency on BDD size. Toggle overcomes these challenges by automatically generating constraints that guide simulation without requiring detailed knowledge of the design. Furthermore, these constraints are small in general and input vectors can be efficiently derived using a novel SAT-based approach.

The high-level flow of Toggle is illustrated in Figure 4.1. First, we apply low-effort synthesis to the behavioral specification of a design, so as to leverage gate-level tools. Then, we apply random simulation vectors to the netlist. To analyze the toggle activity of each signal in the netlist, we have introduced a novel entropy-based coverage metric. Using an entropy calculation, we search for internal signals that suffer from low switching

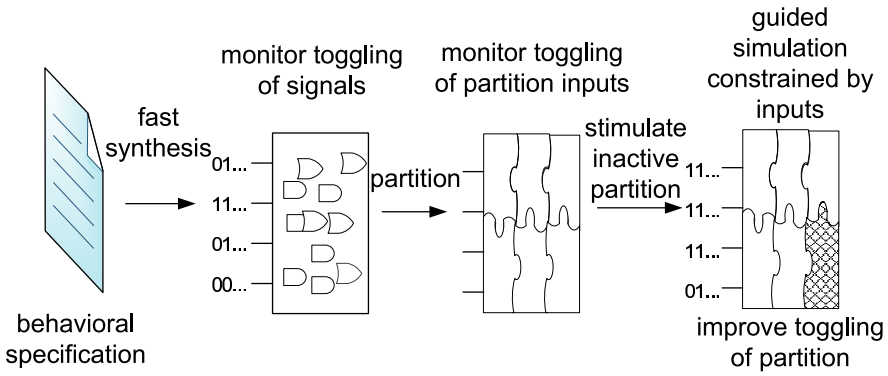

Figure 4.1: Our Toggle framework automatically identifies the components of a circuit that are poorly stimulated by random simulation and generates input vectors targeting them.

activity, because they produce signatures that are less-likely to be capable of distinguishing functionally different nodes. These signals are used to guide a partitioning of the design, so that signals experiencing low activity are grouped together. Low-activity partitions are then targeted by our SAT-based guided simulation, as shown at the end of the flow. By targeting several partitions with guided simulation, we seek to evenly sensitize the different components of a design.

The key theoretical result that we leverage to increase the switching activity of a partition is from [81]. It enables us to derive an even distribution of input vectors stimulating a partition by automatically generating small random constraints that involve XORing sets of inadequately stimulated signals. We then derive targeted stimuli by invoking a SAT solver. Our technique is flexible in that it can evenly sensitize parts of the design while incorporating other designer-specified constraints. We apply our analysis to commonly-used benchmark designs and demonstrate that many of them experience very low toggle coverage under random simulation. In contrast, our technique achieves higher simulation coverage than random simulation and is orders of magnitude faster, when compared to a

guided simulation approach.

The chapter is organized as follows. In Section 4.2, we propose a solution for monitoring activity in a design based on entropy. In Section 4.3, we introduce a strategy to re-simulate areas of a circuit to increase its toggling activity. Finally, experimental results comparing Toggle to constrained-random simulation are shown in Section 4.4.

4.2 Finding Inactive Parts of a Circuit

In this section, we adapt the notion of Shannon's entropy [75] to estimate simulation coverage within a gate-level circuit and propose its use to find inadequately-stimulated regions. We then show that obtaining high entropy corresponds to evenly sensitizing a design and thus minimizes unintended simulation bias, which can help in exposing corner-case behavior.

4.2.1 Toggle Activity of a Signal

The toggle coverage for a single signal s in a circuit C can be determined by the distribution of 0s and 1s seen under input stimuli. Each 0 corresponds to a *maxterm* and each 1 to a *minterm* of the function implementing s. We capture this distribution with two frequency values, and estimate the uncertainty (or toggling) of the signal using *Shannon's entropy* [75] E_s:

$$E_s = -\frac{nOnes}{K}\log_2(\frac{nOnes}{K}) - \frac{nZeroes}{K}\log_2(\frac{nZeroes}{K}) \tag{4.1}$$

where *nOnes* (*nZeroes*)is the number of simulation cycles for which $s = 1$ ($s = 0$) and K is the total number of simulation cycles examined. E_s assumes values ranging from 0 to 1, where higher entropy indicates a more even distribution of ones and zeroes. If s is the

output of a function depending on the set of Boolean variables X, we can relate E_s to the entropy of the inputs E_{X_i} by the following:

$$E_s \leq min(\sum_{i=1}^{|X|} E_{X_i}, 1) \tag{4.2}$$

Based on Equation 4.2, if the input vectors applied to X are uniformly distributed (and thus $E_{X_i} = 1$ for every input), the maximum entropy for E_s is 1. For instance, an even distribution of input vectors applied to an XOR function results in high E_s; in contrast, for an AND function, E_s is low because it has 1 minterm and $2^{|X|} - 1$ maxterms. If s fans out to other parts of the circuit, the signal's low entropy can be a limiting factor in achieving high switching activity in downstream logic, as indicated by Equation 4.2. We observe that the signal entropy can be increased by setting the signal to either 0 or 1 (depending on which value occurs less frequently) and then deriving an input vector that satisfies this condition with a SAT solver.

As a practical example of guiding simulation based on signal entropy, consider the impact of random simulation on an 8-bit bidirectional counter, as shown in Figure 4.2a. The results indicate that after many simulation vectors, random stimuli do not adequately toggle the most significant bit of the counter. We toggle the output bit of the counter with the smallest entropy by deriving a sequence of counter operations that flip this value using Toggle. Figure 4.2b shows that the techniques described in this work achieve an even distribution of entropy across each counter bit after only 300 additional simulation vectors, while the same result requires over 10000 vectors in a pure random simulation environment.

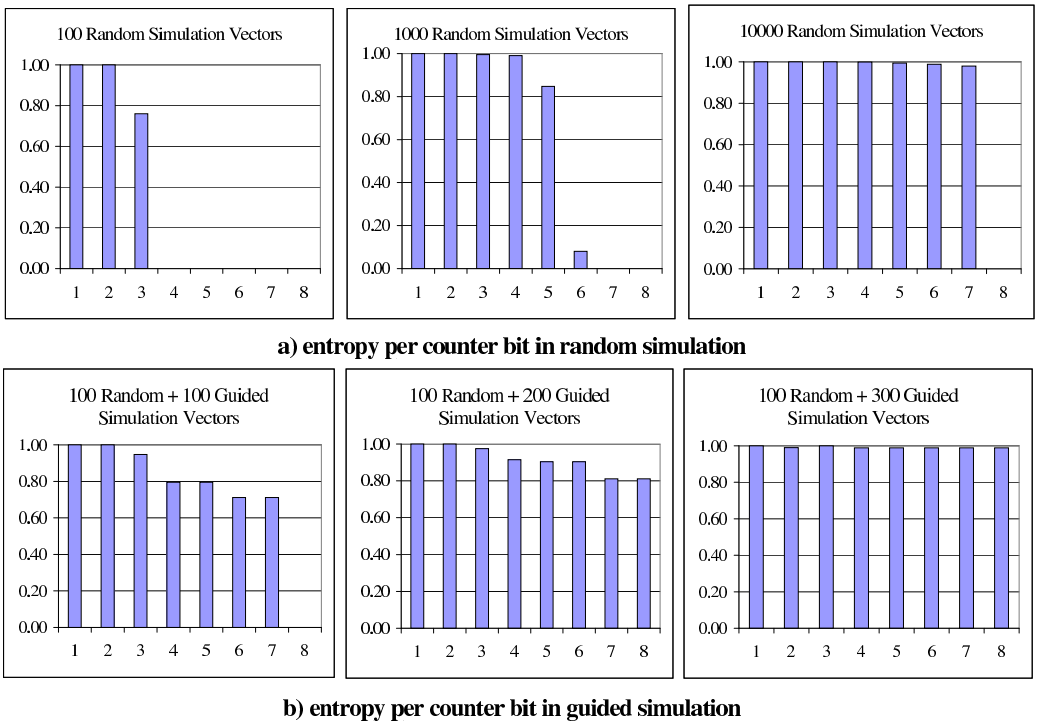

Figure 4.2: The entropy of each bit for an 8-bit bidirectional counter after 100, 1000, and 10000 random simulation vectors are applied is shown in part a). Part b) shows the entropy achieved after 100, 200, and 300 guided vectors are applied after initially applying 100 random vectors.

4.2.2 Toggle Activity of Multiple Bits

We extend the notion of signal entropy for a single signal to a set of signals that experience low activity when correlated to each other. We first identify these sets of signals as small cuts in the circuit determined by automatic netlist partitioning. We then define a coverage metric to assess the activity along the partition inputs that accounts for signal correlation.

Automatic circuit partitioning. Circuit partitioning has been explored in physical placement applications where net-cut minimization strategies generally lead to smaller wire-

length. The Fiduccia-Mattheyses (FM) min-cut partitioning algorithm [31] is commonly used for circuit partitioning and runs in linear time on the netlist size per optimization pass. Typically, only a few passes are required to achieve a good partition. Furthermore, multi-level extensions of this algorithm scale near-linearly to very large designs.

In our Toggle flow, we only need to generate a partition of the circuit once, hence its runtime is amortized by the subsequent input generation. To partition the circuit, we perform recursive bisection, *i.e.*, apply multiple cuts in the circuit until it is partitioned to a desired granularity (specified by a designer). The goal of this procedure is to minimize the total number of nets connecting the different partitions while ensuring that partitions have similar sizes. This leads to the generation of large partitions with few input signals, so that the activity for a large section of logic can be determined by only a small set of controlling signals.

To identify input cuts that experience low activity, we use the signal entropy defined in Equation 4.1 to guide the partitioning objective function. We note that the maximum entropy E_F of a set of signals F is bound by $\sum_{s \in F} E_s$. After assigning E_s as a weight to net s, we can employ netlist partitioning to find cuts with small entropy. This creates partitions with inputs whose total entropy is small, which, in turn, exposes parts of the circuit that are inadequately sensitized.

Estimating cut activity and biasing through entropy. The activity along a derived cut can be analyzed to assess the amount of simulation coverage for each partition. Consider the following metric for cut activity on partition F:

$$A_F^c = num\_diff\_vecs(< F_1, \cdots, F_m >) \tag{4.3}$$

where *num_diff_vecs* is the number of different value combinations that occur on partition F's m-input cut. We observe that this formula does not consider the frequency of certain combinations — it only provides the number of different combinations that is simulated.

Consider the following Boolean function $F(g_n(X), h_m(Y))$ where $X \cap Y = \emptyset$ and g_n and h_m are n and m output functions respectively. We analyze the activity along the outputs of g and h respectively using Equation 4.3. According to Equation 4.3, high activity would occur if $A_g^c = 2^n$ (if all output combinations are possible), and likewise if $A_h^c = 2^m$. Function F has $n+m$ inputs. Assuming maximum activity along the outputs of g and h, the maximum activity along F's input cut would be $2^n 2^m$. Because the activity metric does not account for the frequency of combinations, there is no insight on whether repetitive value combinations occur frequently for $F(g_n, h_m)$. Avoiding repetitive combinations is desirable so that the broadest span of behavior in the circuit is explored. However, we do know that there are at least $min(2^n, 2^m)$ different combinations.

We improve this coverage metric to account for repetitive value combinations, so that we can better guide our re-simulation efforts. To account for this repetition, which we call simulation bias, we develop a measure based on the amount of information (entropy) associated with the signals along a partition inputs under K simulation vectors. We compute the entropy of F as:

$$E_F^K = - \sum_{vec \,:\, occ(vec) \neq 0} \frac{occ(vec)}{K} \log_2 \left(\frac{occ(vec)}{K} \right) \tag{4.4}$$

where *occ* is the number of occurrences of a particular vector *vec* representing a value combination along the input cut, which is represented by an integer value. Under this formulation, the entropy is high when there are several different value combinations.

Using the entropy metric we can define *even sensitization* formally under entropy as:

Definition 4.2.1 *A set of inputs X to function F is evenly sensitized if* $\lim_{K\to+\infty} E_F^K = |X|$ *where* $2^{|X|}$ *is the number of possible input combinations along* X.

When the number of possible input combinations is less than $2^{|X|}$, due to limited controllability, the entropy corresponding to even sensitization is $\log_2$(the number of maximum different value combinations). Because the number of input vectors K applied during simulation is typically much smaller than the number of possible input combinations, a set of inputs is evenly sensitized under K input vectors when $E_F^K \approx K$.

Considering the Boolean function $F(g_n(X), h_m(Y))$, we can determine the maximum entropy along the outputs of g_n as $E_g^K = |X|$, where $|X|$ is the number of inputs to g. In other words, we see that the outputs of a function can be sensitized at best as evenly as its inputs. For a *reversible* circuit, there is a one-to-one mapping between input and output combinations, so that the entropy over the inputs is equal to entropy of the outputs. The entropy for the $n+m$ inputs of F, E_F^K, can now be bound as follows:

$$min(E_g^K, E_h^K) \leq E_F^K \leq E_g^K + E_h^K \tag{4.5}$$

Unlike the metric in Equation 4.3, by using entropy we can provide a bound to measure how even is the sensitization of F. In other words, we encapsulate more information about the behavior of circuit by using entropy rather than simply counting the number of input vectors (we later explain how to estimate the number of possible input vectors so that the maximum possible entropy is known). There is an additional benefit to using the entropy metric. By stimulating the partition with the smaller entropy, either g_n or h_m, we increase the lower-bound in Equation 4.5 for downstream logic.

4.3 Targeted Re-simulation

Toggle uses the entropy measure previously described to find parts of the design with low activity. We now introduce a SAT-based strategy that uses random XOR constraints to produce an even distribution of simulation vectors along a partition cut with low activity. The motivation for producing an even distribution is to find corner-case behavior, that could not be be exposed previously without detailed knowledge of the design and the generation of complex constraints.

Deriving a distribution of input vectors that evenly sensitize certain signals using a SAT solver is challenging because state-of-the-art SAT solvers do not provide any guarantees on the quality of the distribution. On the other hand, traversing a BDD to derive an even distribution of input vectors as in [86] may require prohibitive amounts of memory to represent the circuit. These challenges can be partially addressed by using techniques developed in the AI community, where a SAT solver is modified to evenly sample the solution space [25, 44]. However, these approaches are incompatible with DPLL-based SAT solvers, which are often more effective in solving EDA instances. This limitation is partially addressed in [36], which uses randomly generated XOR constraints to modify the SAT instance so any SAT solver can sample its solution space more evenly. At first sight, these techniques are not directly applicable to IC verification since we desire to derive input vectors that expose corner-case behavior in a circuit, but our work provides several missing links to make this connection.

4.3.1 Random Simulation with SAT

In this section, we first discuss the theoretical underpinnings that are used in our strategy to evenly sample the SAT solution space. We then propose a strategy to improve

the sensitization quality of a set of signals in a circuit, while satisfying the circuit's input constraints.

Theoretical background. Consider a SAT instance with $N > 1$ solutions. According to [81], it can be transformed to an instance that admits only one of those N solutions, requiring only a randomized polynomial-time algorithm that adds a limited number of XOR constraints. The algorithm succeeds in producing such an instance with probability $\geq 1/4$. Below, we discuss an aspect of this result that is relevant to our work, that is, adding a random XOR constraint reduces the solution space roughly by half with high probability.

Assume that we are given a SAT instance f with variables $x_1, x_2, ..., x_n$, and with solutions $v \in \{0,1\}^n$. To reduce the solution space, we randomly choose an assignment of the variables $w_1 \in \{0,1\}^n$ and add the following constraint to f: $v \bullet w_1 = 0$ in base-2 arithmetic (where $\bullet$ is the dot product). This can be expressed as follows:

$$f \wedge (x_{i_1} \oplus x_{i_2} \oplus \cdots \oplus x_{i_j} \oplus 1) \tag{4.6}$$

where i_j represents the indices of x_i where w_1 is 1. This results in an XOR constraint whereby an even polarity of x_{i_j} determined by w_1 needs to be assigned to 1. Alternatively, a CNF representation can be given as:

$$f \wedge (y_1 \Leftrightarrow x_{i_1} \oplus x_{i_2}) \wedge (y_2 \Leftrightarrow y_1 \oplus x_{i_3}) \wedge \cdots \wedge (y_{j-1} \Leftrightarrow y_{j-2} \oplus x_{i_j}) \wedge (y_{j-1} \oplus 1) \tag{4.7}$$

where the y_j variables are additional auxiliary variables required to express the XOR constraint.

Example 1 *Consider the CNF formula* $(a+b+c')(b'+d)(a'+d')(a+c+d)$*, where the solutions are :* $\{abcd\} : \{0001, 0101, 0111, 1000, 1010\}$*. The number of solutions can be reduced by generating an XOR clause corresponding to the randomly generated* $w_1 : a = 0, b = 1, c = 1, d = 0$*. The resulting CNF would be* $(a+b+c')(b+d')(a'+d)(a+c+d)(y \Leftrightarrow b \oplus c)(y \oplus 1)$ *where only* 3 *solutions* $\{0001, 0111, 1000\}$ *remain.*

If S_F represents the set of all solutions of F, then the addition of k random w_k vectors, or equivalently of k random XOR constraints, reduces the size of the solution space to $\sim 2^{-k}|S_F|$.

Random simulation with SAT. Through XOR-based reductions to a *unique-SAT* (U-SAT) instance (an instance with one solution), any particular solution[1] can be generated, which is the basis for our approach for deriving an even distribution of simulation vectors. Based on the results in [81], we can estimate that adding n XOR constraints for a CNF with n variables produces a U-SAT instance. (Since this is an estimate, some instances may have no solutions *i.e.,* are over-constrained, and some have multiple solutions. Instances with no solutions are called *UNSAT*.) Therefore, we can add multiple sets of n different XOR constraints to derive U-SAT instances where the unique solutions are evenly distributed following from the randomness of the reduction. In a circuit application where we wish to generate random input patterns, the XOR constraints need only involve primary input signals since the different ways to stimulate a circuit is completely determined by the assignments to the primary inputs of the circuit. Consequently, if an entire circuit is mapped to a CNF, the XORs added will not involve internal signals and therefore they will typically only increase the size of the original instance by a small amount. In principle, any

[1]The constraints derived from Equation 4.7 are satisfied when an even number of variables in each constraint is assigned 1, which always permit the all-0s solution.

SAT solver can be used to derive solutions for this modified SAT instance.

While our approach does not always produce instances with a unique solution, this happens very frequently [81]. In our strategy, if an unsatisfiable instance is produced, we derive another one. If an instance has multiple solutions, the SAT solver selects one of the remaining solutions. Using a SAT solver, we can derive an even distribution of simulation vectors as we show empirically in Section 4.4. However, if one desires only a small number of input vectors, a more efficient procedure can be used that requires the addition of fewer constraints and minimizes the number of unsatisfiable instances produced. For example, consider the case where only 64 evenly distributed input vectors are desired for circuit C with n primary inputs where $2^n > 64$. In this case, 6 XOR constraints can be added to approximately reduce the solution space to $\frac{1}{64}$ of the original size. By adding different random sets of 6 XOR constraints 64 times, we can still achieve an even distribution of solutions for the number of solution vectors desired, with faster simulation runtimes as shown in Section 4.4. In general, if we seek K simulation vectors, we solve K SAT instances each with different sets of $log_2(K)$ XOR constraints.

The addition of designer-specified constraints for targeting design properties to the original SAT instance does not affect the XOR formulation previously described. Therefore, an even distribution of input vectors can be derived that satisfies these additional constraints. Consider a circuit C with $|S_C|$ solutions and a circuit constrained with additional designer-specified constraints C^* that has $|S_{C^*}|$ solutions. When $|S_{C^*}| << |S_C|$, solutions that exist in S_C, S_{C_i}, may rarely exist in S_{C^*} as illustrated in Figure 4.3a. By adding $\log_2(K)$ XOR constraints, we can derive K vectors $S_{C_i^*}$ that are evenly distributed. If numerous UNSAT instances occur, implying that $K > |S_{C^*}|$, then one can alternatively

exhaustively enumerate all the solutions.

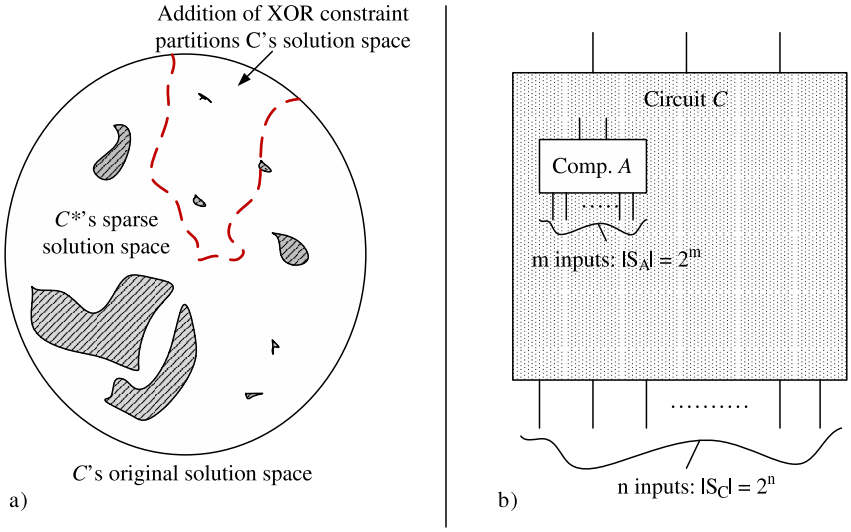

Figure 4.3: a) XOR constraints are added to reduce the solution space of a SAT instance C^*, which is sparser than the solution space of C. b) Component A is targeted for simulation, so that its m inputs are evenly sensitized within circuit C.

4.3.2 Partition-Targeted Simulation

We now propose an approach to automatically stimulate internal partitions while satisfying input constraints.

Stimulating a component within a design. Evenly stimulating an inadequately sensitized component by choosing certain input vectors is not straightforward, because the relationship between the distribution of stimuli on the primary inputs and on the inputs of the component is often complex. In Figure 4.3b, we show a component A with m input signals that that we desire to stimulate and that is deeply buried in the design hierarchy. We denote the solution space of A with respect to the input constraints as S_{AC}, and denote S_A as the solution space when not considering the input constraints. By applying random vectors to the m signals and checking whether the input constraints are satisfied, we can evenly stimulate A. However, limited controllability could mean that the input constraints are

rarely satisfied leading to prohibitive runtimes. To this end, we propose a new SAT-based methodology that expands upon our circuit simulation strategy in Section 4.3.1. For circuit C and its subcircuit A, we observe the following relation between CNF formulae:

$$CNF(C) = CNF(C \setminus A) \wedge CNF(A) \tag{4.8}$$

Therefore a solution to C implies a solution to A. Since the m signals uniquely determine every legal input combination to A, we can reduce the solution space of S_A and subsequently S_{A^C} by adding XOR constraints involving the variables m:

$$CNF(C \setminus A) \wedge CNF(A) \wedge (m_{i_1} \oplus m_{i_2} \oplus \cdots \oplus m_{i_j} \oplus 1) \tag{4.9}$$

This formulation reduces S_A roughly in half, and since the input constraints are accounted for by the constraint $CNF(C \setminus A)$, subsequently reduces S_{A^C} in half. Although many S_{C_i} may map to one S_{A_i}, the intention of this formulation is to generate input vectors that evenly sensitize the component, not the entire circuit.

Algorithm. Function `partition_sim()`, shown in Figure 4.4, generates an even distribution of simulation vectors by adding multiple random XOR constraints according to Equation 4.9. The number of random XOR constraints added is determined by the number of simulation vectors (`num_sims`) desired. After constructing the CNF, designer-specified constraints can be added (`add_additional_constrs()`). Then, we add different sets of XOR constraints for each pass of the while loop by function `add_xor_constrs()`. When large XOR constraints are added, the increased cost of propagating implications on a large set of clauses can be mitigated by (easily) adding specialized data structures and decision procedures. However, our experiments indicate that small XOR constraints are

most common in our application and these usually do not slow down the SAT solver appreciably. Therefore, specialized solver extensions for XORs, as in [35], are unnecessary.

```
partition_sim(Partition part, Circuit C, int num_sims){
  num_xor = log2(num_sims);
  CNF = construct_cnf(C);
  add_additional_constrs(CNF);
  while(num_sims){
      add_xor_constrs(num_xor, part, CNF);
      if(Solve(CNF, solution)) {
          add_vector(solution);
          num_sims − −;
      }
      remove_xor_constraints(part, CNF);
  }
}
```

Figure 4.4: Partition simulation algorithm.

If the instance is satisfiable, we add the new simulation vector and decrement `num_sims`. If the solution space considered is small relatively to the number simulation vectors applied, the SAT solver frequently derives the same simulation vector again. This can be eliminated by adding blocking clauses to the SAT instance. Also, numerous unsatisfiable instances are produced when the number of desired simulation vectors is similar to the size of the solution space. We can avoid these scenarios by estimating the size of the solution space as described below.

Controllability estimation with XORs. To maximize the effectiveness of our SAT-based simulation, we seek to target poorly-sensitized regions where the number of possible vectors is also greater than the ones observed so far, so that our SAT-based simulation has the potential to generate many new vectors. Ensuring this requires estimating the number of solutions for partition A with respect to its input constraints $|S_{A^C}|$.

We now show how to estimate if $|S_{A^C}| > (1+\Delta) * num\_diff\_vecs$ using XOR con-

straints.[2] In other words, we use XORs to find whether less than half of the possible vector combinations have already been observed along the partition's inputs. To do this, we use the result from [37] to estimate the number of SAT solutions with random XOR constraints. For instance, if the addition of x different XOR constraints does not produce an UNSAT result, we can estimate that the solution space is of size $\geq 2^x$. By examining multiple sets of different XOR constraints, we can obtain bounds with high accuracy, as proved in [37]. Since we desire a lower-bound computation and need XOR constraints that only involve the partition inputs, we can improve the efficiency of [37] for our specific circuit application.

