STATISTICAL GUARANTEES OF PERFORMANCE FOR RTL DESIGNS

BY

JAYANAND ASOK KUMAR

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

ABSTRACT

Traditional hardware verification is a non-probabilistic process that verifies the adherence of a design to a non-probabilistic performance specification. However, adaptive techniques like voltage and frequency scaling, process variations due to shrinking chip geometries and input variations for accommodating "better-than-worst-case" design contribute significantly to the stochastic nature of contemporary chips. Therefore, it is desirable to incorporate probabilistic reasoning in the verification paradigm. Moreover, if such probabilistic verification can be applied at the register transfer level (RTL), it would facilitate better choices early in the hardware design cycle.

In this thesis, we introduce the SHARPE (Statistical High-level Analysis and Rigorous Performance Estimation) methodology. Our SHARPE methodology is a rigorous, systematic approach to verify design correctness in RTL in the presence of variations. We construct macromodels to transfer low-level hardware information, such as timing, to the higher RTL. We treat the RTL source code as a program and use static program analysis techniques to propagate signal probabilities. We combine the information obtained from both static analysis and the macromodels and model the RTL design as a statistical entity. We then employ formal probabilistic analysis on this probabilistic RTL model in order to compute the required probabilistic metric. The use of formal analysis makes our methodology high in confidence as opposed to simulation-based methods.

Our SHARPE methodology can be applied to both combinational and sequential designs. In combinational designs, formal probabilistic analysis is performed by exhaustively testing all possible input vectors. In sequential designs, we represent the probabilistic RTL models as discrete time Markov chains (DTMCs) that are then checked formally for probabilistic invariants using PRISM, a probabilistic model checker. When posed with a query, PRISM performs formal probabilistic analysis by exploring all possible transitions of the DTMC model.

Formal probabilistic analysis is known to be infeasible for large RTL designs. We improve the scalability of our SHARPE methodology by using sound symmetry reductions, automatic compositional reasoning and value-based interval abstractions in the design source code. For very large RTL designs, we employ statistical model checking, a scalable simulation-based alternative to probabilistic model checking. Since simulation-based analysis is not exhaustive, verification by statistical model checking is inexact. However, the statistical model checker guarantees the verification results to be within specified bounds of error.

In this thesis, we demonstrate several applications of our SHARPE methodology. We first show how our SHARPE methodology can provide statistical timing estimates in RTL in the presence of either input variations or process variations. We then modify the macromodels used by our methodology and use them to obtain RTL estimates of aging resulting from negative bias temperature instability (NBTI) effects. We describe how our methodology can be extended to compute bit error rates (BER) that are commonly-used performance metrics for RTL designs of wireless communication systems. For RTL of large multicore designs, we employ statistical model checking to provide guarantees regarding the performance of dynamic power management schemes. We perform our experiments on both benchmark RTL designs and real-world RTL designs.

To my family and friends, for their love and support

ACKNOWLEDGMENTS

At long last, after nearly seven glorious years at Illinois, I am ready to bid adieu. In these formative years of my life, several people have influenced my decisions for the better. I would like to use this space to express my heartfelt gratitude to all of them.

First, I would like to thank my adviser, Prof. Shobha Vasudevan. I got introduced to Shobha at a critical juncture in my graduate life when I was contemplating leaving with just a master's degree. Shobha rekindled my passion for research and inspired me to pursue a PhD degree with her. Over the course of my PhD, my interactions with her have helped me mature into an able and confident researcher. Her infectious enthusiasm spurred me onto greater heights than I thought were possible for me. I am certain that the lessons I learned from her would help me succeed in the next stages of my career. Of all people, I owe her the most for the completion of this thesis.

I would like to thank my dissertation committee members, Prof. David Nicol, Prof. Bill Sanders, and Prof. Josep Torrellas for agreeing to be part of my PhD exam process. Their comments and suggestions have proved invaluable in the shaping of this thesis.

I am sincerely grateful to Ken Butler (Texas Instruments) and Prof. Sachin Sapatnekar (University of Minnesota) for collaborating with me in my work on aging analysis. This work won the Silver Medal at the ACM Student Research Competition held at DAC, 2011. If not for Ken's and Sachin's expertise in the area of aging analysis and their regular feedback, I would not have been able to develop my work to a point where it became worthy of such recognition.

I would like to thank my labmates, Lingyi Liu, Viraj Athavale, Adel Ahmadyan, Sam Hertz, Parth Sagdeo, David Sheridan and Chen-Hsuan Lin for helping provide a wonderful work environment. My research discussions with them, even when they were for casually bouncing ideas back and forth, have greatly furthered my understanding of hardware verification. I would like to specially thank Lingyi, Viraj and Adel who were collaborators in some parts of my research.

My PhD life would have been fairly dull, to say the least, if not for my wonderful circle of friends at Illinois. They were my pillars of support during the rare "ups" and frequent "downs" that come with life as a graduate student. The outings, game sessions, tennis matches and movie nights lent some semblance of sanity during crazy stretches of work. I will not mention any names here. You know who you are.

Last but not the least, I thank my mom, dad, sister, brother-in-law and niece for loving me unconditionally despite my frequently being absent from my duties as a son, brother and uncle. I met my fiancèe, Meenakshy, only towards the very end of my PhD. However, at the advice of people far more worldly-wise than me, I decided that it is prudent to acknowledge her as well for the completion of this thesis.

All the choices that I make are centered around the well-being of my family. It is from their love and happiness that I derive the meaning for my life. I dedicate this thesis to them.

Now, at the end of my PhD, I think back to my very first meeting with Shobha. During the meeting, I expressed my doubts regarding my ability to complete a PhD. Among her words of encouragement was a famous quote. I believe it is only fitting that this quote finds a place in my thesis.

"Every battle is won or lost before it
is ever fought."
- Sun Tzu, *The Art of War*

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Motivation

1.1.1 Electronics in daily life

With the advent of electronic devices in the twentieth century, the data processing power of computing machinery has increased exponentially resulting in several significant advances in the scientific community. Vacuum tubes were the electronic building blocks of the first generation of computers in the 1930s. These computers were behemoths that occupied entire rooms. Moreover, the costs of constructing and maintaining these computers were astronomical. Therefore, access to such computing power was limited to small groups of people and was beyond the reach of an average individual.

With the invention of transistors in 1947 and integrated circuits (ICs) in 1958, electronic devices became faster, smaller and more efficient, and could be produced more reliably. Advances in semiconductor manufacturing technology enabled mass production of such devices which significantly brought down the production costs and gave rise to a diverse range of affordable consumer electronic devices. Electronic computing devices ceased to be items of luxury and became more commonplace in daily life.

Wireless internet is becoming ubiquitous and is rapidly replacing the need for a wired internet connection. As a result, the electronics market is driven to produce powerful computing devices that are portable. Fixed desktop workstations have given way to portable personal computers such as laptops and tablets. With the increased bandwidth afforded by 3G/4G mobile communication networks, a new generation of “smart” phones is becoming increasingly popular. These smartphones also double as portable personal computers, media players/recorders and GPS navigators.

In addition to supporting sophisticated functional features, portable electronics devices are required to be compact and have a long battery life. Therefore, in addition to performance, power, area and reliability are also important criteria in the design of present day electronic devices.

1.1.2 The hardware design cycle

The design of an electronic device typically starts with a complex document called a *specification*. The specification document systematically outlines the high-level intent for the design. This specification is then realized into physical hardware through a hardware design process that has been refined over several decades. Figure 1.1 broadly outlines the major steps in this process.

From the specification, a behavioral description of the design is generated. Currently this process has to be done manually. Normally the behavioral description is captured in a high-level language such as SystemVerilog or even C/C++. Once the behavioral description has been captured the designer has a formal unambiguous hardware description that can be executed and therefore tested against the specification. This is an important stage in the process because already at this early stage the design can be tested for misconceptions. If a major conceptual error is discovered late in the design process, it would result in significant time and cost over-runs. Therefore, in the modern design process, it is critical to bring in verification as early as possible.

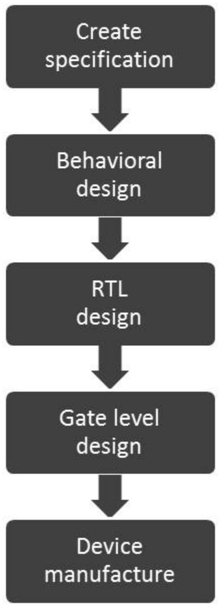


Figure 1.1: Block diagram depicting the various stages in the hardware design cycle.

Once the behavioral design is verified, the next step in the process is to construct the register transfer level (RTL) description. While RTL is still a level of abstraction above the gate level, the design blocks, combinational logic and registers can be clearly identified. In fact, 80% of a design's structure is typically fixed at the RTL stage. Transforming a design from a behavioral description into RTL is still a largely manual process. RTL descriptions are captured in a *hardware description language* (HDL) such as Verilog or VHDL.

RTL provides a bit-accurate and cycle-accurate description of the hardware design. Therefore, most of the functional verification is done at the RTL stage. HDL simulation is the most frequently used functional verification technique. The use of formal verification techniques is limited due to their restricted scalability. Property checking can also be performed to verify that the RTL implementation matches the specification.

After completing the RTL design, the next step is to transform this description into a gate-level description in the target technology. This process is known as *logic synthesis* and is now predominantly a fully automatic process, with several commercial software tools available in the market today. The constraints of the design such as clock period/frequency are listed in a file which is provided as input to the synthesis tool. The tool maps the RTL design and optimizes the resulting gate-level netlist in order to satisfy these constraints.

Again the functionality of the system is verified using the gate-level netlist. This process usually involves using timing data from the technology library. The purpose of the functional verification at this stage is to determine that the system operates correctly with the timing constraints of the physical hardware devices. Gate-level simulations take significantly longer than RTL simulations. Formal verification is used to check the synthesized netlist against the RTL design in order to ensure that synthesis has not altered the functionality. This verification process is referred to as *equivalence checking*.

The other operation that is performed on the gate-level description is *fault simulation*. The purpose of this is to create a set of tests which will be used to verify the actual device is free from production defects once it has been manufactured. These tests can bear little relationship to the functionality of the device. Fault simulations take much greater simulation effort than normal functional simulations.

Since the design is now mapped to actual hardware elements, the electrical and physical characteristics of the design can now be quantified. The synthesized netlist is analyzed to check whether the requirements for area, power and timing are satisfied.

The synthesized gate-level netlist is input to a *place-and-route* tool. This tool first determines the ideal physical locations for each hardware component on the device. The tool then determines the optimal wiring pattern for interconnecting these components. Typically, the delay and power consumption of these wires is non-negligible. Therefore, the netlist that is output by the place-and-route tool is once again analyzed to check that the power/timing requirements are still satisfied.

The final stage in the process, once the physical layout has been completed, is to manufacture the actual devices, after which the tests created by fault simulation can be run to verify the manufacturing process.

1.1.3 Sources of variations in hardware

There are several sources of variations that contribute to the stochastic nature of contemporary hardware [1]. Some of the dominant sources of variations are as follows.

- **Input variations:** Hardware designs perform computations on data that is input to it by the surrounding environment. Computation delay/timing in hardware is data-dependant. Variations in input patterns introduce randomness in data computation which in turn causes hardware behavior to vary statistically [2],[3],[4].

- **Process variations:** The manufacturing process introduces variations in feature sizes and electrical properties of devices. These process variations manifest as randomness in the timing of hardware computations. Aggressive scaling of technology has resulted in rapid shrinkage of chip geometries. Correspondingly, the variations introduced by the manufacturing process have grown significantly [5],[6],[7].

- **Transient faults:** Physical hardware is susceptible to impairments, commonly referred to as *faults*, that can modify hardware behavior. A transient fault is a type of fault temporarily injected in hardware due to exposure to cosmic radiation. The location and duration of transient faults are statistically distributed. Therefore, transient faults introduce randomness in hardware behavior [8],[9].

- **Adaptive strategies:** Several low-power designs use adaptive strategies for dynamically managing the power consumed by hardware. Strategies for dynamic power management (DPM) may involve either powering down idle blocks or scaling the operating voltage and frequency adaptively. Such adaptive strategies often employ statistical choices, and thereby introduce variations in hardware [10],[11],[12],[13].

1.1.4 Hardware performance in the presence of variations

Traditionally, hardware is designed to be error-free in the presence of timing variations. This is achieved by tuning the design to the worst-case timing scenario. Typically, such worst-case scenarios are distant from the statistical average-case scenario and occur with very low probability. Therefore, such worst-case design methodologies are pessimistic and are not sufficient to satisfy the performance, area and power requirements for present day systems.

In order to meet the requirements of present day systems, a “better-than-worst-case” [2],[3],[4],[6],[7] design methodology can be used. In this methodology, the design is tuned to the statistical average-case scenario instead of the infrequent worst-case scenario. Such methodologies allow for errors resulting from timing violations which can be corrected later with some penalty. The performance of “better-than-worst-case” designs depends on the probability of occurrence of these these errors.

In traditional error-free hardware designs, the performance is defined by the worst-case timing scenario regardless of the probability with which it occurs. However, in better-than-worst-case designs, the performance depends on the statistical distribution of timing and is therefore, a statistical metric.

In contexts different from timing variations, the notion of performance as a statistical metric is well known. In communication systems [14], the transmitted data bits are not always correctly decoded by the hardware in the receiver. The performance of such hardware, measured by a *bit error rate* (BER), is a statistical metric. In dynamic power management, the schemes are designed to meet soft probabilistic constraints. Therefore, the performance metrics for such schemes are also statistical in nature. In the presence of transient faults, reliable systems are designed to tolerate errors that occur with low probability. The reliability of such systems is commonly measured by a metric called *soft error rate* (SER) [9]. SER is also a statistical metric.

With growing sources of variations in hardware, the notion of performance as a statistical quantity needs to be widely adopted. In order to complement methodologies for design in the presence of variations, it is desirable to formulate a rigorous statistical analysis methodology that can provide guarantees regarding hardware performance.

1.1.5 Guaranteeing statistical performance of hardware

A register transfer level (RTL) design provides a behavioral description of the hardware components and is a common starting point for hardware designers. Later in the design flow, these RTL descriptions are mapped to actual hardware elements (e.g., gates and transistors) using a process called *synthesis*. With regard to performance estimation, RTL designs provide a good tradeoff between the hardware-unaware algorithms and the post-synthesis designs that provide an excess of detail. Designers iteratively analyze and revise the RTL designs in order to meet the performance requirements. However, this process is both time- and resource-intensive since the hardware is typically required to meet other criteria, such as area and power, as well. Therefore, it is desirable to have a methodology where performance estimation of RTL can be performed quickly and with a high degree of confidence.

Although there is variation-awareness in high-level synthesis techniques [7] as well as architecture-level power and performance analysis [15], this is not observed in RTL design verification methodologies. The behavioral levels of the design as in RTL lack awareness of the underlying stochastic nature of the circuit. Although there are multiple techniques for checking functional specifications, to the best of our knowledge, there are no well-known methods to verify the adherence of a design to a given statistical performance specification in RTL.

Traditional RTL verification is a non-probabilistic process that verifies design correctness using purely non-probabilistic finite state machine (FSM) models to represent the design. Such models are sufficient to express desired properties or invariants across the design that answer non-probabilistic (Yes/No) questions of the type: "Does the delay at an RTL block/ module signal ALWAYS meet a timing specification T?" However, this purely non-probabilistic model at the higher level does not address the multiple growing sources of variation at the lower levels of design.

In order to develop an alternate notion of design correctness, we require probabilistic verification of RTL designs that incorporate statistics from the lower levels. It is therefore necessary to establish *probabilistic invariants* [16] across the design. These invariants would attest certain properties of the design that hold true in the presence of the underlying variations. They can provide quantitative answers to questions of the type: "What is the probability of an RTL block/module signal meeting a timing specification?" or "What is the average delay of a signal in an RTL block/module?" This information, if available in RTL, can provide quick and early estimates of delay distribution of different blocks, thereby facilitating better design choices and reducing the overhead in post-synthesis simulation methods [17].

1.2 Thesis contributions

In this thesis, our contributions (Figure 1.2) can be broadly classified into the following categories.

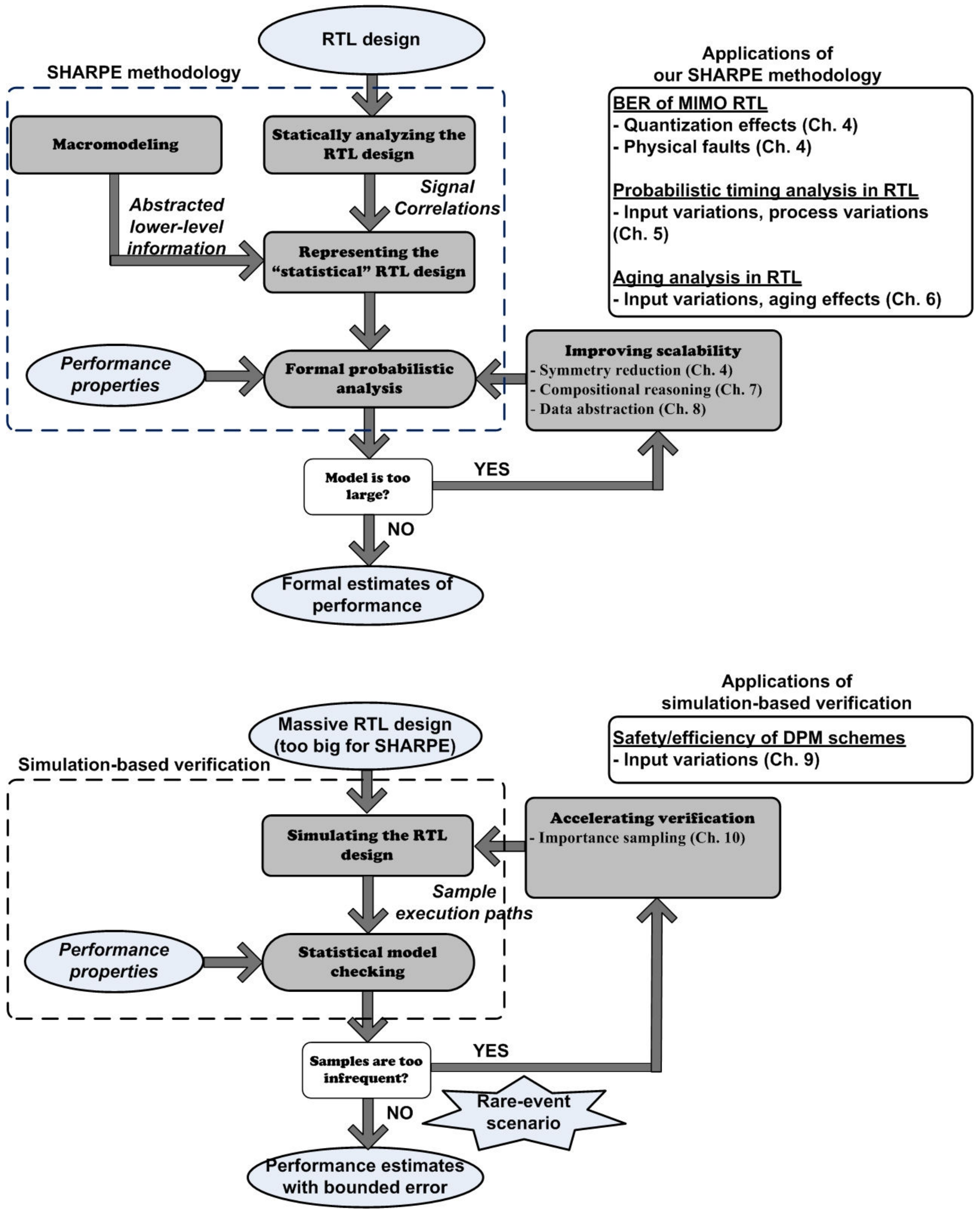

RTL design
SHARPE methodology
Macromodeling
Statically analyzing the RTL design
Signal Correlations
Abstracted lower-level information
Representing the "statistical" RTL design
Performance properties
Formal probabilistic analysis
Model is too large?
YES
NO
Formal estimates of performance
Applications of our SHARPE methodology
BER of MIMO RTL
- Quantization effects (Ch. 4)
- Physical faults (Ch. 4)
Probabilistic timing analysis in RTL
- Input variations, process variations (Ch. 5)
Aging analysis in RTL
- Input variations, aging effects (Ch. 6)
Improving scalability
- Symmetry reduction (Ch. 4)
- Compositional reasoning (Ch. 7)
- Data abstraction (Ch. 8)
Massive RTL design (too big for SHARPE)
Simulation-based verification
Simulating the RTL design
Sample execution paths
Performance properties
Statistical model checking
Samples are too infrequent?
YES
NO
Rare-event scenario
Performance estimates with bounded error
Applications of simulation-based verification
Safety/efficiency of DPM schemes
- Input variations (Ch. 9)
Accelerating verification
- Importance sampling (Ch. 10)

Figure 1.2: Block diagram depicting an outline of our contributions in this thesis. We list the thesis chapters in which each contribution is described.

1. **SHARPE methodology and applications:** We introduce SHARPE (**S**tatistical **H**igh Level **A**nalysis and **R**igorous **P**erformance **E**stimation), a methodology for formally estimating performance of RTL designs. With the SHARPE methodology,

we intend to introduce probabilistic reasoning in the RTL verification paradigm by incorporating empirical statistical evidence from the lower levels of design. Our SHARPE methodology provides a broad platform for rigorously analyzing various performance metrics by modeling different sources of variations in RTL. We present a detailed analysis of a set of applications of our SHARPE methodology (Section 1.3).

2. **Improving scalability of the SHARPE methodology:** Our SHARPE methodology employs formal probabilistic verification to provide rigorous statistical guarantees in RTL. Scalability issues of the formal engine restrict the feasibility of our methodology to small RTL designs. In this thesis, we devise a slew of novel techniques that we employ to mitigate the scalability issues faced by the SHARPE methodology (Section 1.5).

3. **Simulation-based statistical verification of RTL:** Although we present several scalability improvement techniques, large RTL designs such as those pertaining to multicore systems are still beyond the feasibility of the SHARPE methodology. For such massive RTL designs, we employ a simulation-based alternative to formal probabilistic verification. Simulation-based techniques are known to be inefficient for providing guarantees about low probability (infrequent) events. In such cases, we also demonstrate a technique to accelerate simulation-based verification (Section 1.6).

In the SHARPE methodology, we exhaustively analyze all possible probabilistic behavior of the RTL design. Therefore, the analysis is formal unlike in simulation-based statistical verification. We employ simulation-based verification on designs for which formal analysis using the SHARPE methodology is not feasible. In the rest of this thesis, we consider the formal SHARPE methodology separately from simulation-based statistical verification.

1.3 SHARPE methodology and applications

In this thesis, our primary contribution is the formulation of the SHARPE methodology. We believe that the potential range of applications for our methodology is vast. In this thesis, we analyze a few of these applications in depth.

1.3.1 Our SHARPE methodology

The steps in our SHARPE methodology (Figure 1.2) are as follows. We determine the correlations among the RTL signals by using static analysis of RTL source code. Such information can be used to propagate probability distributions throughout the RTL design. We construct macromodels [18],[19] in order to map the relevant effects from the lower physical levels (i.e., gates and transistors) to the higher RTL. We combine the information obtained from both static analysis and the macromodels and then employ formal probabilistic analysis in order to compute the required probabilistic invariant with respect to performance.

RTL designs are *abstract interpretations* [20] of the lower levels of hardware implementation. Therefore, macromodeling can be viewed as an *abstraction* that maps a design from a *concrete* lower-level description to an *abstract* RTL description. With the SHARPE methodology, we estimate performance of hardware designs by restricting analysis to the abstract domain. Typically, macromodeling involves a loss in lower-level information, and therefore our analysis is not exact. However, with clever mapping, we can provide quick estimates that are reasonably accurate (<10% error).

- *Our SHARPE methodology is of immense value since the performance estimates are obtained early in the design cycle, facilitating informed choices at the higher level itself.*

An important part of our methodology is the formal probabilistic analysis that we use to compute probabilistic invariants. When posed with a query, our formal analysis technique explores all possible statistical behaviors of the design.

- *The use of formal probabilistic verification makes SHARPE analysis high in confidence as opposed to simulation based methods for generating probability distributions.*

1.3.2 Formal probabilistic analysis for RTL

In this thesis, we consider the two broad classes of RTL designs:

- **Sequential designs:** In these designs, the value of the output depends not only on the present input values but also on the history of the input. Therefore, these designs are said to have "memory" or *state*.

- **Combinational designs:** In these designs, the value of the output is a function of only the present input values. In contrast to sequential designs, these are memoryless designs.

In combinational designs, all the interesting statistical behavior can be derived directly from the probability distribution of the input vectors. For such designs, formal probabilistic analysis computes the required probabilistic invariant by exhaustively checking all possible input vectors.

In sequential designs, the interesting behavior typically arises from sequences of states that the hardware transitions through. In order to employ formal probabilistic analysis, it is essential to model all such possible sequences of the design. In this thesis, we represent the statistical behavior of sequential designs using discrete time Markov chain models (DTMCs) [21].

- *In sequential designs, each state in the corresponding DTMC is a concatenation of the present input values and the internal state of the design.*

We shall describe these DTMCs in more detail in Section 3.4.1.

In our SHARPE methodology, the steps for performing formal probabilistic analysis on sequential designs are as follows. We convert a sequential RTL module into a DTMC [21] with the transition probabilities derived from the signal probability distributions and information obtained from the lower levels of implementation. We express the performance metrics of our interest as properties in probabilistic computational tree logic (pCTL) [22]. Finally, we use a probabilistic model checking engine, PRISM [23], to compute the required probabilistic invariants. Probabilistic model checking explores all possible DTMC sequences that are relevant to the property, and therefore the computation of the probabilistic invariant is high in confidence.

1.4 Applications of our SHARPE methodology

The SHARPE methodology can be used to compute invariants with respect to a wide range of statistical metrics. Our methodology would remain mostly the same, with the major difference being in the construction of the macromodels. In this thesis, we analyze a select few applications of the SHARPE methodology.

1.4.1 SHARPE for performance analysis of faulty MIMO RTL

Multiple-input multiple-output (MIMO) communication systems [24] are required to satisfy stringent *bit error rate* (BER) [14] performance specifications. BER is an average measure of the probability with which a transmitted data bit is decoded in error at the receiver. BER is inherently probabilistic in nature due to the randomness introduced by corruption of the data signals that reach the receiver. The data is further corrupted in the receiver blocks due to internal fixed-point quantization errors. BER estimates that are reasonably accurate can be obtained by simulating the MIMO RTL designs [25] over many cycles. However, this technique is time-consuming and incomplete. We use our SHARPE methodology for quick performance estimation of MIMO RTL by using probabilistic model checking over simulation-based techniques.

The hardware realizations (i.e., the post-synthesis designs) of RTL designs are subject to several forms of impairments, such as physical faults [8], that distort the BER performance of the system. We enhance the SHARPE analysis by including models for the occurrence of physical faults and determine whether the BER is still within acceptable limits. Therefore, we can use our SHARPE methodology to determine the reliability of MIMO RTL designs in the presence of "real" physical impairments.

- *To the best of our knowledge, ours is the first work that provides a unified framework which incorporates the effects of physical faults while formally analyzing BER performance from the RTL description.*

1.4.2 SHARPE for probabilistic timing analysis in RTL

Timing speculation techniques [2],[3] achieve "better-than-worst-case" [4] performance by tuning the circuits to meet the delay corresponding to the most frequently applied input patterns. The performance of speculation-based designs depends on the probability with which the timing constraint is satisfied. For such designs, it is not sufficient to determine just the worst-case delay. Instead, in order to estimate performance, it is important to determine the statistical distribution of delay.

We employ our SHARPE methodology and compute probabilistic delay distributions at the outputs of RTL blocks/modules. There exist probabilistic timing analysis techniques [26] that can be used to estimate these delay distributions. However, these techniques are entirely at the gate level. With SHARPE, we define and construct macromodels from the gate level in order to shift statistical timing information to the higher RTL. By interpreting the steps in SHARPE in a slightly different manner, we also consider process variations as the primary source of statistics.

- *SHARPE analysis can be used to provide early estimates of performance as compared to existing probabilistic timing analysis that is entirely at the gate level. To the best* *of our knowledge, our SHARPE methodology is the first approach for variation-conscious timing verification in RTL.*

1.4.3 Aging analysis in RTL using SHARPE

Negative bias temperature instability (NBTI) [27] in PMOS transistors has become a significant aging concern in the design of reliable digital circuits. Circuit simulations show that NBTI effects over a lifetime of 10 years can degrade delay by up to 10% which may potentially violate timing constraints and hence result in a circuit failure [28],[29]. Therefore, it is important to have a methodology that can predict aging effects at the time of design itself.

There are several timing analysis techniques at the transistor level and the gate level [30],[31],[32],[33],[34] that predict delay degradation of a circuit by computing signal probabilities of the circuit nodes. However, if such analyses are present at RTL, the signal probabilities that are computed would be representative of the actual usage statistics/workload of the design. As a consequence, RTL analysis provides a wider perspective of the effects of NBTI since the delay degradation of a small block is estimated in the context of a larger operating environment specified by the RTL.

We apply our SHARPE methodology to formally compute signal probability distributions in RTL. We use these distributions with appropriate macromodels derived from the gate level in order to provide RTL estimates of NBTI-induced delay degradation. Since the analysis is at the higher RTL, our SHARPE methodology is much faster compared to existing aging analysis that is purely at the lower levels.

- *We apply our SHARPE methodology to obtain early, RTL estimates of circuit aging. To the best of our knowledge, there is no technique that estimates delay degradation of designs in RTL.*

1.5 Improving the scalability of the SHARPE methodology

Formal probabilistic analysis is known to be intractable for large designs. For large combinational designs, the number of input vectors that need to be analyzed becomes prohibitively large. In the context of sequential designs, the probabilistic model checking tools that we employ are known to encounter the problem of state-space explosion [35]. We find that such restrictions on formal probabilistic analysis typically limit the SHARPE methodology to small hardware designs. In order to encourage the widespread adoption of our methodology, it is essential to have a set of techniques that can improve the scalability of formal probabilistic analysis.

For sequential designs that exhibit structural symmetry, we employ well-known symmetry reduction techniques [35],[36] and obtain smaller equivalent DTMC models. In this thesis, we also present a set of novel techniques that we employ in order to further improve the scalability of the SHARPE methodology.

1.5.1 Automatic compositional reasoning

We wish to verify that a design M satisfies a performance property Φ, denoted by $M \models \Phi$. For large M, this can be achieved by decomposing the problem into a set of simpler subproblems, $M_1 \models \Phi_1$ and $M_2 \models \Phi_2$. This forms the basis for compositional reasoning.

We statically analyze the RTL source code and structurally decompose M into smaller components M_1 and M_2. We employ an *assume-guarantee* form of reasoning [37] where we guarantee that $M_2 \models \Phi_2$ under the assumption that $M_1 \models \Phi_1$. We model the dependence between M_1 and M_2 by computing the conditional probability distribution of the shared RTL variables between M_1 and M_2. We provide an argument for the correctness of our technique. We demonstrate our technique on both combinational and sequential designs.

In the context of probabilistic model checking, several approaches for compositional reasoning have been proposed [37],[38],[39],[40],[41],[42]. However, all these techniques require a significant amount of manual effort to identify the separate components (M_1 and M_2) and their corresponding properties (Φ_1 and Φ_2).

- *We present a sound algorithm for automatic decomposition that uses static analysis of the RTL source code in order to obtain the smaller components and derive their corresponding properties. To the best of our knowledge, there exists no technique to automatically decompose hardware systems in order to make formal probabilistic analysis feasible.*

1.5.2 Abstractions in RTL source code

We are interested in performance properties Φ of the form $P[f_P(V) < T]$, where *exp* is a real valued function that is defined over the set of RTL variables V and T is a user-specified value. When we compute $M \models \Phi$, we are actually computing the probability of all input vectors where the valuation of $f_P(V)$ is less than T. Therefore, the relevant input vectors of M correspond to these states where the predicate $f_P(V) < T$ evaluates to a *true* value. Each input vector of M corresponds to a unique assignment of values to the input variables in the RTL design. Instead of considering all possible input vectors of M, it is sufficient to consider an abstract model M^A that comprises only the relevant input vectors of M.

We consider the relevant input vectors of M by constructing an abstraction whose inputs will be restricted to the set of values that correspond to relevant vectors. We perform our abstractions by deriving *value-based intervals* for the RTL input variables. In order to do this, we symbolically propagate the constraint $f_P(V) < T$ backwards through the RTL source code description. The model that we generate using the restricted input intervals is the abstract model M^A. We verify $M^A \models \Phi$ which is equivalent to verifying $M \models \Phi$. We demonstrate our approach on data-intensive RTL designs, which are mostly combinational in nature.

Our property-specific abstraction works at the source code level description of hardware designs, unlike most of the existing techniques for abstraction in probabilistic verification [43]. In [44], RTL designs are verified by restricting data to intervals that are imposed by the execution of the RTL program. Therefore, these intervals are not property-specific.

- *To the best of our knowledge, our value-based interval abstraction is the first property-specific abstraction for scaling formal probabilistic analysis of RTL designs. In the context of RTL verification, our abstraction is novel since it is intended for probabilistic verification. Moreover, our abstractions work at the source code level, unlike traditional abstractions in probabilistic verification.*

1.6 Simulation-based statistical verification of RTL

In sequential designs, probabilistic model checking uses numerical techniques to exhaustively analyze the DTMC models and compute the exact values of probabilistic invariants. However, these techniques require the state vector and the transition matrix of the DTMC to be constructed. Therefore, their scalability is limited by memory requirements of the probabilistic model checking tool. Although we present several techniques that considerably improve the scalability of formal probabilistic analysis for RTL designs, these techniques are not sufficient to make the SHARPE methodology feasible for massive designs such as RTL of multicore processors.

We provide statistical guarantees for large RTL designs by employing *statistical model checking* [45], a simulation-based alternative to probabilistic model checking. Statistical model checking tools operate on sample execution paths of the system that are drawn according to the statistics of the system. We extend the statistical model checking tool to work with sample paths that are generated by a commercial RTL simulator. The statistical model checking tool uses *hypothesis testing* [46] to infer whether these sample paths provide statistical evidence to decide if $M \models \Phi$ is TRUE.

Statistical model checking avoids the explicit construction and analysis of the DTMC transition matrix. Consequently, statistical model checking is highly scalable compared to probabilistic model checking. On the downside, the sample paths utilized by statistical model checking cover only a limited set of statistical behaviors. Therefore, unlike formal probabilistic verification, verification by statistical model checking is inexact. However, the statistical model checking tool utilized sufficient sample paths to guarantee that the error in the verification result is within specified bounds.

- *To the best of our knowledge, we are the first to apply statistical model checking to verify statistical properties in RTL designs.*

1.6.1 Verifying performance of dynamic power management schemes in RTL

Dynamic power management (DPM) schemes such as clock gating, power gating, dynamic voltage and frequency scaling (DVFS) are important strategies to save power in multicores [10],[11],[12],[13]. The performance of a DPM scheme depends on runtime statistics of the RTL design. For example, power gating is implemented by predicting the duration of an upcoming idle period based on recent history of inactivity. Overestimation of the idle period duration can affect the safety of a DPM scheme while underestimation can affect the efficiency. Safety and efficiency of a DPM scheme then are statistical properties that must be checked.

The probabilistic model checking engine used in the SHARPE methodology cannot be made to scale for verifying safety/efficiency properties of power gating in multicore RTL designs. Instead, we demonstrate that statistical model checking can be employed as a scalable alternative to verify these properties within specified bounds of error.

- *We demonstrate statistical model checking as a scalable technique to verify performance of dynamic power management schemes in large multicore RTL designs.*

1.6.2 Accelerating statistical model checking

Simulation-based estimation techniques are known to be inefficient and time-consuming in *rare-event scenarios*, i.e., scenarios that pertain to failures with very low probability ($<10^{-4}$). In such scenarios, a very large number of failing samples need to be generated in order to gather the statistical evidence required to estimate the failure rate with high confidence.

In rare-event scenarios, simulation-based estimation of the failure rate can be accelerated by increasing the frequency with which failing samples are generated. Such techniques are commonly referred to as *importance sampling* [47],[48],[49]. In importance sampling, the statistical distribution of the design is modified to frequently sample the region in which the failures are present. In order to preserve the correctness of the result, this statistical modification must be factored in during the estimation of the failure rate.

An SRAM cell is designed to be highly reliable (i.e., low failure rate) in the presence of process variations. In an SRAM cell, the effects of process variations are typically modeled using Gaussian random variables [50],[51]. Handling rare-event scenarios is known to be an importance challenge in the design and verification of reliable SRAM cells [50].

In this thesis, we describe an importance sampling approach for accelerating statistical model checking in rare-event scenarios of SRAM cells. In our approach, we sample the design by assuming uniform distributions for the variables in place of the original Gaussian distributions. We show that this modification of the distribution enables us to estimate the failure rate with high confidence using a relatively smaller number of samples.

- *We describe a practical approach for accelerating statistical model checking in rare-event scenarios of SRAM cells. We show, with empirical evidence and analytical bounds, that our approach provides significant speedup over regular statistical model checking.*

1.7 Defining statistical properties for hardware

In this thesis, we verify properties based on two types of probabilistic invariants for RTL:

- **Type A:** Probability that a state satisfies some condition.
- **Type B:** Probability that some sequence of states satisfies some condition on that sequence, such as "every state in a prefix subsequence starting from the initial state satisfies some condition C until a state S is encountered".

For sequential designs, we consider both the types of properties described above. For combinational designs, only the first type of property is meaningful. Since there is no notion of state for combinational designs, we interpret the **Type A** invariant as the "probability that input values satisfy some condition".

We now list the various statistical properties that we verify in this thesis. We shall later describe these properties in detail.

1.7.1 Performance analysis for MIMO RTL (sequential designs)

We verify performance properties related to the occurrence of bit errors in MIMO communication systems. We consider sequential designs such as the Viterbi decoder.

We compute the following error related metrics:

- Probability that no decoding error occurs in a sequence of T data bits (**Type B**).
- Probability of being in a state where the decoded bit is in error (**Type A**).
- Probability that number of errors occurring in T data bits is greater than a predetermined value (**Type B**).

1.7.2 Probabilistic timing analysis (combinational designs)

In "better-than-worst-case" designs, the circuit is tuned to the average-case delay instead of the worst-case delay. In such designs, computations that exceed the average-delay result in a timing violation. This may result in a computation error which can be later corrected with some penalty. Typically, in the control logic of hardware, timing violations are not allowed since the ensuing computing error can have disastrous consequences. However, if data computations are corrupted, the consequences are not so severe. Therefore, better-than-worst-case design typically is applied to the datapath which predominantly comprises of combinational logic.

In this thesis, we perform probabilistic timing analysis on combinational designs. We wish to identify

- Probability of input values such that the delay is less than a specified timing constraint (**Type A**).

1.7.3 Performance of dynamic power management schemes (sequential designs)

We verify the following safety and efficiency properties of power gating, a dynamic power management scheme:

- If a block is idle for K_{TO} cycles, then it will become active within a further K_{BE} cycles with probability less than p_S (**Type B**).
- If a block becomes idle, then it will become active again within K_{TO} cycles with probability less than $(1 - p_E)$ (**Type B**).
- IF a block becomes idle AND is predicted to be idle for atleast K_{BE} cycles THEN it will become active within K_{BE} cycles with probability less than p_S (**Type B**).
- IF a block becomes idle AND is predicted to be idle for less than K_{BE} cycles THEN it will become active within K_{BE} cycles with probability greater than p_E (**Type B**).

We verify these properties in the RTL of a large multicore processor design. The DTMC model for the multicore is too large to be constructed. Therefore, we verify the safety and efficiency properties using statistical model checking instead of probabilistic model checking. In statistical model checking, the DTMC model is not constructed.

1.8 Thesis organization

The remainder of this thesis is organized as follows.

In Chapter 2, we we present some definitions and background on the techniques used in the thesis.

In Chapter 3, we describe all the steps in our SHARPE methodology. We define the statistical models that we consider for both combinational and sequential designs. We also present the formal probabilistic analysis techniques that we employ for these designs.

In Chapter 4, we apply our SHARPE methodology in order to compute the BER performance of MIMO RTL designs. We first consider fault-free designs where the bit errors occur only due to data corruption resulting from external noise and quantization effects. We illustrate our technique on seminal components of a MIMO system, using a Viterbi decoder [52] and MIMO detector [25] as case studies. We enhance our BER performance estimation framework by including models for the occurrence of physical hardware faults. We consider the Viterbi decoder as a case study and rigorously analyze the vulnerability of BER performance to the presence of faults at various locations in the design. We present an error profiling technique where we define a pCTL property that can be used to systematically identify the broad reason for severe degradation in BER.

In Chapter 5, we apply our SHARPE methodology in order to compute probabilistic delay distributions at the outputs of RTL modules. We consider variations in input patterns as the primary source of statistics. We demonstrate that the RTL delay estimates provided by our methodology are within 10% of those obtained by simulating the gate-level designs over sufficiently large number of cycles. We also show how our methodology can be modified to consider process variations as the primary source of statistics.

In Chapter 6, we apply our SHARPE methodology to provide RTL estimates of delay degradation in the presence of aging effects. We use SHARPE to efficiently compute probability distributions at RTL. We construct appropriate macromodels that use these probability distributions to estimate the underlying delay degradation. We demonstrate that the RTL estimates provided by our approach closely track those obtained by simulating the gate-level designs over sufficiently large number of cycles.

In Chapter 7, we present an algorithm for automatic compositional reasoning for probabilistic model checking of hardware designs. We demonstrate the effectiveness of our approach by considering designs from Chapter 4 and Chapter 5 as case studies. For example, we are able to compute delay invariants for a 64-bit adder with over 10^{40} possible input vectors.

In Chapter 8, we present our technique for property-specific data abstraction that employs static analysis of RTL source code. We show that our technique is sound with respect to the properties of interest. We demonstrate the effectiveness of our technique using benchmark RTL designs as well as several modules of the H.264 decoder.

In Chapter 9, we describe how statistical model checking can be used to verify performance properties in large sequential RTL designs without constructing DTMC models for them. We demonstrate this approach by verifying safety and efficiency properties of a dynamic power management scheme, namely power gating. We consider the RTL of OpenSPARC, an industry-strength multicore processor.

In Chapter 10, we describe an approach to accelerate statistical model checking in rare-event scenarios for SRAM cells. We show that our approach provides up to around 10x speedup over regular statistical model checking.

In Chapter 11, we present work that is closely related to our contributions in this thesis.

In Chapter 12, we present a summary of the work and conclude this thesis.

CHAPTER 2

BACKGROUND

In this chapter, we present some definitions and background on the techniques used in the thesis.

2.1 Statistics of hardware variables

An n-bit variable v can be assigned any one of 2^n possible values with associated probabilities. Collectively, these define the *probability distribution* (called PMF) of v. The *expected* value of v is the probability-weighted sum of the possible values. The *joint probability* of a set of variables is the probability with which a set of values are collectively assigned to the variables. Two variables a and b are *independent* if the probability of an assignment to a is not affected by the assignment to b. Variables that are not independent are called *correlated variables*. Probability distributions that do not vary across time are said to be *stationary*.

We assume knowledge of the distribution of primary input variables and that they are independently distributed. We also assume stationary probability distributions for our inputs, and therefore for all variables in the system. A function of stationary variables is also stationary [21]. The values assigned to stationary variables in two different time steps are independent of each other.

In order to find the PMFs of a set of variables O from the PMFs of a set of variables I, we need to find the function f such that $O = f(I)$. We also need the joint probability distributions of the variables in I.

Definition 1. If the variables in I are independent, their joint PMF is simply a product of their individual PMFs.

2.2 Discrete time Markov chains

A *DTMC* is a state machine where each transition is associated with a probability. A DTMC *state* is a unique assignment of values to a set of variables called *state variables*. A DTMC is said to be *finite* if it has a finite number of states. A transition in a DTMC is a movement from one state to another, i.e., an assignment of a different set of values to the state variables. A detailed treatment of DTMCs can be found in [53].

A DTMC can be completely specified by using a triple (S, T_P, μ_0) where S is the set of state variables, T_P: $S \times S \to [0, 1]$ is the probabilistic state transition relation and μ_0 is the initial state. Each state μ of the DTMC corresponds to a unique assignment of values to the variables in S.

Definition 2. $Next(\mu)$ is the set of states reachable from μ in one transition.

A DTMC transition from one state μ_i to another state μ_j, denoted by $\mu_i \to \mu_j$, corresponds to a new assignment of values to the state variables. The probability of a transition $\mu \to \mu'$, denoted by p_{ij}, is equal to the probability with which the corresponding new values of the state variables are assigned. All possible state transitions allowed by $Next()$, along with their corresponding probabilities constitute T_P for the DTMC M.

Definition 3. Given a initial state μ_0, an *execution path* Λ of M is a sequence of transitions

$$\Lambda \ = \ \mu_0 \to \mu_1 \to \mu_2 \dots$$

where μ_i are the states of M. In a discrete-time system, these transitions occur in fixed discrete time steps.

Definition 4. The length of an execution path Λ, denoted by $length(\Lambda)$, is the number of transitions taken by Λ.

The probability of path Λ, denoted by $D(\Lambda)$, is the product of all the transition probabilities in Λ.

A DTMC is said to have attained a *steady state* when the probability of being in a state at any time step is independent of the both the time step and the initial state. All *finite*, *irreducible*, *aperiodic* DTMCs are guaranteed to reach a steady state [53].

Definition 5. A *state reward* is defined as a cost associated with being in a state of the DTMC.

2.3 Register transfer level (RTL) designs

Register transfer level (RTL) designs provide a behavioral description of hardware functionality. We consider RTL designs written in Verilog HDL. We view the RTL design as a Verilog program [54] on which we can perform static analysis techniques. In this thesis, we use RTL design interchangeably with Verilog program; similarly we use signal and variable interchangeably.

We consider the synthesizable subset of Verilog for our analysis. A Verilog program statement is a conditional (*if-else, case*) statement or an assignment.

Definition 6. If a variable is on the right-hand side (RHS) of an assignment to a variable v, it is an *operand*. The set of all the operands is called $RHS(v)$.

An example RTL code fragment is shown below:

```
module (I1,I2,I3,I4,O1,O2,O3)
input [9:0] I1,I2,I3,I4;
wire [9:0] Z1,Z2,Z3,Z4;
output [9:0] O1,O2,O3;
always @(posedge clk)
begin
Z1 <= I1; Z2 <= I2 - I3;
Z3 <= I3 + I4; Z4 <= I3 & I4;
O1 <= Z1 | Z2; O2 <= ~Z3; O3 <= Z4;
end
endmodule
```

A *module* in RTL corresponds to a hardware block that performs an independent function and has inputs and outputs defined. In the example RTL module shown above, $I1$, $I2$, $I3$ and $I4$ are the RTL inputs. $O1$, $O2$ and $O3$ are the RTL outputs and $Z1$, $Z2$, $Z3$ and $Z4$ can be viewed as temporary variables in the hardware system. All variables are of 10 bits and can therefore be assigned 1024 different numeric values.

The `always @(posedge clk)` blocks can be thought of as processes that are executed in parallel at every rising edge of the hardware clock signal which is considered as a time step. At any time step t, the `<=` operator evaluates the *right-hand side* (RHS) value and assigns it to the *left-hand side* (LHS) variable in the next step $t + 1$. The `&`, `|`, and `~` operators perform logical disjunction, conjunction and negation, respectively, in a bitwise manner on the operands.

An RTL variable v that can be assigned N values is defined to have size N, denoted by $|v|{=}N$. We refer to the probability distribution of v as the PMF of v. We define a set of RTL variables V to be independent if all the variables in V are mutually independent (Definition 1). The joint PMF of an independent set V is simply a product of the individual PMFs of the variables $v \in V$.

Let V be the set of all RTL variables in the source code and $I \subset V$ be the set of RTL input variables. It is reasonable to expect that all variables $v \in V$ are functions of some of the input variables $i \in I$. Moreover, the value of a sequential variable in any clock cycle may depend on its value from some previous clock cycle.

Definition 7. The *support* of v, denoted by $Sup(v)$, is the set of all input variables and sequential variables that can affect the value of v. For combinational designs $Sup(v)$ is simply a subset of I, i.e., $Sup(v) \subseteq I$. The support of a signal is independent if all the variables in the support are independent.

Definition 8. For a signal v, the *signal function* $f(Sup(v))$ is the symbolic expression that includes the variables $Sup(v)$, or the "formula" that corresponds to its evaluation. $f(I)$ may comprise Boolean or arithmetic operators that are allowed in the system.

In order to find the PMFs of a variable $v \in V$ from the PMFs of I, we need to find the signal function $f(Sup(v))$.

Definition 9. For a set of inputs I, an input vector to the design is a unique assignment of values to the variables I.

Two variables v_1 and v_2 are structurally independent if they do not share any common variables between their respective supports, i.e., $Sup(v_1) \cap Sup(v_2) = \mathbf{0}$. If v_1 and v_2 are structurally independent, their values are assigned by disjoint sets of input variables. Therefore, v_1 and v_2 are also statistically independent. In the rest of this thesis, we use the term "independence" to refer to both statistical and structural independence.

2.4 Statistical verification of DTMCs

Statistical systems such as cyber-physical systems (CPS), computer networks and even some hardware circuits [55] require probabilistic verification for properties pertaining to metrics such as reliability, safety, stability and performance. Probabilistic model checking [23] employs numerical analyses to provide statistical guarantees for such systems. Although exact, such analyses may become computationally intensive for very large systems.

Statistical model checking [45],[56],[57] is an inexact alternative strategy that provides guarantees by simulating several sample paths of the system. Statistical model checking is highly scalable and can be used to rigourously analyze massive systems that cannot be verified exactly using probabilistic model checking.

We wish to verify that a DTMC M satisfies probabilistic properties ϕ of the form

$$\phi = P_{\geq\theta}(\rho)^{\dagger} \tag{2.1}$$

where θ is a user-specified constraint on the probability that ρ is TRUE in M. Let p denote the actual probability that ρ is TRUE in the execution paths of M (Definition 3). If $p \geq \theta$, then M satisfies ϕ, denoted by $M \models \phi$.

Since p is unknown, the crux of probabilistic verification is in checking $p \geq \theta$ efficiently.

2.4.1 Statistical model checking

We now present a brief background on statistical model checking [45] in the context of DTMCs.

Conventional probabilistic model checking [23] uses numerical techniques to compute the exact value of p and then compares it against θ. However, these techniques require the DTMC model to be constructed, and therefore their scalability is limited by memory requirements.

Statistical model checking tools operate on sample execution paths (Definition 3) of the system that are drawn according to the statistics of the system. Statistical model checking uses *hypothesis testing* to infer whether these sample paths provide statistical evidence to decide if $M \models \phi$ is TRUE.

Let H_0 represent the hypothesis that $p \geq \theta$, i.e., $M \models \phi$. Let H_1 represent the alternate hypothesis, i.e., $M \not\models \phi$. The statistical model checking algorithm checks whether ρ (in Equation 2.1) is TRUE or FALSE in each sample path. Since the sample paths are generated based on the statistical distribution of the system, the fraction of sample paths in which ρ is TRUE can then be used to determine whether H_0 or H_1 should be "accepted" as the valid hypothesis for the system. We refer the reader to [45] for further details of the hypothesis testing algorithm.

Typically, an indifference region of width $2.\delta$ is specified. If p lies within the range $[\theta - \delta,\ \theta + \delta]$, then it is correct to accept either H_0 or H_1 as the valid hypothesis. Therefore, H_0 is redefined as $p \geq \theta + \delta$ and H_1 is redefined as $p \leq \theta - \delta$.

†In place of $\geq$, other relational operators can be used as well.

Since all possible execution paths of M are not analyzed, the verification results provided by statistical model checking are not exact. However, with a sufficient number of sample paths, the likelihood of error from hypothesis testing is bounded as

$$\begin{aligned} &P[(M \models \phi) \mid (M \not\models \phi) \text{ is claimed based on sample paths}] \leq \alpha \\ &P[(M \not\models \phi) \mid (M \models \phi) \text{ is claimed based on sample paths}] \leq \beta \end{aligned} \tag{2.2}$$

α and β are bounds for the likelihood of error in the verification result, and therefore represent the *verification accuracy.* α is guaranteed to be the maximum probability that statistical model checking verifies the property to be FALSE based on the sample paths when in fact the property is TRUE in the system. Similarly, β bounds the probability that a property is incorrectly verified to be TRUE. Tight error bounds (small α, β and δ) can be provided by using a sufficiently large number of sample paths [45].

Ymer [58], a statistical model checking tool, implements the sequential probability ratio test (SPRT) [45] algorithm for hypothesis testing. In SPRT, hypothesis testing is performed sequentially on each sample path. The SPRT algorithm terminates as soon as it discovers sufficient evidence to accept (or reject) a hypothesis within the specified error bounds. In [59], Ymer is shown to be significantly faster than statistical model checking tools that use other hypothesis testing algorithms. Therefore, we use Ymer as the statistical model checking tool for our experiments (Chapter 9).

CHAPTER 3

OUR SHARPE METHODOLOGY

3.1 Introduction

SHARPE is a CAD methodology for formally verifying statistical properties in RTL. SHARPE is an acronym for **S**tatistical **H**igh-level **A**nalysis and **R**igorous **P**erformance **E**stimation. We introduce with the SHARPE methodology, the notion of "variation-aware" design verification in RTL.

Our SHARPE methodology is broad in scope and can be applied to estimate a diverse range of statistical metrics by considering different sources of variations in hardware. In this chapter, we describe all the steps in our methodology by considering variations in input patterns to be the primary source of statistics.

3.1.1 Steps in our SHARPE methodology

A summary of the steps in the SHARPE methodology (Figure 3.1) are as follows.

- **Macromodeling:** We obtain macromodels [17],[18] to incorporate empirical statistical evidence from the lower levels of design into the higher RTL. We use these macromodels to restrict probabilistic analysis entirely to RTL. The macromodels that we derive depend on the statistical metric of interest as well as the source of variations. Macromodeling is typically a one-time characterization effort for a given technology library.
- **Source code static analysis:** We employ static analysis to traverse the RTL source code and determine the correlations among the RTL signals. We use these correlations to tie the statistics of an RTL signal to the PMFs of the primary inputs.
- **Formal probabilistic analysis:** In order to formally verify a statistical performance property, we exhaustively analyze all possible statistical behaviors in RTL. In combinational designs, we achieve this by checking all possible values that can

be assigned to the inputs. In sequential designs, we perform formal statistical verification by first representing an RTL module as a DTMC (Section 2.2) with the transition probabilities derived from input PMFs. We input these DTMCs into a probabilistic model checking engine, PRISM [23] to compute probabilistic invariants with respect to the performance property of interest.

Macromodels are but plugins that can be used with the SHARPE methodology. In this thesis, we apply our methodology to a diverse range of applications by using appropriate macromodels in each case. We derive macromodels for modeling hardware faults (Chapter 4), delay (Chapter 5) and aging effects (Chapter 6) in RTL. We shall later describe these macromodels in detail in the context of the corresponding application for the SHARPE methodology. In this chapter, we describe the remaining components of the SHARPE flow.

3.1.2 Chapter organization

The rest of this chapter is organized as follows. In Section 3.2, we describe our algorithm for statically analyzing the RTL source code. In Section 3.3, we describe the steps in the SHARPE methodology for verifying statistical properties of combinational designs. In Section 3.4, we describe the steps in our methodology for verifying statistical properties of sequential designs. We describe the DTMC models that we use for representing sequential designs. In Section 3.5, we outline how the probabilistic invariants computed by our methodology can be used to revise a hardware design at the early RTL stage.

3.2 Source code static analysis

We consider input variations as the primary source of statistics in hardware. Statistical properties for RTL are typically expressed in terms of a set of RTL variables V. In order to verify these properties, we must relate each variable $v \in V$ to the source of statistics $Sup(v)$ (Definition 7). We can achieve this by determining the signal function $f(Sup(v))$ (Definition 8).

We statically analyze the Verilog program [54] to determine the support $Sup(v)$ and signal function $f(Sup(v))$ for each variable v that is of interest with respect to the statistical property. Through static analysis, we can also identify correlated variables and propagate the PMFs from the input variables to other variables in the slice and the RTL module outputs.

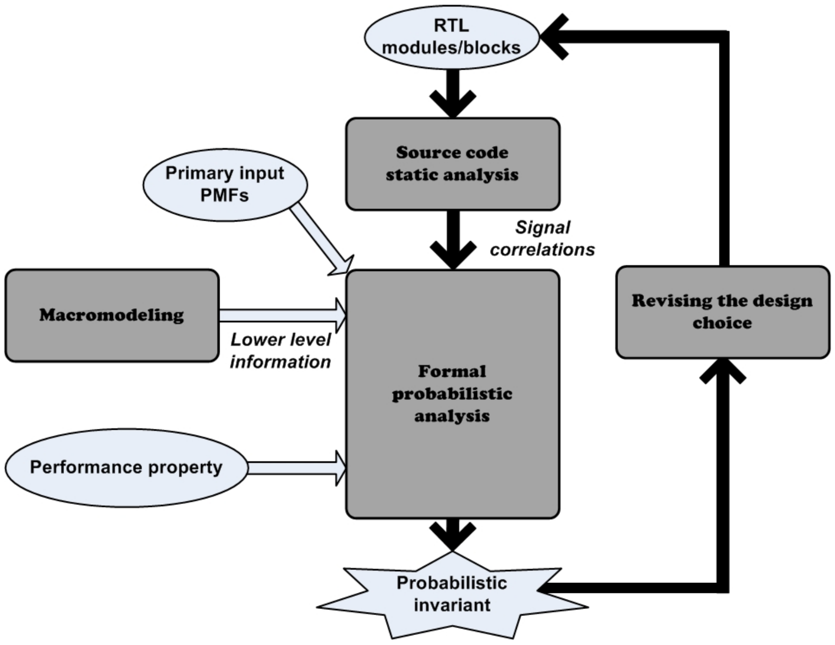


Figure 3.1: Block diagram depicting the steps in the SHARPE methodology for both combinational and sequential designs.

We statically traverse the source code for computing $Sup(v)$ and $f(Sup(v))$. For a Verilog statement where a value of v is assigned, $RHS(v)$ gives the set of variables that are included in the right-hand side of the assignment statement. These variables are annotated with a value $t-1$ to denote that we step the design backwards by a step. In programs, this "stepping back" refers to moving in the space of the program source code. The hardware interpretation of this is that the "previous cycle" values of variables are being obtained. This is a temporal step back, as opposed to the purely spatial one in software.

If all the variables in $RHS(v)$ are either primary inputs or sequential variables, we define $RHS(v)$ to be $Sup(v)$ and the analysis is complete. If not, we step the circuit backwards by one more step. Now each variable in $RHS(RHS(v))$ is annotated by $t-2$. If the union over the set of all variables tagged with $t-2$ comprises only either primary inputs or sequential variables, we define the union to be $Sup(v)$ and obtain the corresponding $f(Sup(v))$. The algorithm terminates when the support is obtained. If the support is independent, the joint PMF can be calculated as in Definition 1 and this can be used to compute the PMF of v.

The value of a sequential variable at time t will depend on its value at some time $t-j$. With static analysis, we identify such dependencies in order to identify all the sequential variables in the RTL source code. We also compute the signal functions for these sequential variables.

In order to be able to use PMFs of variables assigned in one time step over future time steps, we need to assume that these PMFs are stationary, i.e., they do not change over time (Section 2.1).

```
module (A,B,C,D,E,F)
input A,B,C;output D,E,F;
always @(posedge clk)
begin
D <= A ;E <= B & C;F <= D + E;
end
endmodule
```

In the code fragment shown above, let F be the variable of interest. Here, A,B and C are the primary inputs. D and E are the elements of $RHS(F)$. Since D and E are not primary inputs, we step back once more and determine that A, B and C are the elements of the union of $RHS(D)$ and $RHS(E)$. Therefore, we obtain $Sup(F)$=$\{A, B, C\}$ with F=$f(A, B, C)$=$A + (B\&C)$.

3.3 Formal probabilistic analysis for combinational designs

In combinational designs, all the properties of interest are broadly of the same type. This type of properties can be verified by computing the following probability.

- Probability that an expression $f_P(V)$ defined over RTL variables V satisfies some condition.

Here $f_P(V)$ may contain both arithmetic and Boolean operators. $f_P(V)$ and the condition it needs to satisfy are related to the statistical property and the macromodel that we consider. For example, $Delay < T$ is the condition that is specified in probabilistic timing analysis (Chapter 5), where $Delay$ is a real-valued function of RTL variables and T is a user-specified time constraint.

From static analysis, we obtain the signal function $f(Sup(v))$ for each variable $v \in V$. Let $f_P(f(Sup(V))$ be the expression obtained by substituting each occurrence of v in $f_P(V)$ with $f(Sup(v))$. The required probability is equal to the probability that $f_P(f(Sup(V))$ satisfies the condition specified in the property.

We compute the required probability as follows. We exhaustively analyze all possible input vectors (Definition 9) for the set of inputs $Sup(v)$. We use the PMFs of the inputs to compute the probability of occurrence of each input vector. We use the expression $f_P(f(Sup(V))$ to check whether an input vector satisfies the specified condition or not. We compute the required probability by summing the probabilities of all input vectors that satisfy the condition.

We use the above approach to propagate PMFs from the inputs to other RTL variables. Let v be the variable whose PMF we wish to compute. We use the above approach to compute the probability that the signal function $f(Sup(v)$ is equal to i, where i is a possible value that can be assigned to v. We repeat this for all i and obtain the PMF of v.

Definition 10. With respect to a property defined over variables V, the *size* of a combinational design is the number of possible input vectors for the set of inputs $Sup(V)$. We use the size of a combinational design as a metric for estimating the complexity of formal probability analysis.

3.4 Formal probabilistic analysis for sequential designs

Sequential designs are hardware designs that contain memory elements. Typically, memory elements are represented in RTL using register variables. The values stored in the memory elements determine the "state" of a sequential RTL design.

Formal probabilistic analysis for sequential RTL designs typically involves computing either one of the two following types of probabilistic invariants.

- **Type A:** Probability of being in an RTL state in which the expression $f_P(V)$ satisfies some condition.
- **Type B:** Probability that some sequence of RTL states satisfies some condition on that sequence, such as "every state in a prefix subsequence starting from the initial state satisfies some condition *C1* until a state that satisfies condition *C2* is encountered".

Sequential RTL designs can be modeled as finite state machines (FSMs) that transition from one state to another. Due to the sources of variations in hardware, each state transition is associated with a probability. Traditionally, FSMs that have been used to model all the non-probabilistic functionality of the RTL design. Although these FSM models are

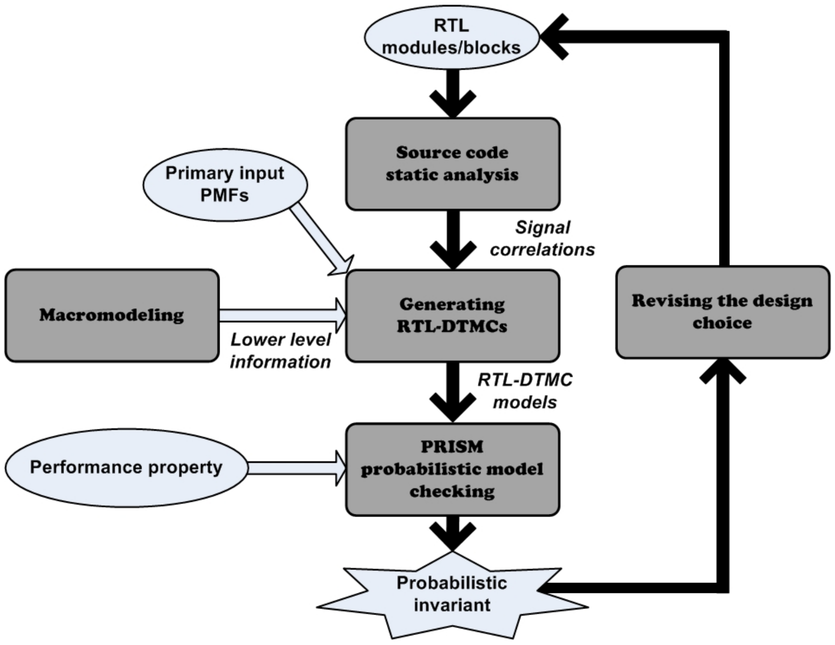


Figure 3.2: Block diagram depicting our SHARPE methodology for sequential designs. In this case, formal probabilistic analysis is performed by modeling the sequential design as a DTMC and then employing probabilistic model checking on the DTMC model.

sufficient for employing non-probabilistic formal verification in RTL, they cannot be used to provide any statistical guarantees. In order to employ formal probabilistic verification on sequential RTL designs, the states and their probabilistic transitions must first be represented.

In our SHARPE methodology, we use DTMCs (Section 2.2) to represent the statistical behavior of sequential RTL designs. DTMCs are FSMs in which each state transition is labeled with the corresponding probability. We perform formal probabilistic analysis of a sequential design (Figure 3.2) by employing probabilistic model checking on the corresponding DTMC model.

3.4.1 Modeling sequential RTL designs as DTMCs

We consider input variations to be the primary source of statistics. Therefore, the PMFs of the input variables determine the probability with which the hardware transitions from one state to another. We now describe the process of converting an RTL hardware system into a finite-state probabilistic system with respect to input variations.

We represent the sequential RTL design formally as a DTMC model, which we call an

RTL DTMC. Each state in the DTMC should model the state information of the sequential RTL design. Additionally, we require each DTMC state to also carry information regarding the values assigned to the input variables.

Definition 11. A state in an RTL DTMC is a concatenation of the present input values and the internal state of the sequential RTL design.

We construct the RTL DTMC model M for a variable v, with the support of v (Definition 7) being the state variables, denoted by S (Section 2.2). Therefore, $S = Sup(v)$. We define the initial state by setting the value of all state variables to 0.

In each hardware clock cycle, we assume that new values are assigned to the RTL inputs. Therefore, each hardware clock cycle corresponds to a time step in which new values may be assigned to all the RTL variables based on their respective signal functions (Definition 8). Each such time step corresponds to a DTMC transition from one state μ to another state μ' that corresponds to the new assignment of values to $Sup(v)$.

Let $S_I \subset S$ denote the set of input variables in $Sup(v)$. The probability of a transition to a new state μ' is equal to the probability with which the corresponding new values are assigned to S_I. Since all the input variables are assumed to be independent, we obtain the state transition probabilities by taking the product of the individual probabilities of all variables (Definition 1). If we do not assume independence for the inputs, we would use the specified joint PMF of the inputs. All such possible state transitions labeled with the corresponding probabilities constitute T_P for the DTMC M.

3.4.2 Generating RTL-DTMCs

Let V be the set of RTL variables over which the statistical properties are defined. The probabilistic behavior of V can be completely represented in terms of the PMFs of the support of each variable $v \in V$. We construct the RTL DTMC M using state variables S given by

$$S = \bigcup_{v \in V} Sup(v) \tag{3.1}$$

Definition 12. With respect to a property defined over variables V, the *size* of a sequential design is the number of states in the RTL DTMC model for V. We use the size of a sequential design as a metric for estimating the complexity of formal probability analysis.

In every state (Definition 11) of the RTL DTMC M, the value assigned to v can be determined by the signal function $f(Sup(v))$, which we obtain using static analysis (Section 3.2).

We compute the PMF of v as follows. The probability of $v{=}i$ is the probability of being in a state where $f(Sup(v)){=}i$, where i is one of the possible values that v can be assigned. We tag every state in the model where $f(Sup(v)){=}i$ is satisfied. In doing so, we create a state-based reward model (Definition 5) for the RTL-DTMC. We assign a reward of 1 to the states we want to tag, and 0 otherwise. The *expected* value of the reward at any time step thus translates to the probability that a tagged state is reached in that step. Repeating this for all i gives the complete PMF of v.

More generally, let $f_P(V)$ be an expression that is defined over variables V, as described in Section 3.3. We wish to compute the probability of being in a state where $f_P(V)$ satisfies some condition specified in the **Type A** property. We tag (i.e., set reward=1) all states where $f_P(V)$ satisfies the condition. Therefore, the expected reward at any time step represents the probability of being in a tagged state in that time step. This is equivalent to the probability that we wish to compute.

- *In principle, we could generate an RTL DTMC M by discarding the input variables S_I in $Sup(V)$ and using only the sequential variables in $Sup(v)$ as the state variables for M. This reduction in the set of state variables will result in a more compact state-space representation for M. However, since input values are now absent from the DTMC state, we would need to define a complex, statistical reward model.*

3.4.3 Probabilistic model checking of the RTL-DTMCs

The next step in our flow is to compute the probabilistic invariant in order to verify whether the RTL design satisfies the given statistical property. We achieve this by employing probabilistic model checking on the RTL DTMC M.

Probabilistic model checking is a formal verification technique for the analysis of stochastic systems. We use PRISM [60], a symbolic model checking tool that uses efficient algorithms and data structures based on binary decision diagrams (BDDs). BDDs allow for compact representation and efficient manipulation of DTMCs, increasing the scalability of this tool considerably.

In order to verify a **Type A** property, we would like to find the probability of being in a tagged state (reward=1), at the end of N transitions. This is equivalent to finding the expected value of the reward at the N^{th} time step. PRISM computes the expected value of a reward at the N^{th} time step by performing an exhaustive exploration of all the possible paths of length N. In contrast, simulation-based techniques typically explore a limited number of paths of length N, providing only an incomplete analysis.

Definition 13. The FSM model of a sequential design is said to be *strongly connected* if for any pair of FSM states μ_i and μ_j, there exists a sequence of input values that takes the design from μ_i to μ_j. In other words, any state μ_j of the FSM is "reachable" from any another state μ_i.

For a sequential design, the RTL DTMC model is just the FSM model with probabilities assigned to the state transitions (Section 3.4.1). If the FSM is strongly connected, the RTL DTMC is irreducible and may converge to a steady-state distribution (Section 2.2). Therefore, if the FSM of a sequential design is strongly connected, the PMFs of the all the variables in the design may converge to an equilibrium probability distribution.

A **Type A** property typically refers to the equilibrium PMFs of variables in an RTL design. We start the RTL DTMC model from a known initial state that we specify. For our experiments, we set N large enough such that the RTL DTMC reaches a steady state by the N^{th} time step. We find that the computed PMFs do not change significantly beyond this time step.

- *In this thesis, we employ our SHARPE methodology to verify a* **Type A** *property (steady-state) on a sequential design only if the design has a strongly connected FSM. We do not guarantee that our SHARPE methodology can compute steady-state probabilities for other types of sequential designs.*

3.5 Revising the design choice

The probabilistic timing invariants computed by the SHARPE methodology are provided as feedback to the RTL designer. These invariants are utilized to revise and if necessary, modify the RTL design. This process can be iterative until the statistical design requirement are satisfied. The use of macromodels precludes the need to synthesize the RTL design in each iteration. Therefore, our methodology avoids all the overheads (in resource and time) that are associated with synthesis. Additionally, the performance of different gate-level implementations of an RTL function can be explored by comparing the invariants obtained. Such information can be passed down to facilitate lower-level design.

3.6 Chapter summary

In this chapter, we have described SHARPE, a comprehensive methodology for rigorously analyzing statistical performance at the higher levels of hardware design, namely RTL. The SHARPE methodology can be viewed as an integrated solution for formal statistical verification in RTL by considering different sources of statistics in hardware. The use of formal probabilistic analysis make our analysis rigorous as compared to simulation-based techniques that analyze only a limited set of possible behaviors.

CHAPTER 4

SHARPE FOR PERFORMANCE ANALYSIS OF FAULTY MIMO RTL

4.1 Introduction

In this chapter, we describe how our SHARPE methodology (Section 3.4) can be applied to obtain RTL estimates of performance for multiple-input multiple-output (MIMO) communication systems [24]. The designs and properties that we consider are sequential in nature.

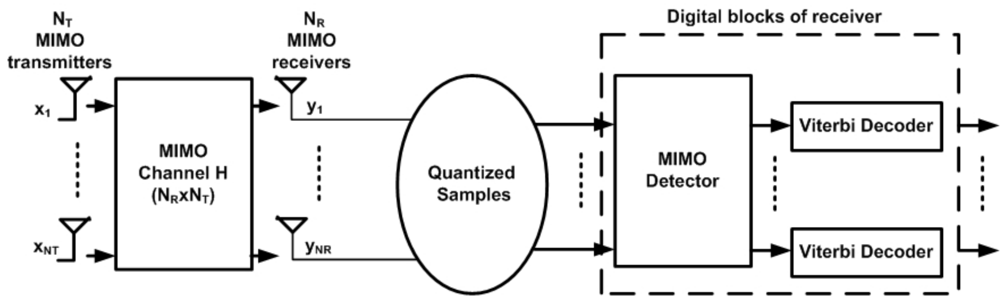


Figure 4.1: Block diagram showing several components of a MIMO system.

4.1.1 Performance of MIMO communication systems

There is an ever growing demand to design reliable communication links that operate at high data rates. The communication and digital signal processing (DSP) systems in the physical layers of these links are required to be area and power efficient. *Bit error rate* (BER) is a commonly used performance metric for these systems. BER is an average measure of the probability with which a transmitted data bit is decoded in error. In wireless communication systems, BER requirements can be as low as 10^{-7}. MIMO systems [24] are designed to meet these requirements.

MIMO systems are complex and comprise a large number of digital components implemented in RTL. The process of making MIMO RTL designs meet the BER requirements is both time- and resource-intensive. This is due to other criteria, such as area and power, that also need to be met. Therefore, it is desirable to have a methodology where performance estimation of MIMO RTL can be performed quickly and with a high degree of confidence.

Performance metrics are inherently probabilistic in nature due to the randomness introduced by corruption of the data signals that reach the receiver. The data is further corrupted in the receiver blocks due to internal fixed-point quantization errors. Conventionally, performance estimation is done by performing Monte Carlo simulations [61] of MIMO RTL using random input vectors. Estimates that are reasonably accurate can be obtained by simulating the MIMO systems [25] over many cycles. However, such simulation-based techniques are time-consuming and incomplete.