4.4 Empirical Validation

We show that adding XOR constraints can evenly stimulate a design and that Toggle can improve activity for poorly stimulated partitions, while being considerably more efficient than a guided random simulation approach. In our experimental evaluation, we use MiniSAT [29] to derive simulation vectors and hMetis [46] to perform circuit partitioning. We examine circuits from the IWLS 2005 suite [102] and consider only their combinational portions.

Efficiently generating random stimuli with XOR constraints. In Table 4.1, we show the results of performing our SAT-based simulation on the primary inputs of circuit *alu*4 . We report the entropy, the number of different simulation vectors (`diff sim`) generated, and the runtime in seconds. For the results under `SAT-based`, we add 14 random XOR constraints to generate U-SAT instances, until we derive `#sim vectors` (number of simulation vectors). We compare this approach with random simulation and achieve com-

[2] we consider $\Delta = 1$ in this work.

#sim vectors	rand			SAT-based			approx SAT-based			
	diff sim	entropy	time(s)	diff sim	entropy	time(s)	#xor	diff sim	entropy	time(s)
64	64	1.00	<1	63	0.99	2	6	58	0.97	<1
128	128	1.00	<1	128	1.00	4	7	119	0.98	1
256	253	1.00	<1	256	1.00	6	8	240	0.98	1
512	499	0.99	<1	499	0.99	13	9	485	0.99	2
1024	991	0.99	<1	989	0.99	26	10	968	0.99	5

Table 4.1: Generating even stimuli through random XOR constraints for the 14 inputs of *alu*4. We normalize the entropy seen along the inputs by $\log_2(\#simvectors)$, so that 1.0 is the highest entropy possible.

petitively high entropy. Since many of the reductions using 14 XOR constraints produce UNSAT instances, this formulation is computationally expensive. Therefore, we show, under `approx SAT-based` in Table 4.1, that by adding fewer XOR constraints, determined by $\log_2(\#simvectors)$, we can significantly improve the runtime of the previous SAT-based formulation with nominal degradation to the entropy. Although random simulation is sufficient for this simple example, we now show that even distributions of simulation can be efficiently generated for internal signals while satisfying input constraints.

Identifying inactive parts of the circuit with Toggle. In Table 4.2, we show circuits that are partitioned using the signal-entropy weighting objective using 1024 random simulation vectors. After extensively experimenting with partitions of different sizes, we chose partitions that are ~ 100 gates in size. Compared to partitions of larger or smaller size, we observed empirically that this partition size most effectively balances our desire for examining the coverage of large parts of the circuit while minimizing the number of the signals considered for entropy analysis and re-simulation. Our results are averaged over 5 independent runs.

We show the average and worst entropy, where 10.0 is the maximum entropy possible

circuit	#gates	average entropy	worst entropy	guide+32		rand+32	
				new comb	%entr incr	new comb	%entr incr
spi	3010	9.5	6.4	+26.2	2.67	+1.6	0.12
systemcdes	3196	9.1	5.6	+15.0	0.48	+14.0	0.19
tv80	6847	8.9	1.6	+18.6	5.17	+0.8	-0.17
systemcaes	7453	9.7	5.2	+26.6	1.01	+12.4	0.16
ac97_ctrl	10284	10.0	9.5	+24.8	0.43	+21.8	0.35
usb_funct	11889	9.9	7.4	+26.4	1.10	+12.2	0.24
aes_core	20277	7.5	4.1	+17.0	2.60	+4.6	0.14
wb_conmax	28409	8.8	6.2	+25.2	2.12	+3.6	0.26
ethernet	37634	9.9	1.6	+26.2	2.12	+1.4	0.22
des_perf	94002	9.1	5.0	+13.4	0.55	+5.0	0.17

Table 4.2: Entropy analysis on partitioned circuits, the number of new input combinations found and the percentage of entropy increase after adding 32 guided input vectors versus 32 random ones.

with 1024 random input vectors. The results indicate that, while the average entropy for each circuit is close to 10.0, there is usually at least one partition that is considerably lower, as in `tv80`. We can then perform simulation mainly over these few poorly covered partitions.

Improving activity with Toggle. In the next part of Table 4.2, we assess the improvement of our SAT-based targeted re-simulation on a partition with low entropy and a sufficiently large solution space by deriving 32 additional simulation vectors. Our guided simulation is compared to generating 32 more random vectors. In `new comb`, we report the number of new combinations seen at the partition inputs averaged over 5 independent runs with different random seeds, and in `%entr incr` we report the percentage increase in entropy for the partition. Our approach outperforms random simulation on almost every circuit. Random simulation performs poorly, *e.g.*, `ethernet` and `tv80`, indicating strong bias under random simulation. If no improvements to verification coverage for a partition are possible with random simulation, the percentage increase in entropy hovers around 0. Our

circuit	guided+32 time(s)	part.random+32 time(s)	entropy time(s)
spi	1	210	<1
systemcdes	1	110	<1
tv80	1	time-out	<1
systemcaes	1	110	1
ac97_ctrl	1	2	<1
usb_funct	1	18	<1
aes_core	2	time-out	1
wb_conmax	4	232	1
ethernet	10	107	2
des_perf	20	23	2

Table 4.3: Comparing SAT-based re-simulation with random re-simulation over a partition for generating 32 vectors. The time-out is 10000 seconds.

approach can still re-derive some previously seen vectors, but we minimize these occurrences by our estimation of the partition's solution space size, which prevents re-simulation on partitions with limited controllability. Even for `ac97`, which is evenly sensitized by random simulation, we see some improvements because the worst-case entropy for the partition targeted for re-simulation is not at the maximum value of 10.0.

Runtime efficiency of Toggle. In Table 4.3, we show that evenly simulating a partition by randomly assigning values to its inputs and checking whether the primary input constraints are satisfied, is often much slower than using SAT-guided simulation. The results indicate that the SAT-based simulation scales well for larger circuits, in part, because the size of the XOR constraints required is typically small compared to the size of the circuit. Also, our SAT-based simulation often achieves orders of magnitude runtime improvement over random simulation, such as `wb_conmax` and `ethernet`. On the other hand, some benchmarks time-out at $10,000$ seconds, such as for `tv80` and `aes_core`. These results indicate that the solution space of the partition stimulated is sparse with respect to the input constraints. We expect our technique to perform even better when additional

designer-specified constraints are added, since this would further reduce the size of the solution space. For completeness, the last column shows the runtime of the entropy calculation in Equation 4.4. Clearly, this calculation is fast and scales to large designs.

4.5 Concluding Remarks

Our framework shows that certain theoretical results, not used in verification and simulation previously, hold the potential to significantly improve simulation coverage. This is done through careful feedback on coverage and biasing of input vectors to better stimulate poorly-sensitized parts of the circuit. By improving the quality of the simulation, we can expose interesting corner-case behavior in the circuit and encode it in signatures. To achieve these goals, we have introduced 1) an entropy metric to characterize the verification coverage of internal signals and 2) a novel simulation framework that uses XOR constraints to generate even distributions of stimuli while satisfying complex constraints. Our coverage metric reveals circuit regions that are inadequately stimulated under random simulation. We also show that adding only a few XOR constraints is often sufficient to evenly sensitize a design. Finally, our results indicate that guided simulation can compensate for coverage bias and can outperform purely random simulation in quality and runtime.

CHAPTER V

Enhancing Simulation-based Abstractions with Don't Cares

In this chapter, we introduce a strategy to efficiently derive and encode don't-care values in a bit signature using logic simulation. Using don't-cares facilitates more powerful synthesis transformations as shown in Chapter VIII.

Computing don't-cares for a node is challenging because a node's don't-care set is determined with respect to the primary inputs and outputs, which we will refer to as *global* don't-care analysis. Table 5.1 compares our analysis and its capabilities in computing don't-cares with previous work. One common theme among previous approaches is that they typically do not consider the entire fanin and fanout cone of a node because of the high cost of computation; we will refer to these solutions as *local* don't-care analysis. In [34, 59], a solution is proposed that can exploit satisfiability don't-cares (SDCs), but not observability don't-cares (ODCs) by relying on a combination of SAT solving and simulation in equivalence checking. In [61, 95], ODCs are considered, but only for a few levels of logic in the node's fanout cone.

In our approach, we can handle large circuits and derive all SDCs and ODCs with respect to the input vectors used in producing logic signatures. To this end, we develop a novel approximate simulator whose performance scales linearly with the size of the circuit.

Property	Simulation-guided SAT [34, 59]	Window-based ODC+SDC [61]	Local SAT-sweep [95]	Our solution
Don't-cares computed	global SDCs	local SDCs local ODCs	global SDCs local ODCs	global SDCs global ODCs
Computational engines	simulation + SAT	primarily SAT	simulation + SAT	simulation + SAT
Complexity limited by	SAT engine	windowing strategy	levels of downstream logic	moving-dominator incremental SAT (Chapter VI)
Primary application domain	verification	synthesis	verification	verification; logic & physical synthesis

Table 5.1: Comparisons between related techniques to expose circuit don't-cares. Our solution can efficiently derive both global SDCs and ODCs.

We evaluate the accuracy of our simulator both analytically and through empirical results.

5.1 Encoding Don't Cares in Signatures

When using signatures, there is no need to identify SDCs explicitly because impossible input combinations are not generated during logic simulation. However, some of the bits in signatures do not affect the outputs of the circuit and therefore they represent ODCs. To account for ODCs, we maintain an *ODC mask* S_f^* for node f in addition to its signature S_f.

Definition 5.1.1 *For input vector X_i, $S_f^* = \{X_1 \notin ODC(f), \ldots X_K \notin ODC(f)\}$ denotes the ODC mask for function f. $ODC(f)$ is a set of input vectors for which node f has an observability don't-care.*

When an input vector X_i is in the set $ODC(f)$, the corresponding bit position is denoted by a 0.

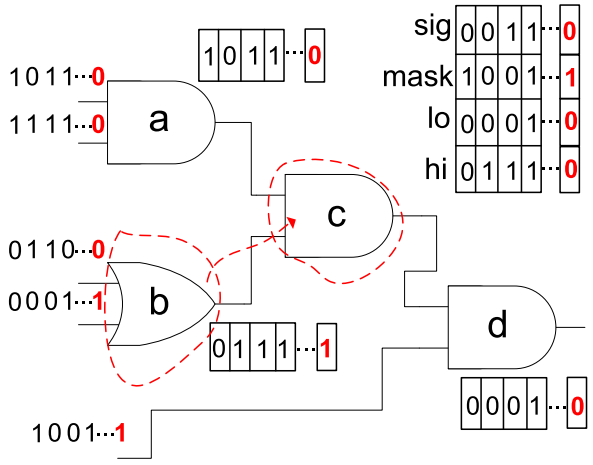

Figure 5.1: Example of our ODC representation for a small circuit. For clarity, we only show ODC information for node c (not shown is the downstream logic determining those don't-cares). For the other internal nodes, we report only their signature S. When examining the first four simulation patterns, node b is a candidate for merging with node c up to ODCs. Further simulation indicates that an ODC-enabled merger is not possible.

Figure 5.1 shows a circuit with signatures for each node and, in addition, a mask for node c. Each ODC for a node is marked by a 0 in the ODC mask. We express the logic flexibility of a given node by maintaining an *upper-bound signature* S^{hi} and *lower-bound signature* S^{lo}. $S^{hi}_f = S_f | \neg(S^*)_f$, where $|$ represents bit-wise OR, and $S^{lo}_f = S_f \& S^*_f$, where & represents bit-wise AND.

S^{lo}_f and S^{hi}_f of node f correspond to the range of Boolean functions $[f^{lo}, f^{hi}]$ that can implement f without modifying the circuit's functionality because the logical difference between any pair of functions within $[f^{lo}, f^{hi}]$ is a subset of the $ODC(f)$.

After simulation generates the signatures, potential optimizations can be identified. In the example in Figure 5.1, after the first four simulation patterns, node b is identified as a *candidate* for implementing c, meaning that we have a potential node merger. However, in this example, further simulation reveals that the candidate merger is not viable because

b and *c* are not compatible with respect to their last signature bit. In Chapter VIII, we present a node-merging application that efficiently exploits this don't-care encoding.

5.2 Global ODC Analysis

Below we describe a simulator with linear runtime complexity, that finds ODCs for each node of a circuit. Generating ODC masks S_f^* efficiently is integral to maintaining the scalability of our signature-based framework. While each node's signature can be computed from its immediate fanin, computing each node's ODC mask often requires analyzing its entire fanout cone.

The mask S^* can be computed for each node by using Equation 2.1,[1] where the X_i are the random simulation vectors. This approach requires circuit simulation of each X_i for each circuit's node. For K simulation vectors and n internal nodes, the time-complexity is $O(n^2K)$.[2] Although the simulation can be confined to just the fanout cone of the node, this approach is computationally expensive.

5.2.1 Approximate ODC Simulator

To improve upon the baseline algorithm described above, we developed an approximate ODC simulator whose complexity is only $O(nK)$ (n is the number of nets in the design and K is the number of simulation vectors). Our approach computes the ODCs of one node at a time in a manner that reuses previous computation. An outline of the algorithm for generating the masks in our approximate simulator is shown in Figure 5.2.

The function `set_output_`S^*`()` initializes the masks of nodes directly connected to the input of a latch or primary output to all 1s. The nodes are then ordered and traversed

[1] $C(a) = \bigcup_{i:D(X_i)\neq D^*(X_i)} X_i$.

[2] $O(nK)$ time-complexity for computing the mask for each of the n nodes.

```
gen_odc_mask(Nodes N){
  set_output_S*(N);
  reverse_levelize(N);
  for_each node ∈ N {
    node.S* = 0;
    for_each output ∈ node.fanout {
      temp_S* = get_local_ODC(node, output);
      temp_S* = temp_S* & output.S*;
      node.S* |= temp_S*;
    }
  }
}
```

Figure 5.2: Efficiently generating ODC masks for each node.

in reverse topological order as generated by `reverse_levelize()`. The immediate fanout of each `node` is then examined. The function `get_local_ODC()` performs ODC analysis for every simulation vector for `node`, as defined by Equation 2.2,[3] except only the subcircuit defined by `node` and `output` is considered. This local ODC mask is bitwise-ANDed with `output`'s S^* and is subsequently ORed with `node`'s S^*.

The algorithm requires only a traversal of all the nets given by the two `for_each` loops and the K simulation vectors considered for each net in `get_local_ODC()`, resulting in the $O(nK)$ complexity. This algorithm enables our global ODC simulator to be more efficient than what can be achieved simply extending the local observability calculations in [95] to perform global ODC analysis.

We can apply our algorithm to the circuit in Figure 5.1 to compute the ODCs of node a from the ODC information shown for node c. Because node c has don't-cares for the second and third simulation bit, node a also has don't-cares for those bits. When `get_local_ODC` is executed, the first simulation bit of node a is also a don't-care because

[3] $ODC(a) = \bigcup_i X_i - C(a)$.

node *b* has a controlling value of 0.

5.2.2 False Positives and False Negatives

Since we do not consider logic interactions that occur because of *reconvergence*, it is possible for the algorithm in Figure 5.2 to incorrectly produce 0s (*false positives*) or 1s (*false negatives*) in S^*. For the example shown in Figure 5.3, node *a* misses a don't-care (false negative) in the third bit of S_a^*. Notice that node *b* and *c* do not have any ODCs and no local ODCs exist between *a* and *b* or *a* and *c*, resulting in no ODCs being detected by the approximate simulator. However, the reconvergence of downstream logic makes the third value of node *a* a don't-care. In a similar manner, false positives may occur due to the interaction of multiple signals with local ODCs at a reconvergent node.

False positives do not affect the correctness of signature-guided transformations because each transformation is formally verified by equivalence checking. However, false negatives limit the pool of potential optimizations available for resynthesis. We show empirically in this chapter and in Chapter VIII that false negatives and positives occur rarely and seldom affect the results produced.

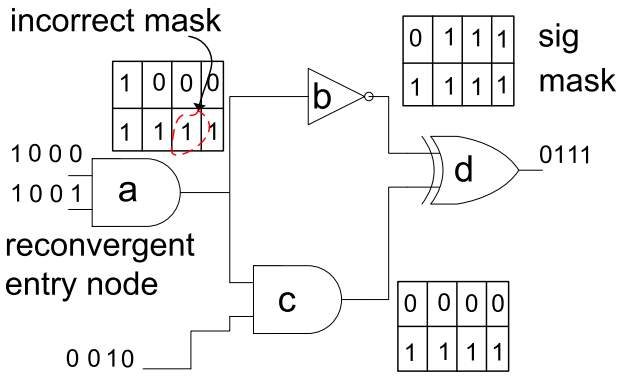

Figure 5.3: Example of a false negative generated by our approximate ODC simulator due to reconvergence. S^* and S are shown for all internal nodes; only S is shown for the primary inputs and outputs.

5.2.3 Analysis and Approximation of ODCs

We have observed that most ODCs require only a few levels of downstream logic to be computed. Indeed, consider two nodes in a circuit, f and g, where f is in the fanin cone of g. If X denotes the set of inputs to g, the number of X_i in the ODC set of f where g is the output, $ODC_g(f)$, is given by the following:

$$|ODC_g(f)| = nZeroes\big(g(f=0,X) \oplus g(f=1,X)\big) \tag{5.1}$$

where g is expressed as a composition of X and f.

Assuming a uniform input distribution, the probability that $X_i \in ODC_g(f)$ is equal to $\frac{|ODC_g(f)|}{2^{|X|}}$. In other words, this gives the probability that the output of f is a don't-care for input vector X_i. We offer a more insightful analysis by considering a subset of Boolean functions that have simple disjunctive decompositions [8], which are defined as:

Definition 5.2.1 *Function $g(X)$ (input set X) has a disjunctive decomposition if g can be expressed as $g^*(h(Y),Z)$ where $X = Y \cup Z$ and $\emptyset = Y \cap Z$.*

In practice, many functions in a circuit can be expressed as disjunctive decompositions [11, 70].

We assume that g has the disjunctive decomposition of $g^*(f(Y),Z)$[4] and note the following theorem based on Equation 5.1:

Theorem 1 $|ODC_g(f)| = 2^{|Y|} nZeroes\big(g^*_{f'}(Z) \oplus g^*_f(Z)\big)$.

Proof. $g^*(f(X),Z)$ can be expressed as a $Z+1$ input function $g^*(w,Z)$ where $w = f(Y)$. The Z-input functions g^*_w and $g^*_{w'}$ correspond to $g^*(w=1,Z)$ and $g^*(0,Z)$ respectively.

[4]This is always the case when f is a primary input.

$nZeroes(g^*_{w'} \oplus g^*_w)$ is the number of input combinations Z_i where $g^*_{w'}$ and g^*_w are equivalent (independent of w). Since Y is independent of Z, there are $2^{|Y|}$ different sets of these Z_i combinations where the value of w does not affect the output of g. □

For the disjunctive decomposition $g^*(f(Y),Z)$, the probability $X_i \in ODC_g(f)$ can be expressed as:

Corollary 1 $P_{X_i \in ODC_g(f)} = \frac{nZeroes\left(g^*_{f'}(Z) \oplus g^*_f(Z)\right)}{2^{|Z|}}$

Notice that this expression is independent of the function of f.

We can now develop a lower bound on the probability that f has an ODC for a given input vector X_i indicated by the following theorem:

Theorem 2 $P_{X_i \in ODC_g(f)} \geq \frac{|nOnes(g) - nZeroes(g)|}{2^{|Z|+1}}$.

Proof. $nZeroes(g^*_{w'} \oplus g^*_w)$ gives the number of input vectors where w is independent from g (a don't-care with respect to g). The difference in the number of minterms and maxterms in g gives a lower bound to the number of input vectors where w is independent of g. □

Note that the entropy of $g^*(w,Z)$ corresponds to a lower bound of the probability of ODCs. Functions with low entropy, *i.e.* high information loss, have a high percentage of input vectors with ODCs.

Example 1. Consider f as a primary input and $g(X)$ as a $|X|$-input AND gate, which has 1 minterm and $2^{|X|}-1$ maxterms. $\frac{2^{|X|}-2}{2^{|X|}}$, given by Theorem 2, is the lower bound of $P_{X_i \in ODC_g(f)}$. In this case, the lower bound is also the probability as given by Corollary 1. For a two-input AND, the probability is $\frac{1}{2}$ and for five-input AND the probability is $\frac{15}{16}$. If g is implemented with a set of two-input AND gates where f is at the first logic level, we see that the first few logic levels account for most of f's ODCs. □

In this example, we observe that most ODCs are due to only a few levels of logic. This trend is made clearer by considering the ODCs of f with respect to other nodes when f has more than one fanout. Consider $ODC_{g_1}(f)$ and $ODC_{g_2}(f)$. If we assume that $g_1(f(Y),A)$ and $g_2(f(Y),B)$ are disjoint decompositions and that $A \cap B = \emptyset$, we can express the probability of an ODC for f relative to outputs g_1 and g_2 by the following theorem:

Theorem 3 $P_{X_i \in ODC_{g_1 g_2}}(f) = P_{X_i \in ODC_{g_1}(f)} P_{X_i \in ODC_{g_2}(f)}$.

Proof. According to Corollary 1, $P_{X_i \in ODC_g(f)}$ is independent of the implementation of f. The probability that an ODC exists for input vector X_i is the joint probability that there is an ODC with respect to both g_1 and g_2. Since, A and B are independent, the two relative probabilities are independent, which results in the above relation. □

Example 2. If g_1, g_2, ... g_m are n-input ANDs that are fanouts of primary input f, $P_{X_i \in ODC_{g_1 g_2 \cdots g_m}}(f) = (\frac{2^n-2}{2^n})^m$. As in the previous example, the probability is $\frac{1}{2}$, for $n = 2$ and $m = 1$. However, the addition of just one fanout significantly decreases this probability to $\frac{1}{4}$, for $n = 2$ and $m = 2$. □

In this example, the presence of fanout counteracts the mechanism shown earlier for producing don't-cares. When a circuit contains many nodes with fanout, which is often the case, the ODCs of a node are often due to the impact of only a few levels of fanout logic.

Our approximate simulator can be inaccurate for circuits with reconvergent paths because our per-node computation treats the fanout cone of each immediate output as being disjoint from each other. We now show why our approximate simulator rarely produces

false positives and negatives.[5] To understand the impact of reconvergent paths on the accuracy of our simulator, consider the function $G(g_1, g_2)$, where $g_1(f(y), A)$ and $g_2(f(y), B)$ as before. G represents a reconvergent node. We note the following:

Theorem 4 *The probability that approximate simulation produces an error in f's ODC set is $P_{error} \leq (1 - P_{X_i \in ODC_{g_1}(f)})(1 - P_{X_i \in ODC_{g_2}(f)})$.*

Proof. If $X_i \in ODC_{g_1}(f)$ and $X_i \in ODC_{g_2}(f)$, then f is not observable at the inputs G. If $X_i \in ODC_{g_1}(f)$ or $X_i \in ODC_{g_2}$, then the inputs to G can be accurately analyzed independently since only one of the inputs experiences an observable difference. An error in the approximate simulator can only occur when neither $X_i \in ODC_{g_1}(f)$ nor $X_i \in ODC_{g_1}(f)$. □

According to Example 1, there is a high probability that f is unobservable with respect to a single node after a few levels of logic for certain commonly used functions with low entropy, such as ANDs and ORs. When there are few reconvergent nodes, with respect to the total number of nodes in the circuit (we have observed this empirically), the upper-bound for P_{error} becomes very small. We also showed in Example 2 the impact of multiple outputs on the observability of node f. There may be nodes other than g_1 and g_2 that fanout from f and increase f's observability independent of the reconvergence, reducing the probability of error in the approximate analysis.

5.2.4 Performance of Approximate Simulator

In Table 5.2, we report the empirical data on the runtime efficiency of our approximate ODC simulator. The first column indicates the benchmarks examined. The second column, `sim`, gives the time required to generate only signature S for each node. We use this as

[5] In Section 5.2.2, we explain that false positives and negatives can be tolerated in our framework.

circuit	runtime(s)		
	sim	simodc	our approx
ac97_ctrl	1	6	1
aes_core	2	79	1
des_perf	9	410	7
ethernet	4	76	2
mem_ctrl	1	119	1
pci_bdge32	1	28	1
spi	0	39	0
systemcaes	1	48	1
systemcdes	0	24	0
tv80	1	130	1
usb_funct	1	11	1
wb_conmax	3	69	4

Table 5.2: Efficiency of the approximate ODC simulator.

circuit	considering x downstream levels [95]					our global
	2	4	8	16	32	algorithm (s)
ac97_ctrl	1.0	1.0	1.0	1.0	1.0	1.0
aes_core	3.0	3.1	3.4	6.3	7.9	3.0
spi	0.4	0.5	0.5	1.8	11.2	0.4
systemcaes	2.3	2.4	2.6	11.9	1300.0	2.3
systemcdes	0.3	0.3	0.3	0.5	0.6	0.3
tv80	2.2	2.3	2.6	8.2	363.0	2.2
usb_funct	2.2	2.3	2.4	2.8	3.3	2.2

Table 5.3: Runtime comparison between techniques from [95] and our global simulation.

a baseline to assess the cost of generating masks. The third column, `simodc`, shows the time required to generate S^* for each node using Equation 2.2. The fourth column, `our approx`, shows the time to compute S^* using the approximate simulator. The results indicate that the approximate simulator's runtime is comparable to that of `sim` and is much faster than `sim_odc`. These results were generated by running 2048 random simulation vectors.