4.1.2 Using SHARPE for formal performance analysis of MIMO RTL

MIMO RTL designs are typically sequential designs. Therefore, in order to apply our SHARPE methodology (Section 3.4), we represent RTL designs of MIMO systems as RTL DTMCs (Section 3.4.1). We define BER-like performance metrics that can be expressed as properties in pCTL [22]. We then use PRISM to verify the pCTL properties on the DTMC models. This formally guarantees the statistical performance of MIMO RTL designs.

In [62], we apply our methodology to evaluate performance when bit errors result only due to algorithmic choices or RTL design choices made by the designer. We wish to analyze the performance of the "real" hardware implementation. Therefore, we need to consider the bit errors that can also arise due to physical impairments at the lower levels of implementation(i.e., gates, transistors).

The hardware realizations (i.e., the post-synthesis designs) of RTL designs are subject to several forms of impairments [1],[8],[33],[63] which we model as *physical faults*. In order to provide performance estimates for MIMO RTL designs that are faithful to reality, the presence of faults must be modeled. Such estimates can then be used to verify whether the performance of the MIMO RTL design is within an acceptable range even in the presence of physical follies.

We enhance the RTL DTMCs in order to represent the occurrence of faults. We then employ the BER performance properties to formally compute performance metrics for these "faulty" DTMC models and check if the deviation is within acceptable limits. We use this approach to rigorously analyze the vulnerability of MIMO RTL performance to faults that are present at different locations in the design.

When the BER performance property fails, it implies that the design does not meet a specified BER requirement. We present an error profiling technique where we define a pCTL property that can be used to distinguish between the different causes for the occurrence of bit errors. We then use this technique in order to devise a mechanism to systematically identify the broad reason for the severe degradation in BER. Since our identification mechanism can be performed entirely in software, hardware components are not required. This results in savings in both area and power in the post-synthesis designs.

We illustrate our technique on seminal components of a MIMO system (Figure 4.1), using a Viterbi decoder [52] and MIMO detector [25] as case studies.

4.1.3 Chapter organization

The rest of this chapter is organized as follows. In Section 4.2 through Section 4.5, we describe our methodology for formal performance analysis in the absence of physical faults. In Section 4.2, we describe the modeling of bit errors in the absence of physical faults. In Section 4.3, we describe all the steps in our methodology. In Section 4.4, we illustrate our methodology on several fault-free MIMO components. We present the corresponding experimental results in Section 4.5. In Section 4.6, we describe how we enhance the methodology in order to analyze the vulnerability of performance to hardware faults. In Section 4.7, we present a technique that can be used to identify the broad cause for BER degradation.

4.2 Macromodels for bit errors in fault-free designs

In a communication system with digital blocks in the receiver, an analog to digital Converter (ADC) first translates the received analog signals into bits by *discretizing* it in time (*sampling*). Digital blocks can represent data only using finite precision, i.e., a finite number of bits. Therefore, the received samples are discretized in value (*quantization*) as well. The digital blocks in the receiver then process these quantized samples to decode the transmitted bit. A bit error is said to occur if the decoded bit does not match the actual transmitted data bit. In this work, we confine our analysis to the digital blocks by assuming knowledge of the statistical performance of analog blocks

However, imperfections such as thermal fluctuations in current and voltage, and timing errors of the ADC sampler, are present in the circuitry preceding the receiver. Collectively, these imperfections constitute a *noise* that corrupts the received sample. The

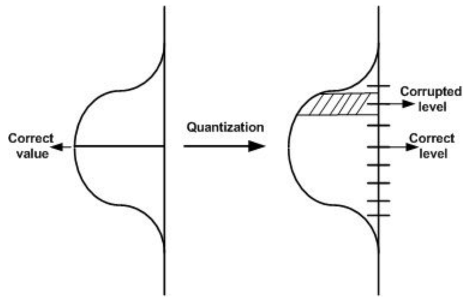


Figure 4.2: Quantization of noisy samples.

corrupted sample may be mapped to a quantization level that deviates from the correct level depending on the value of noise, as shown in Figure 4.2. For a sufficiently high value of noise, the decoded bit computed using the corrupted quantization level may be in error. Such bit errors occur due to the corruption of the data before it is processed by the receiver and are called *external data corruption errors*. These type of bit errors are not caused due to data corruption within the digital blocks of the receiver.

Data processed by the receiver is represented internally using a finite number of bits. The area and power consumed by digital blocks is proportional to the number of such bits. Therefore, it is advantageous to reduce the total number of bits that are used. However, the reduction of bits results in a loss of precision of data. For example, consider that 0.55 and 0.95 are the outputs of two adders in the receiver structure. Due to insufficient number of bits, both these values may be represented as 0.75. If these outputs are compared at a later stage in the receiver, the loss in precision can result in a bit error. These bit errors constitute a second class of errors and are called *internal data corruption errors*. Since we use a bit-accurate RTL description, we are able to capture these internal errors in addition to the external errors.

Noise is commonly represented as a single random variable, with a zero-mean Gaussian distribution [14], that is added to an uncorrupted received sample. This is called an *additive white Gaussian noise* (AWGN) model. *Signal-to-noise Ratio* (SNR) represents the level of the uncorrupted signal relative to that of the noise. For high values of SNR, the noise is insignificant compared to the signal, resulting in a low BER. Given the SNR, the shape of the Gaussian distribution can be determined. The area of the shaded region in Figure 4.2 can be computed. This is equal to the probability that the corresponding quantization level is processed by the receiver.

In this work, we assume that the analog blocks exhibit ideal behavior. We also assume an AWGN model and a *binary phase shift key* (BPSK) signaling scheme [14]. However, our methodology is not limited to these assumptions.

In Section 4.6, we shall define hardware faults as another potential cause for bit errors. In the rest of this work, we use the term “errors” exclusively to refer to bit errors in the communication system.

4.3 Formal performance analysis using our SHARPE methodology

The steps involved in our formal performance analysis methodology (Section 3.4) are:

- **RTL DTMC modeling**: We represent the target MIMO RTL design as an RTL DTMC (Section 3.4.1). We assume that every transition of the DTMC model corresponds to a single time step (modeled by an explicit clock in RTL). For a given SNR, we obtain the variance of the Gaussian distribution of noise. We use this to calculate the probability of a received sample being mapped to a particular quantization level which in turn can be used to label the transitions of the RTL DTMC model.

- **Property specification**: We define a set of BER-like performance metrics to rigorously analyze the error-related performance of a system. We write pCTL properties corresponding to these metrics, as functions of the state variables in the RTL DTMC.

- **Property-preserving reduction**: We determine property-preserving reductions by analyzing certain components of a MIMO system. We show that our reductions are sound with respect to the pCTL properties. We identify that these reductions can be extended to a large class of designs for checking the same properties.

- **Probabilistic model checking**: We use PRISM to verify the specified properties by rigorously analyzing the RTL DTMC model. We explore the transitions of the RTL DTMC until it reaches a steady state (Section 3.4.3).

4.4 Case studies: MIMO systems

A MIMO system with N_R receiver antennas and N_T transmit antennas can be modeled as

$$\overline{y} = \mathbf{H}\overline{x} + \overline{n} \tag{4.1}$$

where $\overline{y} = [y_1, .., y_{N_R}]^T$ is the vector of received signals and $\overline{x} = [x_1, .., x_{N_T}]^T$ is the vector of transmitted signals. $[\ ]^T$ denotes the transpose of a vector. $\mathbf{H}$ represents the N_RxN_T channel matrix and $\overline{n}$ is an AWGN noise vector. We assume a commonly used *flat fading Rayleigh channel* model [24] and obtain the probability distribution of the elements of $\mathbf{H}$.

In this chapter, we present the following case studies.

- Estimation of error properties of a Viterbi decoder
- Estimation of error properties of a MIMO detector

A case study for estimation of convergence properties of a Viterbi decoder is presented in [62].

4.4.1 Estimation of error properties of a Viterbi decoder

In some channels, the received sample at any time step contains components from signals transmitted in adjacent steps. This interference can be mitigated using digital blocks, such as Viterbi decoders, in the receiver. We briefly describe the Viterbi algorithm [52].

In this case study, we consider a transmitter whose output at time step n is obtained by adding the data bit from the current time step, $x[n]$, with the data bit from the previous time step (i.e., $x[n-1]$). This system is defined to have a memory (m) equal to 1.

$q[n]$ is the quantized sample at the receiver in time step n. By itself, $q[n]$ is insufficient to determine the value of the actual data bit with a low probability of error. Therefore, the Viterbi decoder waits for the samples received in the next L-1 time steps before decoding the value of the data bit. Heuristically, selecting L greater than $5m$ is assumed to be sufficient for decoding the data bit with high confidence. In this example, we consider L=6.

The Viterbi decoder maintains two internal states (0 and 1) corresponding to the possible data bits (0 and 1, respectively) in each time step. We associate the variables $prev0$ and $prev1$ with the internal states 0 and 1, respectively. Since the data bit in each time step can be a 0 or a 1, a transition can occur from any one of the two internal states to another. Each transition is associated with a probability that is a function of $q[n]$. By comparing the transition probabilities, the decoder assigns values to $prev0$ and $prev1$ that point to the corresponding most-probable previous internal state. For example, if internal state 0 is reached with a higher probability by a transition from internal state 1 than from internal state 0, the decoder assigns a value of 1 to $prev0$.

A *trellis stage* comprises the variables $prev0$ and $prev1$ corresponding to a single time step. In each time step, the Viterbi decoder stores the variables corresponding to the previous L-1 trellis stages as well. Starting at one of the internal states, the decoder can traverse a path of length L through a sequence of previous internal states using the variables in the L trellis stages. This operation is called *traceback*. A *path metric* is the cost associated with a traceback path. We use $pm0$ and $pm1$ to store the path metrics associated with internal states 0 and 1, respectively.

In each time step, the Viterbi decoder uses $q[n]$ to compute the internal state transition probabilities and then assign values to $prev0$ and $prev1$ of the corresponding trellis stage. The decoder then increments $pm0$ and $pm1$ as a function of the the computed transition probabilities. The decoder chooses the internal state with the least corresponding path metric, as the starting point for traceback.

At the end of the traceback operation, the Viterbi decoder decodes the data bit. However, there is a decoding latency of L-1 time steps. Therefore, to verify the correctness of the decoded bit in each time step, we need to keep track of the actual data bits corresponding to the previous L-1 time steps.

We now describe, in detail, all the steps involved in our methodology.

DTMC modeling

We represent the Viterbi decoder as a DTMC model M with the following state variables (Section 2.2):

- $pm0$ and $pm1$: 7-bit variables to store pathmetrics
- $prev0_i$ and $prev1_i$: Variables used to store values of $prev0$ and $prev1$ in the i^{th} trellis stage, where $0 \leq i \leq L-1$. i=0 corresponds to the trellis stage in the current time step.
- x_i: Data bit in the i^{th} trellis stage.
- $flag$: Variable that is set to 1 if the decoded bit is in error.

We define the initial state μ_0 of M by assigning an initial value (typically, equal to 0) to each state variable. In each time step (i.e., each clock cycle in RTL), the state variables of M are assigned new values. For each variable, the set of possible values is finite. Therefore, M is a finite DTMC model. The assignment of new values denotes a transition from state μ to another state μ'. The following assignments collectively define T_P.

- Data bit and path metrics: x_0 is assigned a value of 0 or 1 with equal probabilities (equal to 0.5). Based on the SNR, a quantization level q_0 is probabilistically chosen. The probability distribution of q_0 for a given SNR can be computed by determining the areas of the shaded regions, as shown in Figure 4.2. q_0 is used to obtain values of $pm0$ and $pm1$, given by

$$(pm0', pm1', x_0') = \Gamma_p(pm0, pm1, x_0) \tag{4.2}$$

where Γ_p is a probabilistic function with the combined probabilities of x_0 and q_0. The probability of a transition from state μ to another state μ' is equal to the probability that x_0 and q_0 are assigned their new values x_0' and q_0', respectively. Since q_0 is an intermediate variable that we discard, we do not store it as a state variable. However, the required probability information of q_0 can be recovered from $pm0$, $pm1$ and Γ_p.

The remaining state variables are then assigned with non-probabilistic functions that do not affect the state transition probabilities.

- Transmitter: F_S computes the new values of $prev0$ and $prev1$ in the current trellis stage, as functions of the new path metrics $pm0'$ and $pm1'$. This is given by

$$(prev0_0', prev1_0') = F_S(pm0', pm1') \tag{4.3}$$

- *Writeback*: Values of variables denoting stage i of the trellis are written to those in stage $i+1$. This action represents the entire trellis structure being advanced by one time step.

$$(prev0_{i+1}', prev1_{i+1}', x_{i+1}') = (prev0_i, prev1_i, x_i) \tag{4.4}$$

- *Traceback*: F_E determines the decoded bit as a function of the values of $prev0$ and $prev1$ across L trellis stages. F_E sets $flag$ to 1, if the decoded bit is not equal to the corresponding actual data bit x_{L-1}.

$$flag' = F_E(prev0_i', prev1_i', x_{L-1}) \tag{4.5}$$

States where the decoded bit is in error are of interest to us. To tag the states of interest, we use $flag$ to define a *reward* model on the DTMC. A reward is defined as a cost associated with being in various states of the DTMC. For each state, we assign a reward equal to the value of $flag$ in that state.

Property specification

We define the following BER-like metrics for T time steps and write the corresponding pCTL properties that we use in PRISM.

- **P1** (Best case error): P=? [G $\leq$ T (!flag)]: Probability that no error occurs in any of the T steps.

- **P2** (Average case error): R=? [I = T]: Probability that an error occurs at exactly the T^{th} step.
- **P3** (Worst case error): P=? [F $\leq$ T (flag>1)]: Probability that number of errors occurring in T steps is greater than a pre-determined value (value equal to 1, in this case).

Simulation-based techniques employ a counting process to estimate BER. Over T time steps, BER is computed as

$$\text{BER} = \frac{\text{Number of bit errors}}{T} \tag{4.6}$$

For sufficiently large T, BER converges to a fixed value. Therefore, BER is not a time-bounded metric. BER can then be interpreted as the probability of a bit error occurring at any time step.

We define **P2** as a reward property that computes the *expected* instantaneous value of $flag$ after exactly T transitions (time steps) of the DTMC model. Therefore, **P2** is equal to the probability of being in a state with $flag$=1 (i.e., a bit error) in time step T. However, once the DTMC model attains a steady state, **P2** is independent of the value of T.

In Section 4.5, we demonstrate that our systems do attain a steady state, and therefore **P2** computed using our methodology corresponds to the BER of the system. Since we use a simple reward model that assigns rewards of 0 or 1, we do not need to express **P2** using the reward-based extension of pCTL [64].

BER (and **P2**) is a measure of the average number of errors in a system in steady state. For example, a BER of 10^{-3} implies that on an average, 1 bit is in error in a transition path (i.e., sequence of transitions) that is 1000 time steps long. However, there may be transient paths of length 1000 that have either 0 errors (best case) or 10 errors (worst case). BER does not provide any information about the frequency with which these paths occur. Therefore, we define **P1** and **P3**.

In addition to average case (**P2**), we analyze best and worst case error scenarios. This enables us to make stronger claims regarding the error-related performance of MIMO RTL. The properties are checked by performing an exhaustive exploration of all the possible paths of length T.

Property-preserving reduction

For error properties, it is sufficient to determine whether a bit is in error or not. Reductions can be defined for checking error properties, that compute bit errors without actually determining the values of the decoded bits. In such cases there is no comparison of values with the transmitted bits. Therefore, in designs with decoding latency, variables storing past values of transmitted bits can be discarded from the model.

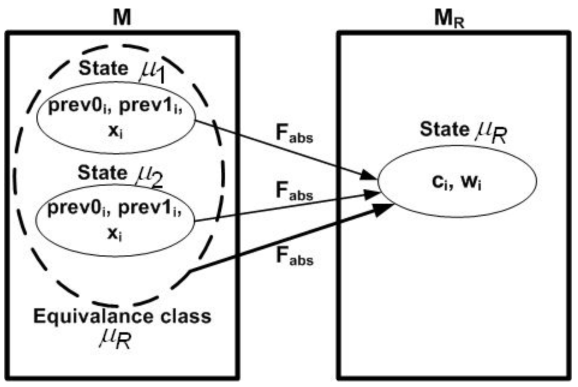


Figure 4.3: Reduction from M to M_R.

We obtain the reduced Viterbi decoder model M_R by replacing the variables $prev0_i$, $prev1_i$ and x_i (excluding x_0) with the variables c_i and w_i (Figure 4.3). We need the variables c_i and w_i to indicate whether $prev0_i$ and $prev1_i$ point to the previous internal state corresponding to the actual data bit x_i. This information is sufficient to check the correctness of the traceback operation, and thereby check the correctness of the decoded bit. We construct an abstraction function F_{abs} to assign values to c_i and w_i, given by

$$(c_i', w_i') = F_{abs}(prev0_i', prev1_i', x_i') \tag{4.7}$$

Multiple states in M (μ_1, μ_2,..) are mapped to the same state μ_R in M_R, by the function F_{abs}. This illustrates how we achieve a reduction in the state-space. Variables $pm0$, $pm1$ and x_0 from model M, are retained in the reduced model M_R. The values of these variables are the same in states μ_1, μ_2 and μ_R. Therefore, the probabilistic function Γ_p is also preserved by our reduction. The non-probabilistic state transition assignments for M_R are given by

$$(c_0', w_0') = F_{cw}(pm0', pm1', x_0') \tag{4.8}$$

$$(c_{i+1}', w_{i+1}') = (c_i, w_i) \tag{4.9}$$

$$flag' = F_{E_R}(c_i', w_i', x_0') \tag{4.10}$$

where F_{E_R} is a slightly modified version of F_E from model M.

M_R does not have information to obtain the values of the decoded bits. However, $flag$

in M_R indicates the correctness of the decoded bit, as in M. Through this reduction, the variables x_1 to x_{L-1} can be discarded. Therefore, the size of M_R is smaller than that of M.

Proof of correctness

We need to show that M_R is a probabilistic bisimulation of M. We prove this in two parts. In **Part A**, we establish that the variable based on which the error property is defined (i.e., $flag$), is preserved by the reduction. In **Part B**, we show that M_R also preserves the probabilistic behavior of M. We then employ the *strong lumping theorem* [65] to complete the proof.

All the states in M that are mapped to the same state in M_R through the function F_{abs}, constitute an *equivalence class* [36]. Two states are in the same equivalence class if and only if they are equivalent under a given equivalence relation. F_{abs} is the equivalence relation that establishes a one-to-one correspondence between such an equivalence class in M and the corresponding state in M_R (Figure 4.3). We use μ_R to refer to both a state in M_R and the corresponding equivalence class in M.

Part A: We need to prove that the value of $flag$ assigned to a state in M_R is the same as in the corresponding equivalence class in M. We do this by verifying that Equations 4.5 and 4.10 are equivalent[1].

Part B: We need to prove that the equivalence classes preserve the probabilistic behavior of the states of M. Consider a transition in M, from μ to a destination state μ'. μ' is mapped by F_{abs} to the corresponding state μ'_R in M_R according to

$$(c'_0, w'_0) = F_{abs}(prev0'_0, prev1'_0, x'_0) \tag{4.11}$$

Combining Equations 4.4 and 4.7,

$$\begin{aligned}(c'_{i+1}, w'_{i+1}) &= F_{abs}(prev0'_{i+1}, prev1'_{i+1}, x'_{i+1}) \\ &= F_{abs}(prev0_i, prev1_i, x_i) \\ &= (c_i, w_i)\end{aligned} \tag{4.12}$$

We verify that Equation 4.8 and 4.11 are logically equivalent. This implies that if two states (μ_1, μ_2) belong to the same equivalence class in M, their respective destination states (μ'_1, μ'_2) are also equivalent under F_{abs}.

Any state transition ($\mu \rightarrow \mu'$) in M corresponds to a transition between the respective equivalence classes ($\mu_R \rightarrow \mu'_R$). In our example, for each equivalence class μ'_R, $\mu' \in \mu'_R$

[1]Since our functions are Boolean, we use an equivalence checker [66].

is a unique destination state that $\mu \in \mu_R$ can transition to. We write the transition probability as

$$\begin{aligned} P(\mu_R \to \mu'_R) &= P(\mu \to \mu') \\ &= \sum_{\mu' \in \mu'_R} P(\mu_R \to \mu') \end{aligned} \tag{4.13}$$

States in equivalent classes related by F_{abs} transition to the same set of equivalence classes related by F_{abs} and with the same probability distribution. According to the strong lumping theorem, any quotient DTMC that comprises these equivalence classes as its states, is a probabilistic bisimulation of M. In fact, M_R is such a DTMC. Since Γ_p is preserved by our abstraction, the associated probabilities in Equation 4.13 also hold true for state transitions in M_R. Therefore, M_R is a probabilistic bisimulation of M for checking error properties.

Probabilistic model checking

We use PRISM (Section 3.4.3) to verify the properties **P1**, **P2** and **P3** on the reduced DTMC model M_R.

4.4.2 Estimation of error properties of a MIMO detector

For the MIMO system in Equation 4.1, detectors estimate the *most likely* $\overline{x}$, given the received vector $\overline{y}$. This maximum likelihood (ML) MIMO detection algorithm can be expressed as

$$\widehat{x} = argmin\,|\overline{y} - \mathbf{H}\overline{s}| \tag{4.14}$$

where $\widehat{x}$ is the detected vector and $\overline{s}$ is a possible value of $\overline{x}$.

We consider a 2x2 MIMO sytem with BPSK signals. Therefore, each vector element x_i can be a 0 or a 1. The ML algorithm can be implemented as in [25],

$$\begin{aligned} \widehat{x} &= argmin(|y_1 - h_{11}s_1 - h_{12}s_2| \\ &+ |y_2 - h_{21}s_1 - h_{22}s_2|) \end{aligned} \tag{4.15}$$

where both s_1 and s_2 are elements of $\overline{s}$ and can have values of 0 and 1. We split Equation

4.15 further into real and imaginary parts,

$$\begin{aligned}\widehat{x} &= argmin(|y_{1,R} - h_{11,R}s_1 - h_{12,R}s_2| \\ &+ |y_{1,I} - h_{11,I}s_1 - h_{12,I}s_2| \\ &+ |y_{2,R} - h_{21,R}s_1 - h_{22,R}s_2| \\ &+ |y_{2,I} - h_{21,I}s_1 - h_{22,I}s_2|) \\ &= argmin(M_{1,R} + M_{1,I} + M_{2,R} + M_{2,I}) \qquad (4.16)\end{aligned}$$

where the metrics $M_{1,R}..M_{2,I}$ are computed for each of the four possible values of the vector $\overline{s}$. The *argmin* function determines that the most likely transmitted vector, $\widehat{x}$, is the vector $\overline{s}$ that corresponds to the least sum of metrics as in Equation 4.16.

We construct the DTMC model for the MIMO detector, as in Section 3.4.1. We use the transmitted bit vector $\overline{x}$ and the real and imaginary parts of the elements of both $\overline{y}$ and $\mathbf{H}$, as DTMC state variables. We determine $\widehat{x}$ using Equation 4.16 and compare it with x to assign the value of $flag$. We assume knowledge of the probability distributions of the elements of $\mathbf{H}$ and $\overline{n}$ (based on SNR). We combine this with Equation 4.1 and compute the probability distributions of the state variables. In our MIMO detector model, the values of the states variables in a time step are independent of their values in the previous time step. Therefore, the probability of a transition from state μ to μ' is equal to the joint probability with which the state variables are assigned their values in μ'. We use the state variable $flag$ to define the DTMC reward model.

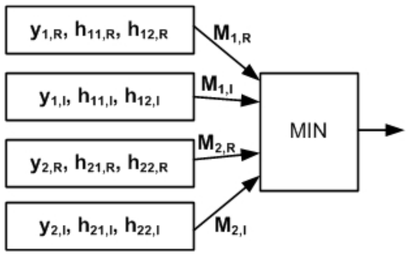


Figure 4.4: Symmetry in MIMO detector.

Consider a state μ_1 of the DTMC model of the MIMO detector. The variables $y_{1,R}$, $h_{11,R}$ and $h_{12,R}$ constitute the block that computes $M_{1,R}$ (Figure 4.4). Let us interchange the values of these variables with those of the corresponding variables from the block that computes $M_{1,I}$ (i.e., $y_{1,I}$, $h_{11,I}$ and $h_{12,I}$ respectively). This new assignment of values corresponds to another state μ_2 of the DTMC.

From Equation 4.16, we observe that the computation of $\widehat{x}$ (and $flag$) is unaffected by the interchange operation between states μ_1 and μ_2. We also observe that the probabilistic assignments to the corresponding variables in the two blocks, are symmetrical. Therefore, the states μ_1 and μ_2 exhibit symmetrical probabilistic transitions. This proves that the blocks for the metrics $M_{1,R}$ and $M_{1,I}$ are symmetric with respect to error properties that are defined based on $flag$. In fact, this is true across all the four blocks in the detector. In general, for any N_RxN_T MIMO detector, there are 2xN_R symmetric blocks.

We employ symmetry reduction [36] to reduce the size of the DTMC model, as seen in Table 4.2. MIMO designs that exhibit such symmetries, constitute a large class of systems where symmetry reduction can be applied. In this case study, we check only the average case property **P2**.

Table 4.1: Error properties for a Viterbi decoder.

	States (Original model)	States (Reduced model)	Time (sec)	Result
P1	$53,558,744$	$8,505,363$	90.80	3x10^{-15}
P2	$53,558,744$	$8,505,363$	184.13	0.2394
P3	$107,504,890$	$16,435,490$	365.68	≈ 1

4.5 Experimental results: Fault-free performance

We perform our experiments on a 3 GHz, 3.25 GB machine. In all our experiments, we assume that the system variables are equal to 0 at the start of operation. We represent this by initializing all DTMC state variables to 0.

For an SNR of 5dB, we check the error properties for the Viterbi model over T=300 time steps (Table 4.1). The times listed account for both model construction and model checking. **P2** indicates that the system has a high BER, equal to 0.2394. **P1** and **P3** provide measures over a window of 300 consecutive bit tranmissions at the start of system operation. **P1** shows that the fraction of error-free paths in this window is only 3x10^{-15}. **P3** shows that almost all paths in this window have strictly more than 1 bit error. We can assign the initial state of the system to measure **P1** and **P3** over any window of operation. **P1**, **P2** and **P3** together confirm the poor error-related performance of the system for the given SNR.

Table 4.2 shows the reduction factors achieved in the MIMO detector. We consider 1x2 (SNR=8dB) and 1x4 (SNR=12dB) MIMO ML detectors. In the 1x4 detector, PRISM discards states that are reached with a probability less than 10^{-15}.

Table 4.2: Symmetry reduction of MIMO detector.

MIMO	States (Original model M)	States (Reduced model M_R)	Reduction factor
1x2	569,480	32,088	18
1x4	524,288	1,320	400

PRISM performs a reachability analysis first and a fixpoint is achieved. The fixpoint is referred to as *reachability iterations* (RI). After this fixpoint, no new states are reached in further iterations. Tables 4.3 and 4.4 show the computations of **P2** for different values of T. We observe that for values of T much greater than RI, the computed values do not change significantly. Once steady state is attained, we consider **P2** as the BER of the system.

The SS columns in Tables 4.3 and 4.4 show the exact values of **P2** computed using the steady-state solver in PRISM. We compare the computation times for the steady-state solver with the model checking times of our time-bounded approach. For the 1x4 MIMO detector, our time-bounded model checking completes in less than 0.5 sec for all values of T while the steady-state solver takes 53.27 sec. In all experiments, we observe that our approach is faster than using the steady-state solver. Moreover, our results deviate only negligibly from the steady-state solutions. For applications where performance evaluation is offline and not time-critical, either approach can be used.

We observe that the DTMC model for the Viterbi decoder is finite, irreducible and aperiodic. Therefore, the model is guaranteed to converge to a steady-state probability distribution (Section 2.2). Although at a slower rate than for the MIMO detector, the computations for the Viterbi decoder converge reasonably quickly. To check error properties, all MIMO RTL designs will be represented as DTMC models of a similar structure. Therefore, a steady-state solution is guaranteed, although the exact time steps required to attain this may vary.

The values computed in our approach closely match those obtained by performing simulations over a large number of time steps. We simulate 10^7 time steps to estimate a BER of 1.07x10^{-5} for the 1x4 MIMO system in Table 4.4. We observe zero bit errors in 10^5 time steps. This clearly illustrates the efficiency of our approach as compared to simulation-based techniques, particularly for very low BER requirements.

Table 4.3: **P2** for the Viterbi decoder (RI=263).

Viterbi	T=100	T=300	T=600	T=1000	SS
P2	0.2373	0.2394	0.2397	0.2398	0.2398
Time (sec)	23.72	27.66	28.28	29.13	901.02

Table 4.4: **P2** for MIMO detectors (RI=3).

MIMO	T=5	T=10	T=20	SS
1x2	0.277	0.291	0.296	0.296
1x4	1.08x10^{-5}	1.08x10^{-5}	1.08x10^{-5}	1.08x10^{-5}

4.6 Analyzing the effects of faults on BER performance

We now describe an approach to enhance our formal RTL performance analysis framework [62] in order to include the effects of physical hardware faults. Permanent physical faults [67] are hardware defects that arise during the manufacturing process. Transient physical faults [8] in hardware are radiation-induced and arise due to neutrons from cosmic rays and alpha particles from packaging material.

For RTL designs of communication systems, we wish to measure BER which is an application-specific statistical metric. Therefore, the average effect of faults on BER performance is of interest to us. In particular, we wish to verify that the BER performance remains within acceptable limits even in the presence of faults. Since faults can potentially degrade the BER by several orders of magnitude, we measure the deviation from fault-free BER as a multiplicative factor given by

$$\text{Deviation Factor} = \frac{\text{BER of faulty RTL design}}{\text{Fault-free BER}} \tag{4.17}$$

For example, a deviation factor of 1.20 corresponds to a deviation of 20% from fault-free performance. We define the BER property (**P2**) to have failed in the presence of faults if the deviation factor exceeds a value specified by the designer.

We illustrate our approach by computing the BER performance (property **P2**) of a Viterbi decoder in the presence of physical faults.

4.6.1 Fault macromodeling

We use the well-known *single stuck-at* fault model to represent permanent physical faults. In this model, we assume that a fault corrupts the value of only a single bit in the entire

RTL design and we define this as the *fault location.* The bit where the fault is assumed to occur is tied to a constant logical 1 or 0 value. For example, a bit with a stuck-at-1 fault denoted by *s-a-1* is modeled to have a constant logical value of 1 throughout the execution of the system regardless of the values other signals may try to assign to it. In RTL designs, stuck-at faults can be modeled by setting the bits of RTL variables to be either a constant 1 or 0.

Transient faults are typically modeled as *single event upsets* (SEUs) where only one bit in the RTL design is assumed to be flipped (inverted) in a given clock cycle. These bit flips do not occur in every clock cycle. Instead, they are associated with a probability of occurrence p_{SEU} that depend on electrical and timing parameters at the lower levels of implementation [68]. For example, a value of p_{SEU} equal to 10^{-3} implies that the bit flip occurs approximately once in every 1000 clock cycles. We assume that the bit remains flipped until it is overwritten by a new assignment of values to the corresponding RTL variable.

The inputs and outputs of all gates in both combinational and sequential logic are potentially locations for occurrence of permanent and transient faults. However, the effects of faults in combinational logic elements are of consequence only if they propagate to memory elements in sequential logic (latches and flip-flops) which in turn affects the state of the system [8]. In RTL designs, the register variables are the memory elements, and therefore we consider only bits of these RTL register variables as potential fault locations. The state variables that we define for the unreduced DTMC model are in fact the register variables in the RTL design (Section 5.4.1). Additionally, in each experiment we only consider the occurrence of either a permanent fault or a transient fault but never both simultaneously.

4.6.2 Effect of permanent faults on BER

We analyze the effects of permanent faults by injecting them into one fault location bit at a time. We select the state variable from DTMC M that corresponds to the RTL variable where we wish to inject the stuck-at fault. We then define the value of the fault location bit in the state variable to be a constant 1 or 0. The modified DTMC M^{PF} now represents the RTL design with the single stuck-at fault.

We wish to compute the BER performance using M^{PF}. Therefore, the variable $flag$ (Section 3.4.1) is still of our interest and we retain the DTMC reward model that we use for computing performance in the fault-free scenario. We model check property **P2** on M^{PF} in order to formally compute the BER performance of the faulty RTL design.

Table 4.5: BER of the Viterbi decoder in the presence of permanent faults(SNR 12dB).

Fault Location	Fault Type	BER	Deviation Factor
$prev0_4$	s-a-1	0.4401	43231.8
$prev1_0$	s-a-0	0.4844	47583.5
LSB of $pm0$	s-a-1	$1.191\text{x}10^{-5}$	1.17
sign bit of $pm0$	s-a-1	0.4744	46601.2
LSB of $q[n]$	s-a-0	0.07046	6921.4
sign bit of $q[n]$	s-a-0	0.5	49115.9

Table 4.5 shows the effects on BER of a Viterbi decoder due to the introduction of single stuck-at faults at different locations in the RTL design. We consider that the decoder is operating at an SNR of 12dB. The deviation factors are calculated with respect to the fault-free BER which we formally compute to be equal to $1.018\text{x}10^{-5}$.

$prev0_4$ and $prev1_0$ (Section 3.4.1) are Boolean variables that control the direction of the traceback operation. Therefore, stuck-at faults at these locations permanently point traceback to the same previous state irrespective of the value of the actual transmitted bit. Therefore, we observe that the BER is degraded by several orders of magnitude as indicated by the large deviation factors.

$q[n]$ is the quantized sample at the input of the Viterbi decoder (Section 3.4.1). For the Viterbi decoder design that we consider, $q[n]$ can have seven possible values. Although, $q[n]$ is not used as a DTMC state variable in Section 3.4.1, we shall consider it as a potential fault location. A fault in the least significant bit (LSB) of $q[n]$ offsets the value of $q[n]$ by 1. This translates to the decoder incorrectly using an adjacent quantization level in place of the actual level indicated by the fault-free value of $q[n]$. Since there are only seven quantization levels in total, the resulting effect is quite significant. However, the same value of $q[n]$ is used to update the values of both the pathmetrics $pm0$ and $pm1$. Therefore, a change in value of $q[n]$ is nullified to a small extent and the BER degradation is not as severe as compared to faults in $prev0_4$ and $prev1_0$.

$q[n]$ is a *signed* variable that can be assigned both positive and negative integer values. The most significant bit represents the sign and is called the *sign bit.* A fault at the sign bit of $q[n]$ can result in a positive value being incorrectly interpreted as a negative one (or vice versa) resulting in severe worsening of the BER performance. Similar performance degradation can be observed for a fault at the sign bit of $pm0$ which is also a signed variable.

The least significant bit (LSB) of $pm0$ does not affect the value of $pm0$ significantly since it is a 7-bit variable (Section 3.4.1). Consequently, a fault at the LSB of $pm0$ (or $pm1$, by symmetry) does not affect BER performance significantly as indicated by the low deviation factor (equal to 1.17). If a deviation factor of up to 1.20 is acceptable, then the faulty design falls within the tolerance range that is specified for BER performance (Equation 4.17). Therefore, the designer can choose not to employ fault-tolerance techniques in order to protect the LSB of $pm0$, and thereby avoid the associated hardware overheads.

4.6.3 Effect of transient faults on BER

We now analyze the effects of transient faults on BER performance. We wish to modify the DTMC model M that corresponds to the fault-free RTL design in order to model the occurrence of transient faults. As in the case of permanent faults, we consider only one transient fault location bit at a time.

We modify the DTMC M by defining a new Boolean state variable *seu_event* which we set to 1 with probability p_{SEU} and to 0 with probability 1-p_{SEU}. In all DTMC states where *seu_event*=0, we assign all the DTMC state variables their corresponding fault-free values. In states where *seu_event*=1, we flip the bit of the state variable at which we wish to model the occurrence of the transient fault. When the DTMC reverts to a state where *seu_event*=0, we retain the bit flip unless a new value is assigned to the state variable. Figure 4.5 represents the state transition diagram corresponding to our model for the injection of transient faults. The modified DTMC, which we call M^{TF}, now represents the RTL design with the SEU fault. We then model check property **P2** on M^{TF} in order to formally compute the BER performance of the faulty RTL design.

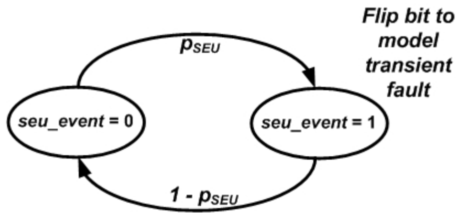


Figure 4.5: State transition diagram depicting injection of transient faults.

Table 4.6 shows the change in BER of the Viterbi decoder (SNR=12dB) for different locations of transient faults. In this analysis, we do not use an exact value of p_{SEU} that is derived using empirical data from the lower levels of design. Instead, we show that our analysis can easily be applied for any value of p_{SEU}.

For p_{SEU}=10^{-2}, the transient fault occurrence is quite frequent and degrades the BER of the system significantly. As p_{SEU} decreases, the system is more frequently in a fault-free state of operation, and therefore the degradation of BER performance reduces in severity.

We observe that the LSB of $pm0$ is resilient to transient faults as well as permanent faults. For p_{SEU}=10^{-6}, the deviation factors are approximately equal to 1 regardless of the fault location. The occurrence of faults is not statistically significant, and therefore the BER performance of the faulty system is hardly discernable from that of the fault-free system.

Table 4.6: BER of the Viterbi decoder in the presence of transient faults(SNR 12dB). We use DF to denote the deviation factor.

	$p_{SEU} = 10^{-2}$		$p_{SEU} = 10^{-3}$		$p_{SEU} = 10^{-6}$	
Fault Location	BER	DF	BER	DF	BER	DF
$prev0_4$	2.135x10^{-2}	2097.25	1.824x10^{-3}	179.18	1.020x10^{-5}	1.002
$prev1_0$	3.174x10^{-2}	3117.87	2.347x10^{-3}	230.55	1.021x10^{-5}	1.003
LSB of $pm0$	1.018x10^{-5}	1.00	1.018x10^{-5}	1.00	1.018x10^{-5}	1.00
sign bit of $pm0$	2.819x10^{-2}	2769.16	2.258x10^{-3}	221.81	1.020x10^{-5}	1.002
LSB of $q[n]$	5.426x10^{-3}	533.01	4.813x10^{-4}	472.79	1.018x10^{-5}	1.00
sign bit of $q[n]$	3.981x10^{-2}	3910.61	3.568x10^{-3}	350.49	1.024x10^{-5}	1.005

4.7 Identifying the cause for BER degradation

We now outline an mechanism to systematically identify the cause for degraded BER performance. The causes for bit errors in MIMO RTL designs can be broadly classified as follows:

A) External data corruption (Section 3.2)

B) Internal data corruption (Section 3.2)

C) Physical hardware faults (Section 3.6.1)

Our compositional reasoning approach (Chapter 4) can be used to analyze BER performance for each component of a large MIMO RTL design. We can then reason individually with components for which we observe a degraded BER performance. Therefore, the scalability of our mechanism can significantly be improved. However, in this work, we shall illustrate our mechanism in the context of computing BER performance for the Viterbi decoder component (Section 3.4.1).

The input to the Viterbi decoder ($q[n]$) is quantized to seven levels which we label using integers from -3 to +3. Therefore, the $q[n]$ can be assigned one of seven integer values $\{-3, -2, -1, 0, 1, 2, 3\}$ with the corresponding probabilities computed as shown in

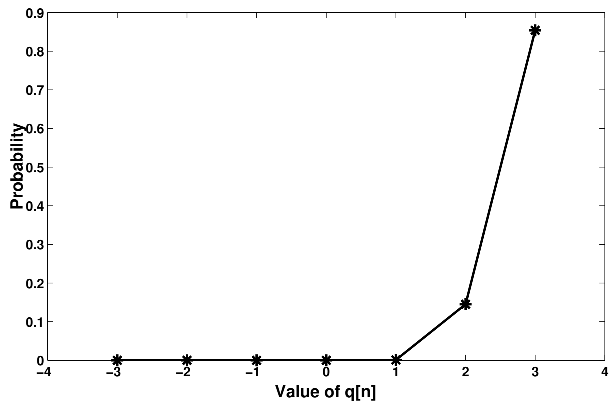


Figure 4.6: Probability distribution of input to Viterbi decoder (SNR = 5dB).

Figure 4.2. Figure 4.6 shows the probability distribution of $q[n]$ at an SNR of 5dB when a data bit equal to 1 is sent by the transmitter. With respect to decoding accuracy, the +3 quantization level is ideal since it is the least corrupted, and therefore the least likely to be misinterpreted as a data bit equal to 0. Similarly, the -3 level is the ideal level when a data bit equal to 0 is transmitted. In the rest of this section, we shall illustrate our mechanism by assuming that the transmitted data bit is equal to 1. By symmetry, our approach also applies to the case where the transmitted bit is 0.

In the absence of internal data corruption and faults, bit errors occur only due to external data corruption. In this case, an ideal quantization level at the input will result in an incorrect decoded bit with negligibly low probability. Therefore, if we are able to determine that an ideal quantization level contributes to a bit error with a non-zero probability, we can infer that the data has been corrupted while being processed internally in the hardware component. In this case, we can characterize all bit errors as errors due to either internal data corruption or physical faults. This forms the basis of our identification mechanism.

4.7.1 Diagnostic properties for BER degradation

We compute a probabilistic measure of the contribution of each quantization level j, $j \in \{-3, -2, -1, 0, 1, 2, 3\}$, to the occurrence of bit errors. We call this an *error characterization profile* for $q[n]$, i.e., the input. These profiles can be used to distinguish between the causes of bit errors since they provide the probability with which an ideal level (and levels close to the ideal level) at the input results in a bit error at the output. We now describe how these profiles can be obtained using probabilistic model checking.

We introduce a new Boolean state variable $dataflag$ into the DTMC model M of the Viterbi decoder. We set $dataflag$ to 1 for all states in which $q[n] \geq j$ and to 0 otherwise. We shall refer to this modified DTMC model as M^j.

We define the following pCTL property $\mathbf{ED}^j$ that we use in PRISM

$$\mathbf{ED}^j\text{: P=? [(dataflag=0) }\mathbf{U}\text{ (flag=1)]}$$

where **U** represents the *Until* pCTL operator. $\mathbf{ED}^j$ is the probability of the occurrence of a DTMC transition path where a state with $flag$=1 is encountered before a state with $dataflag$=1. In other words, $\mathbf{ED}^j$ is a measure of the fraction of paths where a bit error occurs before any quantization level less than j is obtained at the receiver. For example, $\mathbf{ED}^{+3}$ is the probability of occurrence of a path in which the decoded bit is in error despite the fact that only ideal quantization levels are received until at that point.

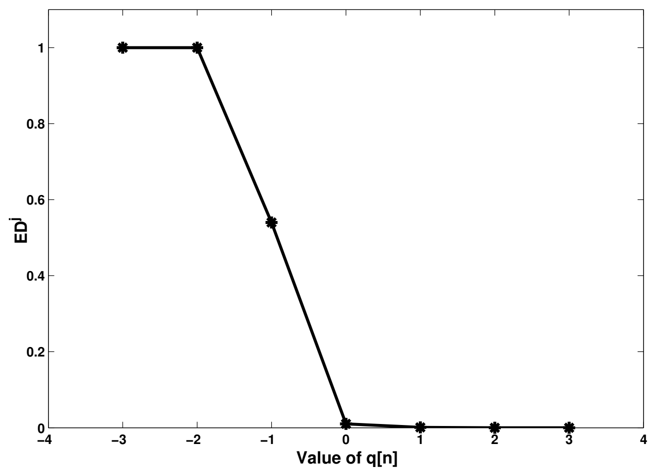


Figure 4.7: Error characterization profile for Viterbi decoder with only external data corruption (SNR = 5dB).

We use PRISM to verify the diagnostic property $\mathbf{ED}^j$ on the DTMC M^j. We repeat this for all values of j and obtain the error characterization profile for $q[n]$. Since there is a decoding latency in Viterbi decoders, the reception of a quantization level does not coincide with the decoded bit in the same time step. Hence, we use the *Until* operator in our diagnostic property.

4.7.2 Profiles for the different causes of bit errors

We now show the distinct error characterization profiles that we obtain by isolating the different causes for bit errors.

Table 4.7: BER for Viterbi decoder for different number of bits assigned to pathmetric state variables (SNR 12dB).

Number of bits	BER
4	0.0653
5	0.00131
6	3.08x10^{-3}
7	1.018x10^{-5}
8	1.021x10^{-5}

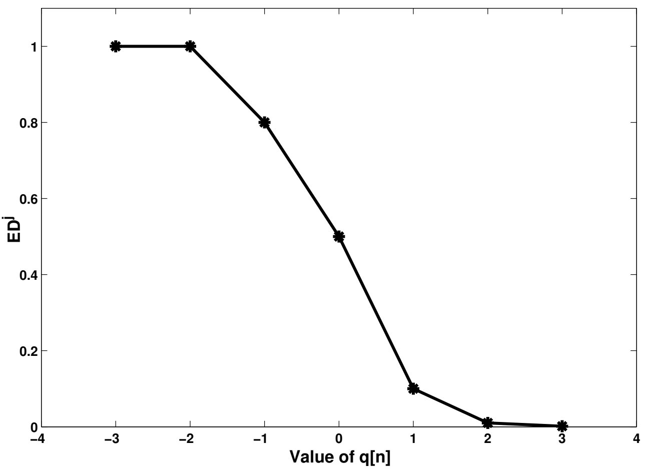


Figure 4.8: Error characterization profile for Viterbi decoder with only internal data corruption (SNR = 12dB).

We start by considering a fault-free Viterbi decoder. We first obtain the error characterization profiles when there is only external data corruption. In order to achieve this, we remove the effects of internal data corruption by using a sufficiently high number of bits for data representation.

Figure 4.7 shows the error characterization profile for input of the Viterbi decoder operating at 5dB SNR. We observe that the value of $\mathbf{ED}^{+3}$ is equal to 2.828x10^{-9} which is negligibly small compared to the BER. An ideal quantization level results in a bit error only with negligibly low probability. Therefore, we can use the profile to attribute the poor BER performance of the Viterbi decoder (Table 4.3) to external data corruption that results due to low SNR. We change the operating point to 12dB SNR and recompute the BER of the system. We observe that the BER now reduces to 1.018x10^{-5}.

We now compute the error characterization profile for the fault-free Viterbi decoder operating at 12dB SNR. Since external data corruption is negligibly low due to the increased SNR, internal data corruption is the only cause for bit errors.

Table 4.7 presents the change in BER of the Viterbi decoder resulting from a change in the number of bits used to represent $pm0$ and $pm1$. The use of more than seven bits for representing $pm0$ and $pm1$ does not significantly improve the BER of the system. We observe that the BER performance degrades noticeably even when using 6-bit variables. Figure 4.8 shows the error characterization profile for the Viterbi decoder that uses 5-bit variables for $pm0$ and $pm1$. The value of $\mathbf{ED}^{+3}$ is equal to 1.32x10^{-3} which is significantly high compared to the BER (equal to 1.018x10^{-5}). Therefore, the error characterization profile indicates that bit errors occur with high probability even when an ideal quantization level is received. From this, we can infer internal data corruption to be the cause of BER degradation.

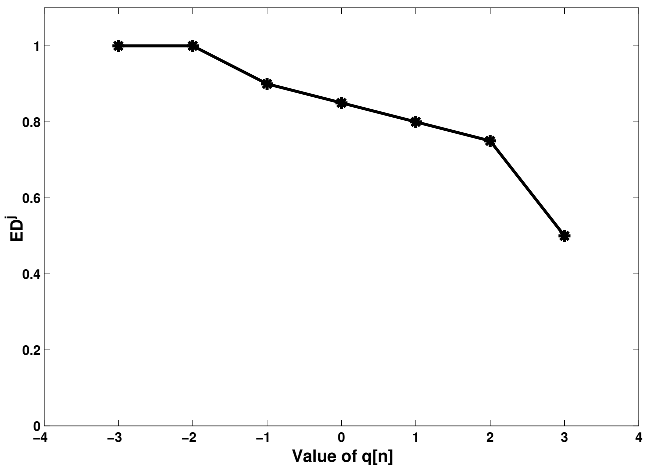


Figure 4.9: Error characterization profile for Viterbi decoder (SNR = 12dB) with s-a-1 fault at $prev0_4$.

We now consider the Viterbi decoder that is injected with physical faults. We assume that 7-bit variables are used for the path metrics and that the decoder is operating at 12dB SNR. Therefore, we remove both internal and external data corruption as causes for bit errors.

Figure 4.9 shows the error characterization profile for the Viterbi decoder with a s-a-1 fault at $prev0_4$. The $\mathbf{ED}^{+3}$ value equal to 0.5 clearly indicates the presence of data being corrupted by the hardware. Figure 4.10 shows the error characterization profile for the Viterbi decoder with a transient fault at $prev0_4$ that occurs with probability p_{SEU}=10^{-3}. Although the value of $\mathbf{ED}^{+3}$ is only 2.25x10^{-3}, it is significant compared to the BER of the fault-free system. Therefore, the error characterization profile indicates that bit errors occur with high probability even when an ideal quantization level is received. From this, we can infer that faults are the cause of BER degradation.

The profile for faults may appear similar to that for severe internal data corruption. We distinguish between these two causes of bit errors by considering faulty and fault-free designs independently.

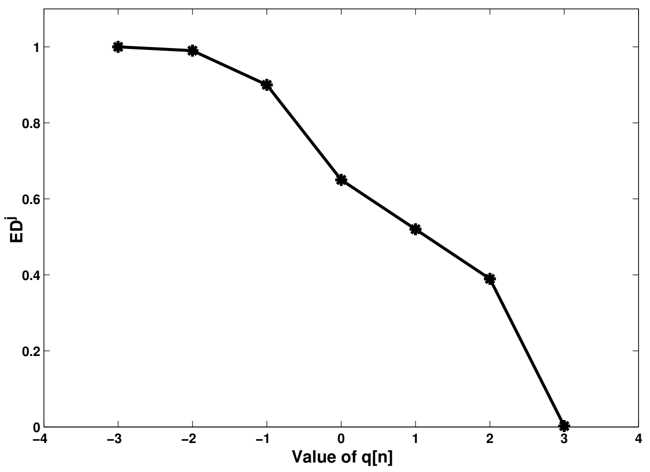


Figure 4.10: Error characterization profile for Viterbi decoder (SNR = 12dB) with a transient fault at $prev0_4$.

4.7.3 Removing the causes of bit errors

We briefly outline a sequence of steps that can be used to remove the causes of BER degradation in order to make an RTL design satisfy the BER performance property, i.e., the BER requirement.

Step 1: Remove internal data corruption

We start with the fault-free RTL design. We use the error characterization profiles to check for presence of internal data corruption. If internal data corruption is present, structural changes, such as assigning more bits for RTL data representation, can be made to reduce the occurrence of bit errors.

Step 2: Remove errors due to physical faults

We now consider the faulty RTL design and use the error characterization profiles to check for presence of physical faults. The designer can choose from several existing fault-tolerance techniques [69],[70] in order to design systems that are resilient to permanent and transient faults. The bit errors that remain are all due to external data corruption.

Step 3: Remove external data corruption

Algorithmic changes can be made to improve the performance of the system at the given operating SNR. For example, more complex Viterbi decoders and higher order MIMO detector algorithms can be used.

4.8 Chapter summary

In this chapter, we have applied our SHARPE methodology in order to formally analyze the performance of MIMO RTL designs both in the the absence and presence of low-level hardware faults. We present sound property-preserving reductions that can be used to improve the scalability of our approach. When the BER performance requirement is not met, we obtain error characterization profiles that can distinguish between the broad causes of BER degradation.

CHAPTER 5

SHARPE FOR PROBABILISTIC TIMING ANALYSIS IN RTL

5.1 Introduction

In this chapter, we apply our SHARPE methodology to perform probabilistic timing analysis in RTL. We describe how delay macromodels can be used to shift timing analysis from the lower gate level to the higher RTL.

Timing cannot be compromised in microprocessor control logic. It is only in the datapaths that probabilistic timing is acceptable. Therefore, in this chapter, we apply our SHARPE methodology primarily to combinational designs (Section 3.3).

5.1.1 Probabilistic timing analysis

Traditional design methodologies guarantee error-free performance by tuning the circuit to the worst-case delay regardless of how infrequently it may occur. In recent designs, the skew between the worst-case delay and the average-case (nominal) delay has become significantly high. For example, process variations can cause an adder delay to vary by up to 27% of its nominal delay [71]. In such cases, worst-case design strategies can result in pessimistic circuits which are over-designed for the most part. Instead, significant power/area/performance benefits can be obtained by "better-than-worst-case-design" strategies that tune the circuit to its average-case delay [72].

In the case of process variations, the fraction of chips that violate the timing constraint are discarded. With respect to input variations, recent design methodologies [2],[3],[73],[74] allow long computations to make errors which are corrected later with small performance penalty. In both cases, if the timing violations occur infrequently then the resulting loss in chips (or the penalty) may be tolerable since they provide an average gain in overall performance. Therefore, it is desirable to couple such design strategies with a methodology that can evaluate them by quickly and accurately estimating the probability of timing violations.

In the context of better-than-worst-case design, timing verification must account for the underlying variations in delay. At present, this stochastic nature is addressed only at the lower gate-level of the hardware design cycle [5],[72],[75],[76],[77] where appropriate circuit design measures are taken to allow for variation-dependence in timing. However, such gate-level techniques do not offer a scalable solution for statistical timing analysis at the higher levels of design. Although there is variation-awareness in high level synthesis techniques [7],[78],[79] as well as architecture level power and performance analysis [15], this is not observed in RTL design verification methodologies.

5.1.2 The SHARPE methodology for probabilistic timing analysis in RTL

SHARPE is a methodology for formally computing probabilistic distributions of statistical metrics in RTL. In this chapter, we compute probabilistic delay invariants with variations in input patterns being the primary source of statistics [80]. We also show how the SHARPE methodology can be extended to consider process variations as the primary source of statistics. In this context, we interpret the steps in our methodology in a different manner.

In order to restrict timing analysis to RTL, we obtain delay macromodels [17], [18] from the gate level. With these macromodels, we introduce awareness in RTL regarding the underlying statistics in timing. We use these macromodels with our SHARPE methodology and compute probabilistic invariants with respect to timing in RTL.

We illustrate our SHARPE methodology on a variety of data-intensive RTL designs such as the ALU of an OR1200 processor, some non-trivial components of communication systems and a few high-level synthesis benchmark designs.

5.1.3 Chapter organization

The rest of this chapter is organized as follows. In Section 5.2, we describe the macromodels that we obtain for modeling delay in RTL. We consider input pattern variations as the source of statistics in timing. In Section 5.3, we present the experimental results demonstrating the accuracy and scalability of our SHARPE methodology. In Section 5.4, we describe how our methodology can be used to compute the probabilistic distribution of RTL path delays. In Section 5.5, we show how our approach can be used to estimate the performance of better-than-worst-case-design strategies such as timing speculation. In Section 5.6, we illustrate how the SHARPE methodology can be extended for performing statistical timing analysis in the context of process variations.

5.2 Delay macromodeling

In this chapter, we consider a "floating" delay model in which delay depends only on present input values [26]. With a minor modification in the SHARPE methodology (Chapter 3), delay models that depend on both present and previous input values can also be used.

We construct delay macromodels in order to propagate the notion of delay from the lower levels to the higher RTL. We consider RTL blocks/modules (Section 2.3) as the basic units for delay computation. A module usually performs one or more independent functions and has inputs and outputs defined. Every statement in a module uses RTL operators (addition, subtraction, bitwise operators etc).

To restrict the timing analysis to RTL, we define delay in terms of the RTL statements of interest. In order to find delay of a Verilog module (RTL block), we need a delay characterization of all the statements that appear in the module. The delay of a statement depends on the operation that it performs and the values of the operands (Definition 6). For every RTL operator, we determine a function that computes the delay as a function of the values of its operands. We refer to this function as a *delay macromodel* for that operator.

Constructing a library of macromodels

In order to obtain delay macromodels, we first synthesize each RTL operator into logic gates and find delay of this implementation for various input transition patterns by gate-level simulation. We tabulate the delays per input pattern of this implementation. In order to abstract this tabular information into a higher level, we need to find a function that estimates the delay of the operator as a function of operand transitions. We arrive at polynomial functions that predict the delay of an output with small average error for each RTL operator.

For instance, for a ripple-carry adder implementation of the RTL add operator, we determine a Boolean function to reflect the number of bits that "ripple" through to the most significant bit (MSB). A 5^{th} order polynomial function predicts delay at the MSB as a function of the *ripples*, with very small average error. This is the delay macromodel of the ripple carry implementation of the add operator. We consider only MSB for this function definition since it has most impact. In Section 5.3, we obtain such functions for other operators as well.

We obtain the delay for multiple gate-level implementations for every operator. In order to model the effects of synthesis optimizations, we consider three scenarios: Optimization in 1) logic only (technology independent), 2) delay only (technology dependent) and 3) both logic and delay. For each scenario, we perform the the corresponding type of optimization on the gate-level circuit of the RTL operator. We then extract the macromodel, which we refer to as the *optimized macromodel*, from this optimized gate-level circuit. If more scenarios are considered, a richer library of macromodels can be constructed.

This macromodeling and delay characterization is a one-time effort and can be done offline for a given technology library. We use the NANGATE 45nm Open Cell Library for obtaining the delays for the gate-level implementations.

Although we present simplistic delay macromodels in this paper, we demonstrate that they provide RTL estimates that are within 20% of the gate-level estimates. However, the SHARPE methodology is not restricted to these macromodels. Rigorous regression-based techniques [81] can be employed to obtain more complex macromodels that can be used in our methodology.

Delay of an RTL block

We model the delay of a Verilog statement that assigns the output to a block. We consider *blocking* as well as *non-blocking* assignments. Blocking assignments assign the value in RHS in the current cycle to the LHS, in the same cycle. Non-blocking assignments postpone this assignment to the next cycle.

The *delay* of an RTL assignment statement is the time taken from the (rising) clock edge for the effect of the statement execution to be observed. We consider the rising edge of clock as the reference point for measuring delay. In the following code segment, the non-blocking assignment to F is performed with the values of D and E evaluated synchronously at the rising edge of clock.

```
always @(posedge clk)
B <= A;D <= C & B;F <= D + E;
```

Consider the following set of blocking assignments

```
always @(posedge clk)
B = ;D = C & B;F = D + E;
```

The assignment to F is "blocked" till the assignment to D is completed. The AND gate skews the arrival time of operand D with respect to E. In this work, we make an approximation used by high-level SSTA [7],[78],[79] to deal with this situation. If there is a transition at the gate output, we assume that the signal arrival time T_{out} at the output is $T_{out} = T_{op} + max(T_{in1}, T_{in2})$ where T_{op} is the operator delay and T_{in1},T_{in2} are the arrival times of the operand signals. The $max()$ function is a coarse approximation to determine signal arrival time. More sophisticated functions are used at the gate-level and can easily be used with our technique too.

Let T be a timing constraint at the output of a design. We use the library functions obtained using delay macromodeling to express delay at the output as a function $f_P(V)$ defined over RTL variables V. We wish to compute the probability of input vectors that satisfy $f_P(V) < T$.

Definition 14. In the presence of input variations, the probability that the delay at an output signal meets a specified timing constraint is defined as *critical probability.* Critical probability is the timing related invariant that we are interested in.

Probabilistic timing analysis is most meaningful for datapath where timing violations can be tolerated. The PMFs of control signals (combinational and sequential) play a role in determining the delay of the datapath. Therefore, we compute PMFs for both combinational (Section 3.3) as well as sequential signals (Section 3.4). However, we confine statistical timing analysis to combinational, datapath signals.

5.3 Experimental results: With input pattern variations

We perform all our experiments on a 3 GHz, 3.25 GB machine.

5.3.1 Comparing techniques for formal probabilistic analysis of combinational designs

For designs that are purely combinational, formal probabilistic analysis can be performed by exhaustively analyzing all possible input vectors (Section 3.3). However, as an alternative to this *exhaustive input vector analysis* approach, we can manipulate PRISM to perform this analysis by modeling combinational designs as RTL DTMCs (Section 3.4.1). We shall refer to this as the *DTMC + PRISM* approach.

- *For combinational designs, a state in the RTL DTMC model is just the present input vector (Definition 9).*

Once the RTL DTMC is defined for a combinational design, the DTMC + PRISM approach is the same as that described for sequential designs (Section 3.4). For combinational designs, it takes two time steps to detect convergence of the RTL DTMC.

PRISM exhaustively analyzes all possible states of the DTMC model. Therefore, the probability computed by PRISM is exactly equal to that computed by exhaustively analyzing all input vectors.

We compare both the approaches by considering an N-bit adder design as a case study. Let A and B represent the N-bit inputs to the adder. Each input can be assigned one of 2^N integer values in the range 0 to 2^N-1 (Section 2.3). Therefore, there are a total of 2^{2N} possible input vectors. Since each input vector is mapped to a unique state, the corresponding RTL DTMC model has 2^{2N} states.

We assume a uniform probability distribution on the inputs A and B. Therefore, each of the 2^{2N} possible input vectors occurs with a probability equal to $\frac{1}{2^{2N}}$.

We wish to compute the probability that A+B= 2^{2N}-2. We know that there is exactly one input vector (A=2^N-1, B=2^N-1) that satisfies this condition. Therefore, the required probability is equal to $\frac{1}{2^{2N}}$. We use this analytical estimate to verify the accuracy of our probabilistic analysis approaches.

Table 5.1: Comparing formal probabilistic analysis approaches for a combinational adder design. We consider two approaches: 1) Exhaustive input vector analysis and 2) DTMC modeling followed by PRISM.

	Exhaustive input vector analysis			DTMC + PRISM		
Adder size (N)	Number of input vectors	Time (sec)	Computed probability	Number of DTMC states	Time (sec)	Computed probability
8-bit	2^{16}	< 0.01	1.53x10^{-5}	2^{16}	0.56	1.53x10^{-5}
10-bit	2^{20}	< 0.01	9.54x10^{-7}	2^{20}	12.75	9.54x10^{-7}
12-bit	2^{24}	0.07	5.96x10^{-8}	2^{24}	298.06	5.96x10^{-8}
14-bit	2^{28}	0.82	3.72x10^{-9}	2^{28}	6274.38	3.72x10^{-9}
16-bit	2^{32}	11.72	2.33x10^{-10}	2^{32}	-	-
18-bit	2^{36}	186.20	1.46x10^{-11}	2^{36}	-	-
20-bit	2^{40}	3800.00	9.09x10^{-13}	2^{40}	-	-

In Table 5.1, we compare the performance of the two approaches for adders of different sizes. For large adder designs (>14-bit), PRISM runs out of memory while constructing the RTL DTMC model. Since exhaustive input vector analysis does not require any state vector to be constructed, such memory bottlenecks can be avoided. Therefore, exhaustive input vector analysis scales better than the DTMC+PRISM approach. The probability computed by both approaches (when both are feasible) are same and exactly equal to $\frac{1}{2^{2N}}$.

The runtime for the DTMC+PRISM approach is dominated by the time PRISM takes to build the state space of the RTL DTMC. Once the state space is constructed, the actual model checking times for the 8-bit, 10-bit and 12-bit adders are only 0.01 sec, 0.7 sec and

7.54 sec, respectively. If there are several invariants that need to be computed using the same RTL DTMC, the model construction overhead can be amortized over them. Therefore, for the smaller adders, both approaches can be viewed to have comparable runtimes.

For very large designs ($N > 20$), the exhaustive analysis also becomes impractical due to large runtimes. The exhaustive input vector analysis is implemented as a software loop that iterates through all possible input vectors. This loop can be trivially parallelized resulting in up to a linear speedup with respect to the number of parallel computing units.

5.3.2 Validating the SHARPE results for an OR1200 processor

We apply our SHARPE methodology to the datapath of the OR1200, an embedded RISC processor. We refer to this RTL design as RTL_1.

We first analyze the accuracy of the probability distributions computed through static analysis. For this, we consider both datapath and a few sequential control signals. We consider a set of signals to serve as primary inputs whose probability distributions we define. We do not explicitly model the 32-bit instruction registers. All signals that are directly driven by the values stored in these registers are included in the set of primary inputs. We assume that primary input signals like rst and mac_stall take on their atypical values (equal to 1) once in 100 million clock cycles. Therefore, we assume that in each clock cycle, these signals are assigned a value of 1 with a probability of 10^{-8}. In the datapath, we consider the output signal ($result$) of the ALU. The data signals a and b, that serve as synchronously arriving inputs to the ALU are modeled as primary inputs with uniform probability distribution.

Table 5.2: Accuracy of PMFs computed using our SHARPE methodology.

	Simulation Length		
Signals	10^2	10^5	10^8
if_freeze, ex_freeze $lsu\_unstall$, $flushpipe$ $extend\_flush$, wb_freeze	100%	100%	12.8%
$result$	43.2%	11.6%	4.5%

We compute the PMFs of both sequential signals and combinational signals using our methodologies described in Section 3.3 and Section 3.4, respectively. We determine that the control signals in Row 1 of Table 5.2 are assigned one of their infrequent values with

Table 5.3: SHARPE analysis of 6-bit ALU (RTL_1).

	Critical Probabilities		
Constraint (% of WCT)	SHARPE	Gate-level simulations	Percentage Error(%)
100%	1.0	1.0	0
90%	0.9992	0.9929	0.63%
80%	0.9990	0.9748	2.50%
70%	0.9772	0.9180	6.45%
60%	0.9531	0.8993	5.98%

a probability of 10^{-8}. We find that the RTL-DTMC models for these sequential signals converge to a steady-state distribution in 10 time steps. We find that the datapath signal *result* is assigned all its 64 possible values with equal probability. Table 5.2 compares the percentage of error between the probability distributions estimated using our SHARPE methodology, against the values obtained through RTL simulations for different lengths.

As we increased the simulation length, the percentage error reduces significantly for both control and datapath signals, implying that our analytical estimation is of high-confidence. With simulation lengths less than 10^8, we find the control signals are not assigned their infrequent value (rare event) even once and hence we record an error of 100% as compared to estimates obtained using the SHARPE methodology.

For timing estimation, we obtain the delay macromodels for shifters (shift-rotators) and 2^K:1 MUX as functions of the delay of basic 2:1 MUX blocks. For bitwise operators, we use the delay models of the corresponding basic blocks (AND, XOR or OR). For a 6-bit ripple carry adder (subtractor), we obtain the macromodel as described in Section IV.A.1. We fit a curve, shown in Figure 5.1, to the simulated delay measurements at the MSB, as a function of the *ripples*. The mean error between the delay predicted by this analytical model and the measured delay is less than 1% of the worst-case timing for the

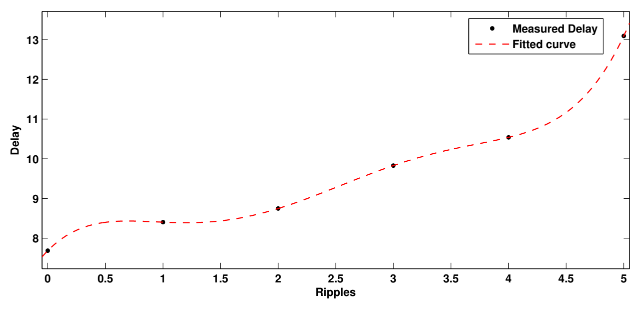


Figure 5.1: Delay macromodel for ripple carry adder.

adder.

We use our SHARPE methodology (Section 3.3) to estimate the timing invariants for the MSB of the 6-bit ALU output signal, *result*. We set the maximum timing delay observed through simulations as the worst-case time (WCT). For different timing constraints, we compare our results (Table 5.3) with the critical probabilities (Definition 14) obtained by gate-level simulations that utilize delay values from the NANGATE 45nm standard cell library. The estimates obtained through both approaches differ only by a maximum of 6.45%. The 6-bit ALU has 3.5x10^6 possible input vectors and the total runtime for our methodology is 0.43 sec. We find that formal probabilistic analysis, without compositional reasoning, is infeasible for ALUs with larger bitwidth.

We now illustrate our technique for input signals with different arrival times as in Section 5.2. We modify the ALU module by introducing a 2:1 MUX after each of the inputs. This introduces independent timing offsets to their arrival times to the original ALU module (RTL_1) and we shall call this modified design, RTL_2. The calculated critical probabilities are listed in the Table 5.4. Even in this case, the SHARPE estimates deviates from simulation-based estimates by less than 6.07%.

Let RTL_2P be a pipelined version of RTL_2, where registers are introduced after both the MUXes that offset the inputs. By definition, we consider only the delay from the register outputs triggered at the rising edge of clock. Therefore, the delay distribution curve of interest in RTL_2P is exactly the same as that of RTL_1 (Table 5.3). This demonstrates how an RTL designer can make use of the probabilistic timing invariants to modify the RTL (introduce pipelining, for example) and obtain a better delay distribution curve.

We consider an alternate gate-level implementation of RTL_1, which has a carry look-ahead adder, with 2 3-bit ripple carry adder blocks. In this case, we reuse the signal PMFs computed for RTL_1 since they are unaffected by the gate-level implementation. We plug in the required delay macromodel and determine the new critical probabilities. It is

Table 5.4: SHARPE analysis of 6-bit ALU (RTL_2).

	Critical Probabilities		
Constraint (% of WCT)	SHARPE	Gate-level simulations	Percentage Error(%)
100%	1.0	1.0	0
90%	0.9990	0.9810	1.83%
80%	0.9890	0.9562	3.43%
70%	0.9531	0.9122	4.48%
60%	0.9529	0.8985	6.07%

observed that only 93.2% of the inputs satisfy the 90% WCT constraint. This information prompts the gate-level designer to choose a ripple carry adder implementation. The use of macromodels in the SHARPE methodology avoids the need to synthesize the design to the gate-level, and thereby eliminates all the associated overheads.

5.3.3 Demonstrating the scalability of our SHARPE methodology

In Chapter 7, we demonstrate that a compositional reasoning approach can be used to scale formal probabilistic timing analysis for a ripple carry adder. We now briefly outline the results of applying this approach. We present a more detailed analysis in Section 7.6.

We use our SHARPE methodology (Section 3.3) to compute delay invariants at the MSB of a ripple carry adder for 90% WCT constraint. The 32-bit adder has 2^{64} possible input vectors and is too large (Definition 10) for efficient, exhaustive analysis. For a 64-bit adder, the number of input vectors can be as high as 2^{128} (approximately 10^{40}). We employ compositional reasoning by splitting the adder design into a set of smaller adders. For example, we can compute the delay invariant for the 64-bit adder in 3.70 sec by decomposing the design into eight 8-bit adders each of which corresponds to a design with only 65,536 possible input vectors.

With the improved scalability afforded by compositional reasoning, we use our SHARPE methodology to analyze a 32-bit ALU corresponding to RTL_1. We present a comparison of the estimated critical probabilities in Table 5.5. SHARPE estimates deviate by a maximum of 7.42% from the simulation-based estimates. The 32-bit ALU has over 10^{20} possible input vectors. Without compositional reasoning, exhaustive input vector analysis is infeasible for this design. However, with decomposition, our SHARPE methodology computes the delay invariants in less than 2 minutes.

Table 5.5: SHARPE analysis of 32-bit ALU (RTL_1).

	Critical Probabilities		
Constraint (% of WCT)	SHARPE	Gate-level simulations	Percentage Error(%)
100%	1.0	1.0	0
90%	0.9763	0.9649	1.17%
80%	0.9652	0.9448	2.11%
70%	0.9434	0.8735	7.41%
60%	0.9248	0.8562	7.42%

In ongoing work, we are incorporating into our SHARPE methodology, several property-preserving reduction techniques [36],[82]. We believe that the scalability of our SHARPE methodology can be further improved through these approaches.

5.3.4 Applying the SHARPE methodology to other types of designs

In Table 5.6, we show that our SHARPE methodology can be applied to the datapath in circuits that are seen in communication systems (Chapter 4). In a Viterbi decoder, $VD\_Path\_Compute$ and $VD\_Path\_Compare$ are the path metric computation and comparison units respectively. We also analyze a MIMO detector and the adder output of a pipelined FIR filter. The input to these designs is data that is corrupted over a communication channel. Therefore, we assume a Gaussian probability distribution [14] for the inputs which is a common model used in communication systems.

We perform timing analysis only for the combinational logic in the datapath. If there are sequential variables at the inputs of this combinational logic, we compute their PMFs using our formal methodology described in Section 3.4. Once the PMFs of the sequential variables are determined, the combinational logic can be analyzed separately using our methodology described in Section 3.3.

5.3.5 Demonstrating robustness to synthesis optimizations

We compare our RTL estimates against those obtained using the synthesized gate-level netlists that have undergone heavy optimizations. We consider three optimization scenarios: Optimization for 1) logic only (technology independent), 2) delay only (technology dependent) and 3) both logic and delay. For each scenario, we obtain our RTL estimates by choosing the corresponding optimized macromodels (Section 5.2).

We derive the optimized macromodels by considering synthesis optimizations on an operator-by-operator basis. Although this does not model global optimizations, a large

Table 5.6: SHARPE methodology for probabilistic timing analysis in communication systems.

	Critical Probabilities		
Circuit	90% WCT	80% WCT	70% WCT
$VD\_Path\_Compute$	0.9690	0.8005	0.6150
$VD\_Path\_Compare$	0.8447	0.7975	0.6749
$MIMO\_Detector$	0.9378	0.8514	0.8019
$FIR\_pipelined$	0.9448	0.8734	0.7637

number of local optimizations are captured by this approach. Therefore, as shown in Table 5.7, our RTL estimates for critical probabilities faithfully follow the gate-level estimates even in the presence of synthesis optimizations.