In Table 5.3, we compare our simulator that considers ODCs by examining all downstream logic with the implementation in [95] where a local don't-care analysis is per-

formed per node considering only a few levels of downstream logic. We show runtimes for [95] as a function of downstream levels considered. Notice that our simulator accounts for more don't-cares while achieving better runtimes. In some circuits, like `systemcaes`, considering more levels of logic is prohibitive using [95] due to the depth of the circuit. The runtime similarities between considering only 2 levels and our implementation suggests that the contribution of our ODC simulation is insignificant compared to the runtime to generate the initial signatures and parse the design. Furthermore, these results show that computing don't-cares using an $O(N^2) - time$ algorithm can become prohibitive. When ODC analysis needs to be performed repeatedly for each circuit change, such as reliability-guided synthesis [49], inefficient ODC computation can become a significant computational bottleneck.

5.3 Concluding Remarks

In this chapter, we developed an efficient algorithm for computing don't-cares using functional simulation. Our strategy scales to large circuits and can compute global don't-cares whereas previous work is limited to examining smaller windows of logic to compute don't-cares. By efficiently analyzing don't-cares throughout the circuit, we can potentially expose more optimizations, which is especially important late in the design flow where fewer opportunities for improvement exist.

Part III

Improving the Efficiency of Formal Equivalence Checking

The results of bit signature-based circuit analysis and transformations must be verified by formal methods to ensure correctness for all input combinations. Generating high-quality signatures using the techniques of the previous chapters increases the likelihood that our the transformations suggested by our abstraction are correct. However, even if the abstraction is generally accurate in guiding optimizations, verification is still necessary and can be prohibitively time-consuming, especially for larger designs. In this part of the dissertation, we propose a solution to accelerate the verification of signature-based abstractions. Chapter VI introduces a strategy to minimize the size of the logic block considered when verifying an abstraction. In Chapter VII, we propose a parallel methodology for general-purpose SAT solving that relies on increasingly prevalent multi-core systems as a means to partially counteracting the increasing cost of verifying complex optimizations in larger designs.

CHAPTER VI

Incremental Verification with Don't Cares

In previous chapters, we have introduced techniques for improving the quality and flexibility of bit signatures. These signatures can efficiently identify logic optimizations because of their ability to distinguish nodes with a small number of input vectors. However, if one desires to determine whether two nodes are equivalent when their corresponding signatures are equal, a formal proof mechanism is needed to check for possible corner-case behavior not captured by the given signatures. Therefore, refining the simulation [59] is an important mechanism to limit the number of signatures that falsely suggest equivalence, thus minimizing the number of expensive proofs.

Additionally, incorporating observability don't-cares into signatures introduces new challenges to both producing high-quality simulation and verifying the correctness of the abstraction. In this chapter, we address these challenges by introducing an incremental verification strategy that dynamically adjusts the complexity of the verification instance based on the amount of downstream logic required to prove equivalence up to don't-cares. In Section 6.1, we outline and formalize some of the challenges involved in verifying abstractions with don't-cares. In Section 6.2, we introduce our incremental verification methodology, and provide concluding remarks in Section 6.3.

6.1 Verifying Signature Abstractions

Using a SAT solver to verify equivalence can be computationally expensive. However, a high-quality selection of simulation vectors limits the number of *false positives*.[1] In general, random simulation generates signatures capable of distinguishing two independent random functions f and g with n inputs. In this case, the probability that the signatures incorrectly indicate equivalence, P_{error}, is simply the joint probability that $S_f = S_g$ and $f \neq g$. Under k input vectors this is:

$$P_{error} = P\big((S_f = S_g) \cap (f \neq g)\big) = \frac{1}{2^k}(1 - \frac{1}{2^{2^n}}) \approx \frac{1}{2^k} \tag{6.1}$$

where P_{error} decreases exponentially as k increases. The term $\frac{1}{2^k}$ corresponds to the probability that $S_f = S_g$ for k input vectors. The term $1 - \frac{1}{2^{2^n}}$ corresponds to the probability that two n-input independent random functions are not equivalent (where the number of n-input Boolean functions is 2^{2^n}). For this case, a small number of random simulation vectors is sufficient to distinguish nodes and avoid false positives.

Logic functions implemented by practical circuits exhibit structural properties and are often dependent on one another. We can account for this in our analysis by defining the $DIFFSET$ between function f and g.

$$DIFFSET(f,g) = \big(ONSET(f) \cap OFFSET(g)\big) \cup \big(OFFSET(f) \cap ONSET(g)\big) \tag{6.2}$$

where $ONSET(f)$ is the set of minterms of f and $OFFSET$ is its set of maxterms. Equivalently the $DIFFSET(f,g) = ONSET(f \oplus g)$.

[1] Here we use the term *false positive* to refer to incorrect equivalent nodes suggested because of a signature match.

Given this, we can express P_{error} for $f \neq g$, *i.e.*, $|DIFFSET(f,g)| > 0$, as:

$$P_{error} = \left(1 - \frac{|DIFFSET(f,g)|}{2^n}\right)^k \tag{6.3}$$

In other words, this is the probability that S_f and S_g are equal. $\frac{|DIFFSET(f,g)|}{2^n}$ is the fraction of input combinations where f and g are different. As the number of k input vectors increases, P_{error} decreases.

For functions encountered in practice, $|DIFFSET(f,g)|$ is often fairly large, indicating that random simulation would rarely produce signatures leading to false positives. For instance, OR functions have $2^n - 1$ minterms, AND functions have 1 minterm, and XOR functions have $\frac{2}{2^n}$. These common associative functions can often be distinguished from each other quickly by simulation because they exhibit significant differences.

The NOR function has only 1 minterm, as the AND function; therefore a large number of input vectors k is needed to achieve a low P_{error} when comparing the signatures of AND and NOR. To reduce the size of k needed to distinguish nodes, *simulation refinement* [47, 59] is commonly performed through SAT-generated counterexamples. In simulation refinement, a miter is constructed between two nodes with matching signatures, and a SAT solver attempts to satisfy its output. If a solution is found, the solution vector (*dynamic simulation vector*) is applied to the circuit so that the signature of each node in the circuit increases to size $k+1$. Not only does this new vector distinguish the two nodes, but it typically also improves the quality of the signatures in the nodes' fanin and fanout cones.

The impact of don't-cares. When ODCs in the circuit are taken into account, more input vectors are usually required to achieve the same P_{error} between f and g. Given $ODC(f)$, we wish to check whether g can implement f in the circuit. To do this, we check

if $S_f \& S_f^* \equiv S_g \& S_f^*$ (where S_f^* is the ODC mask of f). In other words, we check if the signatures of the two nodes match, after masking the don't care bits of f. The impact on the probability of finding a match incorrectly is formulated below:

$$P_{error} = \left(1 - \frac{|DIFFSET(f,g) - ODC(f)|}{2^n}\right)^k \tag{6.4}$$

where the elements in $DIFFSET(f,g)$ that are in $ODC(f)$ are removed. As a result, $\frac{|DIFFSET(f,g)-ODC(f)|}{2^n}$ is the fraction of input combinations where f and g have *observable* differences. In some cases, internal nodes in the circuit are not easily controllable, and hence a large k is needed to limit P_{error}.

Limitations of previous approaches. The equivalence of two nodes, f and g, in a network can be determined by constructing a miter [13] between them and asserting the output to 1 as shown by the following formula:

$$\left(F = G\right) \Leftrightarrow \left(\forall_X \, F(X) \oplus G(X) \neq 1\right) \tag{6.5}$$

where X is an input vector.

Since exploiting ODCs entails including downstream logic, verifying ODC-based mergers could require a miter on the primary outputs of the circuit. Figure 2.4 shows how ODCs can be identified for a given node in a network. In a similar manner, we can prove whether the signatures of b and a match up to ODCs. Instead of using a' in the modified circuit D^*, b is substituted for a and miters are constructed at the outputs. If the care-set determined by Equation 2.1 is null,[2] b matches a. A single satisfiable solution is needed to expose a difference between a and b. Notice that this approach requires the entire circuit to be

[2] $C(a) = \bigcup_{i:D(X_i)\neq D^*(X_i)} X_i$.

considered, resulting in large SAT instances.

6.2 Incremental Equivalence Checking up to Don't Cares

To improve the quality of equivalency checkers, we propose an incremental verification framework where the size of the SAT instance is dynamically adjusted between each SAT solver call. We only consider the smallest required logic block to determine equivalence. Furthermore, by reusing internal data structures between SAT calls, decision heuristics used in SAT solving [65] can be refined. Many learnt clauses [80] can also be reused between calls to prune the search space and boost the performance of the SAT solver. Our incremental strategy has an important advantage — equivalence analyses that are not critical can be aborted if their verification takes too much time. In other words, we can use the runtime cost of verification as a factor in determining whether verifying a match is worthwhile.

6.2.1 Moving-dominator Equivalence Checker

We introduce here a SAT framework that determines equivalence in the presence of don't-cares by considering only a small portion of downstream logic. Consider Figure 6.1, where g is a candidate node to be merged with f up to don't-cares. If a miter is constructed across f and g instead of the primary outputs as shown in part a), a set of differences between f and g that results in satisfying assignments is given by $DIFFSET(f,g)$. (A satisfying solution here indicates the non-equivalence for the given circuit nodes.) If one of these differences between the two nodes is observable at the primary outputs (by examining the downstream logic of f), then non-equivalence that considers ODCs is proven. If none of these differences are observable or if the $DIFFSET$ is null, then g can be merged

with f.

However, if f and g have a large size $DIFFSET$, this could lead to a prohibitive amount of simulation since each difference in $DIFFSET$ is propagated from node f to the circuit's outputs. To reduce the size of $DIFFSET$, we construct miters farther from the potential merger site at node f while minimizing the amount of downstream logic considered in the mitered circuit. We introduce the notion of a *dominator set* to define where we place the miters.

Definition 6.2.1 *The dominator set for node f is a set of nodes in the circuit such that every path from node f to a primary output contains a member in the dominator set and where, for each dominator member, there exists at least one path from node f to a primary output that contains only that member. Multiple distinct dominator sets can exist for a given node.*

6.2.2 Verification Algorithm

In part b) of Figure 6.1, we show miters constructed for a dominator set of f. Dominator sets close to the source node f result in simpler SAT instances but potentially require more downstream simulation to check whether the satisfying assignments indeed prove the equivalence of f and g. We devise a strategy that dynamically moves the dominator set closer to the primary outputs depending on the satisfying assignments generated. Our "moving-dominator" algorithm is outlined in Figure 6.2.

The moving-dominator algorithm starts by deriving a dominator set that is close to the merger site given by `calculate_initial_dominator()`. Then the `dom_SAT()` function solves an instance where miters are placed across the current dominator set.

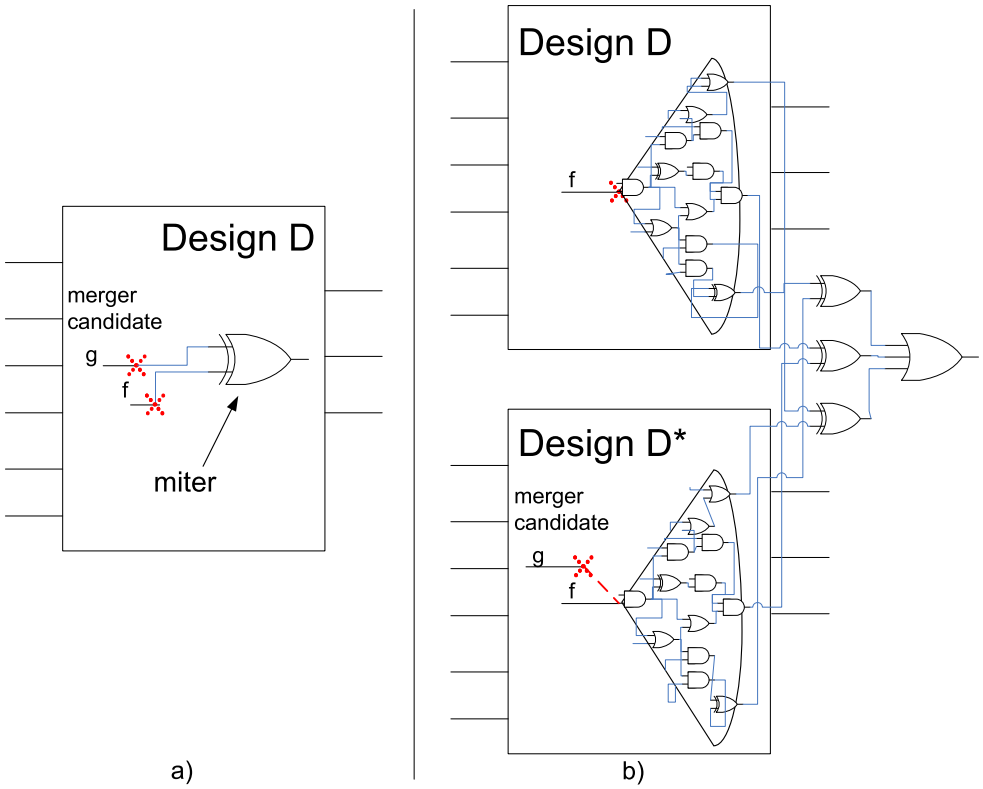

Figure 6.1: An example that shows how to prove that node g can implement node f in the circuit. a) A miter is constructed between f and g to check for equivalence, but it does not account for ODCs because the logic in the fanout cone of f is not considered. b) A dominator set can be formed in the fanout cone of f and miters can be placed across the dominators to account for ODCs.

An UNSAT solution implies that the two candidate nodes are indeed equivalent, and the procedure exits. If a satisfying solution is found, it is propagated on downstream logic from the current dominator set. If the input vector corresponding to the satisfying assignment does not result in an ODC at f, then node g cannot implement f. Otherwise, the procedure must be refined: a new dominator set is generated as determined by `calculate_new_dominator()`, which moves the miters closer to the outputs.

With each invocation of the SAT solver, we add constraints that are particular to the current dominator set, as well as increase the size of the SAT instance to account

```
bool odc_match(f, g){
  current_dom = calculate_initial_dominator();
  while(dom_SAT(miter(current_dom, f, g)) == SAT){
      if(simulation(satisfying_solution){
          return false;
      }
      else{
          current_dom = calculate_new_dominator();
      }
  }
  return true;
}
```

Figure 6.2: Determining whether two nodes are equivalent up to ODCs.

for the additional downstream logic considered. When the dominator set is adjusted by `calculate_new_dominator()`, some of the constraints needed for the previous dominator set are no longer relevant; we remove these constraints and add new ones to the SAT instance. By incrementally building the SAT instance each time the dominator set is moved, we can reuse information learned by a SAT solver between several SAT calls.

ATPG techniques can also be substituted for the SAT-engine described in the previous algorithm. By placing a MUX with a dangling select input between the two nodes in the potential merger, we can generate test patterns for *single-stuck-at faults* (SSF) on the MUX select input. If a test pattern cannot be generated, the merger can take place because both nodes have the same effect on the outputs. Similarly, the circuit considered can be limited by the dominator set, and a test pattern counterexample can be used to refine it.

6.2.3 Calculating Dominators

Using simulation, we calculate a dominator set that attempts to minimize the amount of downstream logic necessary to prove a merger. In general, we check the downstream logic required to prove specific ODCs for certain input combinations and use that to determine

an initial dominator set. We then use counterexamples produced by the SAT solver to refine the dominator set. Details of this approach are outlined below.

In Figure 2.4, ODC(f) is derived by examining observability at the primary outputs. However, by placing miters along a cut defined between f and the primary outputs, it is possible to calculate an ODC-set for f, $ODC_{cut}(f)$, where $ODC_{cut}(f) \subseteq ODC(f)$. Previously, we defined this cut as the dominator set. An ideal dominator set would be the closest cut to the merger site sufficient to prove equivalence. We define the minimal dominator set as follows:

Definition 6.2.2 *The minimal dominator set D_{min} for proving that g can implement f is the closest cut to f such that $DIFFSET(f,g) \subseteq ODC_{D_{min}}(f)$.*

The function `calculate_initial_dominator()` is used to calculate an initial dominator set. We randomly select several input vectors X_i and generate an approximate D_{min} using Definition 6.2.2 by constructing $DIFFSET(f,g)$ and $ODC(f)$ from the X_is. Since not all input vectors are considered, it is possible that the cut obtained is an under-approximation and that the SAT solver fails to detect equivalence. To improve the approximation, `calculate_new_dominator()` extends the cut farther from f for every satisfying assignment found by `dom_Sat()`.

6.3 Concluding Remarks

We introduced an incremental verification methodology to reduce the complexity of SAT instances when verifying our signature abstractions. Since many ODCs occur within few logic levels from the focus circuit node [95], ODC analysis through even a small number of logic levels can bring significant runtime improvements. Our dynamic approach

finds the smallest logic window to verify a node merger that requires ODCs and produces counterexamples to refine signatures accordingly. In later chapters, this incremental verification algorithm is used to verify optimizations in the presence of don't-cares, where it outperforms equivalence checking applied at a circuit's primary outputs.

CHAPTER VII

Multi-threaded SAT Solving

As shown in the last chapter, our bit signature-based transformations rely on SAT-based equivalence checking for validation, occasionally requiring the solution of very complex instances. We observe that SAT computation can be a runtime bottleneck in our signature-based synthesis framework. In this chapter, we propose a novel parallel SAT strategy to exploit increasingly prevalent multi-core architectures, which feature a large shared memory and have the ability to execute several threads simultaneously. Multi-threaded SAT solving can be used to reduce the runtime of verifying signature-guided optimizations, so that more powerful optimizations become practical. We discuss the theoretical underpinnings of our approach to SAT parallelization and how it improves upon previous parallel SAT strategies.

7.1 Parallel-processing Methodologies in EDA

"Intrinsically parallel" tasks, such as multimedia processing, may achieve N times speed-up by using N cores (assuming that sufficient memory bandwidth is available and that cache coherency is not a bottleneck). However, combinatorial optimization and search problems, such as SAT-solving and integer linear programming, are much harder to parallelize. The straightforward solution — to process in parallel different branches of a given

decision — often fails miserably in practice because such branches are not independent in leading-edge solvers that rely on branch-and-backtrack. The recent "View from Berkeley" project [7] designates these problems as one of thirteen core computational categories for which parallel algorithms must be developed. In this chapter, we propose new techniques to parallelize state-of-the-art SAT solving.

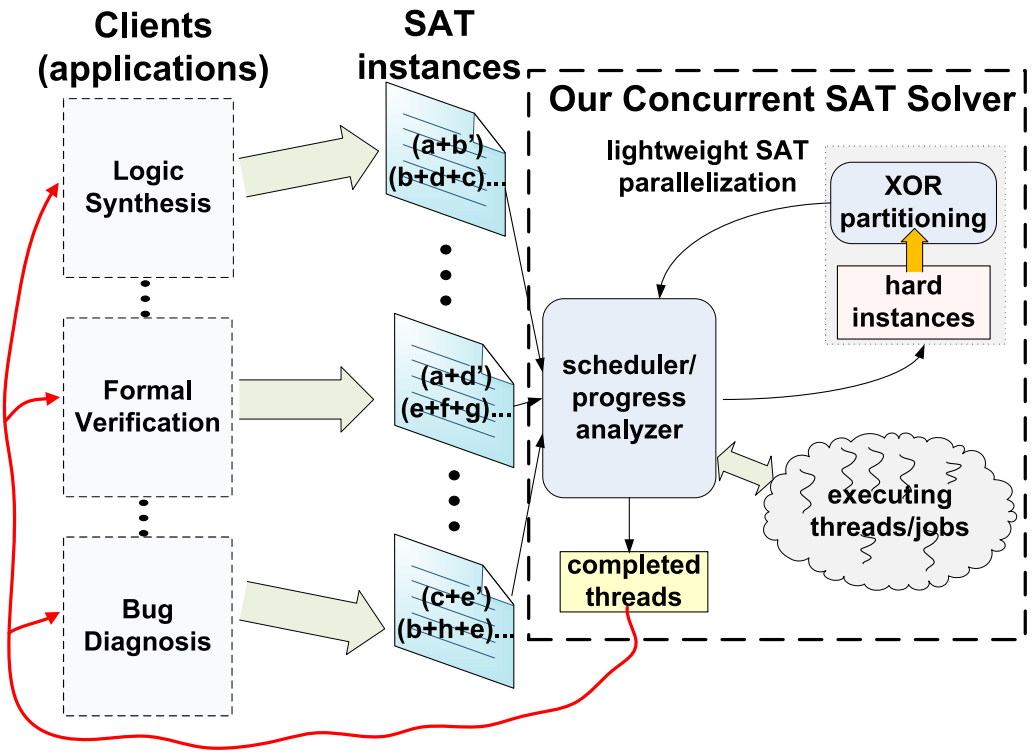

Figure 7.1: High-level flow of our concurrent SAT methodology. We introduce a scheduler for completing a batch of SAT instances of varying complexity and a lightweight parallel strategy for handling the most complex instances.

As of 2008, most EDA frameworks are being rapidly extended to make use of multi-core architectures, *i.e.*, run several cooperating threads in parallel. In particular, state-of-the-art techniques for design optimization such as SAT sweeping [69, 95], SAT-based technology mapping for FPGAs [52] and, in our case, logic resynthesis require solving

multiple SAT instances. In the case of many key EDA algorithms, the computation of these SAT solutions constitutes their bottleneck, and solving them in parallel offers a chance to speed up a broad range of EDA tools. Shared-memory systems and multi-core CPUs are particularly amenable to such parallelization strategies.

We first introduce a novel framework for scheduling and solving multiple instances of hard SAT problems on shared-memory systems such as multiprocessors, as illustrated in Figure 7.1. Different client applications (such as formal verification) produce several SAT instances which are issued to our concurrent SAT solver. These instances are put into a priority scheduler, so that easier instances are finished first, while harder ones are solved using an XOR partitioning strategy.

The first problem we address is that of scheduling of M SAT instances on N processors when $M > N$. Take, for example, the case $N = 1$. If runtimes are known for each instance in advance, then scheduling instances of increasing runtime guarantees the best *batch latency*, *i.e.*, as the sum of completion times of all instances from the beginning of the batch. In other words, a long-running job does not delay numerous small jobs. Scheduling for N processors, and without *a priori* runtime information, is more involved, and our work is the first to address this problem. Furthermore, many applications generate individual SAT instances rather than batches — the technique we propose handles this case as well. The need to parallelize individual SAT instances arises primarily when no other instances remain to be solved to keep all available cores busy. In our framework, we parallelize the hardest SAT instances after they have run sequentially for some time.

In this chapter, we achieve two performance goals: 1) minimization of the average latency for solving a group of SAT problems while ensuring maximum resource utiliza-

tion and 2) minimization of runtime for large problem instances by exploiting concurrent resources. To reduce the average latency of a collection of SAT instances, we introduce a novel scheduling algorithm that combines the benefits of time-slicing and batch scheduling. We achieve a 20% average latency improvement over previous techniques while improving resource utilization. To reduce the runtime for single large instances, we consider a novel partitioning scheme based on including additional constraints to an instance to reduce the size of the search space. We exploit a theoretical result from [81] on randomized polynomial-time algorithms, where adding a limited number of random XOR constraints to a SAT instance can reduce it from one with multiple solutions to one with a single solution. We are the first to apply this result to search-space partitioning in multi-core SAT solving, circumventing a major pitfall common to parallel SAT solver algorithms, *i.e.*, unbalanced partitioning [89]. We further observe that search-space partitioning is best performed when the random restart frequency is low, a common problem when the initial part of the search is conducted sequentially. We validate our parallel methodology by performing extensive experiments on an eight-core system and improve resource utilization by 60.5% over prior work based on solver portfolios.

In Section 7.2, we analyze the issues that are at the core of the high variability of execution for SAT solvers. Section 7.3 introduces our scheduling algorithm for handling multiple SAT instances of varying complexity in a parallel setting. We discuss the limitations of previously proposed parallel solutions in Section 7.4. In Section 7.5, we present a partitioning strategy that provides search-space division along with our strategy for load balancing. We analyze the effectiveness of our approach in Section 7.6 and conclude in Section 7.7.

7.2 Runtime Variability in SAT Solving

While DPLL SAT solvers typically struggle on randomly generated instances, most practical SAT instances possess regular structure and can be solved much faster. However, it has been observed that many practical instances experience exponential runtime variability [39] when using backtrack-style SAT solvers even without algorithmic randomization. This variability can be observed by comparing runtimes of different algorithms on a given instance, and can be formalized through the notion of heavy-tail behavior, summarized below.

Definition 7.2.1 *For a random variable X, corresponding to the search cost for a particular heuristic, a heavy-tail probability distribution exists if $Pr[X > x] \propto x^{-\alpha}$ as $x \rightarrow \infty$ for $0 < \alpha < 2$.*

If the cumulative probability does not converge to 1 quickly enough, the distribution exhibits a heavy-tail. More specifically, the variance of X is ∞, and when $\alpha < 1$ the mean is also ∞. In performance analysis of a single SAT solving algorithm with randomization, or multiple SAT algorithms, the random variable X can capture the number of backtracks required to solve a given instance. Also, since the maximum runtime is exponential, the bounded heavy-tail produces variance that is actually exponential in the number of backtracks.

Random restarting (see Section 2.1.1), which is now extensively used in DPLL-based solvers and involves a worst-case polynomial number of restarts, can eliminate heavy-tail behavior [39]. Intuitively, random restarts prevent a solver from getting stuck in a difficult part of the search space. Portfolio strategies [38] offer similar benefits because each

heuristic tend to explore different parts of the search space. Furthermore, each heuristic can utilize multiple restarting strategies, which in turn can produce more improvement.