When delay minimization is performed during synthesis, the path delays of a design are grouped together tightly. Therefore, even a small reduction in the timing constraint can cause timing violations for several paths. For the 32-bit OR1200 ALU with any delay optimization, the critical probabilities (Columns 5 and 7, Table 5.7) degrade sharply as compared to those without optimization (Column 3, Table 5.5). Correspondingly, SHARPE estimates using the optimized macromodels (Columns 4 and 6, Table 5.7) indicate a sharper degradation compared to SHARPE estimates using the unoptimized macromodels (Column 2, Table 5.5).

Table 5.7: SHARPE methodology for 32-bit ALU while considering three gate-level optimization scenarios. SHARPE estimates for critical probabilities (Columns 2,4,6), obtained using optimized versions of the macromodels, closely agree with those obtained by simulating the optimized gate-level netlists (Columns 3,5,7).

	Logic optimization only		Delay optimization only		Logic + Delay optimization	
Constraint (% of WCT)	SHARPE	Gate-level	SHARPE	Gate-level	SHARPE	Gate-level
90%	0.9458	0.9322	0.9441	0.9263	0.9347	0.9198
80%	0.9239	0.9054	0.8741	0.8562	0.8689	0.8523
70%	0.8754	0.8336	0.7972	0.7638	0.7866	0.7457
60%	0.8246	0.7721	0.7305	0.6852	0.7152	0.6634

We consider the communication systems (from Table 5.6) that have been optimized for delay. Table 5.8 demonstrates that SHARPE estimates closely agree with the gate-level estimates for these designs.

Table 5.8: SHARPE estimates (at 90% WCT) for communication systems with delay minimization.

	Critical Probabilities		
Circuit (% of WCT)	SHARPE	Gate-level simulations	Percentage Error(%)
$VD\_Path\_Compute$	0.9010	0.8781	2.54%
$VD\_Path\_Compare$	0.8224	0.7832	4.77%
$MIMO\_Detector$	0.9245	0.8943	3.27%
$FIR\_pipelined$	0.9176	0.8725	4.91%

The use of optimized macromodels is a significant step towards helping our SHARPE methodology achieve a better correlation with downstream estimates. In future work, we could further refine these macromodels and plug them into our methodology to improve the accuracy of the timing estimates.

5.4 Probability distributions of RTL path delays

An RTL design may comprise several paths corresponding to the assignment of a value to a variable. In any given clock cycle, the assignment to a variable is determined by only one path that is selected based on the values assigned to the control variables.

Consider the following RTL fragment:

```
if(select==0)
F <= D + E;
else
F <= D & E;
```

The variable F is assigned a value by one of two possible paths. If the control variable *select* is equal to 0, then the value of $D + E$ is evaluated and assigned to F. In the other path ($select == 1$), F is assigned the value of $D\&E$.

The delay at the variable depends on the delay of the assignments statements in that path. The probabilities with which each path is selected/excited is determined by the PMFs of the control variables. When the statistical distribution of delay at a variable is computed, it is desirable to also document the probability with which each path delay is excited. This can in turn be used to estimate the probability with which each path contributes to a timing violation, as we shall see in Section 5.5. Such path delay distributions can be passed down to gate-level designers to optimize the timing of blocks in only those RTL paths that are "important". In the RTL example, if $select == 0$ occurs with a very low probability, then optimizing the adder design is not of high priority. For better-than-worst-case designs, such selective optimizations can potentially result in significant power/area savings.

Since the PMFs of control variables are computed by the SHARPE methodology, the critical probabilities inherently account for the path excitation probabilities. However, the critical probability is a single value that does not provide any insight regarding the statistical distribution of delays corresponding to different paths. In this work, we use our SHARPE methodology to determine the path delay distributions, and thereby provide a more comprehensive analysis of probabilistic timing behavior in RTL.

As an artifact of our macromodeling technique (Section 5.2), each RTL block has a discrete (and finite) set of possible delays. For example, the delay of a ripple carry adder is a function of the value of the finite variable, *ripples*, which in turn is a function of the values of the adder inputs. Therefore, the SHARPE methodology can use the PMFs of input variables and compute the probabilities (PMFs) with which different delay paths, within such RTL blocks, are excited.

Figure 5.2 shows the path delay distribution of a 6-bit OR1200 ALU (Table 5.3). With the exception of the shortest path delay, all the other path delays occur with negligibly low probabilities. Therefore, the critical probability of this design does not significantly change when the timing constraint is reduced. However, if the ALU is heavily optimized during synthesis, the path delays will be grouped together. As a result, the critical probabilities decrease sharply when the timing constraint is reduced.

Figure 5.3 shows the path delay distribution of $VD\_Path\_Compute$ (Table 5.6). In this design, some of the longer path delays occur with greater probability than the shorter path delays. Therefore, the critical probability of this design significantly reduces when the timing constraint is reduced even by a little. The Gaussian distribution that we assume for the RTL inputs is the main factor that contributes to this behavior.

Figure 5.4 shows the probability delay distribution of an $FIR\_pipelined$ (Table 5.6). The distribution is similar to that of an OR1200 ALU. However, the decrease in probability with increase in delay value is more gradual than for OR1200 ALU. Therefore, the critical probability of this design decreases gracefully when the timing constraint is reduced.

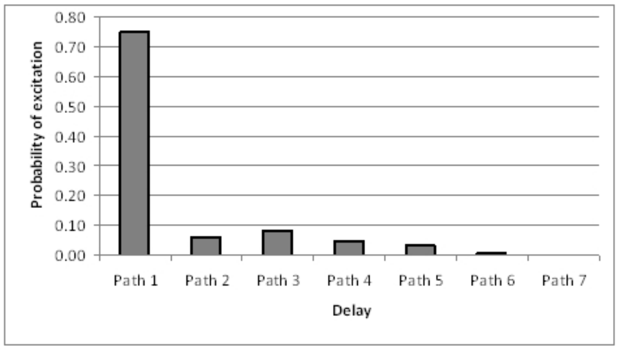


Figure 5.2: Probability distributions for path delays in OR1200 ALU.

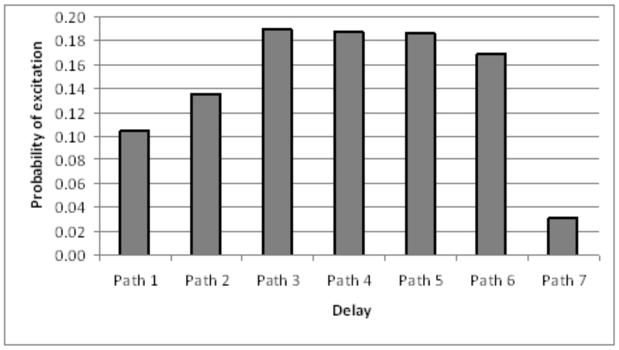


Figure 5.3: Probability distributions for path delays in $VD\_Path\_Compute$.

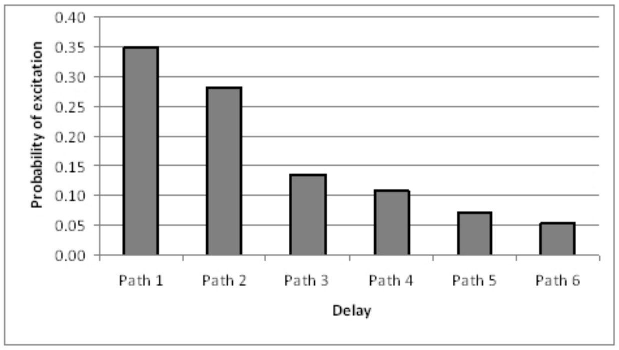


Figure 5.4: Probability distributions for path delays in $FIR\_pipelined$.

5.5 Performance of better-than-worst-case designs

Traditional worst-case design methodologies are insufficient to meet the tight performance constraints of the present day. In these methodologies, circuits are guaranteed to provide error-free operation for all input patterns. However, in circuits where the worst-case input patterns are infrequently applied, the designs are severely under-utilized. In such cases, the circuits can be tuned to a common-case delay and can be allowed to make the timing errors for longer computations. These errors can then be corrected using circuit-level or microarchitecture-level techniques. The average performance of the design may improve or degrade depending on the probability of occurrence of timing errors. This approach for better-than-worst-case design is referred to as timing speculation [2],[3].

We now describe how our SHARPE methodology can be used to estimate the gain in average performance for better-than-worst-case designs.

5.5.1 Timing speculation

In the microarchitecture domain, Razor [2] and Paceline [3] employ timing speculation techniques. In these techniques, additional latches and checker modules are used to detect timing errors. Error correction takes place with a penalty that can range from a few additional clock cycles (Razor) to several hundred cycles (Paceline).

At the circuit level, speculation techniques are employed in *speculative completion* [83] and *telescopic units* [74] to produce variable-latency designs. In these designs, the clock rate is sped-up to match the common case delay, i.e., the delay corresponding to the most frequently occurring input patterns. Longer computations are split over multiple clock cycles.

Circuits designed using speculation techniques have high single-cycle clock frequencies. However, this speed-up is achieved at the expense of extra clock cycles for input patterns that result in a delay longer than the time constraint. If these input patterns occur with a high probability, the average throughput of the system is significantly reduced. An optimal time constraint needs to be found so that the speculation techniques result in an improved throughput for the system.

If the rate of occurrence of the worst-case conditions is known, the performance of such speculation-based designs can be quantified. In other words, the average throughput of the design can be computed if we know the critical probability (Definition 14), i.e., the probability of occurrence of input patterns that finish execution within the specified time constraint. Such estimates can facilitate the selection of an optimal time constraint that maximizes design throughput.

The critical probability of a speculation-based circuit can be estimated by using techniques such as [26]. However, such techniques operate at the gate level and do not scale for large RTL designs. The SHARPE methodology computes critical probabilities in RTL which can then be used to provide early estimates of throughput gain for speculation-based designs.

5.5.2 Estimating performance of speculation-based designs

The gain in average throughput of a design due to speculation-based techniques can be computed as

$$\text{Throughput Gain} = \frac{\frac{1}{T_{clk}^{B}} * (p_{crit} + K * (1 - p_{crit}))}{\frac{1}{T_{clk}}} \tag{5.1}$$

where T_{clk} is the worst-case timing constraint, T^B_{clk} is the better-than-worst-case constraint ($T^B_{clk} < T_{clk}$), p_{crit} is the critical probability and K is the extra number of clock cycles required to correct an error resulting from a timing violation.

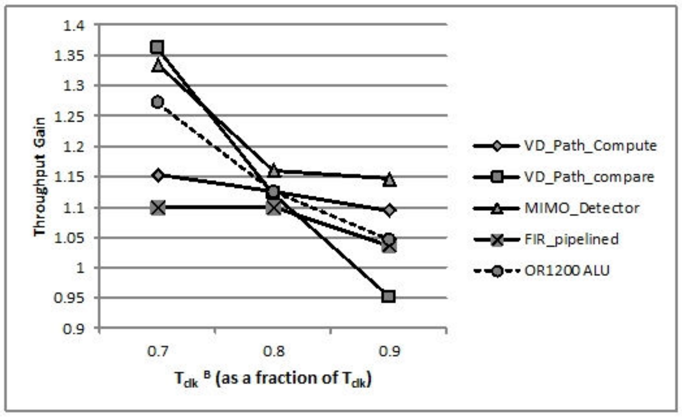


Figure 5.5: Throughput gains for speculation-based designs with an error-correction penalty of K=1 clock cycle.

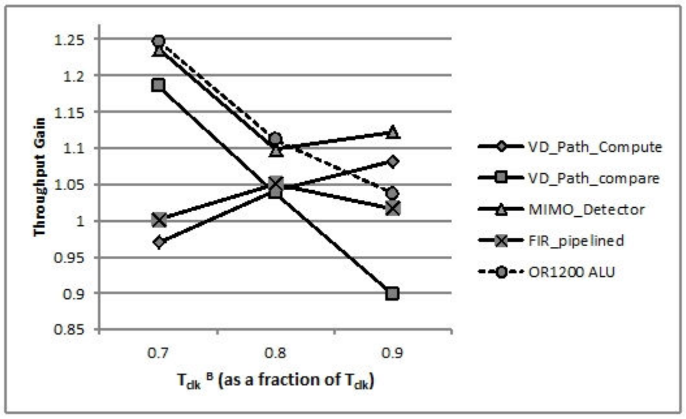


Figure 5.6: Throughput gains for speculation-based designs with an error-correction penalty of K=5 clock cycles.

Figure 5.5 shows the throughput gains for different RTL designs when K=1. For the OR1200 ALU, the gain increases when the timing constraint is reduced. For $FIR\_pipelined$, the gain first increases with a decrease in timing constraint, reaches a maximum and then starts to decrease. Figure 5.6 shows the scenario where K=5. For the range of timing constraints that we observe, the behavior of $VD\_Path\_Compute$ is significantly different from that when K=1 since the gains now decrease when the timing constraint is reduced.

After a point, the throughput of the speculation-based design is even worse (gain < 1) than the worst-case design.

The observed behavior can be explained using the probability distributions in Figures 5.2, 5.3 and 5.4. These figures can be analyzed to determine the set of paths that violate a given timing constraint. Since the figures contain information regarding the excitation probabilities of these paths, we can also estimate the critical probability. The profile (i.e., shape of the curve) of the path delay distributions indicates how sensitive the critical probability is to a decrease in the timing constraint. For example, in the the OR1200 ALU, a small reduction in timing constraint excludes only the longer paths which occur with low probability. Therefore, this design is relatively insensitive to small changes in the timing constraint.

5.6 SHARPE methodology for process variations

We have described our SHARPE methodology by considering input variations as the source of randomness. We now describe how this methodology can be employed for performing statistical static timing analysis (SSTA) in RTL by considering process variations as the primary source of statistics. Although the key steps in our SHARPE methodology (Section 3.3) remain the same, we interpret them differently in the context of process variations.

Our methodology is analogous to the high-level SSTA presented in [7]. We reuse the terminology and delay models described in [7]. However, unlike [7], we apply our methodology to RTL.

5.6.1 Randomness due to process variations

Aggressive technology scaling has drastically reduced the feature size of present day transistors. The corresponding fabrication processes have become significantly complex and introduce significant variations into transistor parameters such as channel length and width [84]. Process variations manifest as variations in delay and must necessarily be considered while performing timing analysis of designs. For example, the delay of an adder can vary by up to 27% of its nominal value [71].

Process variations are commonly classified as *inter-die variations* and *intra-die variations*. Inter-die variations are die to die variations and affect all the devices on the same chip similarly. Intra-die variations correspond to variations within the chip and affect different devices differently on the same chip. However, if two devices are located in close physical proximity on the chip, their variations are highly correlated. In modern day designs, such spatial correlations are significant and cannot be neglected [7].

5.6.2 Delay macromodeling in the presence process variations

We reuse the notion of RTL delay, as defined in Section 5.2. We consider delay of an RTL block by modeling the delay of Verilog statements that assign the output to the block. In Section 5.2, we considered delay variations due to changes in input patterns. Therefore, we obtained delay macromodels that estimate delay of an RTL operator as a function of operand values. However, we now consider variation in delay due to process variations. Therefore, we require a new library of delay macromodels for the RTL operators.

Let D_i represent the delay of an RTL operator i, corresponding to one possible gate-level implementation. Since we no longer wish to model variations with inputs, we consider the worst-case delay with respect to input patterns. However, we still model D_i as a random variable with respect to process variations.

We use a Gaussian model for D_i, which is a widely used model in SSTA. Each Gaussian delay variable can be specified by using the parameters, mean μ_i and variance σ_i^2. However, these parameters model only the individual statistics of the delay of each RTL operator. Due to proximity of physical placement on the chip, the delays of RTL operators may be correlated. Therefore, we must also specify the correlation between the delays of the RTL operators.

We use static analysis to determine a list of all the operators that are used in the RTL source code. Let there be K uses of RTL operators, which may include multiple instances of the same type of operator. We construct a $K \times K$ covariance matrix $\mathbf{C}$ where each element c_{ij} specifies the pairwise correlation between variables D_i and D_j, given by

$$c_{ij} = \{ \begin{array}{cc} \sigma_i^2 & \text{if } i = j \\ \rho_{ij}\sigma_1\sigma_2 & \text{otherwise} \end{array} \tag{5.2}$$

For example, let D_1 and D_2 be two jointly Gaussian [21] random variables with means μ_1 and μ_2 and variances σ_1^2 and σ_2^2, respectively. Let ρ_{12} represent the correlation factor between the two variables. The correlation is completely specified by using the 2x2 covariance matrix $\mathbf{C}$ given by

$$\mathbf{C} = \left[\begin{array}{cc} \sigma_1^2 & \rho_{12}\sigma_1\sigma_2 \\ \rho_{12}\sigma_1\sigma_2 & \sigma_2^2 \end{array} \right] \tag{5.3}$$

In order to construct a delay macromodel D_i, the corresponding RTL operator needs to be first synthesized to obtain a gate-level netlist. Monte Carlo simulations can then be performed on the gate-level netlist to plot a histogram of the set of varying (due to process variations) worst-case delay values at the output. Finally, a Gaussian model can be fit to this histogram and the corresponding values of μ_i and σ_i^2 can be determined.

Realistic estimates of correlation are difficult to obtain. Static analysis of the RTL source code can be used to obtain information regarding the connectivity between the RTL blocks. This information can be used in conjunction with a commercial RTL floorplanning tool in order to estimate the relative physical locations of the RTL blocks on the chip. The correlations factors ρ_{ij} between different RTL operators i and j can then be modeled as a function of the distance between them on the chip [7].

For a given technology, we can construct a library of macromodels for all RTL operators by considering different possible gate-level implementations and synthesis optimizations, as in the case of input variations (Section 5.2). This is a one-time effort and can be done offline.

5.6.3 Decorrelating the delay variables

The set of delay variables D_i are jointly Gaussian [21] and are typically correlated with each other (non-diagonal matrix $\mathbf{C}$). However, for simplicity of probabilistic analysis, we wish to obtain a set of uncorrelated Gaussian random variables. We achieve this by using principal component analysis (PCA), as described in [7].

In PCA, a set of K jointly Gaussian random variables D_i can be expressed in terms of another set of K independent Gaussian random variables, denoted by D'_j. Each variable D_i can be expressed as a linear sum of the variables D'_j given by

$$D_i = \sum_{j=1}^{K} \alpha_{ij} D'_j \tag{5.4}$$

where α_{ij} is a real-valued coefficient. The values of each α_{ij}, along with the mean and variance of the variables D'_j, can be determined by using PCA .

5.6.4 Discretizing the Gaussian random variables

In order to employ our SHARPE methodology (Section 3.3), we require the random variables to be discrete. We "discretize" each delay variable D'_j by considering only a finite set of possible values, as shown in Figure 5.7. We first truncate the Gaussian model to the $\mu - 3\sigma$, $\mu + 3\sigma$ limits. We then determine a set of equally-spaced values in this interval. The probability that a value occurs can be computed from the Gaussian distribution curve. In Figure 5.7, the area of the shaded region under the Gaussian curve represents the probability with which the corresponding discrete value is selected.

In the context of input variations, each input vector (Definition 9) corresponds to a unique set of values assigned to the inputs. For process variations, the discretized variables D'_j are the random variables of interest. Therefore, in this context, the assignment of values to D'_j can be interpreted as the random "input" vector for statistical analysis. Each vector can now be viewed as a unique instantiation of the chip corresponding to a unique set of delay values for the RTL operators.

With increase in the number of RTL operators and the number of discretized values per delay variable, the number of possible random vectors also increases. We mitigate this problem by using a compositional reasoning approach that we present in Chapter 7. Equation 5.4 shows an additive expression which, according to Chapter 7, facilitates an effective employment of our compositional reasoning approach. Such an approach can significantly improve the scalability of our methodology in the context of process variations. However, in this work, we do not further explore this approach in the context of process variations.

We use a simple optimization, similar to the one described in [75], in order to reduce the number of vectors that need to be considered. Consider the delay variables D_i and D_j corresponding to two different RTL operators. If $\mu_i + 3\sigma_i$ is less than $\mu_j - 3\sigma_j$, the variable D_i is highly unlikely to violate the timing constraint when D_j does not. Therefore, D_j dominates D_i and D_i can be discarded from the set of state variables without significantly affecting the accuracy of SHARPE estimates. As a result of this optimization, the total number of vectors can be reduced.

5.6.5 Computing the yield using our SHARPE methodology

We wish to compute the fraction of chips in which the better-than-worst-case timing constraint T^B_{clk} is always satisfied. This is commonly referred to as the yield of the design. We describe our approach by considering the scenario where the RTL contains only non-blocking assignment statements. In this scenario, each assignment operation must be completed within one clock cycle. Therefore, we can express the required invariant, yield, formally as

$$\text{Yield} = \text{Probability}[D_i \leq T^B_{clk}] \ \ \forall i \tag{5.5}$$

Since there is no notion of memory associated with process variations, we treat this analogous to a combinational design. Therefore, we employ formal probabilistic analysis (Section 3.3) which exhaustively analyzes all the random vectors and computes the yield of the design.

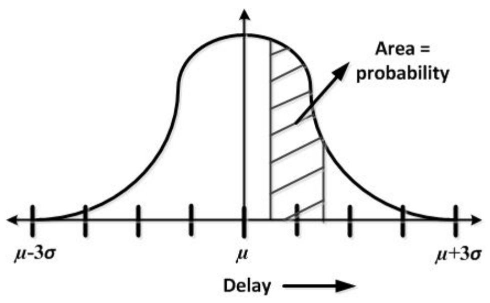


Figure 5.7: Discretizing the Gaussian distribution of a delay variable.

Our methodology can be applied with blocking assignment statements as well. In this case, the D_i in Equation 5.5 is substituted with the composite delay expression (i.e., using $max()$) that is obtained according to the semantics described in Section 5.2. Although an approximation, such delay composition is used in all existing high-level SSTA techniques [7],[78],[79].

5.6.6 Experimental results: With process variations

In [7],[78],[79], the experiments are not intended to demonstrate the accuracy of early statistical estimates in comparison to those obtained at the lower levels. Instead, the authors show that the introduction of variation-awareness at the higher levels can facilitate system exploration. If more accurate macromodels are used, they can provide more accurate timing estimates. In this work, we adopt a similar approach and show that our SHARPE methodology (Section 3.3) can be used to provide statistical timing estimates in RTL.

We wish to use experiments to demonstrate that our methodology can be extended to the context of process variations. For this purpose, we use the values of mean (μ) and variance (σ^2) for adders and multipliers that are given in [78]. We assume a constant value for all pairwise correlation factors (ρ_{ij}). However, any other values of mean, variance and correlation can easily be plugged into our methodology.

We apply our methodology to fft, filter and Kalman, which are RTL designs from the suite of high-level synthesis benchmarks [85]. For each design, we compute the yield for three better-than-worst-case timing specifications: T_{95}, T_{90} and T_{85}. T_{95}, T_{90} and T_{85} are estimates of timing constraints for which the yield of the design is 95%, 90% and 85%, respectively, when spatial correlation is neglected. We estimate these constraints by analytically solving the closed-form expressions for a set of independent Gaussian variables. We consider 25 discrete values for each delay variable. In this work, we assume a constant correlation factor ρ between all pairs of RTL operators in a design.

Table 5.9: SHARPE estimates of yield for *HLS95* benchmarks.

		Yield		
Circuit	Correlation	T_{95}	T_{90}	T_{85}
fft	ρ=0	93.45%	88.38%	84.74%
	ρ=0.1	95.18%	89.64%	86.23%
	ρ=0.5	95.84%	91.52%	87.65%
filter	ρ=0	96.54%	93.45%	88.39%
	ρ=0.1	96.07%	92.14%	86.63%
	ρ=0.5	96.60%	93.39%	89.19%
Kalman	ρ=0	95.70%	91.88%	85.70%
	ρ=0.1	95.13%	91.74%	85.07%
	ρ=0.5	95.94%	92.90%	88.27%

Table 5.9 shows the values of yield that we estimate using our SHARPE methodology. For each design, we compute the yield for three values of correlation factors: ρ=0 (no spatial correlation), ρ=0.1 and ρ=0.5. For ρ=0, we see that the SHARPE estimates of yield are close to 95%, 90% and 85%, which are the values that we assumed while setting the corresponding timing constraints. The slight deviation in the yield estimates is due to the loss of accuracy resulting from discretization. This can be reduced by considering a larger set of discrete values for the delay variables. However, this fine-grained discretization results in a larger set of vectors that need to be analyzed. As ρ increases, we observe that the change in yield is not negligible. This confirms that ignoring the spatial correlation, as in [78], results in inaccurate estimates of yield.

5.6.7 Comparing the SHARPE methodology with existing high-level SSTA

High-level SSTA, such as the one described in [7], that use continuous-valued Gaussian variables can be applied to RTL by using the same notion of RTL delay that we define in this work. In fact, we use such analyses to verify that the SHARPE estimates in Table 5.9 are reasonably accurate despite the discretization of the Gaussian variables. In terms of runtime and scalability, we do not intend our methodology to be a replacement for such analytical techniques in the context of process variations. Instead, in this work, we aim to illustrate the broad scope of our SHARPE methodology by showing that its core engine can be utilized in the context of both process variations as well as input pattern variations.

Our SHARPE methodology for process variations broadly resembles the high-level timing analysis that is described in [7]. However, we use discrete-valued delay variables instead of continuous-valued variables and the engine (probabilistic model checking) that we use for computations is entirely different. Moreover, the technique in [7] is not defined formally for RTL designs. The authors in [78] use discrete-valued variables to simplify the computations involved in statistical timing analysis. However, they ignore the correlations due to spatial proximity of blocks and assume, incorrectly, that the delays are independent. Moreover, none of these high-level techniques can be extended for performing timing verification in the presence of input variations.

5.7 Chapter summary

In this chapter, we have applied our SHARPE methodology for rigorously analyzing statistical timing at the higher levels of system design. The SHARPE methodology can be viewed as an integrated solution for statistical static timing analysis RTL by considering either input variations or process variations as the source of statistics in timing. We also plan to derive more complex delay macromodels so that our methodology accurately estimates the effects on timing that arise in post-synthesis netlists, such as loading of RTL blocks, wire delays and other parasitics.

CHAPTER 6

SHARPE FOR AGING ANALYSIS IN RTL

6.1 Introduction

In this chapter, we apply our SHARPE methodology to estimate aging-induced delay degradation in RTL. The key steps in our methodology, described in Chapter 3, remain the same as in the case of probabilistic timing analysis (Chapter 5). The main difference is in the macromodels that we derive and use with our methodology to shift aging analysis from the lower levels to RTL.

Aging effects depend on the probability distributions of the hardware signals. We have shown that our SHARPE methodology can effectively compute PMFs of variables in both sequential (Chapter 4) and combinational designs (Chapter 5). Therefore, our methodology can be applied to both control logic as well as datapath. However, our methodology is most effective for data-intensive designs since the RTL datapath operators abstract away the details of the complex gate-level implementations. In this chapter, we employ our SHARPE methodology (Section 3.3) for aging analysis of combinational designs.

6.1.1 Effect of signal statistics on aging

Negative bias temperature instability (NBTI) [27] in PMOS transistors has become a significant concern in the design of reliable digital circuits. NBTI affects PMOS transistors in a gate when a negative voltage (logic value 0) is applied at the gate input, causing an increase in threshold voltage which in turn lowers speed of the gate. Circuit simulations show that NBTI effects over a lifetime of 10 years can degrade delay by up to 10% which may potentially violate timing constraints and hence result in a circuit failure [28],[29]. Typically, such failures are detected only while performing extensive pass/fail checks on the circuit after most of the timing closure is achieved. However, at this stage, it is often too late to revise circuit topology or architecture in order to improve circuit reliability by mitigating these failures. Therefore, it is desirable to have a methodology that can provide estimates of delay degradation in earlier stages of the design flow.

The extent to which the delay of each PMOS transistor degrades is a function of the duration for which the transistor is in “stress” mode, i.e., the input is held at a logic value 0 [27]. There are several timing analysis techniques at the transistor level and the gate level [30],[31],[32],[33],[34] that predict delay degradation of a circuit by computing signal probabilities of the circuit nodes. However, if such analyses are present in RTL, the signal probabilities that are computed would be representative of the actual usage statistics/workload of the design. As a consequence, RTL analysis provides a wider perspective of the effects of NBTI since the delay degradation of a small block is estimated in the context of a larger operating environment specified by the RTL. To the best of our knowledge, there is no technique that estimates delay degradation of designs in RTL.

6.1.2 Using our SHARPE methodology to shift aging analysis to RTL

RTL estimation of delay degradation maps the effects of NBTI, an artifact of lower physical level elements, to higher level design choices. Typically, such mapping involves a loss in lower-level information, and therefore RTL estimates are distant approximations. In this chapter, we employ our SHARPE methodology for analyzing aging in RTL. We demonstrate that, with intelligent mapping, RTL analysis can be made up to 18.2x faster (Section 6.4) than gate-level analysis while providing reasonably accurate estimates. Such RTL analysis can be used as an upstream “triage” tool that trades off accuracy for speed in order to provide early estimates of delay degradation. Analysis at the higher level emphasizes on predictability over accuracy. The use of RTL aging analysis can avoid “surprises” at the lower levels where it is too late to implement major revisions in the design. This approach could detect major aging issues early on in the design cycle and relegate finer tuning to lower levels of design.

With our methodology, we obtain RTL estimates of delay degradation based on the RTL signal probabilities. We compute these probabilities entirely in RTL due to their independence from the gate-level implementation. This is significantly faster than gate-level aging analysis that propagates probabilities entirely at the gate-level.

In order to map NBTI effects to RTL within specified bounds of error, we construct macromodels. Our macromodels are subcircuits of the gate-level implementations of RTL operators. We estimate the delay degradation of an RTL operator by propagating the computed RTL signal probabilities through all the gates in the corresponding macromodel.

We demonstrate the effectiveness of our methodology by computing accurate estimates of delay degradation ($<$10% error) for several data-intensive RTL benchmark designs

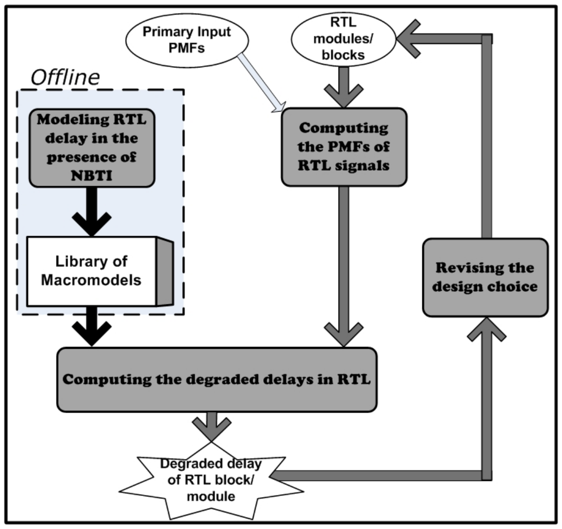

Primary Input PMFs
RTL modules/ blocks
Offline
Modeling RTL delay in the presence of NBTI
Computing the PMFs of RTL signals
Library of Macromodels
Revising the design choice
Computing the degraded delays in RTL
Degraded delay of RTL block/ module

Figure 6.1: Block diagram depicting the application of our SHARPE methodology for aging analysis in RTL.

with up to 18.2x runtime speedup as compared to estimation using gate-level simulations (Section 6.4).

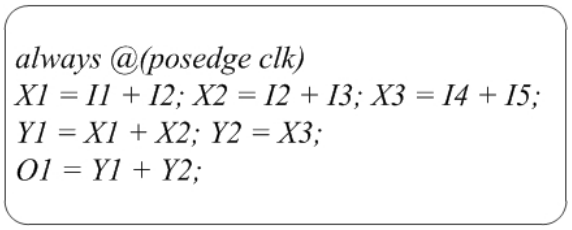
6.1.3 Chapter organization

The rest of this chapter is organized as follows. In Section 6.2, we describe the macromodels that we construct to shift aging analysis to RTL. We show how RTL signal statistics can be used with these macromodels to obtain RTL estimates of delay degradation. In Section 6.3, we describe a methodology to improve the scalability of SHARPE for aging analysis. In Section 6.4, we demonstrate the effectiveness of our SHARPE methodology by estimating the delay degradation for several RTL benchmark designs.

6.2 SHARPE methodology for aging analysis in RTL

We describe our methodology (Figure 6.1) by using the RTL code fragment in Figure 6.2 as a running example. We wish to compute the degraded delay at the output *O1*.

To restrict the aging analysis to RTL, we define degraded delays in terms of the RTL statements of interest. The degraded delay of a statement depends on the operation that it performs and the PMFs of the operands (elements of set $RHS(v)$). In order to find the degraded delay of a RTL block/module, we need a characterization of delay degradation



```
always @(posedge clk)
X1 = I1 + I2; X2 = I2 + I3; X3 = I4 + I5;
Y1 = X1 + X2; Y2 = X3;
O1 = Y1 + Y2;
```

Figure 6.2: An example RTL block where *I1* to *I5* are the RTL input signals and *O1* is the RTL output. *X1* to *X3* and *Y1*, *Y2* are intermediate RTL signals. In general, the RTL may also contain sequential variables. We consider sequential variables while computing the signal probabilities. However, we estimate RTL delay degradation only for the datapath.

for all the statements that appear in the module.

A summary of the steps in our methodology (Figure 6.1) is as follows.

- For every RTL operator, we construct a gate-level circuit that can be used to estimate the degraded delay based on the PMFs of the operands. We refer to this circuit as a *macromodel* for that operator.
- We compute exact PMFs for RTL operands by employing formal probabilistic analysis (Chapter 3) on the RTL design.
- Finally, we compute the degraded delays of each RTL statement by using the PMFs of the corresponding operands and the macromodel for the corresponding operator.

We now describe these steps in detail.

6.2.1 Modeling RTL delay in the presence of NBTI

We construct a macromodels for each RTL operator. A macromodel for an RTL operator is a gate-level circuit with the RTL operands as inputs. We now define necessary terminology and describe our technique for constructing macromodels.

Definition 15. A *logic cone* of a gate input node i_g is the set of all gates and nodes of circuit C that can affect the value of i_g. A logic cone for a path P is the union of the logic cones of all the gate input nodes in P.

Definition 16. A subcircuit of C *contains* the path P if all the gates and nodes in the logic cone of P are present in the subcircuit.

Definition 17. A subcircuit of C is *minimal* with respect to a set of paths Π if it contains only those gates and nodes from C that are present in the logic cone of atleast one path $P \in \Pi$.

Let circuit C be a possible gate-level implementation that is obtained by synthesizing the RTL operator. The corresponding RTL operands are the inputs of C. Let Π be the set of all simple paths of C. The delay of a circuit C is defined by the delay of its slowest path $P_{crit} \in \Pi$, i.e., the critical path. Therefore, P_{crit} determines the delay of the RTL operator.

The circuit C can be used to accurately estimate the degraded delay of the corresponding RTL operator by computing the signal probabilities for all internal nodes [33]. The complexity of accurate probability propagation is known to grow exponentially with the size of the circuit. In order to obtain quick estimates, we consider only a smaller subcircuit of C. We use this subcircuit as the macromodel M for the RTL operator.

In order to compute the delay degradation of a path P in circuit C, we need to propagate signal probabilities from the circuit inputs to the inputs nodes of all the gates on the path [33]. However, we can also achieve this by considering a subcircuit of C which contains (Definition 16) the path P. We construct the macromodel M by extracting a subcircuit that contains a set of relevant paths in C.

Let $P_{crit} \in \Pi$ be the critical path that determines the delay of C in the absence of NBTI. Therefore, a macromodel M that contains P_{crit} can be used to estimate the delay of C in the absence of NBTI. However, in the presence of NBTI effects, the delay of all paths $\in \Pi$ degrade to different extents. Therefore, the new critical path $P'_{crit} \in \Pi$ of circuit C can potentially be different from P_{crit}. Since we intend our macromodels to provide reliable estimates of degraded delay of C, M should ideally contain the path P'_{crit}.

The delay degradation of paths depend on the signal probabilities of the internal nodes which in turn depend on the signal probabilities of the circuit inputs. These input signal probabilities are determined as a function of the design workload which is available only during runtime. Therefore, it is difficult to identify P'_{crit} apriori and include it in the macromodel M.

Let M be a subcircuit of C that contains a subset of K paths from Π, where K is a user-defined value. Consider that M does not contain P'_{crit} for the given value of K. Let $P_{approx} \in \Pi$ be the critical path of M in the presence of NBTI effects. If the degraded delay of $P_{approx} \in \Pi$ closely approximates that of P'_{crit}, we can still use M to obtain reliable estimates of the degraded delay of C. This forms the basis of our macromodeling technique.

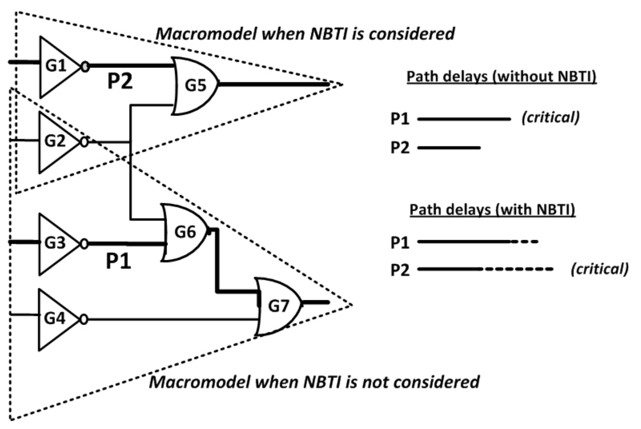

Macromodel when NBTI is considered
G1
P2
G5
G2
G3
G6
P1
G4
G7
Macromodel when NBTI is not considered
Path delays (without NBTI)
P1
(critical)
P2
Path delays (with NBTI)
P1
P2
(critical)

Figure 6.3: Structurally different macromodels obtained using different strategies.

Definition 18. For each gate-level implementation C of an RTL operator, the corresponding *macromodel* is the smallest subcircuit that contains (Definition 16) the set of K slowest paths in C.

NBTI effects may significantly alter the delay distribution of the paths in a circuit. We present two strategies for constructing macromodels by selecting the K slowest paths in C either 1) without NBTI effects or 2) with NBTI effects.

Selecting K slowest paths without NBTI effects

In this strategy, we select the K slowest paths in Π in the absence of NBTI-induced delay degradation. Let Π^K denote this set of K paths. From Definition 18, our macromodel M is a subcircuit of C that contains (Definition 16) all the paths in Π^K and is minimal (Definition 17) with respect to Π^K.

In order to construct M, we first determine the paths in Π^K by performing gate-level static timing analysis on C without considering NBTI effects. We then obtain M by pruning C and removing all gates and nodes that are not in the logic cones of any path in Π^K.

The delay degradation due to NBTI is typically less than 10%. Therefore, it is reasonable to expect that P'_{crit} is among the slowest paths of C even in the absence of NBTI effects. For small values of K, only a small fraction of the paths in Π is present in Π^K. Therefore, M may not contain the path P'_{crit}, i.e., $P'_{crit} \notin \Pi^K$. As the value of K increases, the likelihood of $P'_{crit} \in \Pi^K$ increases. The likelihood of P_{approx} closely approximating P'_{crit} also increases with K. The accuracy of the degradation estimates provided by M depend on how closely P_{approx} approximates the degradation of P'_{crit}.

Since we wish to obtain only an estimate of the degraded delay by using M, it is not necessary to choose K large enough such that M contains P'_{crit}, i.e., $P'_{crit} \in \Pi^K$. Instead, by setting a smaller value of K, we can construct M that provides estimates (using P_{approx}) within a specified range of error. In the limiting case, for very large values of K, M is same as the original circuit C.

Selecting K slowest paths with NBTI effects

In this strategy, we obtain the set of slowest paths Π^K by using static timing analysis on circuit C in the presence of NBTI effects. Since we do not know the actual signal probabilities of the circuit inputs during macromodeling, we assume that they are all equal to 0.5. The advantage of this approach is that the K slowest paths are selected after incorporating some amount of degradation. In certain cases, this could improve the likelihood of $P'_{crit} \in \Pi^K$. Therefore, for the same value of K, the macromodel obtained by using this approach may provide more reliable estimates than the macromodel obtained without considering NBTI effects.

Figure 6.3 shows macromodels that are obtained for an example circuit by using both the methods with K=1. Path **P1** is critical in the absence of NBTI effects whereas path **P2** is the slowest path when NBTI degradation is considered. In this example, the two strategies result in macromodels that are structurally different.

For including NBTI effects during macromodel construction, we assume the signal probabilities to be equal to 0.5. However, the signal probabilities computed at runtime may be significantly different. Since there is an uncertainty in both macromodeling strategies, it is difficult to categorically state that one is better than the other. In Section 6.4, we empirically compare the two strategies and show that both of them are effective.

We select a macromodeling strategy for every RTL operator and construct a library of macromodels. For each macromodel, we specify K (i.e., the size of M) to trade off between the accuracy of the estimates and the complexity of computation. This macromodeling is a one-time effort and can be done offline for a given technology library. We obtain the synthesized gate-level implementations by using a library that is constructed based on the PTM 45nm model files [86].

We obtain a distinct macromodel corresponding to each possible gate-level implementation of an RTL operator. While estimating RTL delay degradation (Section 6.2.3), the user can guide the selection of macromodels by specifying the gate-level implementation for each RTL operator. In the absence of such guidance, our methodology will select the default macromodel for each operator.

6.2.2 Computing the PMFs of RTL signals

In order to compute the degraded delay of *O1 = Y1 + Y2*, we require the PMFs of $Y1$ and $Y2$. We can symbolically express $Y1$ and $Y2$ in terms of the RTL inputs as $I1+2*I2+I3$ and $I4 + I5$, respectively (Definition 8). Therefore, once the PMFs of the input signals are given, we can derive the exact PMFs of $Y1$ and $Y2$ by using these expressions.

In Chapter 3, we described a systematic technique to derive the PMFs for all variables of interest. We statically analyze (Section 3.2) the Verilog program in order to derive the support (Definition 7) and signal function(Definition 8) of each variable v of interest. We use this information and perform formal probabilistic analysis to exactly compute the PMFs of all the variables of interest. The formal probabilistic analysis can be applied to both combinational (Section 3.3) and sequential designs (Section 3.4).

In the combinational RTL example, $RHS(Y1)=\{X1, X2\}$. Since $X1$ and $X2$ are not RTL inputs, we step the design backwards once more. We find that the supports of $Y1$ and $Y2$ are $\{I1, I2, I3\}$ and $\{I4, I5\}$, respectively. The corresponding signal functions for $Y1$ and $Y2$ are $I1 + 2 * I2 + I3$ and $I4 + I5$, respectively. For any value i that can be assigned to $Y1$, the probability of $Y1 = i$ is equal to the probability of input vectors (Definition 9) that satisfy the condition $I1+2*I2+I3 = i$. We repeat this for all possible i and obtain the PMF of $Y1$.

6.2.3 Computing the degraded delays in RTL

We compute the degraded delays in RTL on a statement-by-statement basis. For each RTL statement, we compute the degraded delay of the appropriate macromodel for the corresponding RTL operator. We use the PMFs of the corresponding operands and propagate them through the gates in the macromodel circuit. For each gate, the degraded delay can be analytically computed as a function of the signal probabilities of the gate inputs [33]. We then employ gate-level static timing analysis using these degraded gate delays and compute the degraded delay of the macromodel.

While propagating the signal probabilities through the macromodel, we ignore the correlations among the internal gate nodes. This approximation significantly reduces the complexity of signal probability computation. For small gate-level circuits such as our macromodels, the delay degradation estimates that are obtained with this approximation are not far from the estimates obtained using exact probabilities.

The RTL signal probabilities that we compute are exact and account for all correlations among RTL signals. We approximate the signal probabilities only at the nodes within each macromodel. In our methodology, the computation of RTL signal probabilities is separated from the computation of the signal probabilities of the internal nodes of the macromodels. Therefore, the approximation does not affect the exactness of the RTL signal probabilities. As a result, we restrict the approximation errors to the small macromodel circuits and prevent them from propagating throughout the RTL design. In other words, we localize the approximate errors to within each macromodel. In Section 6.4, we demonstrate that our methodology does not incur significant loss in estimation accuracy.

Once we obtain the degraded delays for all statements in the RTL block, we compose them together and compute the degraded delay of the RTL block. We reuse the semantics, described in Section 5.2, for composing delays of RTL statements.

The *delay of an RTL assignment statement* is the time taken from the (rising) clock edge for the effect of the statement execution to be observed. We consider the rising edge of clock as the reference point for measuring delay.

In the RTL example (Figure 6.2), the values of the inputs $I1$ to $I5$ are available synchronously at the rising edge of clock. However, the assignment to $O1$ is stabilized (completed) only when the assignments to both $Y1$ and $Y2$ are completed. The arrival times for $Y1$ and $Y2$ may be skewed. In this work, we make an approximation to deal with this situation. We assume that the signal arrival time T_{O1} at the output is $T_{O1} = T_{add} + max(T_{Y1},\ T_{Y2})$ where T_{add} is the operator delay and T_{Y1}, T_{Y2} are the arrival times of the operand signals.

6.2.4 Revising the design choice

The estimates of degraded delays computed are provided as feedback to the RTL designer for revising the design. This process can be iterative until the reliability constraints are satisfied. The use of macromodels avoids the need to synthesize the design to the gate-level in each iteration.

6.3 Improving the scalability of our methodology

The use of formal probabilistic analysis can potentially restrict our analysis to smaller designs. In Chapter 7, we present a compositional reasoning approach for probabilistic verification of RTL designs. We now briefly outline a related strategy to significantly improve the scalability of our methodology.

Definition 19. RTL signals v_1 and v_2 are independent if all the signals in the support of v_1 are independent of the signals in the support of v_2. RTL blocks B_1 and B_2 are

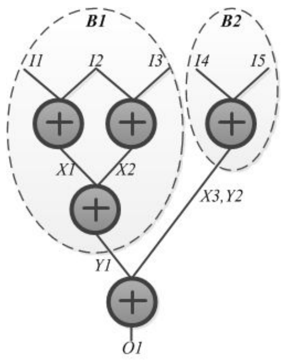


Figure 6.4: In the example RTL block B, $Y1$ and $Y2$ are independent signals. Therefore, B can be decomposed into two independent blocks, $B1$ and $B2$.

independent of each other if there exists no signal in block B_1 that is correlated with a signal in block B_2.

We wish to compute the PMF for the output signal v of an RTL block B. Consider that B can be structurally decomposed into a into a set of independent blocks B_i (Definition 19). We first compute the PMFs for the output signals v_i of these blocks by using their corresponding RTL-DTMCs. We then "compose" the PMFs of all v_i and compute the PMF of v. Instead of performing formal probabilistic analysis on the entire block B, we now verify only the smaller blocks B_i. This forms the basis of our compositional reasoning strategy.

We modify our static analysis algorithm, described in Section 3.2, in order to determine the set of independent RTL signals v_i in RTL block B. These v_i can be viewed as output signals of a set of independent RTL block B_i. Therefore, the modified static analysis algorithm decomposes the original RTL block B into a set of independent blocks B_i.

We illustrate the compositional reasoning approach by using the RTL example from Figure 6.2. Figure 6.4 shows a block diagram that depicts the decomposition of the example RTL block. We wish to compute the PMF of the output signal $O1$. Through static analysis, we determine that $O1{=}Y1+Y2{=}I1+2*I2+I3+I4+I5$. In the absence of compositional reasoning, we compute the PMF of $O1$ by analyzing all input vectors (Section 3.3) with respect to the support $\{I1, I2, I3, I4, I5\}$. Consider that all the signals in the RTL example are of 10 bits each. Therefore, there are 2^{50} possible input vectors that need to be analyzed. With compositional reasoning, we first compute the PMFs of $Y1$ and $Y2$. The number of input vectors that need to be analyzed for computing the PMFs of $Y1$ and $Y2$ are 2^{30} and 2^{20}, respectively. We then treat $\{Y1, Y2\}$ as the independent support for $O1$ and compute the PMF of $O1$ by using $O1{=}Y1+Y2$. This computation now requires analysis of only 2^{20} vectors (values of $\{Y1, Y2\}$).

6.4 Experimental results

We perform our experiments on a 3 GHz, 3.25 GB machine (Intel Core2 Duo CPU). We obtain the synthesized gate-level implementations by using a library that is constructed based on the PTM 45nm model files [86].

We perform the following experiments:

- We validate our macromodels by evaluating them across several gate-level benchmark circuits.
- We use our macromodels and obtain RTL estimates of degradation for several benchmark RTL designs. We demonstrate the effectiveness of our RTL analysis compared to simulation-based gate-level analysis.
- We analyze the NBTI degradation of RTL blocks during 'SLEEP' mode.

Validating the macromodels

We constructed macromodels for several gate-level MCNC benchmark circuits. We estimated the degraded delay for five sets of input patterns. Each each of these patterns corresponds to a different probability distribution. The confidence of validation can be increased by using more input pattern sets. We determine the average error of our macromodel estimates as compared to those obtained by simulating the full gate-level circuit up to 10^5 cycles.

Figure 6.5 shows the accuracy of the macromodels that we obtain without including NBTI effects. Our macromodels contain only a subset of the paths from the original circuit. Moreover, we ignore internal correlations while propagating signal probabilities through the macromodel (Section 6.2.3). Both these factors may contribute to the macromodel estimation error. However, for large K (more paths included in the macromodel), we observe that the estimation error can be removed almost completely. We thus confirm that estimation error due to ignoring internal correlations is negligible. Typically, NBTI effects cause the delay to degrade by about 10%. Therefore, we wish to keep the macromodel estimation error **less than 2%**, i.e., much smaller than 10%.

As K increases, more gates from the gate-level implementation are included in the macromodel (Figure 6.6). Even with K=1, more than one path from the original circuit is contained in the macromodel due to the inclusion of the corresponding logic cones. The paths in `Multiplier16` are highly correlated. Therefore, the macromodel retains almost

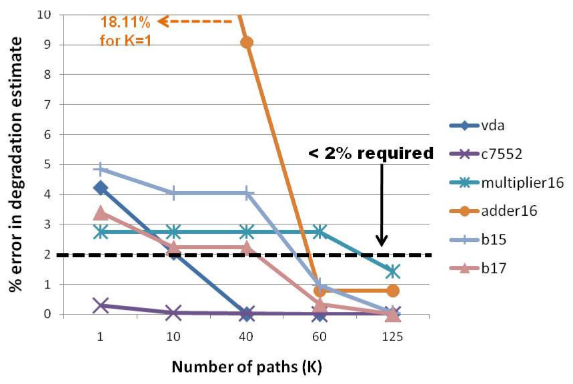


Figure 6.5: Accuracy of the macromodels (without NBTI effects) as a function of K. We select large K such that error in degradation estimates is less than 2%.

95% of the gates from the original circuit even for K=1. Even when K is increased to 40, the size of the macromodel does not increase since it already contains the 40 slowest paths.

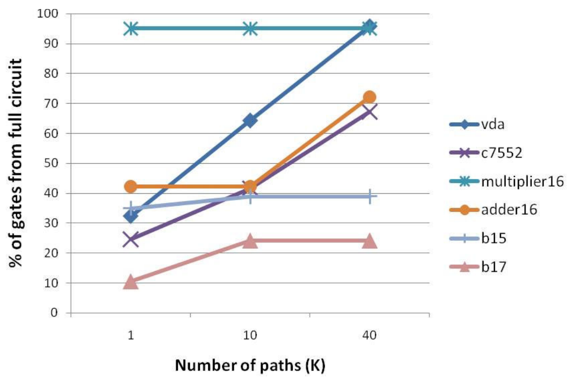


Figure 6.6: Size of macromodels (subcircuits) relative to the entire gate-level circuits, as a function of K.

We ignore internal correlations while propagating signal probabilities through the macromodel (Section 6.2.3). This approximation is a major component in the runtime speedup shown in Figure 6.7. At K=1, `Multiplier16` exhibits a speedup of almost 100x even though the macromodel is not significantly smaller than the original circuit.

Table 6.1 compares the estimation errors for macromodels that are obtained 1) with NBTI effects and 2) without NBTI effects. For `b15`, the critical path in the presence of NBTI effects (P'_{crit}) is not among the slowest paths in the absence of NBTI effects. This is because of the high signal probabilities for the nodes in P'_{crit} as compared to those in other paths. Circuit `b17` is a composition of three `b15` circuits. Therefore, `b17` exhibits a skewing of signal probabilities (and delay degradation) across paths, similar to `b15`.

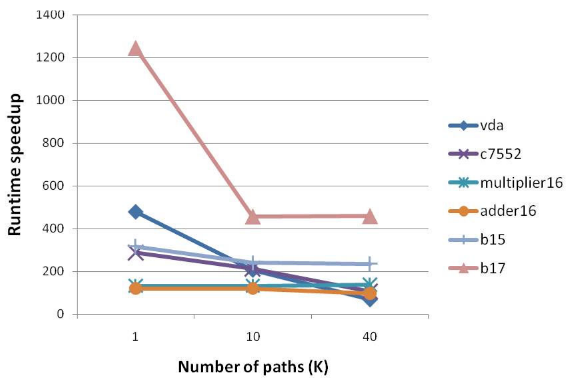


Figure 6.7: Speedup obtained by using macromodels (subcircuits) instead of entire gate-level circuits, as a function of K. The speedup factors reduces linearly with increase in the size of the circuit (Figure 6.6).

For `b15` and `b17`, we obtain more accurate estimates by considering NBTI effects during macromodeling (Section 6.2.1). However, for all the other circuits (only `Multiplier16` and `Adder16` are shown), there is no observable difference between the two macromodeling strategies. For all circuits, macromodeling without NBTI effects provides reasonably accurate estimates and is therefore, a reliable strategy. The inclusion of NBTI effects can be viewed as a strategy for fine-tuning the macromodel construction. The effectiveness of such fine-tuning varies across circuits, as shown in Table 6.1.

Table 6.1: Comparing accuracy of degradation estimates using macromodels 1) with and 2) without NBTI effects. For `b15` and `b17`, we observe that considering NBTI effects during macromodeling reduces the estimation error for the same value of K.

	Circuit	K=1	K=10	K=40
Without NBTI effects	`Multiplier16`	2.76%	2.76%	2.76%
	`Adder16`	18.11%	18.11%	9.09%
	`b15`	4.84%	4.05%	4.05%
	`b17`	3.41%	2.23%	2.23%
With NBTI effects	`Multiplier16`	2.76%	2.76%	2.76%
	`Adder16`	18.11%	18.11%	9.09%
	`b15`	2.80%	2.80%	2.80%
	`b17`	1.00%	0.46%	0.46%

Demonstrating speedup and reliability of RTL estimation

We apply our methodology on several data-intensive RTL designs from the High-Level Synthesis benchmarks suite (HLS95). We consider different gate-level implementations for adders. We use the notations `_bk`, `_mc` and `_cla` to denote designs implemented with BrentKung, Manchester carry and carry look-ahead adders, respectively. For each macromodel, we choose K such that the estimation error is less than 2% (Figure 6.5).

We assume that the RTL inputs are uniformly distributed. We obtain the PMFs of all variables in the RTL design. We start the DTMCs from a known initial state that we specify and then employ probabilistic model checking to compute the PMFs. For N=10 (i.e., 10 DTMC transitions), we find that the PMFs of all variables converge to a steady value.

We use our macromodels to estimate delay in RTL in the absence of NBTI effects (D_{RTL}) as well the average degraded delay (D'_{RTL}). We compute the the average degradation of the design $\Delta D_{RTL}=D'_{RTL}-D_{RTL}$. In Table 6.2, we compare these RTL estimates with those obtained by using gate-level simulations, denoted by $\Delta D_{gate}=D'_{gate}-D_{gate}$. We simulate the gate-level netlist until the estimated signal probabilities converge to within 10% of their steady value (we use 10^5 simulation cycles).

Table 6.2: Accuracy of our RTL estimates for average degradation. Our estimates for % degradation (Column 3) closely agree with those obtained from the gate level (Column 4).

RTL design	Estimation error for D'_{RTL}	$\frac{\Delta D_{RTL}}{D_{RTL}}$	$\frac{\Delta D_{gate}}{D_{gate}}$
`filter_cla`	2.13%	5.56%	5.63%
`kalman_bk`	4.89%	5.41%	6.31%
`kalman_mc`	4.03%	5.68%	6.42%
`fft_bk`	2.10%	5.12%	5.31%
`fft_cla`	2.43%	5.24%	5.31%
`ellipf_bk`	8.82%	4.58%	5.52%
`ellipf_mc`	8.72%	4.75%	5.78%
`ellipf_cla`	9.79%	5.10%	5.23%

In Column 2 of Table 6.2, we list the error in estimating D'_{RTL} (with respect to D'_{gate}). Although we compute exact PMFs in RTL, we ignore the internal correlations while propagating these PMFs through the macromodels at the gate-level (Section 6.2.3). However, the analysis in [87] shows that the estimation error introduced by this approximation is negligible. The error introduced by macromodeling is less than 2% (Figure 6.5) and can be further be reduced by choosing a larger K. The estimation error that remains is due to the sizing mismatch between the gates in the netlists that we use for constructing macromodels and the actual netlists obtained by synthesizing the RTL designs. However, both D_{RTL} and ΔD_{RTL} are affected to the same extent by this error. Therefore, we estimate the % degradation $\frac{\Delta D_{RTL}}{D_{RTL}}$ (Column 3) and find that it is in good agreement with $\frac{\Delta D_{gate}}{D_{gate}}$ (Column 4).

In Figure 6.8, we show that our runtimes are an order of magnitude lower (18.2x faster for `ellipf_cla`) than those for gate-level simulation. The time taken for computing the PMFs of RTL variables dominates the runtimes for our RTL estimation. In practice, the PMFs computed for `ellipf_bk` can be reused for both `ellipf_mc` and `ellipf_cla` since they are independent of the gate-level implementation. However, estimation at the gate-level requires each implementation to be simulated separately in order to propagate the probabilities.

The speedup afforded by our methodology (Figure 6.8) comes mainly from the abstraction of behavior of operators that correspond to large gate-level implementations. Although our methodology can also be applied in the context of control-intensive RTL designs, the speedup provided may be quite modest. For example, an *if-else* construct maps to a MUX block which corresponds to a relatively few number of gates. In this case, it may not be worthwhile to construct a macromodel of the MUX implementation.

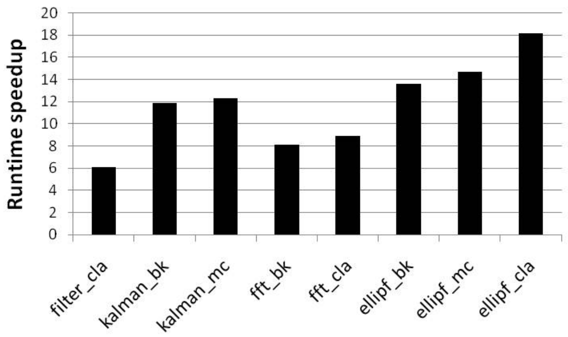


Figure 6.8: Speedup provided by our methodology. Our RTL estimation is up to 18.2x faster than gate-level estimation that uses simulations.

We also compute the estimates of worst-case RTL delay degradation (ΔD_{RTL}-wc). We obtain worst-case macromodels where all the gate inputs are held constant at logic 0. In Table 6.3, we observe that our estimates for % worst-case degradation (Column 3) closely approximate those computed at the gate-level (Column 4). We do not report any runtime speedup here since probability propagations are not required. Such worst-case degradation does not usually occur since it is typically not possible to set all gate nodes to 0 in any useful circuit. However, our estimates can be viewed as upper bounds on degradation that can be used with conservative design methodologies.

Our RTL estimates of % degradation (Tables 6.2 and 6.3) of different gate-level implementations differ from each other at most by about 1%. Therefore, we do not conclude that one gate-level implementation is better than the other. Instead, at the higher level, our focus is on differentiating designs at a coarser level of granularity (differ by 5% or more).

Table 6.3: Accuracy of our RTL estimates for worst-case degradation.

RTL design	Estimation error for $D'_{RTL} - wc$	$\frac{\Delta D_{RTL-wc}}{D_{RTL-wc}}$	$\frac{\Delta D_{gate-wc}}{D_{gate-wc}}$
`filter_cla`	2.21%	9.45%	9.68%
`kalman_bk`	5.02%	9.35%	10.54%
`kalman_mc`	4.76%	9.46%	10.72%
`fft_bk`	2.17%	9.32%	9.98%
`fft_cla`	2.44%	9.63%	10.15%
`ellipf_bk`	8.70%	9.16%	10.04%
`ellipf_mc`	8.96%	8.41%	10.74%
`ellipf_cla`	9.41%	9.53%	9.79%

Analyzing degradation of RTL blocks in 'SLEEP' mode

RTL blocks that are are inactive for several clock cycles can be switched off. During such SLEEP modes, the inputs to the block are held constant in order to reduce switching activity. However, based on this constant input vector, NBTI effects could be exacerbated. Recent work [88] attempt to determine the optimal input vector to be applied during SLEEP mode in order to mitigate NBTI effects. We evaluate the effectiveness of holding all inputs constant at logic 1 during SLEEP mode.

We consider the adder block in the 32-bit ALU of an OR1200 processor. The adder output is used whenever the corresponding value of the *select* signal is assigned. We assume that the adder is in SLEEP mode whenever it is not used. We introduce a 1-bit RTL signal called $SLEEP$ that can be asserted to switch off the adder when it is not in use. We use RTL-DTMCs to compute the signal probability of $SLEEP$ being asserted to be equal to 0.1875. Our methodology estimates that the delay degrades by 3.23%, which is an improvement from the 3.79% degradation if inputs are not held at 1. This conforms to estimates from gate-level simulations which show that degradation reduces from 3.2% to 2.72% if inputs are held at 1 during SLEEP mode.

Our methodology estimates that the degradation of `ellipf_bk` and `ellipf_mc` is 4.44% and 2.88%, respectively, when the inputs are held at 1. Since these are lower than the average-case degradation in Table 6.2, the application of all 1's during SLEEP mode is effective for these designs. However, this input pattern worsens the degradation of `fft_bk` and `fft_cla` to 7.57% and 6.78%, respectively. We confirm all our estimates by using gate-level simulations.

6.5 Chapter summary

In this chapter, we have employed our SHARPE methodology for estimating NBTI-induced delay degradation in RTL. We demonstrated that our methodology provides quick and accurate estimates of delay degradation in RTL. In future work, we plan to analyze NBTI effects in different operating conditions based on temperature, supply voltage and process corners.

CHAPTER 7

AUTOMATIC COMPOSITIONAL REASONING FOR SCALING OUR SHARPE METHODOLOGY

7.1 Introduction

In our SHARPE methodology, we employ formal probabilistic analysis to verify that a hardware design satisfies a statistical performance property. However, we found that the feasibility of this approach is limited to small hardware designs. In order to be practically useful, our methodology must be applicable to larger designs.

In this chapter, we present a sound approach for compositional reasoning that can be used to improve the scalability of our SHARPE methodology. We present our approach and the formal proof of its correctness in the context of combinational designs. However, in our case studies, we demonstrate our approach on both combinational and sequential designs.

7.1.1 Compositional reasoning for our SHARPE methodology

We wish to use SHARPE (Chapter 3) to verify a statistical property on a combinational design. If the size of the design (Definition 10) is too large, then formal probabilistic analysis (Section 3.3) becomes inefficient. Instead, we can obtain an equivalent verification result by decomposing the design into smaller designs and then employing formal probabilistic analysis on each of them. This forms the basis of our compositional reasoning approach.

Let M denote a combinational design and Φ denote the property of interest. Consider that Φ is expressed over a set of RTL variables Π. We use the SHARPE methodology to verify that M satisfies Φ, denoted by $M \models \Phi$. The steps in our compositional reasoning approach are as follows. We partition the set of variables Π into two smaller subsets Π_1 and Π_2 and use them as a basis to structurally decompose M into smaller designs M_1 and M_2. We employ an *assume-guarantee* form of reasoning [37] where we guarantee that $M_2 \models \Phi_2$ under the assumption that $M_1 \models \Phi_1$. Therefore, $M_1 \models \Phi_1$ serves as the environmental constraint while verifying $M_2 \models \Phi_2$. This environment constraint can be viewed as a channel over which the correlation between M_1 and M_2 can be communicated.

We analyze the space of environmental constraints using a *value-based case splitting* approach. Each case in our case-splitting approach corresponds to a possible assignment of numeric values to the variables in Π_1. For each case, we also derive the properties Φ_1 and Φ_2 for M_1 and M_2 respectively. We use formal probabilistic analysis to find if $M_2 \models \Phi_2$ using $M_1 \models \Phi_1$ as the environment. We model the dependence of M_2 on the environmental constraint by computing the conditional probability distribution of the shared variables between M_1 and M_2.

Finally, we compose the results obtained for all cases which is equivalent to verifying $M \models \Phi$. We provide an argument for the correctness of our technique. Our technique spans a broad class of performance properties such as delay (Chapter 5) and BER (Chapter 4).

In our technique, we need to perform several instances of verification of components M_1 and M_2 instead of a single instance of verifying M. However, we keep the size of components sufficiently small in order to ensure the feasibility of each verification instance. We are primarily concerned with hardware systems where the variables Π can be assigned a finite set of Boolean values. Therefore, the space of cases that are based on numeric values is finite. We show using case studies that some amount of designer guidance during case splitting can keep the number of cases reasonably small.

Several compositional reasoning approaches have been presented in the context of probabilistic model checking [37],[39],[40],[41],[42]. However, unlike our technique, these approaches are not automatic.

We show that our technique is extremely effective in practice. We describe case studies of estimating performance metrics like delay and BER of hardware designs such as ripple carry adders (Chapter 5), FIR filters [89], MIMO systems (Chapter 4) and an OR1200 processor (Chapter 5). For example, we are able to determine the statistical delay of a 64-bit adder design with over 10^{40} possible input vectors.

7.1.2 Chapter organization

The rest of this chapter is organized as follows. In Section 7.2, we present a formal description of the properties for which we describe our compositional reasoning approach. In Section 7.3, we describe the intuition for the effectiveness and soundness of our approach. In Section 7.4, we detail each of the steps in our automatic compositional reasoning technique. In Section 7.5, we describe an optimization that we perform to improve the efficiency of our approach. In Section 7.6, we demonstrate the effectiveness on our approach by considering several case studies.

7.2 Formal definition of performance properties

We now formally specify the statistical properties that we consider in this chapter.

Let $V1$ and $V2$ be the set of all variables in RTL designs M_1 and M_2 respectively. M_1 and M_2 are defined to be independent if they do not possess any variables in common, i.e., $V_1 \cap V_2 = \mathbf{0}$. Otherwise, M_1 and M_2 are correlated and the common shared variables are called *interface variables*. The set of interface variables between M_1 and M_2 is given by $\Psi = V_1 \cap V_2$.

Hardware performance metrics like delay and BER are data-dependent, and therefore we express them as functions of the variables of the hardware system. We use real-valued analytical functions f_P that are defined over a set of RTL variables $\Pi \subset V$ in the design M. We refer to Π as the set of *observables* of M.

We define probabilistic invariants Γ [16] based on the performance metrics of the hardware design, given by

$$\Gamma \triangleq P[f_P(\Pi) == c] \tag{7.1}$$

where c is a real-valued constant. "==" is a logical operator that evaluates to TRUE if the values of the operands are equal and FALSE otherwise. In place of the equality operator "==", we allow for the use of inequality operators as well. $P[event]$ denotes the probability of occurrence of an *event*. We consider events of two types:

1. Logical event: The evaluation of a logical comparison operation to TRUE. For example, a logical event occurs if the function $f_P(\Pi)$ evaluates to a numeric value equal to c and we denote this event by $f_P(\Pi) == c$.

2. Substitution event: The assignment of a set of concrete numeric values C to a set of symbolic variables Π, denoted by $\Pi/.C$.

We formally represent the computation of performance metrics by using the corresponding invariants to define probabilistic properties Φ of the form

$$\Phi \triangleq \Gamma \leq p \tag{7.2}$$

where $p \in$[0,1] is a specification of the hardware system. We allow logical comparison operators other than $\leq$ to be used for comparing the probabilistic invariant with p.

For sequential designs, we are interested in computing Γ for values of Π at some time step t. For large values of t, the DTMC may reach a steady state (Section 2.2) where the probability of being in a state is independent of the time step. In Chapter 4, we have shown that the DTMC models for sequential designs that are of interest to us do reach a steady state in which Γ is independent of t.

The PMFs of the variables in Π are stationary, and therefore can also be viewed as probabilistic invariants. In every state, the numeric value assigned to $v \in \Pi$ is given by the signal function $f(Sup(v))$ (Definition 8). Let Φ^k be the property that a numeric value equal to k is assigned to v with probability p^k. Since $p^k = P[v/.k]$ and $P[v/.k] = P[f(Sup(v)) == k]$, Φ^k can be defined as

$$\Phi^k \triangleq P[f(Sup(v)) == k] = p^k \tag{7.3}$$

Therefore, the PMF of variable v of size N can be expressed as the property Φ given by

$$\Phi = \Phi^1 \wedge \Phi^2 \wedge ... \wedge \Phi^k \wedge ... \wedge \Phi^N \tag{7.4}$$

where $\wedge$ represents a logical conjunction.

We employ formal probabilistic analysis (Chapter 3) and verify $M \models \Phi$. The verification procedure for properties described in Equation 8.2 involves the computation of the invariant Γ and comparing it with p. If p is not specified, verifying $M \models \Phi$ is equivalent to the computation of Γ.

In order to verify $M \models \Phi$, it is sufficient to determine the PMFs of the observables Π specified in Φ. We can determine the PMFs of Π by using the support and the signal functions of each variable in Π. In order to do this, it is sufficient to consider a slice of the design that contains the logic cone of Π. Therefore, in the strict sense, M denotes this slice of the design. The statistics of M come from the PMFs of the input variables in the support.

7.3 Our compositional reasoning approach

In this section, we define and establish criteria for compositional verification of probabilistic designs of our interest.

Compositional reasoning states that in order to verify $M \models \Phi$, it is sufficient to achieve two separate probabilistic verification subgoals, $M_1 \models \Phi_1$ and $M_2 \models \Phi_2$. However, if M_1 and M_2 are correlated, an assume-guarantee form of reasoning is required [37]. In this form of reasoning we perform probabilistic verification of M_2 to guarantee that $M_2 \models \Phi_2$ under the assumption that $M_1 \models \Phi_1$. Therefore, $M_1 \models \Phi_1$ serves as the environmental

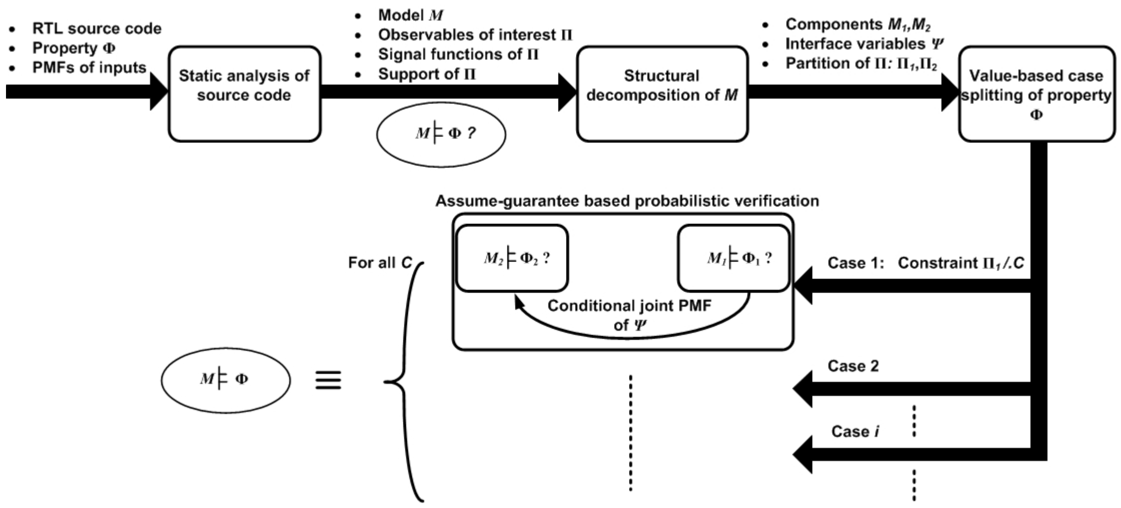


Figure 7.1: Block diagram showing the stages of our assume-guarantee based compositional reasoning approach. The labels on the arrows show the outputs of each stage.

constraint while verifying $M_2 \models \Phi_2$.

Let Π be the set of observable in Φ (Equation 8.2). Consider a partition of Π into two disjoint subsets, Π_1 and Π_2. M_1 and M_2 are the minimal components/slices of M that contain $Sup(\Pi_1)$ and $Sup(\Pi_2)$, respectively (Section 7.2). Therefore, Π_1 and Π_2 can be expressed by using only the variables of M_1 and M_2, respectively. In other words, a partitioning of the observable space into Π_1 and Π_2 can be used to induce a structural decomposition of M into M_1 and M_2.

We analyze the space of environmental constraints using a *value-based case splitting* approach. Since the observables are manipulatable entities, we divide the space of values of all observables Π_1 of component M_1 into separate cases based on their numeric values. Each case corresponds to a unique assignment of a set of numeric values C to the observables Π_1, denoted by the substitution event $\Pi_1/.C$. These cases are then propagated to the interface variables Ψ between M_1 and M_2. In a non-probabilistic context, it is sufficient for the constraints to be propagated only to the numeric values of the interface variables. However, in a probabilistic context, the PMFs of the interface variables need to be considered.

Every case $\Pi_1/.C$ corresponds to a separate constraint on M_1 which in turn acts as an environmental constraint for M_2. For each case, the dependency of M_2 on its environmental constraints can be modeled using the conditional joint PMF of the interface variables Ψ given the event $\Pi_1/.C$. M_2' is the model obtained by augmenting M_2 with the conditional PMF of Ψ. Therefore, the probabilistic behavior of M_2' tracks the occurrence of the event $\Pi_1/.C$. The non-probabilistic FSMs for M_2 and M_2' are identical. However, their corresponding variables do not exhibit identical probabilistic behavior.