Backdoor variables. We now discuss the impact of *backdoors* on the performance of branch-and-backtrack types of SAT solvers.

Definition 7.2.2 *Backdoor [85] variables for a SAT instance are a set of variables that under some assignment produces a sub-problem solvable in polynomial time.*

For example, a backdoor may yield a residual SAT instance that can be solved by a linear-time 2-SAT algorithm.

Definition 7.2.3 *Given a Boolean formula $F(V)$ and a set of variables $B \subseteq V$, B is a backdoor if* $\exists A_B[F_{A_B} \in \mathcal{P} \wedge F_{A_B} \neq 0]$, *where $A_B \in \{0,1\}^{|B|}$ is an assignment to the set of variables B.*

In [85], it was observed that many common problems contain a small *backdoor set*.

Definition 7.2.4 *Given a Boolean formula $F(V)$, a partial variable assignment B is a* strong backdoor *if* $\forall A_B[F_{A_B} \in \mathcal{P}]$.

There are $2^{|B|}$ combinations that need to be examined to solve an unsatisfiable instance for a total runtime of $2^{|B|}\mathcal{P}(F_{A_B})$, where $\mathcal{P}(F_{A_B})$ is the runtime of the polynomial algorithm under a given assignment. Empirical evaluation in [85] suggests that many practical problems have $|B| \propto log(|V|)$ resulting in total runtime of $|V|\mathcal{P}(F_{A_B})$ if the backdoor set is known. Although determining this set is not always computationally feasible, decision heuristics such as VSIDS implicitly look for such sets as they tend to favor variable assignments that lead quickly to a full evaluation of an instance. It was also explained in [85] that

randomly generated instances tend to have considerably larger backdoors, approximately 30% of $|V|$. The efficient determination of a backdoor, explicitly or implicitly, is often key to the performance of a branch-and-backtrack SAT solver.

7.3 Scheduling SAT Instances of Varying Difficulty

The goal of our scheduling strategy can be formally expressed as follows: given M different SAT instances and an N-threaded machine, we wish to solve them in a way that minimizes the total accumulated latency:

$$min\Big(\sum^{M} T_c(m)\Big) \text{ where } \forall_t S_t \leq N \tag{7.1}$$

where $T_c(m)$ is the completion time for problem m, and S_t is the number of instances being solved in a particular time-slice t. Note, when $N = 1$, this formulation considers the case of having a single thread of execution. Ideally, the completion time T_c for the last instance m_f when using $N > 1$ threads, should be N-times smaller than for $N = 1$ to fully utilize the parallel resources.

Optimizing the objective above, subject to resource constraints, can lead to a schedule that minimizes the total latency for completing all SAT instances. Assuming that incoming instances are independent and equally important to solve, minimizing latency is a way to ensure that feedback is provided to as many clients as possible in a timely manner. This may unblock the largest number of clients waiting for results (see also Figure 7.1). In the case where the runtimes for all instances are approximately equal, optimizing the latency objective is trivial as the problems can be solved in any order. However, as shown in Figure 7.2, a block of instances can experience a wide variance in runtime. In particular, by

analyzing the distribution of runtimes from the SAT 2003 competition [51], which contains several benchmark suites, we observe a bipolar trend whereby most instances either finish in the first five minutes or timeout after 64 minutes. An optimal schedule for an N-threaded machine involves scheduling problems in increasing order of complexity on each thread. Unfortunately, predicting actual runtimes beforehand is not possible. However, we will discuss strategies for mitigating this limitation later in this section.

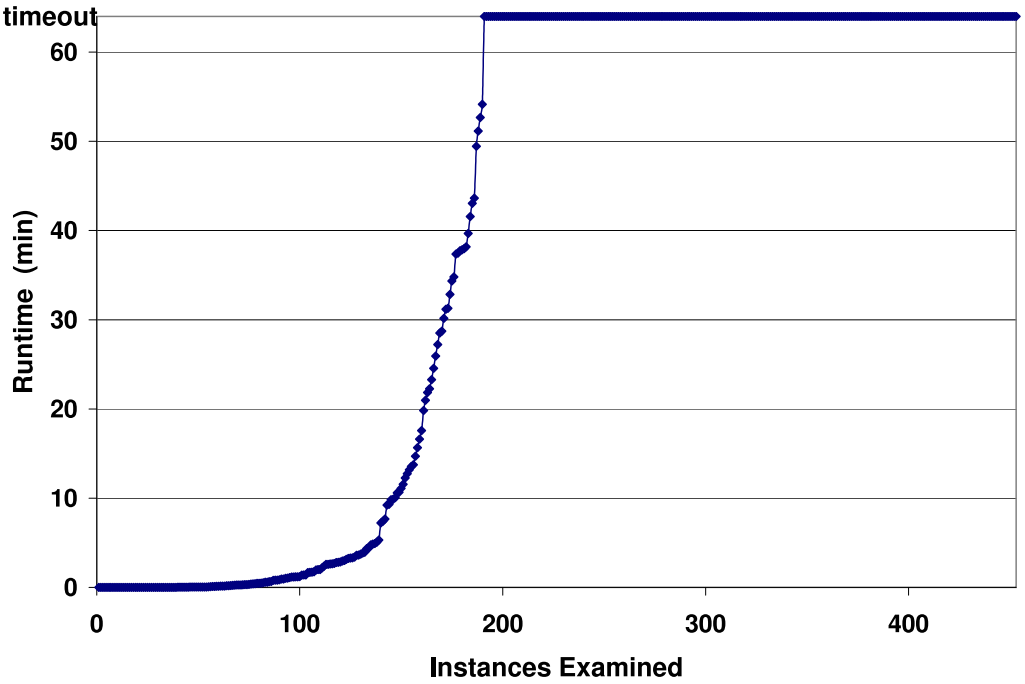

Figure 7.2: Number of SAT instances solved vs. time for the SAT 2003 collection. The timeout is 64 minutes.

Because the distribution of runtimes is uneven, it is possible that, random scheduling could result in some threads completing execution much after others, leading to poor resource utilization. To even the execution latency across threads, we can leverage schedulers available in most operating systems, which usually exploit time-slicing. Through time-slicing, problems with short runtimes finish fairly quickly, while longer instances tend to complete at approximately the same time.

Our solution relies on an estimate of the distribution of SAT runtimes to predict a time threshold beyond which the unsolved problems are likely to have high complexity. We also explore other techniques which are not dependent on predictive distributions to evaluate possible overall better latency. From Figure 7.2, we see that this time threshold should be approximately 5 minutes. Thus, for the first solving period, up to the threshold time, we perform time-sliced scheduling over all the problems, after that we increase the thread priority for only N instances (where N is the number of threads available) so that they run in batch mode.

To further reduce the average latency, we can lower the priority for instances that require large memory resources, and thus negatively impact system performance. This was unnecessary in our experimental evaluation since the instances we considered had low memory profiles.

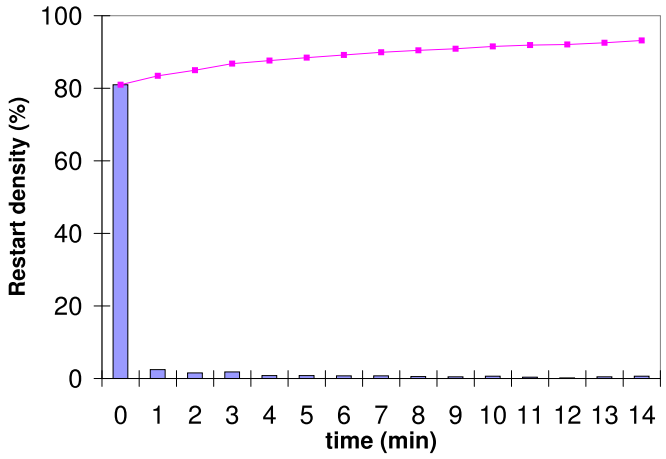

Figure 7.3: Percentage of total restarts for each minute of execution for a random sample of instances from the SAT 2003 collection.

Although not implemented here, scheduling can be based on runtime estimates generated from progress meters found in some SAT solvers [4]. The thread priority for simpler

instances can be increased in this manner. For example, one could consider random restart frequency, or the percentage of restarts performed each minute. In Figure 7.3, we show a distribution of restarts over a randomly chosen sample of instances from the SAT 2003 collection. It reveals an exponential decay in frequency, which can be used as a guide to lower thread priority. When few restarts occur, there are fewer opportunities to quickly arrive at a solution due to a better variable order.

7.4 Current Parallel SAT Solvers

Previous efforts at parallelizing algorithms for solving *random* SAT instances have been effective as indicated in [58], but random instances are not common in EDA applications whose, problems exhibit structure. For such instances, [55] represents the state of the art, proposing a solution that exploits shared memory to enable efficient learning between solvers running on different threads. In this section, we overview some pitfalls of this approach and discuss some limitations of portfolio solvers.

Search space partitioning using guiding paths, as proposed in [55], is limited because the partitioning may be unbalanced. This may circumvent the effectiveness of random restarts by forcing initial assignments to each concurrent solver. Addressing this problem by undoing the initial assignments for a thread after each random restart appears to undermine the benefits of partitioning. The partition itself may also generate sub-problems that demand very different runtimes. Furthermore, learning between threads is not always an effective means of boosting performance. As discussed in [90], using *1-UIP* learnt clauses is often more effective at improving the solver's performance than using minimally-sized learnt clauses. This counter-intuitive result suggests that parallel schemes for learning, which often use the size of learnt clauses as a filtering mechanism, are not an effective

mechanism for boosting the performance of a particular thread of execution.

Implementing these parallelization strategies requires careful selection of a successful sequential solver. Choosing a poor heuristic for parallelization still leads to poor performance, especially in a portfolio where it consistently under-performs compared to other heuristics. Furthermore, the heuristics implemented in the most successful SAT solvers are finely-tuned, which would require much careful and time-consuming development when porting to parallel optimizations. The slightest perturbation to the quality of the sequential algorithm caused by parallelization (such as excessive learning between threads) can significantly degrade runtime performance. For example, learning increases the size of the clause database which, in turn, increases the cost of Boolean constraint propagation. Furthermore, decision heuristics, such as VSIDS, are guided by learning, and can therefore be affected by it.

Portfolio solvers are advantageous because their implementation overhead is minimal and have low risk of performing poorly on instances with highly variable runtime. However, this approach requires that the various heuristics have different performance characteristics on different types of instances. As larger computing systems become available, it is increasingly difficult to find large collections of different heuristics. Furthermore, even where orders-of-magnitude improvements are possible, some instances may show no improvement, resulting in small overall speed-up.

7.5 Solving Individual Hard Instances in Parallel

In this section, we propose an algorithmic methodology that utilizes available resources to reduce the runtime of hard instances. We overcome the limitations described previously by introducing a novel approach for partitioning the search-space, which allows for more

flexible random restarts. Furthermore, our approach can be easily adopted by any state-of-the-art DPLL-based solver.

7.5.1 Search Space Partitioning

Our technique for partitioning the search space of a SAT solver relies on the inclusion of additional XOR constraints to the instance. In this section, we first elaborate on the theoretical underpinnings of adding XOR constraints and then discuss its significance in dividing a search space approximately evenly.

Reducing the search space through XOR constraints. To partition the search space, we extend the work for solution-space reduction that was initially presented in [81]. The authors of [81] show that the inclusion of an XOR constraint ($(x_1 \oplus x_2 \oplus \cdots \oplus x_i \oplus 0)$ as shown in Equation 4.6) to an instance $F(V)$ probabilistically reduces its solution space by approximately half. We call the instance obtained after adding this constraint F_{even} because the assignments to the x_i variables must have even polarity to satisfy F_{even}. Correspondingly, we call F_{odd}, the instance:

$$F_{odd} = F \wedge (x_1 \oplus x_2 \oplus \cdots \oplus x_i \oplus 0) \tag{7.2}$$

where the x_i variables are the same as in F_{even}. $\{F_{even}, F_{odd}\}$ is then a disjoint partition of the solution space. More formally:

Definition 7.5.1 *A disjoint partition exists when (1) $F = F_{even} \vee F_{odd}$, (2) $F_{even} \wedge F_{odd} = 0$, and (3) the set of variables $x_i \in V$ is the same for F_{even} and F_{odd}.*

This partitioning generates two sub-problems that can be assigned to different solvers. The sub-problems can be recursively divided by adding more XOR constraints. As a gen-

eralization of the result in [81], each XOR constraint probabilistically divides the number of possible assignments of the V variables roughly in half, *i.e.*, $2^{|V|-1}$. Hence, the constraint divides the search space approximately in half, probabilistically balancing the workload between different solvers addressing the two sub-problems.

In practice, simply adding large XOR constraints is inadequate for reducing the search space, because no conflict is generated until all of the x_i variables are assigned, approximately $\frac{|V|}{2}$ variables.[1] In other words, such a constraint divides the search space evenly, but it is ineffective at restricting the search until after nearly all assignments have been made. To address this, we investigate smaller XOR constraints, derived from the original complex ones, that still achieve the same theoretical result.

Connection between backdoors and randomized reductions. As an example of how we can add smaller constraints, consider a combinational circuit D with m inputs (a more general strategy is presented in Section 7.5.2). This circuit can be converted to a SAT instance $D(V)$ with V variables where the set of possible solutions determined by assignments to the primary inputs is M, with $|M| = 2^m$. Therefore, the set of solutions $(S_D \in 2^{|V|})$ corresponds to the set of solutions M. In other words, any assignment $A_M \in 2^m$ results in precisely one solution. According to Definition 7.2.4, M is a strong backdoor for $D(V)$. By restricting the set of variables x_i to variables in M, we can construct a partition that gives the same probabilistic guarantees as the original formulation, but produces a smaller XOR constraint while generating conflicts sooner if these variables are assigned first. Namely, an XOR constraint on the variables in M divides the solution space roughly in half.

[1] In [81], each variable is randomly chosen to be in the XOR constraint with the probability of $\frac{1}{2}$.

7.5.2 Lightweight Parallel SAT

For a general SAT instance, we can restrict XOR constraints to involve only the typically small set of backdoor variables, where the XOR constraint can cut the search space roughly in half to $2^{|B|-1}$.

Multi-threaded SAT framework. For a circuit, we showed that the primary inputs can be used to derive small XOR constraints. In the following, we propose a more general approach that approximately determines the backdoor set of variables to generate small XOR constraint. Because computing the smallest backdoor set explicitly is not always feasible, we use, as an approximation, highly ranked variables determined by selection heuristics in modern DPLL-based solvers like VSIDS. Since [85] observed that many backdoor sets have cardinality $\log_2(|V|)$, we choose x_i from the top $\log_2(|V|)$ variables to generate small XOR constraints. To generate variable rankings, we run a SAT solver for a certain amount of time (determined experimentally) before generating these XOR constraints.

Algorithm. In Figure 7.4, we show the pseudo-code of our algorithm using XOR partitioning to improve the performance of SAT in a parallel environment. `psat_solve()` is a SAT solver invoked with the CNF instance (*cnf*), the number of random restarts after which the problem should be partitioned (*passes*), the mode of execution (the default mode is sequential *seq*), and any initial variable assignments *assumps*. This allows very simple instances to be completed sequentially and *trains* the solver so that good variables are chosen for partitioning. When partitioning is required, we add an XOR constraint involving the top $log(|N|)$ variables through `add_xor_constraints()`. Because the XOR constraint is typically small, we don't require a specialized XOR constraint representation as in [35]. We then spawn two threads and wait for their results. Notice that

the threaded mode uses the same infrastructure as the sequential mode with only a few minor changes. To maintain an even division of work between the two threads, we ensure that the partitioning variables `part_vars` are ranked high (we increase their rank after restarting). Because multiple variables are used to drive the partitioning constraint, there is more flexibility in the search procedure than having an exact guiding path. Finally, in the `DPLL_search()` function, we share learnt clauses between threads when conflicts occur to facilitate quick search-space pruning (this is similar to [55]). We expect our partitioning to produce sub-problems with similar characteristics, thereby making our inter-thread learning more powerful. If one thread finds its instances unsatisfiable, we do not repartition the problem. We have observed that frequent repartitioning hinders the effectiveness of the underlying sequential algorithm. In practice, we observe that the even partitioning results in threads that compute for a similar amount of time.

Solution Space. It is possible to exploit the theoretical qualities of our partitioning and note that the number of solutions to the SAT instance under study should be approximately evenly distributed. Therefore, if one sub-problem is found to be unsatisfiable, we can estimate that the other sub-problems have none or very few solutions. This could be used to guide the selection of a portfolio of solvers on-the-fly.

Note that, we do not partition a SAT instance until the batch-mode time threshold in the methodology of Section 7.3 is reached. In addition, we may also partition an instance when its restart frequency is low. This way, we reserve parallel computation only for the hard problems and avoid deterministically partitioning the search space when variable rankings change frequently. As a consequence, we can simplify our procedure in Figure 7.4 to not increase the ranking of variables chosen for the partitioning.

```
bool psat_solve(CNF cnf, int passes, Mode mod=seq, Lit assumps){
  static Var part_vars;
  initialize_assumps(assumps);
  while( not_done() && (passes−− || mod!=seq)) {
     if(mod == parallel) increase_rank(part_vars);
     random_restart();
     result = DPLL_search(mod);
  }
  if(mod == seq && not_done()){
     part_vars = top_vars();
     add_xor_constraints(cnf, part_vars);
     thread(psat_solve, cnf, 0, parallel, neg);
     thread(psat_solve, cnf, 0, parallel, pos);
     while(wait){
        if(SAT) return SAT;
        else if(num_threads−−) return UNSAT;
     }
  }
  return result;
}
bool DPLL_search(Mode mod) {
  while (true) {
     propagate();
     if(conflict) {
        analyze_conflict();
        if(top_level_conflict) return UNSAT;
        backtrack();
        if(mod == parallel) parallel_learn_backtrack();
     }
     else if(satisfied) return SAT;
     else decide();
} }
```

Figure 7.4: Parallel SAT Algorithm.

7.6 Empirical Validation

To evaluate our new solver framework, we consider SAT 2003 Competition benchmarks [51] from the `handmade` and `industrial` categories, both including several suites. The runtime of each benchmark is profiled using MiniSAT 2 [29] on a four-processor dual-core Opteron system clocked at 1GHz with 16 GB of memory running the Fedora 8 SMP OS. We set a timeout for each benchmark at 64 minutes and created a distribution of runtimes over the entire suite. Our results indicate that most benchmarks complete in either less than one minute or over one hour. This highlights the wide variance in runtime performance motivating our proposed methodology. Statistics for the benchmarks as well as runtime distributions can be found in Table 7.1 and Figure 7.2 respectively.

SAT suite	# benchmarks	#SAT	#UNSAT	#TimeOut $> 64min$	total time (min)
handmade	353	48	90	215	13779
industrial	100	19	33	48	3160

Table 7.1: MiniSAT 2 results on the SAT 2003 benchmark suite.

7.6.1 Effective Scheduling of SAT Instances

We first consider an upper-bound on resource utilization by executing several problems concurrently in the ideal case where each benchmark is roughly of the same complexity. Here, we consider only small benchmarks from the suite previously analyzed and we show how a multi-threaded machine can effectively be used so that n threads result in approximately an n-times speed-up. This analysis, shown in Table 7.2, is vital in showing that if n independent problems are available, the corresponding expected speed-up is indeed pos-

sible. The slight deviation from ideal speedups is due to the variation in runtime demands from instance to instance. Below, we show our results for solving a set of instances with a potential wide variance in runtime.

#threads	runtime(min)	speed-up
1	67	1
2	34	1.98
4	18	3.72
8	10	6.70

Table 7.2: Running MiniSAT on a set of benchmarks of similar complexity using a varying number of threads.

Scheduling SAT problems with varying complexity. To evaluate a parallel solving methodology under a realistic distribution of runtimes, we randomly selected a subset of benchmarks with a total runtime of ~ 32 hours, and with the distribution of Figure 7.2. In Figure 7.5, we plot the performance of a non-ideal methodology that schedules the SAT problems as a batch of jobs `batch mode` in an 8-threaded machine. Although the total runtime for all the problems is approximately four hours, we note that several fast problems are not scheduled until late in the batch. In particular, small instances tend to be penalized in their latency. The `time-slice mode` uses the operating system to schedule threads. Notice that although several simple instances finish early, the latency for harder instances increases over batch mode. In our `priority mode`, we transition to batch-mode by adjusting thread priorities after a time threshold is reached. Notice that the integral of our priority mode plot is smaller, indicating better overall latency. We achieve a 20% improvement in average latency over `batch mode` and 29% improvement over `time-slice mode`. Figure 7.5 shows wall-clock time; however, we have observed that the system time is insignificant for each strategy (< 2 minutes). This is due, in part, to the

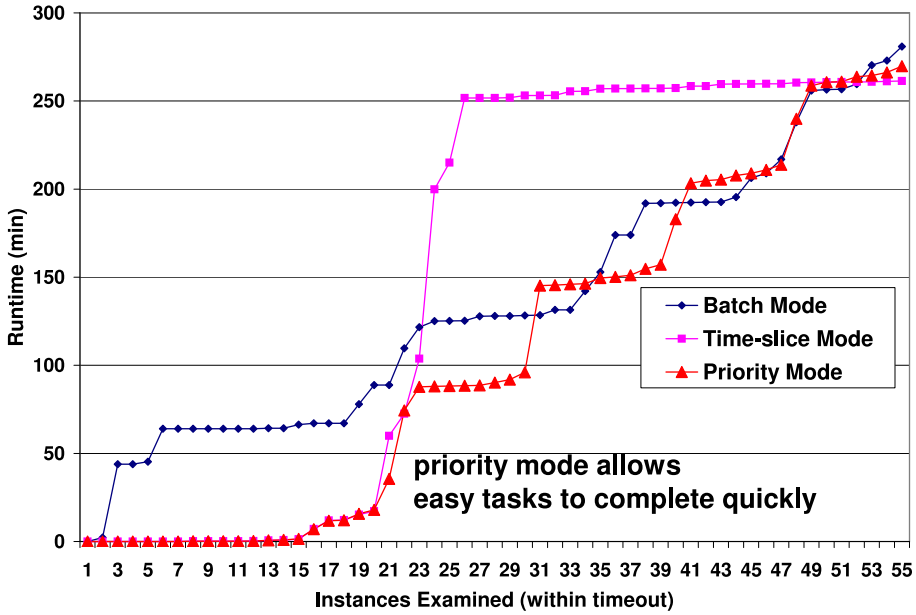

Figure 7.5: The number of SAT instances solved (within the time allowed) by considering three different scheduling schemes for an 8-threaded machine. Our priority scheme gives the best average latency, which is 20% better than `batch mode` and 29% better than `time-slice mode`.

efficiency of the OS scheduler along with the relatively small memory profile required for the random slice of 55 instance considered.

7.6.2 Solving Individual Hard Problems

Ultimately, fast verification turn-around may require a faster solution of individual hard SAT instances. Solvers such as SatZilla [87] try to exploit the fact that some solvers perform better on certain classes of SAT problems than others. By carefully assigning different solvers to each instance, one can improve runtime compared to using any one solver. In the parallel setting, the choice can be simplified by running until one of them completes. However, unlike the single-threaded portfolio variant, it is desirable that the

improved runtime is comparable to the extra computing resources required. Although super-linear runtime improvement over the runtime of MiniSAT is possible due to the high variability of performance of different approaches on a given problem instance, it is important to achieve consistent improvements by exploiting available computational resources. In the following analysis, we choose a subset of instances in the suites we considered where MiniSAT requires significant computation ($\sim$ 1 hour).

solver portfolio		MiniSAT variants		portfolio w/MiraXT		portfolio w/pMiniSAT	
heuristic	# solved	heuristic	# solved	heuristic	# solved	heuristic	# solved
MiniSAT	6	m1	3	MiniSAT	6	**pMiniSAT**	5
Mira1T	0	m2	2	MiraXT	1	Mira1T	1
Jerusat1.3	1	m4	1	Jerusat1.3	0	Jerusat1.3	1
march_ks	0	m5	1	march_ks	0	march_ks	0
picosat	2	m6	2	picosat	2	picosat	2
rsat	0	m7	1	rsat	2	rsat	1
zchaff	2	m8	1	zchaff	2	zchaff	2
HaifaSat	1	m3	1	-	-	-	-
time(min)	321		326		335		**200**
speed-up	1.67		1.65		1.60		**2.69**
%util	20.9		20.6		20.0		**33.6**

Table 7.3: Hard SAT instances solved using 8 threads of computation with a portfolio of solvers.

heuristic	# solved	heuristic	# solved
MiniSAT	7	**pMiniSat**	8
picosat	2	picosat	2
zchaff	2	zchaff	2
Jerusat1.3	1	-	-
time(min)	359		**218**
speed-up	1.50		**2.46**
%util	37.4		**61.6**

Table 7.4: Hard SAT instances solved using 4 threads of computation with a portfolio of solvers.