For each case, the probability of occurrence of the corresponding substitution event $\Pi_1/.C$ can be interpreted as Φ_1 for M_1. For each case, we project the numeric values of Π_1 onto the property Φ by replacing each observable in Π_1 that appears in Φ with its corresponding numeric value specified in C. As a result of this projection, we obtain the second property Φ_2.

We then verify $M_2' \models \Phi_2$ under every environmental constraint. We use the invariants obtained as results of verifying $M_1 \models \Phi_1$, $M_2' \models \Phi_2$ for all cases and compute a probabilistic invariant which we call Γ_{12}. We then verify $\Gamma_{12} \leq p$ which we shall later prove (Section 7.4.6) is equivalent to verifying $M \models \Phi$, denoted by

$$\Gamma_{12} \leq p \equiv M \models \Phi \tag{7.5}$$

We are primarily concerned with hardware systems where the observables can be assigned a finite set of Boolean values. Therefore, our space of constraints is finite. This is a domain-specific advantage over generic systems where the numeric values of observables may range over the entire space of integer/real numbers.

Typically, compositional reasoning of correlated components requires a circular assume-guarantee technique where verifying $M_2 \models \Phi_2$ by constraining M_1 is supplemented with verifying $M_1 \models \Phi_1$ by constraining M_2. In our approach, Φ_1 represents the joint probability of the assignment of numeric values specified by C to the observables Π_1. Since we assume that all variables are stationary, the joint probabilities are constant. Therefore the outcome of verifying $M_1 \models \Phi_1$ does not depend on its environmental constraints and it is sufficient to verify $M_1 \models \Phi_1$ irrespective of any assumptions on the behavior of M_2.

7.4 Algorithm for automatic decomposition of hardware systems

We present our algorithm for automatic decomposition of hardware systems in order to perform compositional reasoning as described in Section 7.3. We describe in detail the stages of our algorithm as shown in Figure 7.1.

We illustrate our technique using the example combinational RTL code fragment shown below:

```
module (I1,I2,I3,I4,O1,O2,O3)
input [9:0] I1,I2,I3,I4;
```

```
wire [9:0] Z1,Z2,Z3,Z4;
output [9:0] O1,O2,O3;
always @(posedge clk)
begin
Z1 <= I1; Z2 <= I2 - I3;
Z3 <= I3 + I4; Z4 <= I3 & I4;
O1 <= Z1 | Z2; O2 <= ~Z3; O3 <= Z4;
end
endmodule
```

In the example hardware system shown above, $I1$, $I2$, $I3$ and $I4$ are the system inputs. $O1$, $O2$ and $O3$ are the system outputs and $Z1$, $Z2$, $Z3$ and $Z4$ can be viewed as temporary variables in the hardware system. All variables are of 10 bits and can therefore be assigned 1024 different numeric values (Section 2.3).

We wish to compute the probability that the sum of the numeric values of the hardware system outputs is equal to 100 (the decimal equivalent of a binary number). Therefore, the required invariant Γ for the property Φ is defined as $P[(O1 + O2 + O3) == 100]$.

7.4.1 Static analysis of source code

Let Π be the observable that we extract from the property Φ. We statically analyze the RTL source code (Section 3.2) and determine the support and signal function for each variable in Π. Given the PMFs of the input variables, the combinational design M models the probabilistic behavior of Π using $Sup(\Pi)$ and $f(Sup(\Pi)$.

Figure 7.2 depicts the static analysis for the RTL example. We extract the observables Π=$\{O1, O2, O3\}$. $Z1$ and $Z2$ are the elements of $RHS(O1)$. Since $Z1$ and $Z2$ are not inputs, we step back once more and determine that $I1$, $I2$ and $I3$ are the elements of the union of $RHS(Z1)$ and $RHS(Z2)$. Since all the variables reached are inputs, $Sup(O1)$=$\{I1, I2, I3\}$. Similarly, we obtain $Sup(O2)$=$\{I3, I4\}$ and $Sup(O3)$=$\{I3, I4\}$. The support of Π is given by $Sup(\Pi)$=$\{I1, I2, I3, I4\}$.

We use the supports to identify the variables that are statistically independent. We define two variables v_1 and v_2 to be independent if their supports are disjoint sets, i.e., $Sup(v_1) \cap Sup(v_2)$=$\mathbf{0}$. In Section 7.5, we shall describe an optimized version of our static analysis technique that can be used to detect independent supports.

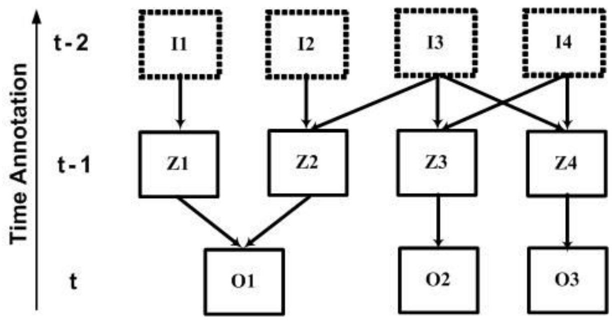


Figure 7.2: Depiction of static analysis for an RTL example.

7.4.2 Structural decomposition of M

We partition the set of observables Π in property Φ into two disjoint subsets Π_1 and Π_2 such that $\Pi_1 \bigcup \Pi_2 = \Pi$. From static analysis, we know the support and signal functions for the observables Π_1 and Π_2. We construct the models M_1 and M_2 by obtaining the minimal slice of M that contains the logic cone of Π_1 and Π_2, respectively. $Sup(\Pi_1)$ and $Sup(\Pi_2)$ are variables in M_1 and M_2, respectively. Therefore, Π_1 and Π_2 are the observables of M_1 and M_2, respectively. Both M_1 and M_2 are structurally different from M. If M_1 and M_2 are correlated, we determine the set of interface variables $\Psi = Sup(\Pi_1) \cap Sup(\Pi_2)$.

For the RTL example, the partition of Π is $\Pi_1 = \{O1\}$ and $\Pi_2 = \{O2, O3\}$. Therefore, M is decomposed into M_1 and M_2 with $Sup(\Pi_1) = \{I1, I2, I3\}$ and $Sup(\Pi_2) = \{I3, I4\}$. The set of interface variables between M_1 and M_2 is $\Psi = \{I3\}$.

RTL designs are inherently modular in nature and [38] treats the *modules* in RTL as the components that need to be composed together. However, our structural decomposition strategy is not restricted to this level of granularity and is therefore able to further decompose such modules resulting in smaller components.

7.4.3 Value-based case splitting of property Φ

We split the property Φ into cases based on the numeric values of the set of observables Π_1. Let $\Pi_1/.C$ denote a case corresponding to an assignment of a set of values C to Π_1.

For each case, we define the computation of the invariant $P[\Pi_1/.C]$ as Φ_1 (Section 4.2.3). Let Φ^C denote the joint PMF of Ψ given the substitution event $\Pi_1/.C$. We call this the *conditional joint PMF* of Ψ given that the event $\Pi_1/.C$ has already occurred. We use the notation $P[A|B]$ to represent the conditional probability of the occurrence of the event A given that event B has already occurred. Joint PMF can be viewed as the invariant corresponding to a property as defined in Equation 7.4.

We update M_2 by substituting the PMFs of the variables Ψ with the conditional joint PMF Φ^C. We call this statistical model M_2'. However, the set of variables in M_2' is the same as that in M_2.

The property Φ in Equation 8.2 can be written as

$$\begin{aligned} \Phi &= P[f_P(\Pi) == c] \leq p \\ &= P[f_P(\Pi_1, \Pi_2) == c] \leq p \end{aligned} \tag{7.6}$$

We modify the property Φ by substituting the values of Π_1 as fixed by the constraint, into the expression for f_P. We define this as property Φ_2 given by

$$\Phi_2 \triangleq P[f_P(C, \Pi_2) == c] \leq p \tag{7.7}$$

Our value-based case splitting technique can be considered similar to the approach described in [38]. We do not make any assumptions regarding the function f_P. Therefore, our case splitting technique is applicable to a broad class of performance metrics.

For the RTL example, the constraints are of the form $O1/.x$ where x is one of the 1024 numeric values that $O1$ can be assigned. Therefore, there are 1024 cases that need to be considered. Consider the case $O1/.5$, where 5 is the decimal equivalent of the 10-bit binary number 0000000101. We obtain the conditional PMF of the interface variable $I3$ given $O1/.5$. We construct M_2' from M_2 by replacing the PMF of $I3$ with this conditional PMF. In this case, we define Φ_1 with the invariant $P[O1/.5]$. We define Φ_2 using the invariant $P[(5 + O2 + O3) == 100]$.

7.4.4 Assume-guarantee based probabilistic verification

For each case $\Pi_1/.C$, we first verify $M_1 \models \Phi_1$ and compute the corresponding probabilistic invariant Γ_1 given by $\Gamma_1 = P[\Pi_1/.C]$. We then compute the conditional joint PMF Φ^C of the interface variables Ψ by verifying $M_1 \models \Phi^C$ under the constraint $\Pi_1/.C$.

We then verify $M_2' \models \Phi_2$ and compute the corresponding probabilistic invariant Γ_2'. We implicitly use a form of assume-guarantee reasoning here since verifying $M_2' \models \Phi_2$ can be thought of as verifying $M_2 \models \Phi_2$ under the assumption that $M_1 \models \Phi^C$.

We compute the product of the invariants Γ_1 and Γ_2'. We repeat this for all possible C and obtain the sum of the product of invariants computed for each case $\Pi_1/.C$. We call the resulting probabilistic invariant Γ_{12} which is given by

$$\Gamma_{12} = \sum_{C} \Gamma'_2 \Gamma_1 \tag{7.8}$$

Finally, we compare Γ_{12} with p and verify $M \models \Phi$ according to Equation 7.5

For the RTL example, we combine the 1024 numeric cases based on the values of x and compute Γ_{12} as

$$\sum_{x} P[(x + O2 + O3 == 100)]P[O1/.x] \tag{7.9}$$

Although our approach can be applied recursively, we have used just two components to simplify the illustration of our technique. In our approach, M'_2 can be further decomposed into smaller models M_3 and M_4. We now define property cases by coupling each case on M_1 with a case based on the numeric values of observables of M_3. We use this new constraint to modify M_4 and obtain M'_4. We project the values of the observables of both M_1 and M_3 onto Φ to construct property Φ_4. We verify $M'_4 \models \Phi_4$ for all cases and then compute the required invariant Γ_{12}. Our decomposition technique can be applied recursively until the size (Definition 10) of the model verified falls below a user-specified threshold.

7.4.5 Complexity analysis

We now analyze the complexity of our decomposition technique when M is decomposed into N components M_1 to M_N. Each property case corresponds to a numeric value assigned to the sets of observables Π_i of the models M_i, $i = 1$ to $N - 1$. In order to compute the invariant Γ_{12}, we verify $M'_N \models \Phi'_N$ for each case.

For large values of N, the size of M'_N can be made very small resulting in faster computation of verification results. However, all of the property cases need to be considered in order to compute the invariant Γ_{12}. For large values of N, only a small number of observables from the original set Π may appear in Π_N. Since case-splitting is performed based on the numeric values assigned to the other $N - 1$ subsets, the number of verification instances can potentially be very large.

The computation of the conditional joint PMF Φ^C can be viewed as a verification instance. The complexity of our approach increases with the space of numeric values that can be assigned to the interface variables Ψ. Therefore, in future work, we plan to investigate the use of Ψ as a criteria when structurally decomposing hardware systems [90]. In Section 7.5, we describe an optimization that may reduce the number of cases which need to be considered for certain properties and designs.

7.4.6 Proof of correctness

We now show the correctness of our automated decomposition technique for combinational designs. In order to achieve this, we prove that verifying $M \models \Phi$ is equivalent to computing Γ_{12} and comparing it with p, as in Equation 7.5.

For properties of the type described in Equation 8.2, verifying $M \models \Phi$ is equivalent to computing the probabilistic invariant Γ and comparing it with the specified probability p. In our decomposition technique, the value of p is not modified. Therefore, we prove the correctness of our technique by showing that Γ_{12} computed in Equation 7.8 is exactly equal to Γ.

For each case $\Pi_1/.C$, the invariant computed while checking $M_2 \models \Phi_2$ is $P[(f_P(C, \Pi_2) == c)]$ which can be written as $P[(f_P(\Pi) == c)|(\Pi_1/.C)]$. However, this is not equal to the invariant Γ_2' computed by verifying $M_2' \models \Phi_2$ since M_2' and M_2 exhibit different probabilistic behavior. Since the PMFs of the interface variables are replaced with the conditional joint PMFs, the invariant Γ_2' is given by

$$\sum_K P[(f_P(C, \Pi_2) == c)|(\Psi/.K)]P[(\Psi/.K)|(\Pi_1/.C)] \tag{7.10}$$

which in turn can be written as

$$\sum_K P[(f_P(\Pi) == c)|(\Pi_1/.C \wedge \Psi/.K)]P[(\Psi/.K)|(\Pi_1/.C)] \tag{7.11}$$

where $P[(\Psi/.K)|(\Pi_1/.C)]$ is the conditional joint probability of $\Psi/.K$ given the event $\Pi_1/.C$.

According to Equation 7.8, Γ_{12} is computed as $\sum_C \Gamma_2' P[\Pi_1/.C]$ which can be written as

$$\sum_{C,K} P[(f_P(\Pi) == c)|(\Pi_1/.C \wedge \Psi/.K)]P[(\Psi/.K) \wedge (\Pi_1/.C)] \tag{7.12}$$

From the *law of total probability* [21] for random variables with discrete values, we know that the expression for Γ_{12} in Equation 7.12 can be reduced to $P[f_P(\Pi) == c]$ which is exactly equal to Γ.

7.5 Optimization

The complexity of our approach increases with the size of the component DTMCs and the number of property cases (Section 7.4.3). In this section, we present optimization strategies that we use to improve the efficiency of our compositional technique. First, we describe an optimized static analysis technique that can be used to construct smaller component models. We also outline a strategy for reducing the number of property cases that need to be considered.

7.5.1 Reduction of size of components

For a variable $v \in \Pi$, static analysis of the RTL source code can be used to determine the signal function $f(Sup(v))$ with respect to the support $Sup(v)$. However, v can also be expressed as a function of a set of independent variables that are not inputs. We call this set the reduced support of v, denoted by $RedSup(v)$. We define the corresponding signal function f_R, such that $v = f_R(RedSup(v))$.

For each variable $r \in RedSup(v)$, the PMF can be obtained as in Equation 7.4. A reduced model M_R corresponding to Π can then be constructed with only the variables given by

$$RedSup(\Pi) = \bigcup_{v \in \Pi} RedSup(v) \tag{7.13}$$

In many hardware systems, we observe that the number of variables in $RedSup(v)$ is less than that in $Sup(v)$ and that $|RedSup(v)|$ is significantly less than $|Sup(v)|$. Therefore, the size (Definition 10) of M_R is smaller than that of M leading to faster probabilistic verification.

For each $r_i \in RedSup(\Pi)$, let M_i be a model containing $Sup(r_i)$ as input and sequential variables. Therefore, r_i is an observable for M_i. Since the variables in $RedSup(\Pi)$ are independent, the corresponding models M_i are also independent. The reduced model M_R can be viewed as a parallel composition of independent components M_i [37]. Therefore, M and M_R are probabilistically equivalent with respect to Φ.

Optimized static analysis:

We now describe an optimized version of the static analysis technique (Section 3.2) that can be used to compute the independent, reduced support for the variables of interest.

We statically traverse the source code for computing $RedSup(v)$ and $f_R(RedSup(v))$ for a variable v of interest. The steps in our optimized static analysis are as follows. These variables in $RHS(v)$ are annotated with the time step value $t-1$. If all the variables in $RHS(v)$ are independent, we define $RHS(v)$ to be $RedSup(v)$ and the analysis is complete. If not, we step the circuit backwards by one more time step. Now each variable in $RHS(RHS(v))$ is annotated by $t-2$. If the union over the set of all variables tagged with $t-2$ comprises only of variables that are independent of each other, we define the union to be $RedSup(v)$ and obtain the corresponding $f_R(RedSup(v))$. The algorithm terminates when the reduced support is obtained. In the worst case, the algorithm proceeds till the primary inputs are reached and we obtain $RedSup(v)$=$Sup(v)$. Since the reduced support is independent, the joint PMF can be calculated as in Definition 1 and this can be used to compute the PMF of v.

The static analysis methodology to establish independence of two variables x and y (as in Section 2.1) is as follows. If x and y are inputs, they are independent by assumption. If x and y are not inputs, they are recursively defined as independent if each variable in the union of $RHS(x)$ and $RHS(y)$ is independent. Conversely, if x and y are correlated, there is at least one common/shared variable among the variables with the same annotation $t - i$.

In the RTL example, $Z1$ and $Z2$ are independent and form the reduced support of $O1$. The model M corresponding to $O1$ has variables S=$Sup(O1)$ that corresponds to 2^{30} possible input vectors. However, the reduced model M_R with $RedSup(O1)$=$\{Z1, Z2\}$ has only 2^{20} possible input vectors that need to analyzed. The PMFs of $Z1$ and $Z2$ itself can be computed using models with input variables $\{I1\}$ and $\{I2, I3\}$, respectively. Therefore, the use of a reduced support provides significant reduction in the size of the models on which verification is performed.

7.5.2 Reduction of cases

A function $f_P(\Pi)$ is defined to be *separable* if it can be written in terms of two other functions $g_P(\Pi_1)$ and $h_P(\Pi_2)$ such that Π_1,Π_2 are disjoint subsets of Π and $\Pi_1 \bigcup \Pi_2 = \Pi$. For example, an *additively separable* function can be expressed as $f_P(\Pi) = g_P(\Pi_1) + h_P(\Pi_2)$.

If the functions $f_P(\Pi)$ defined in the invariant Γ are separable, we can perform a case splitting of the property based on the numeric values of $g_P(\Pi_1)$ rather than the numeric values of Π_1. Such an abstraction [91] can be used to significantly reduce the number of cases that need to be considered.

In some systems, the probabilistic behavior of the components may be symmetric with respect to the different property cases that we construct. All such symmetric cases can be analyzed by performing assume-guarantee based probabilistic verification for just one representative case. In the context of non-probabilistic hardware systems, a similar form of symmetric case reduction has been employed in [38].

We perform such abstractions and symmetry reductions in order to reduce the number of property cases that need to be analyzed thereby reducing the time complexity of our approach.

7.6 Case studies

We present the following case studies in order to highlight the different aspects of our technique:

1. Ripple carry adder (Chapter 5)
2. MIMO Systems (Chapter 4)
3. FIR filter
4. OR1200 processor (Chapter 5)

With the exception of the ripple carry adder, the case studies that we consider are all sequential designs. For these sequential designs, we wish to compute the steady-state probabilities with respect to performance, i.e., we consider **Type A** properties described in Section 3.4.

The results presented in our tables can be interpreted as follows. We terminate experiments for which verification does not complete within 10 hours. In such cases, we verify the accuracy of our computed invariants by comparing them with high-confidence estimates that are obtained using a sufficiently large number of simulations. In the table for each case study, we list both the number of states (or input vectors) in the largest component model and the number of value-based cases (i.e., verification instances).

For sequential, we use our approach to trade off the complexity in space with that in time. We are now able to compute probabilistic invariants for models that were previously "uncheckable" since the model checking tool ran out of memory.

7.6.1 Ripple carry adder

We wish to compute the probability that the delay at the MSB of a ripple carry adder output is less than a specified timing constraint (Definition 14). We model this delay invariant approximately (Section 5.2) as a function of carry bits that ripple through the design, denoted by *ripples*, which in turn can be expressed as a function of adder inputs. For a 32-bit adder, *ripples* can take a value between 0 and 31 depending on the inputs. The 32-bit adder will have 2^{64} possible input vectors and is therefore, too large for efficient, formal probabilistic analysis (Section 3.3). For a 64-bit adder, the number of possible input vectors can be as high as 2^{128} (approximately 10^{40}).

Since we model delay as a function of *ripples*, we can compute the delay invariant by determining the probability distribution of *ripples*. We decompose the 32-bit adder into four 8-bit blocks. While verifying Block 4 (corresponding to the higher bits), we will have to consider all possible values of the lower 24 bits of the adder output. However, Block 4 only requires information about the probability distribution of ripples through the Blocks 1 to 3, i.e., the lower 24 bits. Since there are only 24 possible numeric values for the ripples through Blocks 1 to 3, only 24 cases need to be considered. We apply this approach recursively to compute the ripples through Blocks 1 and 2. At any point, probabilistic verification is performed only on an 8-bit adder block.

Table 7.1: Delay invariants at the MSB of a ripple carry adder.

	Without decomposition			With decomposition				
	Input vectors	Time (sec)	Invariant	Split	Input vectors	Cases	Time (sec)	Invariant
16-bit	2^{32}	2.141	0.9990	2x8-bit	2^{16}	9	0.31	0.9990
				4x4-bit	2^{8}	27	0.93	0.9990
32-bit	2^{64}	>10hrs	-	2x16-bit	2^{32}	17	36.40	0.9688
				4x8-bit	2^{16}	51	1.63	0.9688
				8x4-bit	2^{8}	119	3.81	0.9688
64-bit	2^{128}	>10hrs	-	4x16-bit	2^{32}	99	211.96	0.8730
				8x8-bit	2^{16}	231	3.70	0.8730
				8x4-bit	2^{8}	495	7.92	0.8730

Table 7.1 shows the delay invariants that we compute using our technique for adders of different sizes. We consider several possible splits of the adder designs into smaller equally-sized blocks. We observe that there exists an optimum point for minimizing the total runtime while trading off between the size of the component model (i.e., number of input vectors) and the total number of verification instances.

7.6.2 FIR Filter

We consider a system where a bit $x[n]$ transmitted over a *channel* [14] at time step n can be corrupted by fractions of the bits transmitted in the three previous time steps. Moreover, the channel can also introduce statistical variation, called *noise*, into the data.

Therefore the received data $y[n]$, at time step n, can be modeled as

$$y[n] = \sum_{i=0}^{3} a_i x[n-i] + \eta[n] \tag{7.14}$$

where a_i represents the fractions of the previous bits that corrupt the data. The noise is represented by $\eta[n]$ which is commonly modeled as a Gaussian random variable and $y[n]$ is represented using a variable with N bits.

The received data is processed by a digital hardware block, called an *equalizer* [89] which is a finite impulse response (FIR) filter whose output is represented as

$$z[n] = \sum_{i=0}^{Taps-1} c_i y[n-i] \tag{7.15}$$

where c_i are called the *tap coefficients* and $Taps$ is the total number of such coefficients. The FIR output $z[n]$ can then be used to compute $\widehat{x}[n]$ which is an estimate of the transmitted bit.

We wish to compute the BER that is defined as the probability that the transmitted bit, $x[n]$ is different from the decoded bit $\widehat{x}[n]$. We construct an RTL DTMC (Section 3.4.1) with $x[n-i]$ and $y[n-i]$ as the state variables, denoted by x_i and y_i respectively. We are interested in computing the following invariant:

$$\begin{aligned} \Gamma &= P[\widehat{x}[n] \neq x[n]] \\ &= P[f(y_0, y_1, ...y_{Taps-1}) \neq x_0] \end{aligned} \tag{7.16}$$

where f is the function defined in Equation 7.15. We observe that f is additively separable. Therefore, we can rewrite Equation 7.16 as

$$\Gamma = P[f(f_1(y_0, .., y_{Taps/2-1}), f_2(y_{Taps/2}, .., y_{Taps-1})) \neq x_0] \tag{7.17}$$

where f_1 and f_2 are given by

$$f_1(y_0, .., y_{Taps/2-1}) = \sum_{i=0}^{Taps/2-1} c_i y[n-i] \tag{7.18}$$

$$f_2(y_{Taps/2}, .., y_{Taps-1}) = \sum_{i=Taps/2}^{Taps-1} c_i y[n-i] \tag{7.19}$$

We split the observables into two sets: $\{y_0, .., y_{Taps/2-1}\}$ and $\{y_{Taps/2}, .., y_{Taps-1}\}$. We need to consider all $2^{NTaps/2}$ numeric values of the observables in $\{y_{Taps/2}, .., y_{Taps-1}\}$, where each y_i is represented using N bits. However, we are interested only in the values of $f_2(y_{Taps/2}, .., y_{Taps-1})$. We assume that the output of every addition in the filter is truncated to N bits. Therefore, the number of cases we need to consider is reduced to 2^N.

In Table 7.2, we show the BER invariants computed for FIR filters with different number of data bits (N) and taps ($Taps$). We observe that the number of cases depends only on the number of data bits and not on the number of taps.

Table 7.2: BER of an FIR Filter. We use *O.o.M* to denote *Out of Memory.*

		Without decomposition			With decomposition				
N (bits)	Taps	DTMC states	Time	Invariant	Split	DTMC states	Cases	Time	Invariant
6	6	$> 10^{12}$	>10hrs	-	2x3-tap	2^{22}	128	189.44s	5.02x10^{-3}
6	8	*O.o.M.*	-	-	2x4-tap	2^{28}	128	234.24s	9.67x10^{-4}
8	6	*O.o.M.*	-	-	2x3-tap	2^{28}	512	2713.6s	3.36x10^{-3}

7.6.3 MIMO systems

We wish to obtain the BER performance of the combined operation of a 1x4 MIMO detector and a Viterbi decoder (Chapter 4). Let x be the actual transmitted data bit. Let x' and x'' be the bits estimated by the MIMO detector and Viterbi decoder, respectively. We express the BER invariant of the combined system as a function of x, x' and x'' which can in turn be written as a function of the state variables of the individual components. This function is fairly complex, and therefore we omit its description here. Direct application of our SHARPE methodology (Section 3.4) on the combined system is infeasible due to the large state space of the corresponding RTL DTMC model.

Table 7.3: Individual BER values for a MIMO detector and Viterbi decoder.

	States (Original model)	States (Reduced model)	Time (sec)	Invariant
MIMO 1x4	524, 288	1, 320	53.27	1.08x10^{-5}
Viterbi	53, 558, 744	8, 505, 363	184.13	0.2394

We choose to perform case-splitting based on the numeric values of x and x'. In other words, we model check the Viterbi decoder, by conditioning on whether the MIMO detector output is correct or not. We need to consider four cases of the form $(x = i, x' = j)$, where i,$j \in \{0, 1\}$. However, the occurrence of bit errors at the MIMO detector output is independent on the values of the decoded and transmitted bits. Therefore, due to symmetry, we only need to consider two cases: $(x = 0, x' = 0)$ and $(x = 0, x' = 1)$. In Table 7.4, we show the BER invariant computed using our methodology.

Table 7.4: BER for a MIMO detector operating in conjunction with a Viterbi decoder.

	Without decomposition			With decomposition			
	States	Time	Invariant	States	Cases	Time (sec)	Invariant
MIMO 1x4 + Viterbi	$O.o.M.$	-	-	$8,505,363$	2	493.45	0.2437

7.6.4 OR1200 processor

In the control logic of processors, even single bit flips that occur due to soft errors [9] can be catastrophic. Therefore, it is desirable to have an estimate of the soft error rate (SER) which is defined to be the probability of occurrence of soft errors. In this case study, we consider an OR1200 processor, an open-source embedded RISC processor. We wish to compute the SER invariant for the RTL variable sel_a that determines the address which one set of data values are read from or written to.

Through static analysis, we determine that sel_a depends on whether the address stored in six bits of the fetched instruction if_insn[21:16] is equal to the address stored in rf_addrw. It is reasonable to assume that SER depends on the values of the RTL variables for stall (if_stall), freeze (if_freeze) and pipeline flush ($flushpipe$). Therefore, we define the SER invariant as a function $f(if\_stall, if\_freeze, flushpipe, if\_insn[21{:}16], rf\_addrw)$.

Without decomposition, PRISM is unable to construct the required DTMC model. Therefore, we perform case-splitting based on the numeric values of if_insn[21:16] and rf_addrw. Each of these are 6-bit variables, and therefore we need to consider 2^{12} cases. From static analysis we identify that the numeric values of if_stall,if_freeze and $flushpipe$ depend only on whether rf_addrw is equal to 0 or not. Therefore, we model rf_addrw to have either value 0 or a value i $(i \neq 0)$ that represents the remaining 63 non-zero possibilities. We are only interested in checking whether $if\_insn[21{:}16]{=}rf\_addrw$. Therefore, we consider only three values for $if\_insn[21 : 16]$: 0, i $(i \neq 0)$ and j $(j \neq 0,$

$j \neq i$). We consider 3x2=6 combinations of numeric values for if_insn[21:16] and rf_addrw. A similar data abstraction technique has been used in [38]. Table 7.5 shows the SER computed using our decomposition methodology.

Table 7.5: SER for the sel_a variable in the control logic of an OR1200 processor

	Without decomposition			With decomposition			
	States	Time	Invariant	States	Cases	Time (sec)	Invariant
sel_a	$O.o.M.$	-	-	$3,145,729$	6	16.48	1.0 x 10^{-8}

In general, control logic of processors can depend on datapath variables . Considering a case-splitting based on the values of such variables can potentially lead to a very large number of cases. However, data-dependant control logic typically is concerned with only a few possible data values for a particular case. Therefore, the number of cases to be verified is small in practice.

7.7 Chapter summary

In this chapter, we have presented an automatic decomposition technique to scale probabilistic verification for hardware designs. We have shown that our technique can be used to significantly improve the scalability of our SHARPE methodology by demonstrating it on several case studies.

CHAPTER 8

ABSTRACTIONS IN RTL SOURCE CODE FOR SCALING OUR SHARPE METHODOLOGY

8.1 Introduction

In this chapter, we describe another novel technique to improve the scalability of our SHARPE methodology. We present a *value-based interval abstraction* technique to scale formal probabilistic analysis in the SHARPE methodology. We intend our value-based interval abstraction for datapath-intensive RTL designs which are often purely combinational. However, in principle, our technique could be extended to work with sequential designs as well.

For combinational designs, we wish to determine the probability that input vectors satisfy a condition specified in the statistical property (Section 3.3). In our SHARPE methodology, we compute this probability by analyzing every possible input vector of the design. With our abstraction technique, we reduce the number of input vectors that need to be considered while preserving the correctness of the verification result with respect to the property. As in the case of non-probabilistic RTL verification [44],[92] we perform our property-specific abstraction by using static analysis at the RTL design source code level (Figure 8.1).

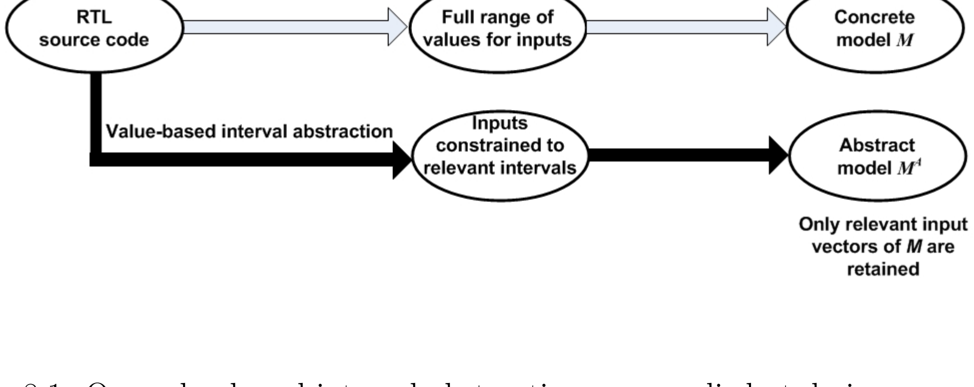

RTL source code
Full range of values for inputs
Concrete model M
Value-based interval abstraction
Inputs constrained to relevant intervals
Abstract model M^A
Only relevant input vectors of M are retained

Figure 8.1: Our value-based interval abstractions are applied at design source code level, leading to a smaller number of input vectors that need to be analyzed.

8.1.1 Value-based interval abstraction

We are interested in properties Φ of the form $P[f_P(V) < T]$, where f_P is a real valued function that is defined over the set of RTL variables V and T is a user-specified value. $f_P(V) < T$ is a *predicate* that evaluates to TRUE or FALSE for an input vector depending on the numeric values assigned to V in that input vector (for example, $Delay < T$). When we verify $M \models \Phi$, we are actually computing the probability of all input vectors where the predicate is TRUE. Therefore, among all possible input vectors of M, only those vectors where the predicate $f_P(V) < T$ evaluates to TRUE are *relevant*. Each input vector of M corresponds to a unique assignment of values to the input variables in the RTL design (Definition 9). We restrict inputs to intervals of values (*value-based intervals*) such that only the relevant input vectors of M are considered.

Value-based intervals for RTL inputs can be used to construct an abstract model M^A by discarding all the irrelevant input vectors. In order to derive these intervals, we first consider the predicate $f_P(V) < T$ as a symbolic constraint on the values of variables in V. We rewrite such symbolic constraints as constraints that are expressed over the input variables in the support $Sup(V)$ (Definition 7). We achieve this by performing *symbolic execution* [93] on the RTL source code. We use an integer constraint solver to obtain lower and upper bound values of the intervals for these inputs by maximizing (or minimizing) the value of the input variable for which the predicate $f_P(V) < T$ is satisfied. We use these intervals and construct the smaller (Definition 10) abstract model M^A from the concrete model M. We then check if $M^A \models \Phi$ which we show is equivalent to verifying $M \models \Phi$.

We demonstrate the application of our technique on multiple practically useful designs that could not otherwise be formally verified due to their large sizes (Definition 10). Datapath verification is notoriously hard [94] to get guarantees for. Our abstractions are intended to verify probabilistic performance on the datapath of RTL designs. We consider filters and FFT blocks that are widely used in communication/DSP systems, as well as an H.264 decoder. We show consistent and significant reductions in size which make probabilistic verification of these RTL designs feasible. For example, we are able to verify a module in the H.264 decoder for which the concrete model corresponds to over 10^{80} possible input vectors. In the the abstract model, there are only approximately 10^9 input vectors that need to be analyzed.

8.1.2 Chapter organization

The rest of this chapter is organized as follows. In Section 8.2, we present some background that describes the framework in which we perform our abstraction. In Section 8.3, we describe the intuition for the effectiveness and soundness of our value-based interval abstraction. In Section 8.4, we detail each of the steps in our abstraction technique. In Section 8.5, we present experimental results that demonstrate the effectiveness of our approach.

8.2 Framework for our value-based interval abstraction

We now present the framework in which we describe our abstraction technique for scaling our SHARPE methodology.

8.2.1 Variables in RTL designs

We shall use the following example combinational RTL source code in order to illustrate the steps of our abstraction technique (Section 8.4).

```
always @(posedge clk)
if (sel)
   O1 <= I1 + I2;
else
   O1 <= 4*I2 + I3;
end
```

where $I1$, $I2$, $I3$ are the inputs, sel is a Boolean control variable and $O1$ is the output. All input and output variables are of 10 bits and can therefore be assigned 1024 different numeric values (Section 2.3).

In RTL source code, a variable v can be assigned integer values in the range $[l\ u]$ where l and u denoted the lower and upper bounds, respectively. For an N-bit variable, we assume the default range of values to be $[0\ 2^N\text{-}1]$.

The values assigned to the control variables activates/selects one among several possible paths in the RTL design. Each path may result in a different assignment to the value of a variable. Therefore, for each path i, a signal function f_i (Definition 8) needs to be defined for each variable v that is of interest. However, $Sup(v)$ is computed by considering all possible paths. In the RTL example, there are two possible paths (*sel*=0 and *sel*=1) and $Sup(O1) = \{I1, I2, I3\}$.

8.2.2 Statistical properties in RTL

For our abstraction technique, we consider the same types of properties described in Section 7.2.

Let $f_P(\Pi)$ be a real-valued expression defined over a set of RTL variables Π. The expression $f_P(\Pi)$ may contain arithmetic or Boolean operators. We define probabilistic invariants Γ [16] based on some condition that $f_P(\Pi)$ must satisfy, given by

$$\Gamma \triangleq P[f_P(\Pi) < T] \tag{8.1}$$

where T is a real-valued constant and $f_P(\Pi) < T$ is the *predicate* that is of interest to us. $P[predicate]$ denotes the probability that the predicate is satisfied (i.e., TRUE) by an assignment of concrete numeric values to Π. Γ is the probability that an input vector will satisfy $f_P(\Pi) < T$. In place of $<$, we allow for the use of other relational operators as well.