Table 7.3 shows the speed-up achieved by running multiple heuristics simultaneously

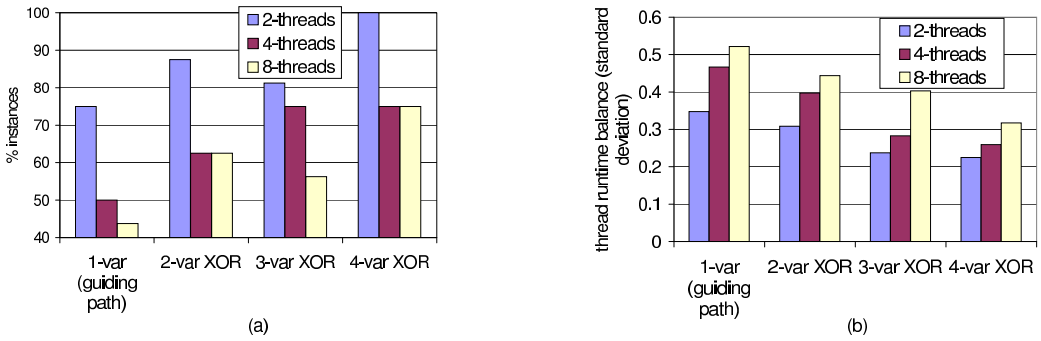

Figure 7.6: a) The percentage of satisfiable instances where the first thread that completes finds a satisfying assignment. b) The standard deviation of runtime between threads. Using XOR constraints as opposed to splitting one variable can significantly improve load balance and more evenly distribute solutions among threads.

where we consider different solver portfolios. We highlight the improvement of incorporating our approach in the last two columns. The total runtime without parallelization for MiniSAT (variant `m1` in Table 7.3) is 537 min. The heuristics columns list different heuristics organized in a portfolio. We report the number of hard instances that a particular heuristic solves the fastest. The first column shows a collection of state-of-the-art SAT solvers. Notice that the speed-up on 8 cores is fairly small at 1.7, meaning that only 20.9% of the 8-times ideal speed-up is realized. The third column shows a portfolio of different variants of MiniSAT given by `m#` produced by adjusting several tunable knobs such as: restart frequency, variable decay rate, and decision heuristic. These results reveal similarly poor utilization where neither randomness nor different heuristics achieve high utilization. We then tried running MiraXT [55] with two threads but did not see additional speed-up in the portfolio (one heuristic is removed from the original portfolio to account for the extra thread required by MiraXT) . Because its performance is dominated

by MiniSAT, parallelizing this solver is ineffective at increasing utilization. Furthermore, the results reported in [55] consider only 2-threads with speed-up much less than 2. Additionally, we have observed that their heavyweight approach for partitioning and learning experiences diminishing returns when considering more threads.

By incorporating our parallel version of MiniSAT, `pMiniSAT`, discussed in Section 7.5 in the solver portfolio, we are able to achieve significant speed-up and higher utilization of 60.5% with respect to the 8 threads of execution compared to the best solver portfolio (pMinisat also requires 2 threads). Furthermore, in Table 7.4, we show that our utilization is even better when considering only 4 threads. This indicates the limitation of large solver portfolios, illustrating that our lightweight approach for parallelization can be beneficial for achieving greater utilization by applying it across multiple heuristics.

7.6.3 Partitioning Strategies

We compared our XOR-based partitioning to a partitioning strategy with a single guiding variable, a special case of *guiding paths* [89]. In Figure 7.6, we show the effectiveness of using XOR constraints for achieving balanced workloads among threads. Figure 7.6a shows the percentage of satisfiable problem instances (out of 16 instances), where the first thread that completes delivers at least one solution. We compare a single variable partitioning strategy against XOR constraints of size $2-4$ and consider parallelization using 2, 4, and 8 concurrent threads. Note that, in the 2-thread case, 100% of the threads that finish first are satisfiable using XORs of size 4, compared to only 75% using one variable. In general. this experiment reveals that our partitioning is more effective at distributing solutions. We expect even better performance in application domains where the number of solutions is much greater than the number of available threads of computation.

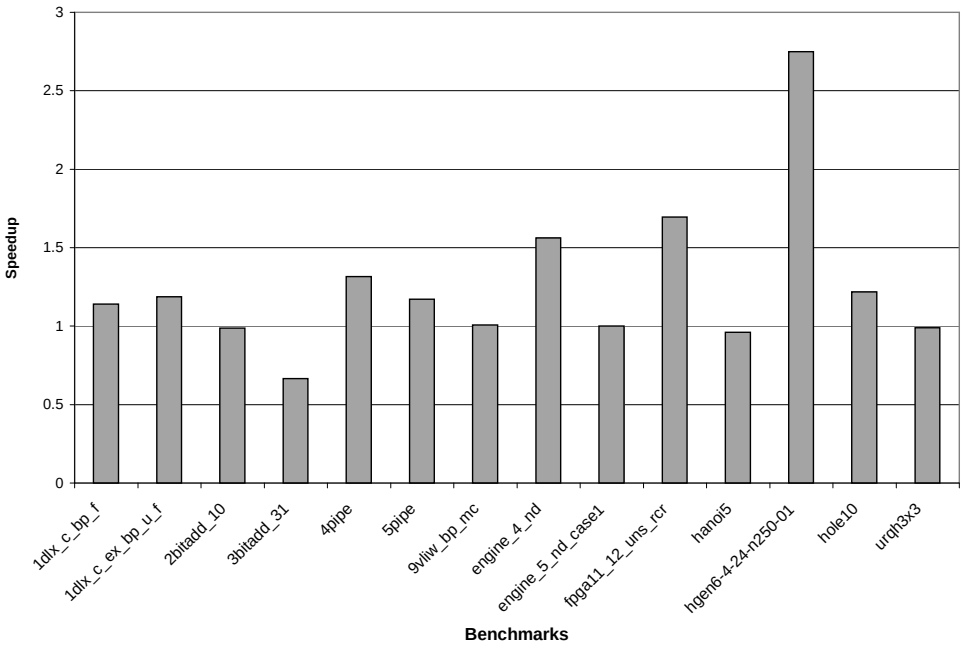

Figure 7.7: The effectiveness of sharing learnt clauses by choosing the most active learnt clauses compared to the smallest learnt clauses.

Figure 7.6b shows the runtime balance between 2, 4, and 8 threads. We examined different partitioning strategies on a set of 29 unsatisfiable problem instances and calculated the standard deviation of thread runtime divided by average runtime. We disabled learning for this experiment to analyze more accurately how the search space is partitioned. For the single variable partitioning for two threads, the normalized standard deviation is 0.35, compared to a much smaller 0.22 for XOR-based partitioning with 4 variables. In general, we observe almost a 2-time improvement in the runtime deviation between single variable strategy and 4 variable XOR when considering different numbers of threads.

7.6.4 Parallel Learning Strategies

We note that efforts in previous parallel learning strategies focus on minimizing communication and subsequently favoring small learnt clauses. The work in [55] incorporates

all learnt clauses within a size threshold. However, according to [90], the size of the clause is not the best indicator for its effectiveness. We consider utilizing VSIDS to choose learnt clauses that more effectively prune the search space relevant to the current sub-problem being solved. We show our results in Figure 7.7 by comparing two different strategies for sharing learnt clauses between 4 SAT solvers executing in parallel. Each SAT solver chooses available learnt clauses ranked either by size as in [22] or by our strategy, which uses the activity of the learnt clauses. We notice that this enhancement results in improvements for most of the benchmarks considered.

7.7 Concluding Remarks

The computational complexity of SAT solving along with the runtime variability exhibited between different solver heuristics challenges state-of-the-art parallel algorithms. We proposed a two-part strategy for exploiting parallel processing more effectively, so that more powerful SAT-based optimizations become practical. First, we introduced a scheduling algorithm that incorporates the approximate knowledge of runtime distributions for a given set of SAT instances to minimize average latency over batch scheduling by 20%. Since several instances require prohibitive amounts of runtime, we also proposed a lightweight parallel SAT algorithm that effectively partitions the search space after first exploring part of the search space sequentially. We observe that our partitioning results in $\sim$ 50% better run-time balance than simply choosing one splitting variable. Our strategy enables us to improve resource utilization over solver portfolios by 60.5%. By incorporating our partitioning strategy with different SAT solvers, solver portfolios can be further improved because the randomness vital to solve SAT instances efficiently is better coordinated.

Part IV

Improving Logic and Physical Synthesis

In the previous chapters, we have introduced strategies to improve bit signatures' ability to distinguish functionally different nodes and to verify the correctness of abstractions more efficiently. We now leverage these advances to enable powerful logic optimizations guided by signatures. First, in Chapter VIII we introduce novel logic transformations which would require prohibitive amounts of computation without using signatures. Then, in Chapter IX we use these novel transformations to enable powerful optimizations in post-placement restructuring to improve critical path delay.

CHAPTER VIII

Signature-based Manipulations

Bit signatures provide an effective means to approximate synthesis transformations. In addition, the ability to encode don't-care information in the signatures enables more optimization opportunities for the transformations considered. In this chapter, we introduce two general techniques for using signatures to enable powerful optimizations. We first describe a node-merging strategy that uses ODCs and achieves area reductions of 25% on average. Then, we discuss a goal-driven synthesis technique, distinct from other logic synthesis approaches, that can efficiently determine whether a logic implementation exists for a topology corresponding to a desired subcircuit. In the next chapter, we leverage both of these synthesis strategies to restructure critical paths after placement.

8.1 Logic Transformations through Signature Manipulations

Algorithms for logic synthesis typically operate on some representation of Boolean functions that represent circuit nodes — algebraic expressions, sums-of-products and other Boolean formulas, such as BDDs, AIGs, etc. After logic synthesis, these representations are converted back to circuits. To justify such manipulations by proxy, one has to ensure that any circuit-based operation is faithfully represented by its counterpart on a given representation. To demonstrate that signature-based abstractions satisfy this condition (where

operations on the signatures correspond to operations on the actual circuit), we formally denote the assignment of a Boolean function F to a value of 0 or 1 by the homomorphism $eval_X$, for an input X. $eval_X$ gives a mapping of the Boolean function space for an input vector to 0 or 1, *i.e.*, $2^{2^{|X|}} \rightarrow \{0,1\}$ for $|X|$ primary inputs.

$$eval_X\big(F(X_i) \cdot G(X_i)\big) = eval_X\big(F(X_i)\big) \circ eval_X\big(G_(X_i)\big) \tag{8.1}$$

The symbol $\cdot$ denotes any Boolean operation in the Boolean function space, $2^{|X|} \rightarrow \{0,1\}$, and symbol $\circ$ denotes the corresponding bit operation.

For example, if $\cdot$ is the Boolean AND operation $\wedge$, then $\circ$ is the bit-wise AND operation &. The relation in Equation 8.1 indicates that, for any input vector, evaluating the output of a Boolean function (composed of Boolean functions F and G) is equivalent to evaluating the outputs of F and G and applying the corresponding bit operation. By extending this relation on one input vector to K input vectors, we produce the following mapping $2^{2^{|X|}} \rightarrow \{0,1\}^K$, which is the signature of a function. Therefore, manipulating the signatures of F and G, $S_F \circ S_G$, is equivalent to generating a signature of $F \cdot G$. The resynthesis of H with inputs F and G corresponds to the generation of S_H from S_F and S_G.

Example 2 *For nodes, H, F, and G assume $S_H = \{0,1,1,0\}$, $S_F = \{1,1,1,0\}$, and $S_G = \{0,1,1,0\}$ under 4 simulation vectors. $S_F \& S_G = S_H = \{0,1,1,0\}$ where & is a bitwise AND. If $eval_X(H) = eval_X(F) \& eval_X(G)$ for all input vectors, $H = F \wedge G$.*

8.2 ODC-enhanced Node Merging

Merging equivalent circuit nodes is an effective technique to reduce the area of a logic circuit. It scales to very large netlists, but, unlike BDD-based techniques to determine

equivalence, it requires non-trivial algorithms to identify potential mergers and verify the results. Such algorithms for node merging were first developed in the context of formal verification to detect possible cut-points in equivalence checking [34, 57]. To this end, the work in [47, 59] uses a combination of SAT solving and simulation. Candidate nodes for merging are first selected by checking whether their outputs correspond when stimulated with random patterns applied to the design's inputs. Then, their actual equivalence can be verified using SAT. The simulation is refined through counterexamples generated by SAT, which reduces the number of checks resulting in non-equivalence. Rather than finding equivalent nodes as a post-processing step, the work in [59] improves equivalence checking by merging equivalent nodes while constructing the mitered circuit. However, incremental approaches, such as [59], do not allow for the detection of ODCs because no information about the downstream logic is maintained. We show that by taking into account ODCs additional node mergers should be possible.

Because of the computational complexity involved in deriving ODCs, previous work [95] tends to emphasize local computation as a synthesis optimization before technology mapping. This emphasis is well justified for AIGs, which have a much larger number of internal nodes, and thus possible mergers, compared to mapped circuits. However, our intended applications are in physical synthesis, where technology mapping can significantly affect circuit delay, and the placement of standard cells is crucial. In this context, fewer nodes are exposed, and one must search for additional don't-cares not found by existing techniques. Thus, our goal is to quickly identify nodes equivalent up to global don't-cares, efficiently verify their equivalence, and use the results to simplify the design structure. Additionally, our implementation can operate on mapped designs without requiring costly

netlist conversions, which otherwise lead to a loss in physical information and delay estimates. In the following, we first explain how merger candidates can be identified using logic signatures and then provide empirical results.

8.2.1 Identifying ODC-based Node Mergers

In this section, we develop the theory involved in ODC-based node merging and describe the use of signatures to identify candidate mergers.

ODC-substitutability. Traditionally, a node merger can occur between node *a* and node *b* when they are functionally equivalent. We define node mergers between *a* and *b* in the presence of ODCs when node *a* is *ODC-substitutable* to node *b*.

Definition 8.2.1 *Node a is ODC-substitutable to node b if* $ONSET(a) \cup ODC(b) = ONSET(b) \cup ODC(b)$.

When *a* is ODC-substitutable to *b*, a merger between *a* and *b* means that *a* can be substituted for *b*. Because the ODCs of only one node are considered, ODC-substitutability is not symmetric as *b* might not be ODC substitutable to *a*.

Using signatures and ODC-masks described in the previous chapter, we can define a *candidate* merger as follows:

Definition 8.2.2 *Node a is a candidate for ODC-substitutability with node b if and only if* $(S_a \oplus S_b) \subseteq \neg S_b^*$. *This can be re-expressed as* $S_a \subseteq [S_b^{lo}, S_b^{hi}]$, *in other words,* S_a *is contained within the range of signatures defined by* S_b^{lo} *and* S_b^{hi}.

where the $\subseteq$ relation is defined using the signatures of two nodes:

Definition 8.2.3 $S_b \subseteq S_a$ *if and only if* $S_b | S_a = S_a$ *where* $|$ *represents bit-wise OR.*

circuit	#candidates	%incorrect (false positives)	%missed (false negatives)
ac97_ctrl	63758	0.0	0.0
aes_core	315917	0.1	0.0
des_perf	296095	0.0	0.0
ethernet	8852009	0.3	0.8
mem_ctrl	867145	1.0	1.4
pci_bdge32	1158654	0.2	0.4
spi	156291	0.0	3.1
systemcaes	285189	0.2	0.2
systemcdes	5288	2.8	0.7
tv80	1348277	1.5	9.0
usb_funct	1685374	2.2	1.8
wb_conmax	1904773	0.0	0.0

Table 8.1: Evaluation of our approximate ODC simulator in finding node merger candidates: we show the total number of candidates after generating 2048 random input patterns and report the percentage of false positives and negatives.

Therefore, by simple application of S_b^*, it can be determined that a is an ODC-substitutable candidate with b. Similar to Definition 8.2.1, if a is an ODC-substitutable candidate with b, it does not imply that b is an ODC-substitutable candidate with a.

The approximate ODC analysis is capable of finding many candidates while filtering out false positives or negatives in the ODC mask due to the approximation of the simulator. Table 8.1 shows the number of ODC-substitutability candidates for all nodes in the circuit identified by our approximate simulator and the percentage of incorrect candidates due to false positives (`%incorrect`) in the ODC mask and missed due to false negatives (`%missed`). In the experiment, we generated 2048 random input patterns to extract the candidates. The results indicate that several candidates exist and that the number of false positives and negatives is typically only a small fraction of the opportunities identified.

Finding candidates with signatures. Constant-time complexity hashing, as in [59], cannot be used to identify ODC-substitutability candidates. Here, each node needs to apply

its mask to every other node to find potential candidates. The result is that for N nodes, finding all ODC-substitutability candidates for a design requires $O(N^2K)$-time complexity, assuming that applying a mask is an $O(K)$-time operation. Thus, we developed a strategy that significantly reduces computation in practice. First, all of the signatures, S, in the design are sorted by the value obtained by treating each K-bit signature as a single K-bit number. This operation requires $O(NKlogN)$-time. Then, for a given node c, candidates can be found by performing two binary searches with S_c^{lo} and S_c^{hi} to obtain a lower and upper bound on the sorted S, an $O(KlogN)$-time operation. Searching for complemented candidates can be accomplished by simply complementing S_c^{lo} and using this to derive an upper bound. Similarly S_c^{hi} must also be complemented and used to derive a lower bound. The following equation defines the set of signatures S_x that is checked for candidacy (we ignore the case of negation for simplicity):

$$\bigcup_x S_x \; if \; num(S_c^{lo}) \leq num(S_x) \leq num(S_c^{hi}) \tag{8.2}$$

where *num* represents the K-bit value of the signature. This set is traversed linearly to find candidates according to Definition 8.2.2.

8.2.2 Empirical Validation

Experimental setup. We developed our solution and relied on a specialized SAT engine based on MiniSAT for validating candidates. We used random simulation patterns to generate the initial ODC signatures. We used testbench circuits from the IWLS 2005 suite [102]. Our experiments run on a Pentium-4 3.2 GHz machine. The ODC-based node-merging algorithm examined each node in a circuit in one topological traversal. Each time a merger is applied, the signatures in the fanout cone of the replaced node could become

inaccurate, due to different don't care sets. However, since signatures are only used to find candidates to be validated by a SAT solver, incorrect signatures can never lead to incorrect mergers and updates are thus not necessary.

For the experiments on combinational simulation and equivalence checking, we extract the combinational portion of the IWLS 2005 testbenches. In the experiment, every internal node with a non-empty ODC-set is examined for merging opportunities; however, we ignore mergers that increase the number of logic levels in the design. After completing the analysis, we check the correctness of the transformations using the ABC's equivalence checking tool [98].

Post-synthesis optimization. In this section, we show that our global ODC analysis discovers node mergers even after synthesis optimizations [62, 98]. These additional reductions can be easily performed in conjunction with layout information to help achieve design closure.

To create a realistic experimental setup, we first optimized the netlist of each circuit by running a synthesis optimization phase in ABC [98], which further compressed the designs (the original netlist was mapped to a barebone set of logic gates).[1] The results of this evaluation are reported in Table 8.2. The first column, `#gates`, gives the number of gates in each design after synthesis with ABC. The second column gives the synthesis optimization runtimes with the `resyn2` script. We then report the number of ODC-based mergers that we detected and applied, and the corresponding reduction in area. The final column gives the additional runtime required by our merger algorithm. We set a timeout of 5000s for the merger algorithm: for a few testbenches we reached this limit and report

[1] We used the `resyn2` script in the ABC package, which performs local circuit rewriting optimization [62].

circuit	#gates	ABC(s)	#merge	%area_reduct	mergers(s)
dalu	1054	0	91	12.0%	10
i2c	1055	0	30	3.2%	3
pci_spoci_ctrl	1058	0	97	9.2%	6
C5315	1368	0	8	0.7%	2
C7552	1541	1	25	3.4%	8
s9234	1560	0	10	1.2%	8
i10	1884	1	38	1.3%	12
alu4	2559	1	469	22.9%	64
systemcdes	2655	1	111	4.7%	9
s13207	2725	1	15	1.8%	17
spi	3342	1	23	1.3%	84
tv80	8279	3	606	7.1%	1445
s38417	9499	2	33	1.0%	275
systemcaes	10093	4	518	3.8%	360
s38584	11306	2	150	0.8%	223
mem_ctrl	12192	5	1797	18.0%	738
ac97_ctrl	13178	3	185	2.0%	188
usb_funct	15514	5	186	1.4%	681
pci_bridge32	19872	6	82	0.1%	1134
aes_core	21957	9	2144	8.6%	1620
b17	24947	6	224	1.6%	5000
wb_conmax	49236	19	2433	6.2%	5000
ethernet	67129	28	45	1.4%	5000
des_perf	80218	50	3148	3.7%	5000
average				4.9%	

Table 8.2: Area reductions achieved by applying the ODC merging algorithm after ABC's synthesis optimization [62]. The time-out for the algorithm was set to 5000 seconds.

the improvements achieved within this time. Despite the ABC-based pre-optimization, we observe that the designs can still be further optimized with improvements of over 10% in some cases.

Table 8.3 reports potential mergers when using don't-cares For this experiment the netlists were generated by Synopsys DesignCompiler [104]. The circuits were synthesized with high effort and the results were mapped using the generic GTECH library. Columns `DC(s)` and `odc(s)` give the runtime for running DesignCompiler and the node merging algorithm, respectively. The runtime overhead of node merging is shown by `%overhead` and it is small for most testbenches. The final two columns give the number of mergers produced and the percentage of gates eliminated. The results indicate that even after state-of-the-art synthesis, our node-merging application, which is a special case of our more general proposed strategy, allows for additional area reductions in many circuits.

ODC locality. We now show that several levels of downstream logic are often involved in proving equivalence with ODCs. Because of our efficient simulation and incremental verification technique, we can enhance the local ODC analysis of [95] by considering node mergers of unbounded depth.

In Table 8.4, we compare the percentage of mergers exposed using K levels of downstream logic, for K=1..5, against using unbounded K. Circuits were optimized as in the previous experiment. The results indicate that most mergers can be detected using only a few levels of logic. However, on average, our solution can detect 25% more mergers by not limiting the depth of logic under consideration.

To evaluate the impact of *circuit unrolling* on merging opportunities, we devised a specific experiment. Circuit unrolling is a key step in bounded model checking and in finding

circuit	#gates	DC(s)	odc(s)	%overhead	#merge	%gate_reduct
pci_spoci_ctrl	281	15	0	0	5	2.5
dalu	315	11	2	18.2	3	1.0
s9234	375	23	1	4.3	0	0.5
systemcdes	437	33	0	0	9	2.5
s13207	487	44	1	2.3	3	1.0
i2c	544	17	1	5.9	8	1.8
alu4	806	18	6	33.3	23	4.1
spi	821	44	2	4.5	4	0.7
C5315	828	14	2	14.3	6	0.7
C7552	1046	17	2	11.8	24	2.4
i10	1185	18	4	22.2	17	1.5
aes_core	1758	293	3	1	29	1.8
tv80	1953	135	15	11.1	16	1.1
pci_bridge32	2079	488	23	4.7	18	1.0
ac97_ctrl	2119	284	12	4.2	35	1.7
systemcaes	2175	135	10	7.4	10	0.6
mem_ctrl	2560	258	23	8.9	19	0.8
s38417	2578	236	36	15.3	28	1.2
s38584	3922	207	20	9.7	69	1.8
ethernet	4163	3053	47	1.5	25	0.6
usb_funct	4718	293	44	15	36	0.8
wb_conmax	9833	885	203	22.9	122	1.3
b17	11133	1041	343	33.0	87	0.8
des_perf	12685	4719	216	4.6	255	2.1
average				10.7		1.4

Table 8.3: Gate reductions and performance cost of the ODC-enhanced node-merging algorithm when applied to circuits synthesized with DesignCompiler [104] in high-effort mode. The merging algorithm runtime is bound to $\frac{1}{3}$ of the corresponding runtime in DesignCompiler.

sequential don't-care opportunities in physical synthesis. This motivated us to investigate if additional netlist compression opportunities were available for unrolled circuits. We expect unrolled circuits to have higher potential for node merging because of the larger amount of combinational logic available. In the experiment, we considered a range of sequential designs and unrolled them between 1 and 5 times; then, for each scenario, we compared the percentage of mergers discovered by considering only five levels of logic

circuit	K=1	K=2	K=3	K=4	K=5	K=∞
dalu	9.9	14.3	19.8	31.9	38.5	100
i2c	36.7	53.3	60.0	66.7	80.0	100
pci_spoci_ctrl	21.6	51.5	67.0	84.5	93.8	100
C5315	87.5	87.5	87.5	87.5	87.5	100
C7552	36.0	64.0	64.0	68.0	72.0	100
s9234	0	0	20.0	20.0	40.0	100
i10	15.8	28.9	60.5	71.1	86.8	100
alu4	13.2	26.9	35.2	42.6	50.1	100
systemcdes	26.1	38.7	60.4	74.8	86.5	100
s13207	13.3	46.7	60.0	80	93.3	100
spi	60.9	82.6	91.3	95.7	100	100
tv80	11.9	23.4	38	49	56.3	100
s38417	12.1	54.5	78.8	100	100	100
systemcaes	21.6	45.8	70.5	72.8	73.9	100
s38584	17.3	55.3	70.7	82.0	85.3	100
mem_ctrl	26.5	43.0	55.4	68.3	77.0	100
ac97_ctrl	63.2	88.1	93.5	96.8	97.8	100
usb_funct	42.5	69.4	81.7	87.6	91.4	100
pci_bridge32	45.1	54.9	68.3	78.0	87.8	100
aes_core	9.7	15.4	22.9	31.6	42.3	100
b17	21.4	30.4	35.7	42.4	44.2	100
wb_conmax	7.9	16.5	26.0	36.5	48.5	100
ethernet	31.1	48.9	68.9	77.8	84.4	100
des_perf	16.8	27.4	39.4	55.7	74.0	100
average	27.0	44.5	57.3	66.7	74.6	100

Table 8.4: Percentage of mergers that can be detected by considering only K levels of logic, for various K.

circuit	unrolling depth				
	1	2	3	4	5
i2c	80.0	57.0	42.8	43.1	43.2
pci_spoc_ctrl	93.8	87.8	86.5	84.8	84.5
s9234	40.0	51.4	42.0	38.2	42.9
systemcdes	86.5	85.3	88.7	86.2	86.3
spi	100	70.7	71.7	64.6	67.5
ac97_ctrl	97.8	83.2	64.2	46.6	38.9
average	83.0	72.6	66.0	60.6	60.6

Table 8.5: Comparison with circuit unrolling. Percentage of total mergers exposed by the local ODC algorithm (K=5) for varying unrolling depths.

versus considering the whole unrolled netlist. As shown in Table 8.5, for a few of the circuits, the percentage of mergers missed by local ODC computation is highly affected by the unrolling depth: the more the circuit is unrolled, the higher the missed fraction. An example is `ac97_ctrl` where, with no unrolling, only 2% of the mergers are missed; however, with an unrolling depth of 5 the missed percentage becomes 60%. On one hand, the local analysis has better performance (we could not show the full range of results for all designs because of timeout conditions). On the other hand, our solution presents better flexibility to adjust to a wide range of design sizes.

Framework assessment. Table 8.6 shows the quality of our signature-based framework on unoptimized circuits by assessing the effectiveness of signatures in finding good merger candidates. `#merge` gives the number of mergers applied to a circuit. We then show the number of SAT calls required to prove the correctness of each merger, along with the corresponding percentage of those calls that confirmed equivalence (columns `#SAT` and `%equiv`). Merger candidates that required over 10 seconds to be verified were timed-out, so to favor faster mergers. The column `#dyn-sim` denotes the number of dynamic simulation vectors derived from counterexamples provided by the SAT-based verification engine. The final column shows how many SAT calls were pruned because of the inclusion of the dynamic vectors.

The results indicate that, on average, almost 50% of the SAT calls result in ODC merging. Moreover, it is clear that the use of dynamic simulation vectors had a great impact on this high-quality result. Reducing the number of SAT calls is key because SAT-based equivalence checks contribute to most of the runtime cost. Furthermore, the dynamic vectors added are typically much fewer than the number of false positives pruned.

circuit	#merge	#SAT	%equiv	#dyn-sim	#prune
i2c	39	206	18.9%	167	36960
pci_spoci_ctrl	170	472	36%	302	34345
alu4	697	1306	53.4%	609	273497
dalu	636	1040	61.2%	404	25808
i10	257	580	44.3%	323	22029
spi	112	557	20.1%	445	78721
systemcdes	255	287	88.9%	32	153
C5315	161	192	83.9%	31	194
C7552	340	524	64.9%	184	107665
s9234	821	1959	41.9%	1138	514875
tv80	658	1781	36.9%	1117	832861
systemcaes	658	750	87.7%	88	8852
s13207	300	1007	29.8%	707	2208345
ac97_ctrl	80	256	31.3%	176	26803
mem_ctrl	2758	4356	63.3%	1580	2710618
usb_funct	246	1739	14.1%	1493	1206172
pci_bridge32	158	1189	13.3%	1031	2951017
s38584	2253	3610	62.4%	1357	3487613
aes_core	2072	2317	89.4%	245	2205
s38417	636	2416	26.3%	1780	11544973
wb_conmax	2313	5068	45.6%	2755	441002
b17	614	3588	17.1%	2974	21984143
ethernet	370	2084	17.8%	1509	2979472
des_perf	2505	2614	95.8%	109	1198
average			47.7%		

Table 8.6: Statistics for the ODC merging algorithm on unsynthesized circuits. The table reports the SAT success rate in validating merger candidates and the number of SAT calls that could be avoided because of the use of dynamic simulation vectors.

8.3 Determining Logic Feasibility with Signatures

In the previous section, we showed that finding equivalent nodes in a circuit may lead to significant area reductions. In this section, we introduce a goal-driven synthesis strategy that can efficiently find a gate-level logic implementation for a given function. The strategy can be applied in logic resynthesis to transform circuit blocks so to optimize one or more physical parameters (*e.g.*, area, timing, etc). In the next chapter, we will apply this technique in building circuit structures optimized for timing delay. To express our goal more formally, we assume to have a subcircuit with m inputs, $\{a_1, a_2, ..., a_m\}$ and output F to resynthesize, and we want to find several restructuring solutions that we can then evaluate based on their parameters. We represent the input subcircuit as a directed graph T_F with m incoming edges, one outgoing edge F, and n internal vertices. Our goal is to determine whether there is a labeling, G^*, of n vertices with gates $g \in G$, such that F is logically equivalent to the subcircuit that implements T_F, with respect to the outputs of the circuit. In defining the data structures necessary to achieve our goal, we leverage a few previous works. In particular, in [93], *sets of pairs of functions to be distinguished* (SPFDs) are introduced as a way of representing a node's functionality which can be used to exploit circuit flexibility in logic optimization. In [77], the authors propose a technique that uses SPFDs to find a logic implementation given a topological constraint, but their resynthesis approach does not incorporate physical parameters such as timing and is limited to only a few neighboring levels of logic to reduce the memory and computational requirements of SPFDs. In an alternative strategy to reduce the memory requirements of SPFDs, the authors in [94] choose a subset of SPFDs for a node using simulation and compatibility don't-cares in a logic rewriting application.

Because we efficiently encode global circuit don't-cares, we are not limited by levels of logic or required to have don't-cares that are compatible. Furthermore, our approach encodes the distinguishing bits in a compact data structure with logic signatures so that these operations can also be performed with bitwise parallelism. This is particularly beneficial in our development of a novel goal-driven synthesis technique where fast evaluation of topological constraints is essential to tightly couple physical optimization and logic synthesis.

We now define several properties for graph T_F that is the input to our strategy. We define the *logic feasibility* of the graph T_F as:

Definition 8.3.1 *T_F is logically feasible if* $\exists_{G^*} ONSET(T_{F^c}) = ONSET(F)$.

where $ONSET$ is the set of input combinations for which the subcircuit produces 1 in output. This definition can be relaxed by considering its relation within the care-set which could be considerably smaller than 2^m, due to controllability and observability don't-cares.

Definition 8.3.2 *T_F is logically feasible up to circuit don't-cares if* $\exists_{G^*} ONSET(T_{F^c}) \cup DC(F) = ONSET(F) \cup DC(F)$.

where DC is the don't-care set.

A naïve algorithm for determining the logic feasibility of T_F requires that every possible labeling G^* is evaluated. For n vertices, this requires checking $|G|^n$ labelings. If the set of two-input logic functions is considered, there are 5^n labelings.[2] Furthermore, performing equivalence checking between $ckt(T_F)$ and T_F is an NP-complete problem. Below, we

[2] Although there are 16 different functions in the two-input Boolean function space over a switching algebra, the tautology and two one-variable identity functions along with the negated form of each function do not need to be explicitly considered.

discuss how signatures can be used to determine a minimal set of inputs that implements a given function and how this can be extended to quickly determine logic feasibility up to the signature approximation.

Pairs of bits to be distinguished.

Definition 8.3.3 *A function F is said to be dependent on an input a_i if: $F_{a=0} \oplus F_{a=1} \neq 0$.*

A similar relationship between the signature S_F of the function F and input signatures $S_1, ..., S_m$ can be established. In [19] it was observed that a set of input signatures can implement a target signature if and only if every pair of different bits in S_f is *distinguished* by at least one of the input signatures S_m.

Definition 8.3.4 *A pair of bits to be distinguished (PBD) is an unordered pair of indices $\{i, j\}$ such that $S_F(i) \neq S_F(j)$.*

Definition 8.3.5 *A candidate signature, S_m distinguishes a PBD in S_F if $S_m(i) \neq S_m(j)$ where $\{i, j\} \in S_F^{PBD}$ where S_F^{PBD} is F's set of PBDs.*

Example 3. Assume a target signal $S_f = \{0,0,1,1\}$ and candidates $S_1 = \{0,0,0,1\}$, $S_2 = \{0,1,0,1\}$, and $S_3 = \{0,1,1,1\}$. The PBDs of S_F that need to be distinguished are $\{0,2\},\{0,3\},\{1,2\},\{1,3\}$. Note that S_1 and S_2 together cannot implement S_F because they do not distinguish $\{0,2\}$. However, if all S_m are used, then all the bit pairs can be distinguished and it is possible to construct a function that generates S_F from the S_i. In this example $S_F = S_3 \cdot (S_1 \bar{\oplus} S_2)$. □

Essential PBDs. Input signatures form an irredundant cover of S_F's PBDs when 1) every PBD is covered by at least one S_i and 2) removing one S_i results in at least one uncovered PBD. The resulting S_i form the support of the function to be resynthesized.

Definition 8.3.6 *A PBD that is distinguished by only one S_i is an* essential *PBD for S_i.*

According to the definition of an irredundant cover and PBDs, each S_i must have at least one essential PBD (or else that input can be discarded). Because there is at least one essential PBD for each input, S_F is dependent on S_i, independently of its implementation, if the following condition holds:

$$\exists_i S_{F(S_i=0)} \oplus S_{F(S_i=1)} = 1 \tag{8.3}$$

In the case of function $F(a_1,...,a_m)$ resynthesis, we note that the cardinality of the irredundant cover can be less than m, because F may be independent of an input a_i up to don't-cares and the signature abstraction might not expose a sufficient number of essential PBDs. Furthermore, several irredundant covers are possible. In this paper, we greedily determine irredundant covers by first selecting signatures that cover most PBDs and continuing until all PBDs are covered.

Determining logic feasibility with essential PBDs. We now describe how the logic feasibility of a given topology can be determined simply using signatures. In the next chapter, we study how to create such topologies and how to verify the corresponding signature-based abstraction. Our strategy assumes that the target library consists of all two-input logic gates, so that each node n has exactly two input edges (although the initial subcircuit can be mapped into any cells). In general, we do not restrict our topologies to be *fanout-free* trees (a topology is fanout-free if each node n in T_F has only one outgoing edge).

Note, however, fanout-free topologies form a critical aspect of our goal-driven synthesis strategy because, under two assumptions, they produce circuits with optimal area

and timing if such a fanout-free circuit exists. First, we assume that each gate in the library requires the same area. Second, we assume that the delay through the subcircuit is solely determined by its path length, that is, we assume that each wire is optimally buffered. With these assumptions, fanout-free topologies have smaller area than their non-fanout free counterparts when implementing a single-output function because they have fewer internal nodes ($m-1$ nodes). Furthermore, fanout-free topologies have the same or smaller delay as non-fanout free trees. The proof of this is straightforward because if a non-fanout free topology has optimal delay based on path length, converting this topology to a fanout-free tree by removing edges and nodes does not increase path length.

In the next few paragraphs, we introduce an algorithm for determining logical feasibility on fanout-free circuits where each primary input has only one outgoing edge F, which can be uniquely defined by an $O(|S_F^{PBD}| * m)$-time algorithm using signatures. Because logic feasibility is not always possible for a fanout-free tree that optimizes a particular performance criterion, we extend our synthesis techniques to handle arbitrary non-tree topologies.

First, we associate a signature S_i to each input of T_F. If we assume that each S_i under simulation distinguishes at least one essential PBD, we note the following for each two-input gate in a fanout-free topology:

Theorem 5 *Given input signatures S_1, S_2, and the two-input function Φ, the signature, $S_{1,2} = \Phi(S_1, S_2)$ has all the essential PBDs of S_1 and S_2.*

Proof. Any cut through T_F gives a set of inputs that implements F. Therefore, the PDBs of S_F must be distinguished by each cut in $ckt(T_F)$ for any feasible topology. Since in a fanout-free tree, S_1 and S_2 do not reoccur in the topology, the output of the node combining

S_1 and S_2, S_{12}, must contain their essential PBDs to distinguish S_F. □

As a direct consequence, each two-input transformation preserves at least two essential PBDs. Furthermore, PBDs that only occur in both S_1 and S_2 must also be preserved to uphold the invariant that every cut through the topology forms an input support. In a similar manner, the work in [77] upholds this invariant in constructing a subcircuit but considers SPFDs (*i.e.*, sets of pairs of functions to be distinguished) instead. We note the following:

Theorem 6 *Given two input signatures where each one has at least one essential PBD, there are at most two two-input Boolean functions (ignoring negated version of these functions) that can preserve all the essential PBDs.*

Proof. A two-input Boolean function has a 4 row truth table with output 0 or 1. One essential PBD adds the following constraint:

$$[\Phi(a,b) = z] \wedge [\Phi(a',b) = z'] \tag{8.4}$$

where a, b, and z are variables with value 0 or 1. In other words, two distinct rows of the truth table must have different values. For a given a and b where an essential PBD is defined, there are only 2 such assignments to z that satisfy this constraint. The remaining 2 rows in the truth table can have any of 4 possible output combinations. Therefore, there is a total of 8 different functions that satisfy this constraint. We ignore negated versions of the Boolean function since that negation can be propagated to the inputs of later gates. Given this, there are 4 distinct functions that can preserve one essential PBD. However, since two essential PBDs must be preserved, the following constraint needs to be satisfied:

$$[\Phi(a,b)=z]\wedge[\Phi(a',b)=z']\wedge[\Phi(d,e)=y]\wedge[\Phi(d,e')=y'] \tag{8.5}$$

If $\{(a,b),(a',b)\}$ is disjoint from $\{(d,e),(d,e')\}$, there are only 4 possible output combinations of z and y that satisfy the constraints, where 2 of them are the negated form. This is also the case if $\{(a,b),(a',b)\}$ is not disjoint from $\{(d,e),(d,e')\}$ (it is impossible for two different functions to have essential PBDs on the same two rows). Therefore, there are at most only 2 distinct Boolean functions that can preserve the essential PBDs of its inputs. □

If the fanout-free tree is traversed in topological order, a choice between two different two-input gates is available for each node. In the worst case, all possible combinations must be evaluated to preserve all the essential PBDs, generating an $O(|S_F^{PBD}|2^m)$-time complexity (there are $m-1$ nodes) for the final algorithm. For the typically small topologies that are considered for resynthesizing portions of the critical paths, this result is a significant practical runtime improvement over trying all possible gate combinations without considering PBDs. Moreover, we note that in many cases the runtime complexity is linear.

Theorem 7 *Consider the following assumptions.*

1. *T_F is an m-input fanout-free tree.*

2. *The m-input function F is completely specified by S_F under simulation.*

Under these conditions, the logical feasibility of T_F can be determined in $O(|S_F^{PBD}|*m)$ time in the worst case.

Proof. A fanout-free topology specifies a disjoint partition of the inputs. If an implementation exists with a disjoint partitioning of inputs, each internal node corresponds to a function that is specified independently of the rest of the implementation. Therefore, when the signatures completely specify F (a complete truth table), each internal node is also completely specified. Because of this, each two-input operation must preserve at least 3 essential PBDs (the minimal number of distinguishing bits a two-input function can have) and therefore only one function satisfies this relation. Because there is only one such candidate function, the complexity of finding an implementation is $O(|S_F^{PBD}| * m)$. □

Although we often resynthesize functions with small supports and therefore small truth tables, a logic signature does not always completely specify a function's behavior resulting in a reduction in the number of bits that need to be distinguished. Also, the ability of simulation to quickly identify circuit don't-cares further reduces the number of bits that need to be distinguished. By not having a completely specified function, we facilitate multiple feasible implementations. Despite the advantages of this flexibility in determining a feasible implementation, an internal two-input operation may only need to preserve 2 essential PBDs rather than 3, which can increase the runtime of finding an implementation. However, in practice, this runtime penalty is minor because the topologies are typically small. Also, in many cases logical feasibility can still be determined in $O(|S_F^{PBD}| * m)$ time depending on which bits need to be distinguished.

Although we work with a functionally complete set of two-input gates, our approach is capable of targeting any standard-cell library. This is done by allowing topologies where each node can have more than two incoming edges. For a completely-specified fanout-free tree, we still only require a linear traversal to discover whether a logically feasible imple-

mentation exists. Alternatively, after first decomposing the implementation to two-input gates (where this decomposition already improves the physical characteristics), further improvement by applying technology mapping using larger cells may be possible.

In some cases the optimal topology with respect to a given performance goal is not logically feasible. Furthermore, some very common functions such as the multiplexor function cannot be implemented using a fanout-free topology. Therefore, a viable technique must handle a broader family of topologies. We therefore describe how essential PBDs can be used to guide synthesis for non-tree topologies where each operation preserves at least one of its inputs' essential PBDs. This facilitates reconvergence and the implementation of useful functions including multiplexors, as shown below.

Theorem 8 *Consider a logic circuit with the following conditions:*

1. *At least one input to each node in the circuit does not fanout to another node at the same or greater logic level.*[3]

2. *The only implementations considered are those where the signatures along each cut through the topology form an irredundant cover.*[4]

Under these conditions, the logical feasibility of an n-node topology T_F can be determined in $O(|S_F^{PBD}| * 3^m)$ time.

Proof. By traversing the graph in topological order, note that at least one essential PBD is transferred to the output. Also, when the implementations are considered are those where

[3]The logic level of a node is determined by the path from the node to the primary inputs with the greatest number of edges.

[4]In general, a topology may have an implementation with redundant covers. However, we focus on implementations that do not use this redundancy to improve the efficiency of our approach.

the signatures along each cut of the topology form an irredundant cover, each signature along the cut has at least one essential PBD. The constraints in Equation 8.4 suggest that there are four distinct two-input functions that preserve one essential PBD. However, one of these functions will correspond to the 1 input identity function, *i.e.*, a buffer (or inverter in the negated case). Ignoring this case, there are three other distinct functions can be tried at each node, which requires no more than 3^m total gate combinations to determine logic feasibility. □

Handling arbitrary topologies with no implementation constraints requires more computation where 5^m gate combinations are examined. However, in practice, our approach is faster than the naïve enumeration described at the beginning of the section because the operations are performed on the signatures, not over the whole truth table. Also, essential PBDs can still significantly prune the search space. Each cut must still cover all of the PBDs. If an edge from internal node or primary input does not appear past a certain logic level in the topology, its signature's essential PBDs must be preserved across that level.

8.4 Concluding Remarks

We introduced two techniques that can enable powerful synthesis optimizations using global don't-cares, which is critical for post-placement optimization where less design flexibility exists. We first presented a node-merging strategy that can operate directly on mapped netlists. Unlike the work in [95], our techniques pursue global ODCs, which are successfully evaluated against logic synthesis transformations. By exploiting global don't-cares, we identify several node mergers even after extensive synthesis optimizations, resulting in up to 23% area reduction. Furthermore, our techniques are not restricted to mapped circuits and can be used directly on AIGs in sequential verification applications.

In this context, global ODC analysis becomes more important because of the greater depth in unrolled circuits.

Finally, we introduced a novel, goal-driven synthesis strategy that quickly finds logic implementations for arbitrary topologies. In the next chapter, we demonstrate the effectiveness of this approach by targeting a critical path delay reduction optimization goal. This synthesis approach coupled with node merging enables significant optimizations in the physical domain.

CHAPTER IX

Path-based Physical Resynthesis using Functional Simulation

In this chapter, we apply the scalable simulation-based framework developed throughout this dissertation to the physical synthesis domain. In particular, we introduce (1) a novel criterion, based on path monotonicity, that identifies those interconnects amenable to optimization through logic restructuring and (2) a synthesis algorithm relying on logic simulation and placement information to identify placed subcircuits that hold promise for interconnect reduction. Experiments indicate that our techniques find optimization opportunities and improve interconnect delay by 11.7% on average, at less than 2% wirelength and area overhead.

As mentioned in [10], many critical paths cannot be improved through cell relocation and better timing-driven placement. Furthermore, the inaccuracy of timing estimates before detailed placement limits the effectiveness of techniques from [40] in eliminating path non-monotonicity. We target these non-monotone paths for resynthesis by generating different logic topologies that improve circuit delay. We use the synthesis strategy introduced in Chapter VIII to efficiently determine whether a logic implementation for the desired topology is possible.

In the example of Figure 9.1, we suggest that by applying our technique, a subcircuit

with a long critical path can be transformed to a functionally-equivalent subcircuit with smaller critical path delay. Unlike most techniques from logic synthesis, the circuit restructuring can work directly on mapped circuits with complex standard cells. Compared to work in [84], our approach exploits global don't-cares to enhance logic restructuring. In [53], redundancy addition and removal (RAR) are used to improve circuit timing. However, these rewiring techniques consider only a subset of our transformations, where we use redundancy and physical information in conjunction to directly guide the resynthesis of subcircuits containing multiple cells.

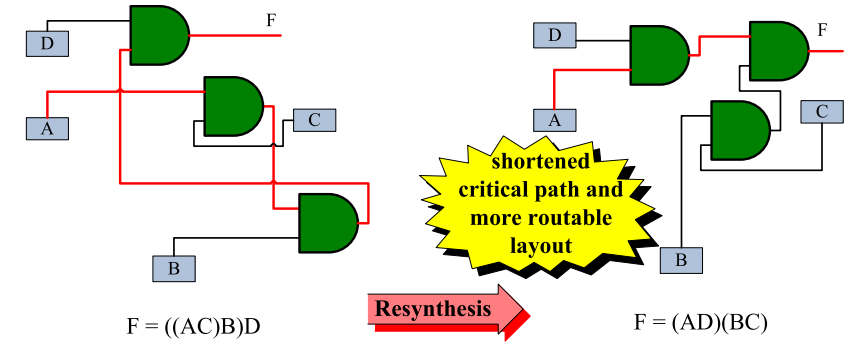

Figure 9.1: The resynthesis of a non-monotone path can produce much shorter critical paths and improve routability.

Our experiments indicate that large circuits often contain many long critical paths that can be effectively targeted with restructuring. Improving these paths results in consistent delay improvements, of 11.7% on average, with minimal degradation to other performance parameters. Furthermore, we achieve almost twice the delay improvement of that achieved by RAR-based timing optimizations. Our techniques are fast and scale to large designs, whereas completely characterizing node functionality with BDDs would require a prohibitive memory footprint.

In Section 9.1, we introduce our interconnect optimization strategy. In Section 9.2,

we propose a metric for finding circuit paths that require restructuring. Section 9.3 and 9.4 integrate these innovations in a novel physically-aware synthesis approach that uses simulation. Empirical evaluation is presented in Section 9.5.

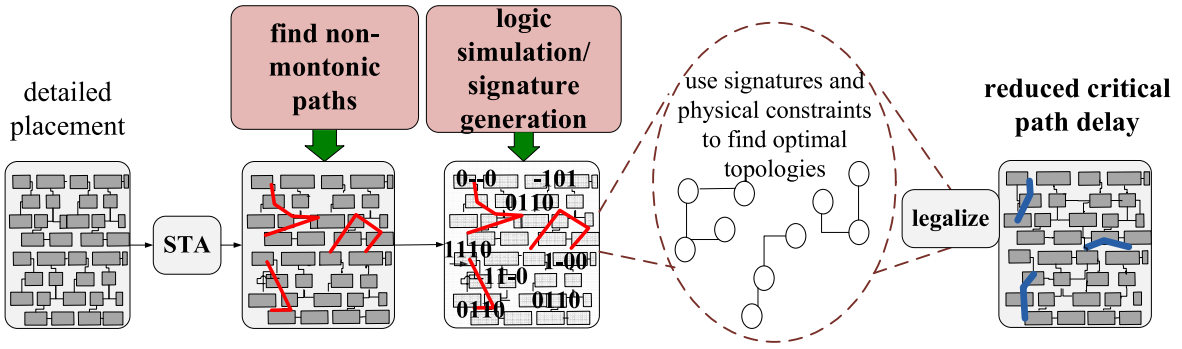

Figure 9.2: Improving delay through logic restructuring. In our solution, we first identify the most promising regions for improvements, and then we restructure them to improve delay. Such netlist transformations include gate cloning, but are also substantially more general. They do not require for the transformed subcircuits to be equivalent to the original one. Instead, they use simulation and satisfiability to ensure that the entire circuit remains equivalent to the original.

9.1 Logic Restructuring for Timing Applications

We introduce a logic resynthesis approach that accounts for physical aspects of performance optimization, by leveraging our simulation-based framework discussed in the previous chapters. We illustrate the approach in Figure 9.2. Starting from a fully placed circuit, we identify critical paths using static timing analysis. We then apply a novel metric, introduced in Section 9.2, that selects subcircuits for which logic restructuring could provide the greatest improvements. We restructure these subcircuits using bit signatures along with physical constraints, to derive a topology that is logically equivalent to the original one but exhibits better performance. Finally, we legalize the altered placement and up-

date the timing information. Beside timing improvements, this technique could can target other objectives as well, such as wirelength reduction. Using signatures for restructuring applications is advantageous because signatures can characterize internal nodes for netlists mapped to standard cells as well as for technology-independent netlists. In contrast, other logic rewriting strategies, such as the one in [62], cannot operate on technology-mapped circuits and do not take physical information into account.

9.2 Identifying Non-monotone Paths

To maximize the effectiveness of our post-placement optimizations, we target parts of the design with critical timing constraints that are amenable to restructuring. In this section, we introduce our fast dynamic programming (DP) algorithm for finding *non-monotone* paths, *i.e.*, paths that are not of minimal length. Unlike the work in [10] that considers only paths with two wire segments, we consider paths of arbitrary lengths and can scale to many more segments in practice. We propose two models for computing path monotonicity: (1) wirelength-based and (2) delay-based. Non-monotonic paths indicate regions where interconnect and/or delay may be reduced by post-placement optimization.

9.2.1 Path Monotonicity

First, static timing analysis is performed to enable our delay-based monotonicity calculation and identify critical and near-critical paths. We use a timing analyzer whose interconnect delay calculation is based on Steiner-tree topologies produced by FLUTE [23][1] and the D2M delay metric [6] that is known to be more accurate than Elmore delay. Before focusing on critical paths, we describe a general approach that examines the monotonicity

[1] Timing-driven Steiner trees can be easily used too [5].

```
inputs
  Nodes: netlist
  Dist: length of paths considered
output
  NMF: NMF between each node
gen_NMF(Nodes nodes, Dist K) {
   levelize(nodes);
   for_each node1 ∈ nodes {
      for_each node2 ∈ range(node1 + 1, node1 + K)
         c_ideal_array[node1,node2] = c_ideal(node1, node2);
   }
   for_each node1 ∈ nodes {
      subtot[] = 0;
      for_each node2 ∈ range(node1_succ, node1_succ + K) {
         subtot[node1,node2] = max(subtot[node1,node2_pred]
         + c(node2_pred, node2));
         factor = subtot[node1,node2] / c_ideal_array[node1,node2];
         NMF[node1,node2] = factor;
      }
   }
}
```

Figure 9.3: Computing the non-monotone factor for k-hop paths.

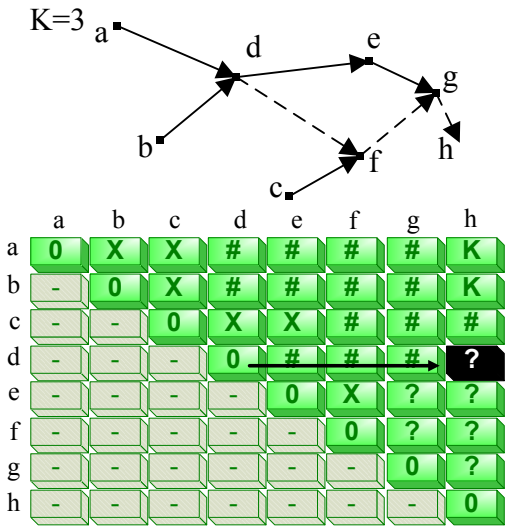

Figure 9.4: Calculating the non-monotone factor for path $\{d,h\}$. The matrix shows sub-computations that are performed while executing the algorithm in Figure 9.3.

of every path. We define the *non-monotone factor (NMF)* for the path $\{x_1,...,x_k\}$ with respect to a given cost metric (such as wirelength or delay) as follows:

$$NMF = \frac{1}{c_{ideal}(x_1,x_k)} \sum_{n=1}^{k-1} c(x_n,x_{n+1}) \tag{9.1}$$

where $c(a,b)$ defines the actual *cost* between a and b and $c_{ideal}(a,b)$ defines an optimal cost. When $NMF = 1$, the path is monotone under the cost metric. We explore two definitions for cost, one based on rectilinear distance and the other on delay.

In the former case, $c(a,b)$ is the rectilinear distance between cell a and b while $c_{ideal}(a,b)$ is the optimal rectilinear distance assuming a monotonic path. For the delay-based definition, $c(a,b)$ is the $AT(b) - AT(a)$, where AT is arrival time. We define c_{ideal} as the delay of an optimally buffered path between a and b as described by [67] and given by the following formula:

$$c_{ideal}(a,b) = dist(a,b)(R_{buf}C + RC_{buf} + \sqrt{2R_{buf}C_{buf}RC}) \tag{9.2}$$

where R and C are the wire resistance and capacitance, respectively, and R_{buf} and C_{buf} are the intrinsic resistance and input capacitance of the buffers. $dist(a,b)$ is the rectilinear distance between a and b. Unlike the distance calculation where the ideal path length between a and b can be equal to the actual path length, the optimal buffered wire between a and b has delay $\leq AT(b) - AT(a)$. We only attempt to optimize paths with large non-monotone factors.

9.2.2 Calculating Non-monotone Factors

We now present our algorithm for calculating the NMF of all k-hop paths in a circuit, for a given $k \geq 2$. Our experiments reveal the existence of high NMFs on even relatively

short paths, which is advantageous since optimizations on these smaller paths often mean fewer perturbations to the existing placement while significant performance benefits are achieved.

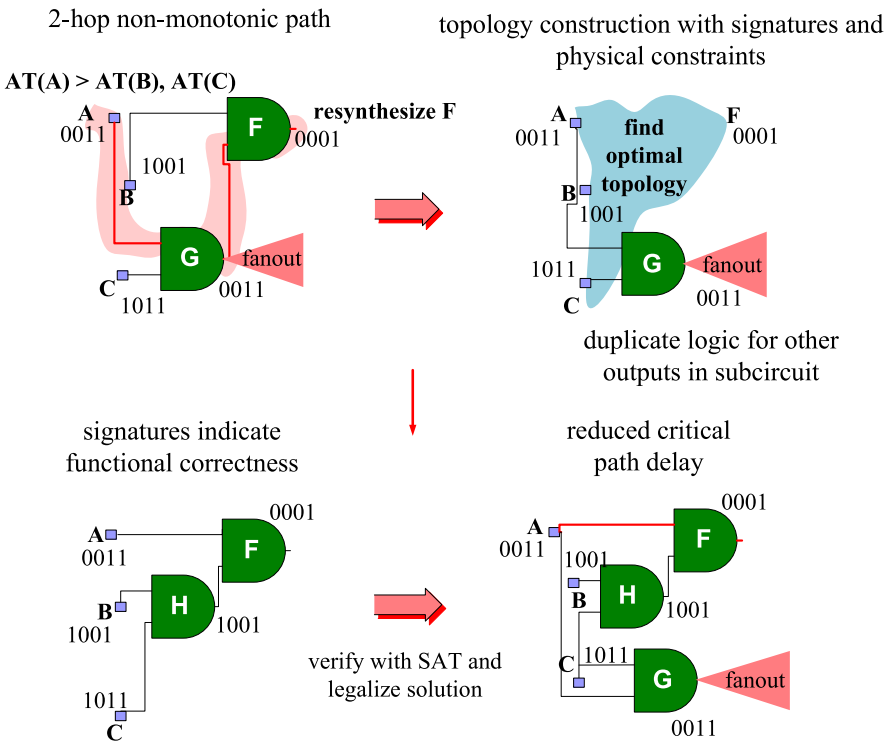

Figure 9.5: Our flow for restructuring non-monotone interconnect. We extract a subcircuit selected by our non-monotone metric and search for alternative equivalent topologies using simulation. The new implementations are then considered based on the improvement they bring and verified to be equivalent with an incremental SAT solver.

The non-monotone factor can be efficiently computed for every path using a $O(nk)$-time algorithm for n nodes in the circuit, as shown in Figure 9.3. First, the circuit is levelized. Then, c_{ideal} is computed for node pairings with a connecting path of $\leq k$ hops, and the values are stored in `c_ideal_array`. All pairs are traversed again, and the `subtot` is generated by computing the maximum cost from `node1` to `node2` through

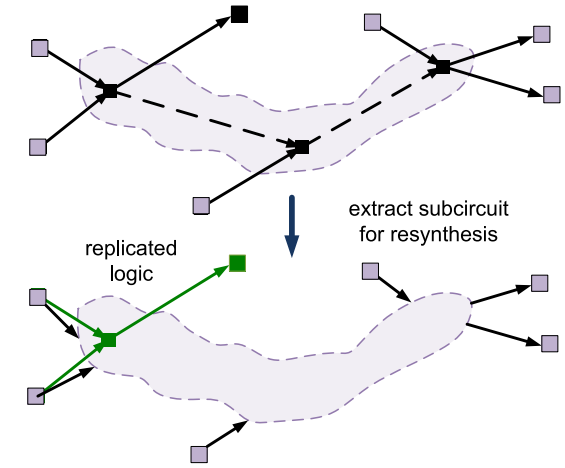

Figure 9.6: Extracting a subcircuit for resynthesis from a non-monotone path.

a recurrence relation. The NMF is computed for the subpath, {`node1, node2`}, by dividing the total cost, `subtot`, by `c_ideal[node1,node2]`. In Figure 9.4, we show an example computation on a subcircuit being traversed using the `gen_NMF()` function where $k = 3$ and the current `node1` is d. The matrix indicates the NMFs already computed with #, and nodes not lying on the same path with X. Because we traverse the graph in levelized order, a, b, c have already been examined. Notice, that nodes that are farther than k hops away are not examined (indicated by K in the matrix). For node d, the non-monotone factor is computed for path $\{d, h\}$ by determining all the incoming sub-paths to h first. In this example, $\{d, h\}$ has the highest NMF if rectilinear distance is the cost function.

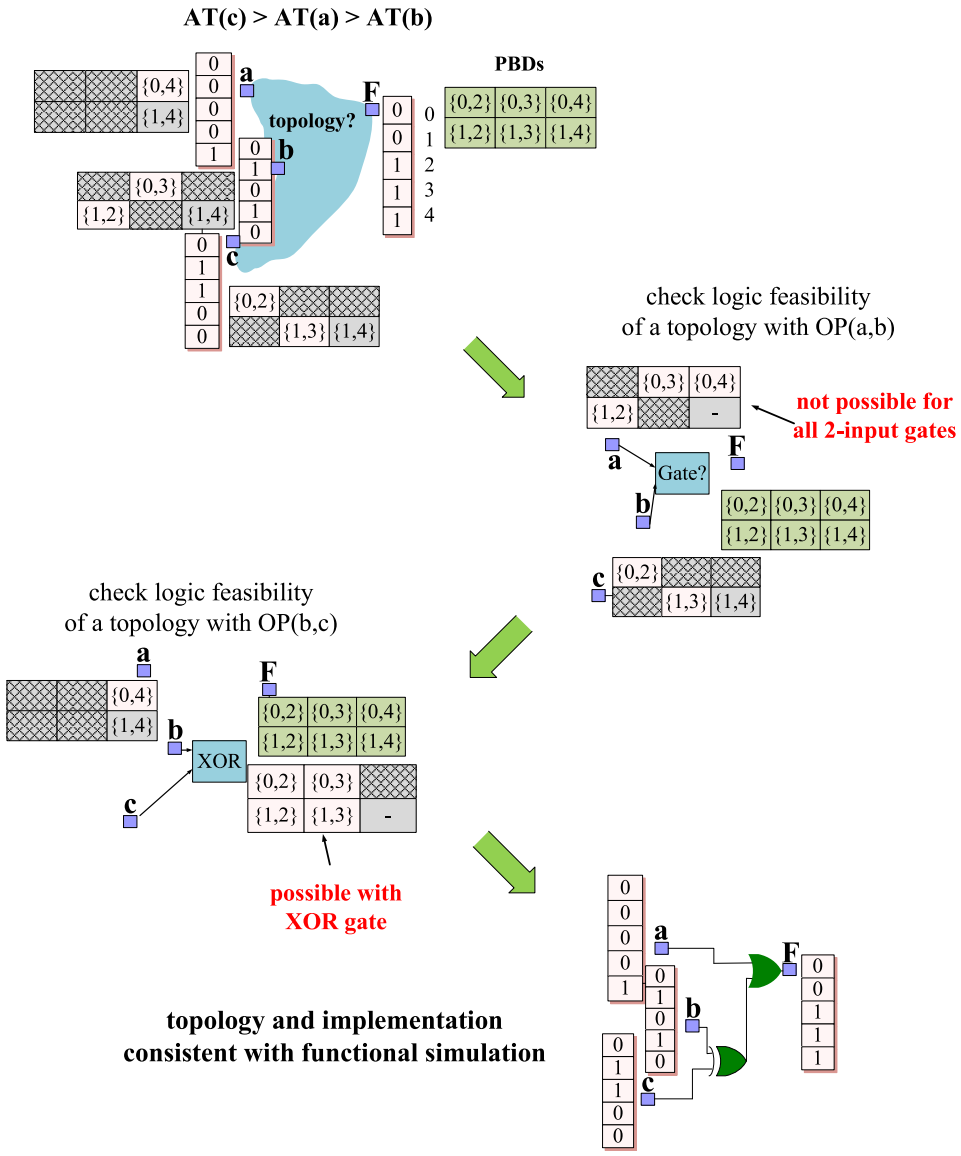

Figure 9.7: Signatures and topology constraints guide logic restructuring to improve critical path delay. The figure shows the signatures for the inputs and output of the topology to be derived. Each table represents the PBDs of the output F that are distinguished. The topology that connects a and b directly with a gate is infeasible because it does not preserve essential PBDs of a and b. A feasible topology uses b and c, followed by a.

9.3 Physically-aware Logic Restructuring

We optimize the subcircuits that are identified by the path monotonicity metric as illustrated in Figure 9.5. We first select a region of logic determined by the non-monotone path for resynthesis. We then use signatures to find an alternative implementation with a topology that improves physical parameters and that it is logically equivalent to the original implementation (up to the signatures). This implementation is then formally verified by performing SAT-based equivalence checking between the original and new netlists.

Previous work on improving path monotonicity used logic replication [42]. However, the technique is restricted to the topology of the extracted subcircuit, and its optimization is independent of the subcircuit's functionality. Furthermore, as observed in [42], cell relocation sometimes cannot improve path monotonicity. In the previous chapter, we introduced the theoretical framework to resynthesize a subcircuit given a set of inputs and a target output by using our algorithm for determining logic feasibility. We now introduce an algorithm for constructing subcircuits using signatures and physical constraints to optimize the interconnect.

9.3.1 Subcircuit Extraction

After identifying the path that is least monotone, we extract a subcircuit (as shown in Figure 9.6) with incoming path edges as inputs and outgoing edges as outputs. The inputs and fanout of the subcircuit are treated as fixed cells, forming the physical constraints. As shown in the figure, if there are outgoing edges at intermediate nodes in the path, this logic is duplicated. In practice, we experience minimal cell area increase because the number of duplicated cells is small, and the resynthesized circuit is smaller than the original in many cases.

9.3.2 Physically-guided Topology Construction

In addition to efficiently *determining* the logic feasibility of various topologies, we propose an algorithm that uses PBDs and physical constraints to efficiently *construct* logically feasible topologies. In this paper, we guide our approach using delay and physical proximity. In the example shown in Figure 9.7, we try to find an optimal restructuring to implement the target function F with inputs a, b, and c. The functionality of the original circuit is represented by signatures. The figure also shows a table associated with each signal showing the PBDs that are distinguished. The non-essential PBDs for each input signature have light-gray background.

The example shows that the arrival time for c is the greatest, followed by a, then b. Therefore, we should consider alternative topologies where c's value is required later. We also consider the proximity of the signals and therefore examine topologies where a direct operation between a and b is performed. Notice that if all possible two-input operations are tried, the essential PBDs are not preserved and hence these are not feasible topologies. We then consider another topology where a can be accessed later and thus it generates an operation connecting c and b first. For this topology, we observe that an XOR gate preserves the essential PBDs. We then can easily derive that an OR gate is needed to implement F.

Algorithm. Figure 9.8 introduces the pseudo-code of the restructuring algorithm for non-monotone interconnect. After identifying the non-monotone paths, `Optimize_Interconnect()` restructures a portion of the critical path. Before restructuring the path, we first simplify the signatures with `simplify_signatures()` by noting that the size of the signature $|S_F|$ can be reduced to the number of different input combinations

that occur across $\{S_1, ...S_i\}$. Thus, only a subset of the signature is needed for restructuring because the small subcircuits considered have a maximum of 2^i possible different input combinations, smaller than the number of simulation vectors applied.[2]

```
Optimize_circuit(){
  gen_NMF(); num_tries = X;
  while(worst_nmf > 1) {
    if(nckt == Optimize_Interconnect(worst_nmf)) {
      if(!check_equiv(nckt)) {
        refine_signatures();
        continue;
      }
      update_netlist();
      legalize_placement();
      update_NMF();
}}}
Subckt* Optimize_Interconnect(Subckt F){
  simplify_signatures(F);
  while(find_opt_topology(constrs)) {
    if(nckt == check_logical_feasibility()) {
      nckt → opt_place();
      return nckt;
    }
    constrs.add(nckt);
}}
```

Figure 9.8: Restructuring non-monotone interconnect.

In `find_opt_topology()`, we find a topology that optimizes delay for the given physical constraints, such as the physical locations of the subcircuit's inputs and outputs The topology is created by a greedy algorithm which derives a fanout-free topology from the current input wires. We examine each pair of wires, apply an arbitrary cell, and estimate the delay to the output of the subcircuit. The topology is then greedily constructed so that wire pairs with earlier arrival times are favored in the early computation stages of the topology. From this initial topology, we can obtain an upper-bound for the best possible

[2]In our experiments, we apply 2048 input vectors and restructure subcircuits with < 10 inputs.

implementation. If a topology can't be found that satisfies the constraints, the function returns.

The topology that is derived is then then checked for logical feasibility using PBDs and signatures in `check_logical_feasibility()`. If the topology is feasible, we associate the appropriate gate with each vertex and place the subcircuit. Our placement routine considers only the legality of the subcircuit (we call a placement legalizer later for the entire design). In our approach, we determine a location for each gate by placing it at the center of gravity of its inputs and outputs and then sifting the gate to different nearby locations. This sifting is done over all the gates and over several passes until a locally optimal solution is achieved, resulting in no overlaps. For the typically small subcircuits considered, this requires a small computational effort.

Finally, if the topology is not logically feasible, we add a *functional* constraint that prevents the construction of similar topologies. The constraint states which wire pairs should not be combined again. For instance, for the multiplexor, $z = a'b + ac$, there is no implementation with a fanout-free topology with inputs $\{a,b,c\}$. If a and b form a wire pair, no implementation can preserve its essential PBDs. However, we can exploit Theorem 8 and consider implementations that eliminate one of the inputs. In this case, if the implementation $a'b$ is attempted, the wire b does not need to reappear in the topology. Therefore, a constraint is added so that the inputs to the topology are now $\{a'b,a,c\}$. With these inputs, a fanout-tree does exist which is logically feasible.

If `Optimize_Interconnect()` returns a subcircuit, we check the equivalence of the entire circuit using a SAT engine. In the case where our candidate produces a functionally different circuit (which is rare, as shown in Section 9.5), we use the counterexample

generated by SAT to refine our simulation, hence improving the signatures' quality. If the resulting subcircuit passes verification, we update the netlist and legalize the placement. We update the timing information and the NMFs if a new critical path is found, in which case we select with the next highest NMF and restructure it.

9.4 Enhancing Resynthesis through Global Signature Matching

Our resynthesis strategy considers the inputs to a non-monotone path for resynthesis. This strategy is convenient because 1) the set of inputs can always implement the target output and 2) the inputs tend to be physically close to the target output. However, local manipulations can be enhanced by incorporating global information. In this section, we explain how to exploit the same advantages of structural hashing for area reductions, by applying matching to the signature abstraction. Furthermore, our approach is more powerful than logic rewriting because the signatures are matched while considering global don't-cares, and our initial physically-guided local rewriting over signatures already exploits don't-cares.

Strategy. To resynthesize non-monotone paths, we exploit signature matching in the following way:

1. Find a set of candidate wires within a certain distance from the output wire to be resynthesized.

2. Check whether any candidate's signature is equal to the output signature up to don't-cares, as discussed in Chapter VIII. If a match is found and the timing can be improved, replace the output wire with the corresponding candidate wire.

3. While checking logic feasibility in topological order, check whether any of the in-

ternal wires can be reimplemented using a candidate wire with a matching signature to further improve timing.

The candidate wires are chosen by proximity to the output wire being resynthesized as determined by its half-perimeter wirelength (HPWL). Any wire annotated with an arrival time after the current output wire's annotated arrival time is not considered. Unlike the resynthesis algorithm that uses a simplified signature, for signature matching, we consider the whole signature except for the don't-cares. In this case, a single comparison between signatures can be performed quickly and it is more efficient than finding a common set of inputs to both wires and then reducing the signatures to the number of simulated different input combinations. Notice that our algorithm is used to enhance the previous resynthesis strategy and improve the timing of a specific implementation, while in general topology construction only the inputs to the subcircuit are considered.

9.5 Empirical Validation

We implemented and tested our algorithms with circuits from the IWLS 2005 benchmark suite [102], with design utilization set to 70% to match recent practices in the industry. Our wire and gate characterizations are based on a 0.18μm technology library. We perform static timing analysis using the D2M delay metric [6] on Rectilinear Steiner Minimal Trees (RSMTs) produced by FLUTE [23]; here FLUTE can be easily replaced by any timing-driven subroutine, without significantly affecting the overall trends of our experiments. Our netlist transformations are verified using a modified version of MiniSAT [29] and placed using Capo 10 [16]. We have considered several different initial placements for each circuit by varying a random seed in Capo and report results as average

improvements over these placements. Our netlist transformations are legalized using the legalizer provided in the GSRC Bookshelf [105].

To evaluate delay improvements, we apply the algorithm of Figure 9.8 to the test-benches. We applied 2048 random simulation patterns initially to generate the signatures. We considered paths of less than or equal to 4 hops (5 nodes) using our delay-based metric, which allowed us to find many non-monotone paths while minimizing the size of the transformations considered. We conducted several optimization passes until no more gains were achieved.

9.5.1 Prevalence of Non-monotonic Interconnect

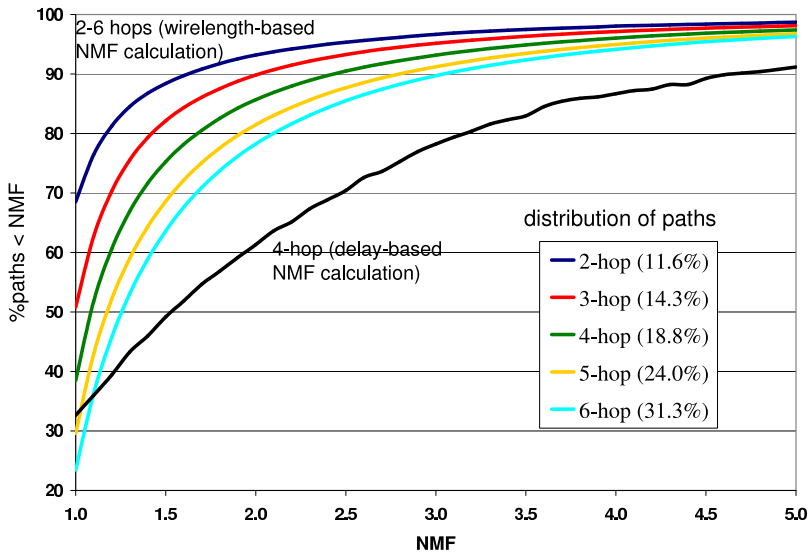

Figure 9.9: The graph plots the percentage of paths whose NMF is below the corresponding value indicated on the x-axis. Notice that longer paths tend to be non-monotone and at least 1% of paths are > 5 times the ideal minimal length.

Our experiments indicate that circuits often contain many non-monotone paths. In Figure 9.9, we illustrate a cumulative distribution of the percentage of paths whose NMFs

is below the corresponding value on the x-axis. We generated these averages over all the circuits in Table 1. Each line represents a different path-length examined, where we considered paths from 2 hops to 6 using the wirelength-based NMF metric. We also show the cumulative distribution for the 4-hop delay-based NMF calculation used to guide our delay-based restructuring. Of particular interest is the percentage of monotonic paths, *i.e.*, paths with NMF = 1.

Notice that smaller paths of 2-hops are mostly monotone, whereas the percentage of monotone paths decreases to 23% when considering 6-hop paths. This indicates that focusing optimizations on small paths only, as in [10], can miss several optimization opportunities. It is also interesting to note that there are paths with considerably worse monotonicity having NMFs > 5, revealing regions where interconnect optimizations are needed. We observe similar trends using our delay-based metric. The inclusion of gate delay on these paths results in greater non-monotonicity when compared to the wirelength-metric. Although not shown, each individual circuit exhibits similar trends.

9.5.2 Physically-aware Restructuring

We show the effectiveness of our delay-based optimization by reporting the delay improvements achieved over several circuits. In Table 1, we provide the number of cells and nets for each circuit. In the `Performance` columns, we give the percentage delay improvement, the runtime in seconds, and the percentage of equivalence-checking calls where candidate subcircuits preserved the functionality of the entire circuit. We also report the overhead of our approach with the percentage of wirelength increase and the percentage of cell count increase.

Considering 8 independently generated initial placements for each circuit, our tech-

circuit	cell count	net count	performance			overhead	
			%delay improv	time (s)	%equiv checks	%wire	%cells
sasc	563	568	14.1	41	100	2.35	3.13
spi	3227	3277	10.9	949	82	4.53	0.73
des_area	4881	5122	12.3	503	93	1.09	0.31
tv80	7161	7179	9.1	1075	71	2.50	0.17
s35932	7273	7599	27.5	476	100	2.14	0.19
systemcaes	7959	8220	13.9	748	95	0.89	-0.07
s38417	8278	8309	11.7	481	84	0.68	-0.21
mem_ctrl	11440	11560	9.2	678	37	0.05	-0.02
ac97	11855	11948	6.3	245	100	0.44	0.02
usb	12808	12968	12.2	605	80	0.30	0.06
DMA	19118	19809	14.5	845	65	0.16	0.08
aes	20795	21055	6.4	603	100	0.13	0.01
ethernet	46771	46891	3.7	142	100	0.08	0.06
average			11.7%		85.1%	1.20%	0.34%

Table 9.1: Significant delay improvement is achieved using our path-based logic restructuring. Delay improvement is typically accompanied by only a small wirelength increase.

niques improve delay by 11.7% on average. For some circuits, such as *s*35932, several don't-care enhanced optimizations enabled even greater delay improvements.

Note that, by optimizing only one output of a given subcircuit, we greatly reduce the arrival time of the critical output, while only slightly degrading the performance of computation of other outputs. Moreover, through our efficient use of don't-cares, several *m*-input subcircuits could be restructured to require fewer than *m* inputs. As a special case of the previous point, sometimes an input to the subcircuit is functionally equivalent to the output of the subcircuit when don't-cares are considered, enabling delay reduction along with removal of unnecessary logic. Signatures are efficient in exploiting these opportunities. Finally, the decomposition of large gates into smaller gate primitives through our restructuring algorithm often produces better topologies because we construct a topology that

meets the physical constraints more precisely.

We also believe that further gains would be enabled by combining buffering, relocation, and gate sizing strategies in our restructuring optimizations. The wirelength and cell-count overhead are minimal because only a few restructurings are needed and the optimizations can simplify portions of logic. In some cases the number of cells is reduced.

The runtime of our algorithm scales well for large circuits due to the use of logic simulation as the main optimization engine. Furthermore, the high percentage of equivalence checking calls that confirmed the equivalence of our transformations indicates that signatures are effective at finding functionally equivalent candidates. Furthermore, we observe that SAT-based equivalence checking requires a small fraction of the total runtime compared to constructing optimal topologies, even for our larger circuit examples. This small runtime can be attributed to the locality of most structural transformations. Because the structures of the original and modified circuits are similar, the SAT instance can be greatly reduced in size and complexity. This limits the complexity of our approach, which tends not to grow with the size of the overall circuit.

To check if our techniques provide comparable improvement when the initial placement is optimized for timing, we performed the following experiment. We first produced 64 independent initial placements optimized for total wirelength. Compared to these 64 wirelength-optimized placements, the best placements achieve 17.0% shorter delay on average and serve as proxies for timing-optimized placements in our experiments. Starting with these initial placements already optimized for delay, our logic restructuring approach can extract further improvements, reducing the delay by 6.5% on average.

9.5.3 Comparison with Redundancy Addition and Removal

We compare our technique with timing optimization using redundancy addition and removal (RAR). We implement redundancy removal using signatures to identify equivalent nodes up to don't-cares. In the context of path-based resynthesis, the inputs to the subcircuit, along with signals that have earlier arrival time and are within a bounding box determined by the HPWL of the output, are considered as candidates for rewiring. If one of these signals is equivalent to the output up to don't-cares in the circuit, rewiring is performed and the timing improved.

circuit	%delay	
	ours	RAR
sasc	13.8	12.1
spi	15.0	12.6
des_area	15.4	11.1
tv80	12.7	3.1
s35932	23.1	21.8
systemcaes	10.1	4.0
s38417	26.3	2.9
mem_ctrl	12.9	8.2
ac97	5.3	3.1
usb	10.8	0.0
DMA	10.7	0.0
aes	5.3	4.7
average	13.5%	7.0%

Table 9.2: Effectiveness of our approach compared to RAR.

In Table 9.2, we compare the delay improvement of our resynthesis strategy to redundancy addition and removal. For this experiment, we report results on a random slice of initial placements from our suite. Note that our technique achieves almost twice as much improvement as RAR in improving delay, and our results are more consistent over all the circuits and are never worse than RAR.

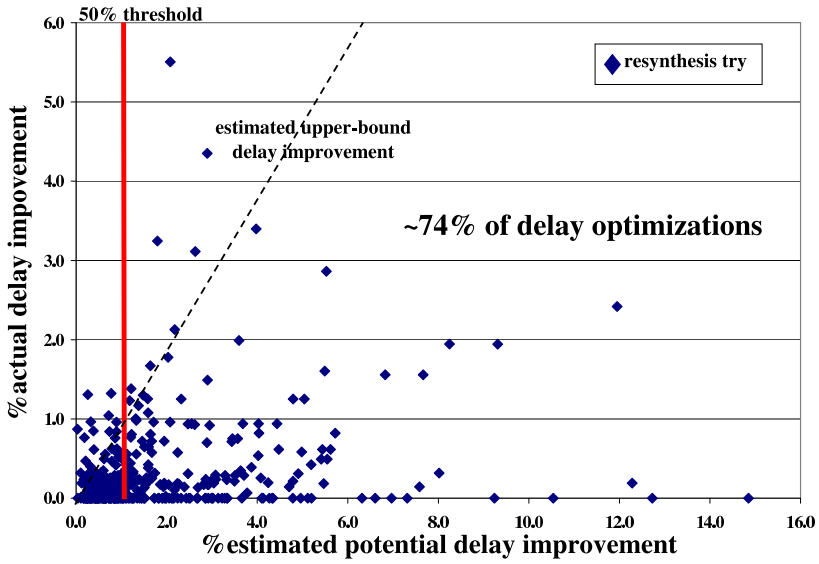

Figure 9.10: The graph above illustrates that the largest *actual* delay improvements occur at portions of the critical path with the largest *estimated* gain using our metric. The data points are accumulated gains achieved by 400 different resynthesis attempts when optimizing the circuits in Table 1.

In Figure 9.10, we demonstrate that our delay-based NMF metric is effective at guiding optimization. Each data point represents a different resynthesis attempt considering all of the circuits in Table 1. The x-axis shows the predicted percentage delay gain possible (determined by the optimal-buffered delay). The y-axis indicates the actual gain. Data points that lie on the x-axis indicate resynthesis attempts that did not improve delay (a better topology could not be found). The 50% threshold line divides the graph so that the number of resynthesis attempts are equal on both sides. The diagonal line indicates an upper-bound prediction for delay gain. Because some of the optimizations reduce the support of the original subcircuit, we can improve the delay beyond the original estimate which considers all of the subcircuit's inputs. Therefore, some of the data points are above the upper-bound line. On the other hand, a resynthesis attempt produces a smaller

than estimated improvement when the ideal topology is not logically feasible or when removing cell overlap degrades the quality of the initial placement. Although the NMF and gain calculations do not directly incorporate circuit functionality, 74% of all delay gains are found on the right half of the graph. The correlation to our metric could be further improved by incorporating the percentage of gain possible with respect to near-critical paths.

9.6 Concluding Remarks

In this chapter, we leveraged our simulation-based framework to improve the quality of delay optimization without sacrificing other performance metrics. In particular, we introduced a novel simulation-guided synthesis strategy that is more comprehensive than current restructuring techniques. We developed a path-monotonicity metric to focus our efforts on the most important regions of a design. Our optimizations lead to 11.7% delay improvement on average, over several different initial placements. Also, our delay-based monotonicity metric indicated that 65% of the paths analyzed were non-monotone. We further observe delay improvements on placements initially optimized for delay, which are consistent with our reported average improvement. We believe that our approach offers an effective bridge between current topological-based synthesis and lower-level physical synthesis approaches. It enables less conservative timing estimates to be made early in the design flow so that other performance metrics can be improved without adversely affecting timing, and it also reduces the amount of buffering required by shortening critical paths.

CHAPTER X

Conclusions

Achieving timing closure is becoming increasingly difficult due to the increasing significance of interconnect delay. For complex designs, failing to achieve timing closure results in costly design-flow iterations and delays market entry of the final product. Previous strategies for achieving timing closure are often incapable of exploiting logic transformations that promise significant delay improvements. In this dissertation, we introduce an aggressive physical synthesis application that employs a broader set of optimizations to reduce interconnect delay while minimizing impact on the remaining circuit. The goal is to improve timing closure, as interconnect becomes more dominant and current methodologies become less adequate. To enable powerful transformations, we leverage logic simulation to characterize the behavior and flexibility of internal nodes using bit signatures. By performing logic manipulations on the signatures instead of the circuit, we abstract away much of the design complexity enabling numerous transformations to be examined. Transformations that result in the greatest delay improvements are verified formally, while the scalability of such verification is dealt with particular care.

10.1 Summary of Contributions

In this dissertation, we developed a comprehensive signature-based framework that efficiently identifies logic optimizations in complex digital designs. This framework consists of the following components.

- A simulation strategy for sensitizing parts of a design that are difficult to control from the primary inputs with the goal of exposing corner-case behavior in the design. We developed a novel metric for determining the information content (entropy) of different groups of signals under given simulation vectors and introduced a SAT-based algorithm for evenly sensitizing these signals. In the experimental evaluation, our techniques evenly sensitized a design where random simulation had not succeeded.

- A strategy for efficiently computing don't-cares and encoding them in signatures to enhance synthesis optimizations. We showed that the approximation used to generate don't-cares was both fast and accurate.

- A SAT-solving methodology that leverages the increasing availability of multi-core systems to enable more efficient verification of signature-based transformations. We introduced a priority scheduler for handling multiple SAT instances of varying complexity and proposed a lightweight parallelization strategy to solve particularly hard instances.

- Techniques for logic manipulation based on signatures. We introduced a node-merging optimization that leads to significant area reductions. We also developed a

signature-based resynthesis strategy that can be efficiently guided by physical optimization criteria, such as delay and wirelength minimization, as well as routability. We then introduced a post-placement resynthesis strategy strategy that uses path monotonicity to identify paths that are most amenable to performance optimization.

Empirical results indicated the effectiveness of our signature-based methodology. We showed that logic simulation can efficiently target hard-to-sensitize regions in a circuit. Furthermore, we demonstrated that signatures are a good approximation of a node's functionality, and can account for both controllability and satisfiability don't-cares. For example, signatures were used to effectively identify nodes that could be merged, and don't-cares facilitated additional node mergers. These results indicated the potential of functional simulation to support fast and powerful design optimizations. In the physical synthesis domain, we demonstrated that the ability to quickly identify numerous resynthesis opportunities is particularly advantageous. Empirical results confirmed that our techniques compare favorably with earler algorithms.

10.2 Directions for Future Research

The use of logic simulation as an abstraction represents a major contribution of our work. This abstraction simplifies search for powerful optimizations subject to functional, temporal, and physical constraints. Our techniques support a variety of performance, power, and manufacturability objectives.

We believe that our don't-care analysis is also useful to enhance verification coverage. In practice, a simulation vector that toggles logic containing a design bug might not produce an observable discrepancy with the golden model at the outputs. By incorporating

observability measures in our coverage analysis to guide a SAT-based resimulation, we could improve the quality of simulation performed.

Finally, we observe that our work in parallel SAT enables the development of a new methodology in CAD tool flows that better utilizes multi-core systems. A future avenue of research would consider multiple incremental optimizations applied in parallel. Such optimizations could be automatically handled by our parallelization strategy to maximize CPU utilization.

INDEX

BIBLIOGRAPHY

[1] M. Abramovici, M. Breuer, and A. Friedman, "Digital system testing and testable design", *W.H.Freeman*, 1990.

[2] M. Abramovici, J. DeSousa, and D. Saab, "A massively-parallel easily-scalable satisfiability solver using reconfigurable hardware", *DAC*, pp. 684-690, 1999.

[3] A. Ajami and M. Pedram, "Post-layout timing-driven cell placement using an accurate net length model with movable steiner points", *DAC*, pp. 595-600, 2001.

[4] F. Aloul, B. Sierawski, and K. Sakallah, "Satometer: how much have we searched?", *TCAD*, pp. 995-1004, 2003.

[5] C. Alpert, A. Kahng, C. Sze, and Q. Wang, "Timing-driven steiner trees are (practically) free", *DAC*, pp. 389-392, 2006.

[6] C. Alpert, A. Devgan, and C. Kashyap, "RC delay metric for performance optimization", *TCAD*, pp. 571-582, 2001.

[7] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick, "The landscape of parallel computing research: a view from Berkeley", *ERL TR*, Berkeley, 2006.

[8] R. Ashenhurst, "The decomposition of switching functions", *International Symposium on the Theory of Switching*, pp. 74-116, 1957.

[9] L. Baptista and J. Marques-Silva, "Using randomization and learning to solve hard real-world instances of satisfiability", *ICPPCP*, pp. 489-494, 2000.

[10] G. Beraudo and J. Lillis, "Timing optimization of FPGA placements by logic replication", *DAC*, pp. 541-548, 2003.

[11] V. Bertacco and M. Damiani, "The disjunctive decomposition of logic functions", *ICCAD*, pp. 78-82, 1997.

[12] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs", *TACAS*, pp. 193-207, 1999.

[13] D. Brand, “Verification of large synthesized designs”, *ICCAD*, pp. 534-537, 1993.

[14] R. Bryant, “Graph-based algorithms for Boolean function manipulation”, *Trans. on Comp.*, pp. 677-691, 1986.

[15] M. Bushnell and V. Agrawal, “Essentials of electronic testing”, *Kluwer*, pp. 129-150, 2000.

[16] A. Caldwell, A. Kahng, and I. Markov, “Can recursive bisection alone produce routable placements?”, *DAC*, pp. 693-698, 2000.

[17] C.-W Chang, C.-K Cheng, P. Suaris, and M. Marek-Sadowska, “Fast post-placement rewiring using easily detectable functional symmetries”, *DAC*, pp. 286-289, 2000.

[18] K. H. Chang, I. L. Markov, and V. Bertacco, “Safe delay optimization for physical synthesis”, *ASP-DAC*, pp. 628-633, 2007.

[19] K.-H. Chang, I. Markov, and V. Bertacco, “Fixing design errors with counterexamples and resynthesis”, *ASP-DAC*, pp. 944-949, 2007.

[20] K.-H Chang, D. Papa, I. Markov, and V. Bertacco, “InVerS: an incremental verification system with circuit similarity metrics and error visualization”, *ISQED*, pp. 487-494, 2007.

[21] S. Chatterjee and R. Brayton, “A new incremental placement algorithm and its application to congestion-aware divisor extraction”, *ICCAD*, pp. 541-548, 2004.

[22] W. Chrabakh and R. Wolski, “GraDSAT: a parallel SAT solver for the grid”, *UCSB Comp. Sci. TR*, 2003.

[23] C. Chu and Y.-C. Wong, “Fast and Accurate Rectilinear Steiner Minimal Tree Algorithm for VLSI Design”, *ISPD*, pp. 28-35, 2005.
`http://class.ee.iastate.edu/cnchu/flute.html`

[24] M. Davis, G. Logemann, and D. Loveland, “A machine program for theorem proving”, *Comm. of ACM*, pp. 394-397, 1962.

[25] R. Dechter, K. Kask, E. Bin, and R. Emek, “Generating random solutions for constraint satisfaction problems”, *AAAI*, pp. 15-21, 2002.

[26] G. DeMicheli and M. Damiani, “Synthesis and optimization of digital circuits”, *McGraw-Hill*, 1994.

[27] N. Dershowitz, Z. Hanna, and A. Nadel, “Towards a better understanding of the functionality of conflict-driven SAT solver”, *SAT*, pp. 287-293, 2007.

[28] N. Een and A. Biere, “Effective Preprocessing in SAT through Variable and Clause Elimination”, *SAT*, pp. 61-75, 2005.

[29] N. Een and N. Sorensson, “An extensible SAT-solver”, *SAT*, pp. 502-518, 2003. `http://www.cs.chalmers.se/~een/Satzoo`

[30] W. C. Elmore, “The transient response of damped linear network with particular regard to wideband amplifiers”, *J. Appl. Phys*, pp. 55-63, 1948.

[31] C. Fiducia and R. Mattheyses, “A linear-time heuristic for improving network partitions”, *DAC*, pp. 175-181, 1982.

[32] Z. Fu, Y. Yu, and S. Malik, “Considering circuit observability don’t cares in cnf satisfiability”, *DATE*, pp. 1108-1113, 2005.

[33] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[34] E. Goldberg, M. Prasad, and R. Brayton, “Using SAT for combinational equivalence checking”, *DATE*, pp. 114-121, 2001.

[35] C. Gomes, W. Hoeve, A. Sabharwal, and B. Selman, “Counting CSP solutions using generalized xor constraints”, *AAAI*, pp. 204-209, 2007.

[36] C. Gomes, A. Sabharwal, and B. Selman, “Near-uniform sampling of combinatorial spaces using xor constraints”, *NIPS*, pp. 481-488, 2006.

[37] C. Gomes, A. Sabharwal, and B. Selman, “Model counting: a new strategy for obtaining good bounds”, *AAAI*, pp. 54-61, 2006.

[38] C. Gomes and B. Selman, “Algorithm portfolios”, *AI*, pp. 43-62, 2001.

[39] C. Gomes, B. Selman, K. McAloon, and C. Tretkoff, “Randomization in backtrack search: exploiting heavy-tailed profiles for solving hard scheduling problems”, *AIPS*, pp. 208-213, 1998.

[40] W. Gosti, A. Narayan, R. Brayton, and A. Sangiovanni-Vincentelli, “Wireplanning in logic synthesis”, *ICCAD*, pp. 26-33, 1998.

[41] W. Gosti, S. Khatri, and A. Sangiovanni-Vincentelli, “Addressing the timing closure problem by integrating logic optimization and placement”, *ICCAD*, pp. 224-231, 2001.

[42] M. Hrkic, J. Lillis, and G. Beraudo, “An approach to placement-coupled logic replication”, *DAC*, pp. 711-716, 2004.

[43] M. Hrkic and J. Lillis, “S-Tree: a technique for buffered routing tree synthesis”, *DAC*, pp. 578-583, 2002.

[44] W. Jordan, “Towards efficient sampling: exploiting random walk strategies”, *AAAI*, pp. 670-676, 2004.

[45] L. Kannan, P. Suaris, and H. Fang, “A methodology and algorithms for post-placement delay optimization”, *DAC*, pp. 327-332, 1994.

[46] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, “Multilevel hypergraph partitioning: applications in VLSI domain”, *TVLSI*, pp. 69-79, 1999.

[47] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, “Robust Boolean reasoning for equivalence checking and functional property verification”, *TCAD*, pp. 1377-1394, 2002.

[48] V. Kravets and K. Sakallah, “Resynthesis of multi-level circuits under tight constraints using symbolic optimization”, *ICCAD*, pp. 687-693, 2002.

[49] S. Krishnaswamy, S. Plaza, I. Markov, and J. Hayes, “Reliability-aware Synthesis using Logic Simulation”, *ICCAD*, pp. 149-154, 2007.

[50] F. Krohm, A. Kuehlmann, and A. Mets, “The use of random simulation in formal verification”, *ICCD*, pp. 371-376, 1996.

[51] H. Hoos and T. Stutzl, “SATLIB: an online resource for research on SAT”, *SAT*, pp. 283-292, 2000.

[52] Y. Hu, V. Shih, R. Majumdar, and L. He, “Exploiting symmetry in SAT-based Boolean matching for heterogeneous FPGA technology mapping”, *ICCAD*, pp. 350-353, 2007.

[53] Y.-Min. Jiang, A Krstic, K.-Ting Cheng, and M. Marek-Sadowska, “Post-layout Logic Restructuring For Performance Optimization”, *DAC*, pp. 662-665, 1997.

[54] H. Lee and D. Ha, “On the generation of test patterns for combinational circuits”, *TR No. 12-93*, Dept. of Electrical Eng., Virginia Polytechnic Inst.

[55] M. Lewis, T. Schubert, and B. Becker, “Multithreaded SAT solving”, *ASP-DAC*, pp. 926-932, 2007.

[56] C. Li, C-K. Koh, and P. H. Madden, “Floorplan management: incremental placement for gate sizing and buffer insertion”, *ASP-DAC*, pp. 349-354, 2005.

[57] F. Lu, L.-C. Wang, K.-T. Cheng, J. Moondanos, and Z. Hanna, “A signal correlation guided circuit-SAT solver”, *J. UCS 10(12)*, pp. 1629-1654, 2004.

[58] P. Manolios and Y. Zhang, “Implementing survey propogation on graphics processing units”, *SAT*, pp. 311-324, 2006.

[59] A. Mishchenko, S. Chatterjee, R. Jiang, and R. Brayton, “FRAIGs: a unifying representation for logic synthesis and verification”, *ERL TR*, Berkeley, 2005. http://www.eecs.berkeley.edu/∼alanmi/publications/

[60] A. Mishchenko, J. Zhang, S. Sinha, J. Burch, R. Brayton, and M. Chrzanowska-Jeske, “Using simulation and satisfiability to compute flexibilities in Boolean networks”, *TCAD*, pp. 743-755, 2006.

[61] A. Mishchenko and R. Brayton, “SAT-based complete don’t care computation for network optimization”, *DATE*, pp. 412-417, 2005.

[62] A. Mishchenko, S. Chatterjee, and R. Brayton, “DAG-aware AIG rewriting: a fresh look at combinational logic synthesis”, *DAC*, pp. 532-536, 2006.

[63] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, “Improvements to combinational equivalence checking”, *ICCAD*, pp. 532-536, 2006.

[64] G. Moore, “Cramming more components onto integrated chips”, *Electronics Magazine*, Vol. 38, No. 8, 1965.

[65] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Chaff: engineering an efficient SAT solver”, *DAC*, pp. 530-535, 2001.

[66] F. Okushi, “Parallel cooperative propositional theorem proving”, *Annals of Mathematics and AI*, pp. 59-85, 1999.

[67] R. Otten and R. Brayton, “Planning for performance”, *DAC*, pp. 122-127, 1998.

[68] M. Pedram and N. Bhat, “Layout driven logic restructuring/decomposition”, *ICCAD*, pp. 134-137, 1991.

[69] S. Plaza, K.-H Chang, I. Markov, and V. Bertacco, “Node mergers in the presence of don’t cares”, *ASP-DAC*, pp. 414-419, 2006.

[70] S. Plaza and V. Bertacco, “STACCATO: disjoint support decompositions from BDDs through symbolic kernels”, *ASP-DAC*, pp. 276-279, 2005.

[71] N. Saluja and S. Khatri, “A robust algorithm for approximate compatible observability don’t care computation”, *DAC*, pp. 422-427, 2004.

[72] H. Savoj and R. Brayton, “The use of observability and external don’t-cares for the simplification of multi-level networks”, *DAC*, pp. 297-301, 1990.

[73] P. Saxena, N. Menezes, P. Cocchini, and D. Kirkpatrick, “Repeater scaling and its impact on CAD”, *TCAD*, pp. 451-463, 2004.

[74] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli, “SIS: a system for sequential circuit synthesis”, *ERL TR*, Berkeley, 1992.

[75] C. Shannon, “A Mathematical Theory of Communication”, *Bell System Technical Journal*, Vol. 27, pp. 379-423, 623-656, 1948.

[76] M. Sipser, “Introduction to the theory of computation, second edition”, *Course Technology*, 2005.

[77] S. Sinha, A. Mishchenko, and R. Brayton, “Topologically constrained logic synthesis”, *ICCAD*, pp. 679-686, 2002.

[78] G. Stenz, B. Riess, B. Rohfleisch, and F. Johannes, “Timing driven placement in interaction with netlist transformations”, *ISPD*, pp. 36-41, 1997.

[79] S. Shyam and V. Bertacco, “Distance-guided hybrid verification with GUIDO”, *DATE*, pp. 1211-1216, 2006.

[80] J. Marques-Silva and K. Sakallah, “GRASP: A search algorithm for propositional satisfiability”, *IEEE Trans. Comp*, pp. 506-521, 1999.

[81] L. Valiant and V. Vazirani, “NP is as easy as detecting unique solutions”, *Theor. Comput. Sci.*, pp. 85-93, 1986.

[82] L.P.P.P van Ginneken, “Buffer placement in distributed RC-tree networks for minimal Elmore delay”, *ISCAS*, pp. 865-868, 1990.

[83] I. Wagner, V. Bertacco, T. Austin, “StressTest: an automatic approach to test generation via activity monitors”, *DAC*, pp. 783-788, 2005.

[84] J. Werber, D. Rautenbach, and C. Szegedy, “Timing optimization by restructuring long combinatorial paths”, *ICCAD*, pp. 536-543, 2007.

[85] R. Williams, C. Gomes, and B. Selman, “Backdoors to typical case complexity”, *IJCAI*, 2003.

[86] J. Yuan, K. Albin, A. Aziz, and Carl Pixley, “Simplifying Boolean constraint solving for random simulation-vector generation”, *IEEE TCAD*, pp. 412-420, 2004.

[87] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown, “SATzilla-07: the design and analysis of an algorithm portfolio for SAT”, *CP*, pp. 712-727, 2007.

[88] H. Zhang, “SATO: an efficient propositional prover”, *CADE*, pp. 272-275, 1997.

[89] H. Zhang, M.P. Bonacina, and J. Hsiang, “PSATO: a distributed propositional prover and its application to quasigroup problems”, *J. of Symb. Comp.*, pp. 1-18, 1996.

[90] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik, “Efficient conflict driven learning in Boolean satisfiability”, *ICCAD*, pp. 279-285, 2001.

[91] Y. Zhao, M. Moskewicz, C. Madigan, and S. Malik, “Accelerating Boolean satisfiability through application specific processing”, *ISSS*, pp. 244-249, 2001.

[92] P. Zhong, M. Martonosi, P. Ashar, and S. Malik, “Using configurable computing to accelerate Boolean satisfiability”, *TCAD*, pp. 861-868, 1999.

[93] S. Yamashita, H. Sawada, and A. Nagoya, “ SPFD: a new method to express functional flexibility”, *TCAD*, pp. 840-849, 2000.

[94] Y.-S. Yang, S. Sinha, A. Veneris, and R. Brayton, “Automating logic rectification by approximate SPFDs”, *ASP-DAC*, pp. 402-407, 2007.

[95] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli, “SAT sweeping with local observability don’t cares”, *DAC*, pp. 229-234, 2006.

[96] “Constrained-random test generation and functional coverage with Vera”, *TR, Synopsys, Inc*, Feb, 2003.

[97] Specman elite — testbench automation, 2004. `http://www.verisity.com/products/specman.html`

[98] Berkeley Logic Synthesis and Verification Group, “ABC: a system for sequential synthesis and verification”. `http://www.eecs.berkeley.edu/~alanmi/abc/`

[99] AMD, “High performance AMD Phenom X4 processors lead the charge to HD desktop gaming and video”, 2008.

[100] Intel, “FDIV replacement program: statistical analysis of floating point flaw”, *White Paper*, 1994

[101] The International Technology Roadmap for Semiconductors, 2005 Edition, *ITRS*.

[102] IWLS OpenCore Benchmarks. `http://iwls.org/iwls2005/benchmarks.html`

[103] Cadence Encounter RTL Compiler. `http://www.cadence.com`

[104] Synopsys DesignCompiler. `http://www.synopsys.com`

[105] UMICH Physical Design Tools. `http://vlsicad.eecs.umich.edu/BK/PDtools/`