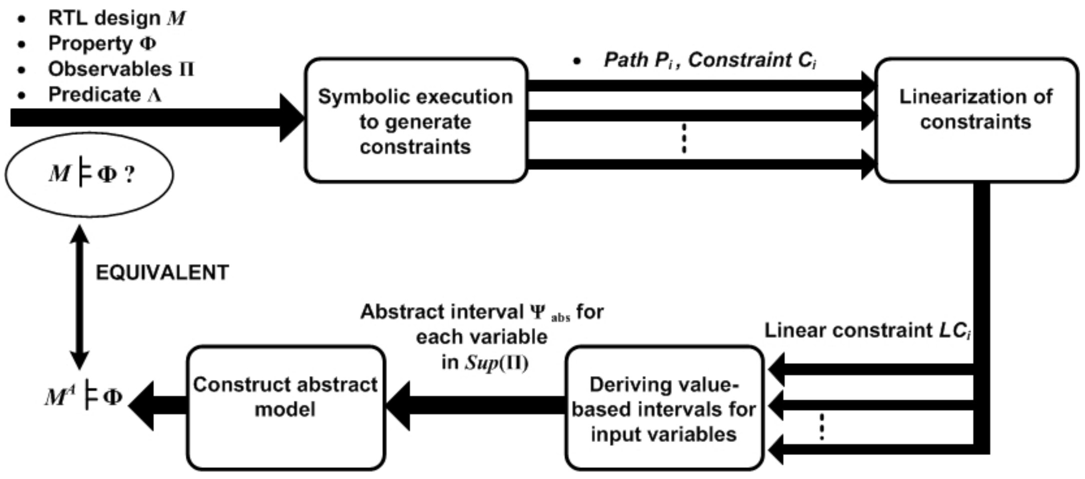
We formally define probabilistic performance properties Φ of the form

$$\Phi \triangleq \Gamma \leq p \tag{8.2}$$

where $p \in$[0,1] is a specification of the design. We allow logical comparison operators other than $\leq$ to be used for comparing the probabilistic invariant with p.

We employ probabilistic verification to verify whether an RTL M satisfies a property Φ, denoted by $M \models \Phi$. The verification procedure for properties described in Equation 8.2 involves the computation of the invariant Γ and comparing it with p. If p is not specified, verifying $M \models \Phi$ is equivalent to the computation of Γ.

We now describe an example of a property Φ. Let Φ be a statistical timing property (Chapter 5). We wish to compute the probability that the RTL delay meets a timing requirement T. The delay of an RTL block can be expressed as a combination of the macromodels of all the operators in the block (Section 5.2). We consider delay in terms of RTL assignment statements. The delay of an RTL assignment depends on the operator and the values of the operands in the RHS. We consider real-valued analytical functions f_P, which we call macromodels, that estimate the delay of an operator based on the value of the operands. For each RTL operator, we derive the macromodel f_P by performing extensive simulations of a gate-level implementation of the operator (Section 5.2). In the



• RTL design M
• Property Φ
• Observables Π
• Predicate Λ
$M \models \Phi$?
Symbolic execution to generate constraints
• Path P_i, Constraint C_i
Linearization of constraints
EQUIVALENT
Abstract interval Ψ_{abs} for each variable in $Sup(\Pi)$
Linear constraint LC_i
$M^A \models \Phi$
Construct abstract model
Deriving value-based intervals for input variables

Figure 8.2: Block diagram showing the stages of our predicate-based data abstraction technique. The labels on the arrows show the outputs of each stage.

RTL example, the delay of $I1 + I2$ can be computed by using the macromodel $f_P(I1, I2)$ corresponding to the specified adder implementation. With a ripple carry adder implementation, f_P is a polynomial function of the number of carry bits in the addition of $I1$ and $I2$.

8.3 Our abstraction using value-based intervals

In this section, we define and establish criteria for performing value-based interval abstractions on probabilistic systems of our interest, namely RTL designs. We perform our abstraction by statically analyzing the RTL source code.

Let Λ be the predicate that is specified in the property Φ. Let Π be the set of RTL variables over which Λ is expressed. We wish to verify whether $M \models \Phi$. In other words, we wish to compute the probability of occurrence of input vectors where Λ is TRUE. This can be achieved by considering a smaller model M^A that contains all the input vectors of M where $\Lambda =$ TRUE. M^A is the abstract model corresponding to the concrete model M. Since each input vector of M corresponds to a unique assignment of concrete numeric values to the input variables $Sup(\Pi)$, the construction of M^A corresponds to retaining only those values of inputs for which Λ = TRUE. All other values of the inputs are inconsequential and can be lumped together by using a single representative value. As a result, the abstract model M^A is typically smaller (Definition 10) than the concrete model$M$. This forms the basis of our data abstraction technique.

$\Lambda =$ TRUE imposes a constraint on the values that can be assigned to the variables in Π.

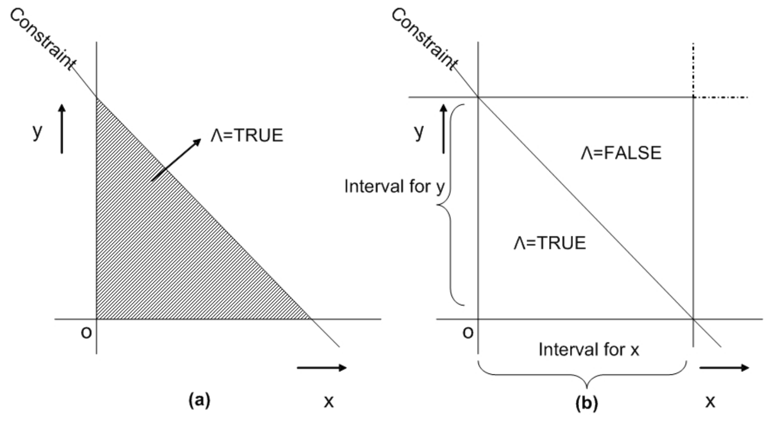


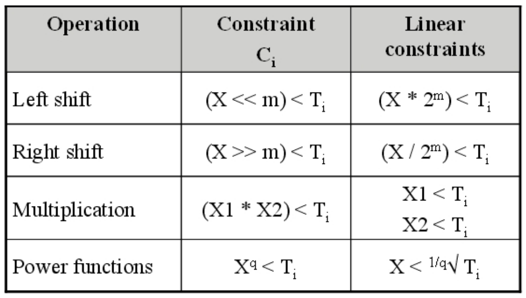
Figure 8.3: a) Exact constraint for values of inputs x,y. b) Conservative value-based intervals for inputs x,y.

In order to construct an abstract model M^A, we wish to use this constraint to determine the concrete values of the input variables $Sup(\Pi)$ that need to be considered. We achieve this by using RTL symbolic execution [93] to rewrite the constraint on variables Π as a constraint on inputs $Sup(\Pi)$. Symbolic execution statically explores all possible paths through the RTL design and determines a constraint C_i on the values of $Sup(\Pi)$, for each path i.

Each constraint C_i specifies an exact bound on the values of $Sup(\Pi)$ for which Λ =TRUE on path i. However, in general, C_i is specified jointly over multiple input variables in $Sup(\Pi)$. Since we define assignments to input variables independently, we wish to obtain constraints that are specified over individual input variables. Therefore, we use a constraint solver (ILP) with C_i to derive *value-based intervals* for each input variable in $Sup(\Pi)$. Since we wish to compute the probability of Λ = TRUE for all paths through the design, we construct an abstract interval ψ_{abs} for v that includes all the values from the intervals computed using each C_i.

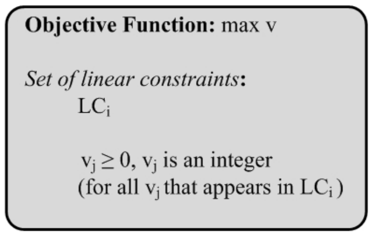
Finally, we use the abstract intervals for each $v \in Sup(\Pi)$ in order to construct the abstract model M^A. We then verify $M \models \Phi$ by checking $M^A \models \Phi$

For each $v \in Sup(\Pi)$, we consider all values of v such that there is a possible assignment of values to the other input variables $\in Sup(\Pi) \setminus \{v\}$ that would satisfy Λ =TRUE. Therefore, the value-based intervals that we construct are conservative (Figure 8.3). M^A may contain input vectors from M in which Λ=FALSE. However, we do not discard any state from M in which Λ= TRUE. Therefore, our abstraction is sound with respect to the probabilistic property of interest.



Operation	Constraint C_i	Linear constraints
Left shift	$(X << m) < T_i$	$(X * 2^m) < T_i$
Right shift	$(X >> m) < T_i$	$(X / 2^m) < T_i$
Multiplication	$(X1 * X2) < T_i$	$X1 < T_i$ $X2 < T_i$
Power functions	$X^q < T_i$	$X < \sqrt[1/q]{T_i}$

(a)



Objective Function: max v

Set of linear constraints:
LC_i

$v_j \geq 0$, v_j is an integer
(for all v_j that appears in LC_i)

(b)

Figure 8.4: a) Rules for linearizing constraints, b) ILP instance for computing upper bound of v.

8.4 Algorithm for value-based interval abstraction

We wish to construct an abstract model M^A in order to determine P[($f_P(\Pi) < T$)], where $f_P(\Pi) < T$ is the predicate of interest, Λ. Figure 8.2 shows the different steps in our abstraction technique. We now describe each of these steps in detail.

We shall illustrate our technique using the RTL example in Section 8.2. Let P[$O1 < 100$] be the invariant that we wish to compute.

8.4.1 Symbolic execution to generate constraints

We use symbolic execution to explore each possible path i in the RTL design and generate a corresponding constraint C_i on the input variables. For each path i, let v=$f_i(Sup(v))$ where f_i is the signal function for variable v. Therefore, a predicate $f_P(v) < T$ can be written as $f_P(f_i(Sup(v))) < T$ which is a constraint C_i on the values of the input variables $Sup(v)$.

Symbolic execution refers to the execution of a single path with symbolic inputs. Symbolic execution of a path generates symbolic expressions that are a logical conjunction of the guards (conditional expression of branches) and assignments to the variables used in guards along that path. Symbolic execution is well known in software [95]. In recent work [93], symbolic execution has been introduced for RTL source code. The RTL symbolic execution engine works on the CFG and expression tree structure of each RTL "program" statement. For each single statement or conditional expression in the design, the expression tree structure exactly records the corresponding assignment or expressions and is linked to corresponding CFG node.

The RTL symbolic execution engine [93] considers exactly one path i of the CFG at any given time. At each CFG node in path i, the corresponding expression tree is traversed and output as symbolic expression. When a variable $v \in \Pi$ is encountered, the corresponding signal function $f_i(Sup(v))$ is output by the engine. Every occurrence of $v \in \Pi$ in $f_P(V) < T$ is substituted with the corresponding signal function $f_i(Sup(v))$. We thus obtain the constraint C_i on $Sup(v)$ corresponding to the path i. We repeat this for all possible paths i in the RTL design and obtain the corresponding constraints on the input variables. Further details of the RTL symbolic execution engine, along with an optimization strategy for path exploration, can be found in [93].

In the RTL example (Section 8.2), we obtain the linear constraints $I1+I2$ <100 and $4 * I2+I3$ <100 corresponding to the paths sel=1 and sel=0, respectively.

8.4.2 Linearizing the constraints

A linear constraint will have *terms* on the left-hand side that are separated by +/- signs. Each term can be a variable multiplied by a constant numeric value. Since datapaths of RTL designs comprise mainly of arithmetic operators, each constraint C_i is typically a linear constraint that is defined over the input variables. However, if the constraints are not linear, we transform them into a set of linear constraints.

In Figure 8.4, we outline a set of rules for transforming non-linear operations into linear constraints. All the rules that we have defined can be extended easily for relational operators other than <. The terms $(X >> m)$ and $(X << m)$ represent shifting the variable X by m bits towards the right and left, respectively. These operations are equivalent to division and multiplication by 2^m, respectively.

If there is a term corresponding to multiplication of non-constants, we split the term into a set of linear constraints. Let $X1 * X2$ be the non-linear term in the constraint C_i. We treat $X1 * X2$ as an input variable and compute its upper bound T_i. We then rewrite this term as two linear constraints $LC1$ and $LC2$, as in Figure 8.4.

Concatenation of variables is supported in RTL designs. Let $X1$, $X2$ be $n1$-bit and $n2$-bit variables, respectively. The variables can be treated as strings of bits and concatenated to get a string of $n1+n2$ bits, represented by the term $\{X1, X2\}$. This algebraic operation corresponding to this term can be rewritten as the linear expression $X1 * 2^{n2} + X2$.

We apply the above rules recursively to each non-linear constraint and derive a set of linear constraints LC_1 to LC_{NumLC}, where $NumLC$ is the total number of linear constraints. The rules that we have defined in Figure 8.4 are not complete, since there RTL designs support several other operators. However, our rules are sufficient for the large class of datapath RTL designs that are used in DSP systems.

In general, the predicate can be expressed as a polynomial function over variables Π. In such cases, we can define rules to convert non-linear terms such as $(X^q < T_i)$ into corresponding linear terms $(X < \sqrt[1/q]{T_i})$ (for $q > 1$ and non-negative X). However, in this chapter, we only consider predicates that are linear functions over Π.

8.4.3 Deriving value-based intervals for input variables

We consider a linear constraint LC_i. For each input variable v that appears in the expression for LC_i, we wish to compute the interval $\psi^{(i)} = [l^{(i)}\ u^{(i)}]$ of values that can be assigned to it. We achieve this by formulating an instance of the ILP problem.

Figure 8.4 shows the ILP instance for computing upper bound of v. Each ILP instance comprises one linear constraint LC_i, and a set of constraints that force all variables v_j (including v) that appear in LC_i to be non-negative integers. The objective of the the ILP problem is to maximize the integer value of v such that all the constraints are satisfied.

If the ILP instance has an optimal solution, we set $u^{(i)}$ to be equal to that solution. If the ILP instance is "unbounded", it implies that all non-negative integer values for v will satisfy the given set of constraints. In this case, we set $u^{(i)}$ to the default upper bound (i.e., 2^N -1, as in Section 8.2.1 and mark v as a *free variable*. Similarly, we compute $l^{(i)}$ by changing the objective function to *min* v.

We perform this interval computation for all linear constraints. If a variable is marked to be free, we do not compute its intervals for any of the subsequent linear constraints.

Finally, we compute the most conservative interval for each input variable v, by computing the union of the intervals $\psi^{(i)}$ that are obtained using the linear constraints LC_i. We call this the *abstraction interval* ψ_{abs} for the variable v.

$$\psi_{abs} = \bigcup_{i=1}^{NumLC} \psi^{(i)} \tag{8.3}$$

In the RTL example (Section 8.2), we compute the intervals [0 99] for both $I1$ and $I2$ based on the *sel*=1 path. Based on the constraint in the *sel*=0 path, we compute the intervals [0 24] and [0 99] for the variables $I2$ and $I3$, respectively. After computing the union of the two intervals for $I2$, we observe that ψ_{abs} for all $I1$, $I2$ and $I3$ is equal to [0 99].

If there is a "-" sign in the left-hand side of the constraint, all the variables that appear in the constraint will be unbounded. For example, it is possible for each value of the variable $X1$ (and $X2$) to satisfy the constraint $X1 - X2 < 100$. However, it is possible to compute a lower bound for $X1$ if the $>$ operator is used in the constraint instead of $<$.

In this work, we have considered *unsigned* arithmetic where RTL variables are interpreted to have non-negative integer values. However, the rules in Figure 8.4 can be easily extended to the case of *signed* arithmetic, where negative integer values are also allowed.

In principle, we can derive value-based intervals of sequential variables as well. In this case, we would need to "unroll" the design and compute the intervals for the sequential variable in each time step. We would then choose the largest interval across the time steps and define it to be the abstract interval for the sequential variable.

We use the abstraction intervals in order to describe the abstract model M^A.

Table 8.1: Sizes of the ILP instances that we use to derive the intervals for input variables. We also list the total abstraction time for each design.

Design name	Predicate name	No. of inputs	No. of paths	No. of constraints	Time
`fir`	p8	6 (8-bit)	1	1	<10s
`elliptic`	p9	12 (8-bit)	1	1	<10s
`fft8`	p10	8 (16-bit)	4	4	<10s
`Inter_pred_LPE`	p1	5 (8-bit)	180	131	<10s
`Inter_pred_LPE`	p2	5 (8-bit)	180	131	<10s
`Inter_pred_LPE`	p3	5 (8-bit)	180	131	<10s
`Inter_pred_pipeline`	p4	32 (8-bit)	1936	1932	<10s
`Inter_pred_pipeline`	p5	3 (8-bit)	8	5	<10s
`Inter_pred_sliding_window`	p6	19 (8-bit)	29	21	<10s
`Inter_pred_sliding_window`	p7	16 (8-bit)	29	21	<10s

8.5 Experimental results

We implement the RTL symbolic execution algorithm using C++. We perform all our experiments on an Intel i5 2.67GHz quad-core machine with 16GB of memory. We use *lpsolve* [96], an open source ILP solver, in order to solve the set of integer linear constraints and derive the value-based intervals for the inputs.

We demonstrate the effectiveness of our methodology by applying it on two sets of data-intensive RTL designs. The first set of designs comprise `fir`, `elliptic` and `fft8` all of which are high-level synthesis benchmarks [85] that are commonly used in communication/DSP systems. Filter coefficients are fixed and stored in a ROM table. We consider constant values for these coefficients. `Inter_pred_LPE`, `Inter_pred_pipeline` and `Inter_pred_sliding_window` are different modules from a real-world H.264 decoder[‡] and constitute our second set of designs. In this work, we analyze each of the H.264 modules independently.

In Table 8.1, *Number of paths* represents the total number of paths that needs to be explored during symbolic execution. This number is with regard to the variables which appear in the predicate (described in Table 8.2) of the specified property. Since our designs do not contain multiplication of variables with each other, there should be exactly one linear constraint per path (Section 8.4.2). However, in some paths, all variables are assigned a constant value, and therefore the predicate is vacuously TRUE or FALSE. We discard these paths and consider only the linear constraints (*Number of constraints*) corresponding to the remaining paths while formulating the ILP instances.

In Table 8.1, *Number of inputs* represents the total number of input variables (and their bitwidths) on which the variables in the predicate depend, i.e., $Sup(\Pi)$. Therefore, each of these input variables appear in at least one of the linear constraints. However, in each linear constraint, at most a small subset of these input variables are present. Therefore, each ILP instance is small and the corresponding runtime of *lpsolve* is also negligibly small. The total abstraction time, which includes the time for both the generation of linear constraints and the ILP solver, is less than 10 sec in all our experiments.

Table 8.2: Description of the predicates that we use to specify properties of our interest. To verify these properties, we compute P[Predicate = TRUE].

Predicate name	Predicate description
p1	bilinear0_A + bilinear0_B < 8
p2	bilinear0_A + bilinear0_B < 6
p3	bilinear0_A + bilinear0_B < 4
p4	8*Inter_blk_mvx + Inter_blk_mvy < 2
p5	Inter_pred_out0 < 200
p6	Inter_pix_copy0 < 2
p7	Inter_H_window_0_0 < 3
p8	y<30
p9	outp < 30
p10	s3r < 127

For each of the designs that we consider, we specify a property that is defined over some internal data variables. Table 8.2 provides a description of all the predicates that we define in order to specify the properties of interest. `fir` and `elliptic` are filter designs in which it is common to check whether the output is less than a user-defined threshold. Therefore, we define the predicates p8 and p9 over the output variables y and

[‡]www.opencores.org

outp, respectively. Although not exact models, these predicates can be viewed as being representative of certain performance properties of the design. For example, y can be an input to an adder block for which the performance constraint requires that $y < 30$, as in p8. For our experiments, we consider predicates that are linear functions over a set of RTL variables and we use the "<" relational operator. For each of the H.264 modules, we consider multiple predicates.

Table 8.3: Reductions in size of models that we achieve by using our abstraction.

		Concrete model	Abstract model
Design name	Predicate name	No. of input vectors	No. of input vectors
`fir`	p8	2^{56}	2^{28}
`elliptic`	p9	2^{96}	$\approx 2^{29.73}$
`fft8`	p10	2^{16}	2^{16}
`Inter_pred_LPE`	p1	2^{40}	$\approx 2^{15.85}$
`Inter_pred_LPE`	p2	2^{40}	$\approx 2^{14.04}$
`Inter_pred_LPE`	p3	2^{40}	$\approx 2^{11.61}$
`Inter_pred_pipeline`	p4	2^{256}	2^{28}
`Inter_pred_pipeline`	p5	2^{24}	$\approx 2^{22.95}$
`Inter_pred_sliding_window`	p6	2^{152}	$\approx 2^{30.11}$
`Inter_pred_sliding_window`	p7	2^{128}	2^{28}

Table 8.4: Demonstrating correctness of our abstractions using smaller, contrived versions of benchmarks designs since the concrete models cannot be verified for the actual sizes. The verification result is P[Predicate = TRUE].

		Concrete model		Abstract model	
Design name	Predicate	No. of input vectors	Result	No. of input vectors	Result
`fir` (small)	$(y < 12)$	2^{24}	$4.65\text{x}10^{-4}$	$\approx 2^{15.57}$	$4.65\text{x}10^{-4}$
`elliptic` (small)	$(outp < 30)$	2^{30}	$9.65\text{x}10^{-7}$	$\approx 2^{14.39}$	$9.65\text{x}10^{-7}$

Table 8.3 demonstrates the reduction in model size Definition 10) provided by our abstraction method. Formal probabilistic analysis (Section 3.3) times out while trying to construct any of the concrete DTMC models, and therefore these designs can not be verified. We estimate the number of input vectors in the concrete model based on the total number of combinations of values that can be assigned to the corresponding input

variables. There is no reason to believe that the RTL inputs, which are data variables, are restricted and we use their full range of values to estimate the size of the concrete model. In all the designs, with the exception of `fft8`, we are able to obtain significant reductions in size by using our abstraction technique and formal probabilistic analysis successfully verifies them. We approximately represent the number of input vectors in the abstract model as powers of 2, in order to facilitate comparison with the concrete number of input vectors.

p1, p2 and p3 are all the same predicates that differ only in the constraint values that are specified in the RHS. We observe that as the constraint values get smaller, the number of relevant data values (and hence states) also decrease. Our technique is extremely effective when the predicate is true for only a small fraction of the possible data values. Although our technique would still be sound for larger constraint values, the reduction that we achieve may be far more modest.

Since verification could not be completed for the concrete models in Table 8.3, we do not present a comparison of the verification results for these designs. Instead, as a proof of concept, we construct smaller versions (smaller bitwidth for inputs) of `fir` and `elliptic` and verify that the results computed using the concrete model and the abstract model are exactly the same (Table 8.4). We choose the smaller bitwidths such that formal probabilistic analysis is feasible for the concrete model. For example, we consider 3-bit data for `fir`(small) and the runtime is <10s. We do not consider a smaller fft since our abstraction does not provide any reductions for it (Table 8.3).

In `fft8`, we are not able to demonstrate any reduction in size using our abstraction. This is due to "-" operator in the RTL design. As described in Section 8.4.2, a "-" sign on the left hand side of a "less than" constraint will result in unconstrained values for all the input variables. `JPEG encoder` is another design for which we cannot obtain reductions. The module of the encoder design that we consider is control-intensive, and therefore the number of paths that need to be explored by the symbolic execution algorithm is huge (Section 8.4.1). For this design, we stopped the symbolic execution engine after 1hr of exploring paths and generating the corresponding constraints. We could not use this incomplete set of constraints since all possible paths in the design need to be considered in order to guarantee correctness of our abstraction.

In all the designs mentioned above, we find that the control paths are independent of the values of data. This is fairly common for a large class of data-intensive designs that are commonly used in DSP systems. For example, typical control variables that we observe are *counters* that are not data-dependent. Since control variables control the selection of paths and since we wish to consider all possible paths, we cannot constrain the values of such variables. In non-DSP designs, the control variables may depend on input data variables, and therefore all such input variables must also be unconstrained. In these cases, the overall reduction achieved by our abstraction technique may not be very large.

In RTL designs, it is possible that arithmetic operations can result in an overflow (or underflow) due to insufficient number of bits that are assigned to store the results. Ideally, such incorrect computations should not be allowed. Our technique cannot detect such overflow errors. Typically, overflows are prevented in DSP designs by assigning sufficient number of bits to the different variables in the design. Our abstraction techniques can be applied on such designs.

8.6 Chapter summary

In this chapter, we have presented a property-specific value-based interval abstraction technique that is applied at the source code level. We intend our abstraction for scaling probabilistic verification of hardware designs. In future work, we plan to broaden the applicability of our abstraction technique by considering designs that use signed arithmetic. We also plan to circumvent the path explosion problem faced by our approach while handling control-intensive designs.

CHAPTER 9

VERIFYING MASSIVE RTL DESIGNS USING STATISTICAL MODEL CHECKING

9.1 Introduction

Our compositional reasoning approach (Chapter 7) and value-based interval abstraction (Chapter 8) significantly improve the scalability of the SHARPE methodology. However, these techniques are not sufficient to make the SHARPE methodology feasible for massive RTL designs such as those pertaining to multicore systems which contain a large number of both combinational and sequential elements.

In this chapter, we employ statistical model checking (Section 2.4.1) to verify probabilistic properties in large RTL designs. Unlike probabilistic model checking that employs numerical analysis on the DTMC model, statistical model checking performs verification by simulating the design. Consequently, statistical model checking is highly scalable. We illustrate our approach by verifying properties of dynamic power management schemes in RTL.

9.1.1 Dynamic power management schemes

Dynamic power management (DPM) schemes such as clock gating, power gating, dynamic voltage and frequency scaling (DVFS) are important strategies to save power in multicores [10],[11],[12],[13]. Due to stringent power constraints of present day systems, DPM schemes have becoming increasingly complex. Therefore, the implementation of DPM schemes is a very challenging problem [97], particularly at the lower levels of design such as RTL. In this work, we present an efficient methodology for verifying a DPM technique, namely power gating, in RTL using statistical model checking [45].

The performance of a DPM scheme depends on runtime statistics of the RTL design. Power gating incurs overheads in delay and power. As a result, power gating provides savings in power only when applied to idle periods that exceed a certain duration [13]. The duration of a component's idle period is statistically distributed based on the runtime behavior of design which itself is based on the distribution of the input values.

Since the exact duration of an idle period is not known apriori, power gating is implemented by predicting the duration of an idle period based on recent history of inactivity. Overestimation of the idle period duration can affect the safety of a DPM scheme while underestimation can affect the efficiency. Safety and efficiency of a DPM scheme then are statistical properties that must be checked.

In order to check the safety and efficiency of power gating, we need to determine the statistical distribution of the duration of a component's idle period. The statistics of an RTL component depends on its interaction with all the other components in the system. Therefore, the safety and efficiency properties must be checked by analyzing the statistical behavior of the full system. Such system-level analysis can become very complex for multicore RTL designs. For example, the RTL design of OpenSPARC [98] contains over 4 millions lines of code and 250K flipflops. State-of-the-art DPM schemes [11],[12] perform RTL simulations, which is the only practical approach, for performing system-level analysis of such massive designs. However, existing simulation-based analysis provides only average-case estimates of safety and efficiency. In this work, we employ statistical model checking to provide a better quality of assurance while checking the safety and efficiency properties of a DPM scheme.

9.1.2 Verifying properties of DPM schemes in RTL

In this work, we apply statistical model checking to verify power gating properties in RTL. Statistical model checking is a scalable verification technique that provides statistical guarantees by simulating several sample execution paths of the full system. By considering a sufficient number of sample paths, statistical model checking guarantees [45] that the verification results are within tolerable bounds of error. Such guarantees cannot be provided by ordinary simulation-based techniques that are currently used to evaluate the performance of DPM schemes.

We specify the safety and efficiency requirements of power gating schemes as properties in probabilistic computational tree logic (pCTL) [22]. We define these properties in terms of relevant RTL signals that indicate the "idleness" (inactivity) of the RTL blocks of interest. We use a commercial RTL simulator to simulate the entire RTL design and observe the values of these relevant RTL signals in each hardware clock cycle. We then employ statistical model checking which uses these observed simulation samples to verify the specified pCTL properties on the RTL design.

We demonstrate our approach on the RTL design of OpenSPARC, a publicly available industry-strength multicore processor [98]. We use the statistical model checking tool

Ymer [58] for all our experiments. We verify the safety and efficiency properties of several power gating schemes by considering several blocks in the floating-point graphics unit (FGU) of the processor core. For example, we verify that timeout-based power gating schemes [13] cannot be more than 50% efficient while having safety greater than 85%. For the efficiency property, Ymer uses approximately 500 sample paths and guarantees that the error in the verification result is less than 1%.

Probabilistic model checking [23] which is exact, unlike statistical model checking, has been used to verify properties of multicore power management [97],[99]. However, these techniques operate on the higher level models where scalability issues are not as severe as in RTL. In RTL designs, probabilistic model checking is typically practical when applied to individual RTL blocks [80]. However, for verifying properties of DPM schemes, we need a scalable methodology that can perform a system-level statistical analysis of the RTL design.

9.1.3 Chapter organization

The rest of the chapter is organized as follows. In Section 9.2, we present some preliminaries on dynamic power management using power gating. We describe the statistical properties pertaining to safety and efficiency of dynamic power management schemes. In Section 9.3, we describe how we apply statistical model checking to verify power gating properties in OpenSPARC RTL. In Section 9.4, we present the results from obtained from verifying power gating properties on several RTL blocks of OpenSPARC.

9.2 Dynamic power management: Power gating

We now present some background on dynamic power management (DPM) schemes in hardware, as discussed in [13]. In this work, we shall focus on power gating schemes that are implemented in RTL. We also describe statistical performance requirements that must be met by these schemes.

Power gating schemes are complex to implement due to the overheads associated with switching the power on and off [13]. For idle periods of short duration, the power overheads due to switching the power supply off and restoring it could outweigh the reduction in power achieved by keeping the power off during the idle period. Therefore, power gating must be judiciously used only for idle periods that are sufficiently long enough to compensate for the power overheads.

Definition 20. The *breakeven period*, denoted by K_{BE}, is the minimum number of cycles required in an idle period such that power gating results in a net reduction of power consumption. Each idle period longer than K_{BE} cycles is an opportunity for saving power.

The duration of an RTL block's idle period varies statistically according to the values assigned to the RTL inputs. Since the idle periods are not known apriori, the best practical approach is to predict the duration of an idle period based on the recent history of idleness/inactivity (e.g., duration of previous idle periods). Power gating is applied if the predicted duration is greater than K_{BE} cycles. However, such approaches may *overpredict* or *underpredict* the duration of the idle period which can potentially degrade the performance of power gating.

An overprediction may predict the upcoming idle period to be longer than K_{BE} cycles (Definition 20) when it is actually shorter than K_{BE}. If power gating is applied during this idle period based on the incorrect prediction, it could result in a net increase in power consumption compared to when power gating is not applied. Therefore, such overpredictions are referred to as *safety concerns* for the performance of power gating schemes. Similarly, an underprediction results in a block being kept on during a long idle period ($\geq K_{BE}$) where power gating could have reduced power consumption. Therefore, underpredictions causes the power gating schemes to miss out on potential power saving opportunities and are referred to as *efficiency concerns* for the performance of power gating schemes.

Typically, overpredictions and underpredictions are tolerable in the system if they occur with low probabilities. In [13], the upper bounds on these probabilities are referred to as the *safety* and *efficiency* requirements of the power gating scheme. These requirements can be broadly defined as:

Definition 21. The *safety* of a scheme is greater than (1-p_S)x100% if the the probability of mistaking an idle period to be a power saving opportunity is less than p_S

Definition 22. The *efficiency* of a scheme is greater than (p_E)x100% if the the probability of missing out on a power saving opportunity is less than 1-p_E

We now consider the safety and efficiency requirements for two power gating schemes [13], namely 1) Timeout-based power gating and 2) Adaptive power gating.

9.2.1 Timeout-based power gating

Timeout-based power gating waits for a fixed number of idle cycles K_{TO} before switching off the power supply. This is based on the idea that if an RTL block is idle for K_{TO} cycles, it is likely to remain idle for a further K_{BE} cycles where power savings can be obtained (Definition 20). With a sufficiently large value of K_{TO}, the safety requirement can be met. However, in such schemes, the first K_{TO} cycles are unexploited for power savings. Therefore, the safety and efficiency properties of the timeout-based scheme are given by

- **P1** (Safety property for timeout-based scheme):

$$P[idle\_cnt < K_{BE} + K_{TO} \mid idle\_cnt \geq K_{TO}] < p_S \tag{9.1}$$

where *idle_cnt* refers to the number of cycles in the current idle period. $P[A|B]$ denotes the probability of event A occurring, given that event B has already occurred.

- **P2** (Efficiency property for timeout-based scheme):

$$P[idle\_cnt < K_{TO}] < (1 - p_E) \tag{9.2}$$

K_{TO} is often pre-determined by statically analyzing the probability distributions of idle period durations. Typically, K_{TO} is selected by trading off safety for efficiency.

9.2.2 Adaptive power gating

Adaptive schemes predict the duration of the upcoming idle period based on the duration of the preceding idle periods. As a result, unlike in timeout-based schemes, the first K_{TO} cycles in each idle period are not wasted. We consider an adaptive scheme that predicts the number of idle cycles K_{pre} as an average of the durations of the previous two idle periods [13], given by

$$K_{pre} = \frac{(idle\_cnt2 + idle\_cnt1)}{2} \tag{9.3}$$

where *idle_cnt*1 and *idle_cnt*2 refer to the durations of the previous two idle periods.

If $K_{pre} \geq K_{BE}$, the RTL block is switched off as soon as the idle period begins. However, the actual idle period *idle_cnt* may be shorter than K_{BE} (overprediction). Similarly, *idle_cnt* may be longer than K_{BE} when $K_{pre} < K_{BE}$ (underprediction). Therefore, the safety and efficiency properties of adaptive power gating is defined as follows.

- **P3** (Safety property for adaptive scheme):

$$P[(idle\_cnt < K_{BE}) \mid (K_{pre} \geq K_{BE})] < p_S \tag{9.4}$$

- **P4** (Efficiency property for adaptive scheme):

$$P[(idle\_cnt < K_{BE}) \mid (K_{pre} < K_{BE})] > p_E \tag{9.5}$$

9.3 Verifying power gating in OpenSPARC RTL

We now describe power gating schemes in the context of the OpenSPARC RTL. We also describe how we formally represent and verify the safety and efficiency properties of these schemes.

We consider the following power manageable RTL blocks in the floating-point graphics unit (FGU) [98] of an OpenSPARC core: **main**, **div**, **mul** and **vis**.

1. **main**: This refers to the whole FGU RTL block itself. This block must be active during any instruction requiring an FGU action.

2. **div**: This RTL block must be active during any SPARC V9 floating point or integer divide or square root type instruction.

3. **mul**: This RTL block must be active during any SPARC V9 or VIS 2.0 floating point or integer multiply type instruction.

4. **vis**: This RTL block must be active during any VIS 2.0 instruction executed in the FGX subunit.

The OpenSPARC RTL design uses clock gating to manage the power of these RTL blocks. Since clock gating has negligible overheads [13], safety and efficiency are not statistical properties as in the case of power gating. Therefore, in this work, we do not verify clock gating schemes.

At present, the OpenSPARC RTL does not use power gating for the FGU components. However, we shall later see (Figure 9.1) that these blocks have several long idle periods that are potential opportunities for saving power using power gating. Therefore, in this work, we assume that power gating can be implemented for these blocks. We now describe how these power gating schemes can be verified.

9.3.1 Expressing properties of power gating in OpenSPARC RTL

We consider both timeout-based and adaptive power gating schemes (Section 9.2) for the FGU components. We identify the RTL signals that indicate the idleness of these blocks. We use these signals to formally express the properties corresponding to the safety (Equations 9.1, 9.4) and efficiency (Equations 9.2, 9.5) requirements for these schemes.

For each RTL block of interest, we use the corresponding clock enable signal to define a new RTL signal called *idle*. Due to the presence of clock gating in OpenSPARC RTL, the clock enable signal is set to 0 in the cycle that the block becomes idle. Correspondingly, we set *idle* equal to 1 to indicate the start of the idle period. Similarly, *idle*=0 indicates the end of the current idle period (when the clock enable signal gets set to 1).

We introduce an RTL signal called *idle_cnt* which counts the number of cycles the block has spent in the current idle period (*idle*=1). We also introduce signals called *idle_cnt*1 and *idle_cnt*2 to store the previous two values of *idle_cnt*, i.e., the durations of the previous two idle periods. We use the RTL signals *idle*, *idle_cnt*, *idle_cnt*1 and *idle_cnt*2 and define the safety and efficiency properties of timeout-based and adaptive power gating (Equations 9.1 - 9.5).

Since we consider discrete-time transitions, we express the properties in pCTL [22]. We now list these pCTL expressions and describe the properties that they are used to verify.

- **P1** (Safety of timeout-based scheme):

$$(idle\_cnt = K_{TO}) \Rightarrow P < p_S[true\mathbf{U}_{\leq K_{BE}}(idle = 0)] \tag{9.6}$$

IF a block is idle for K_{TO} cycles THEN it will become active within a further K_{BE} cycles with probability less than p_S.

- **P2** (Efficiency of timeout-based scheme):

$$(idle = 1) \Rightarrow P < (1 - p_E)[true\mathbf{U}_{\leq K_{TO}}(idle = 0)] \tag{9.7}$$

IF a block becomes idle THEN it will become active again within K_{TO} cycles with probability less than $(1 - p_E)$.

- **P3** (Safety of adaptive scheme):

$$(K_{pre} \geq K_{BE})\&(idle = 1) \Rightarrow P < p_S[true\mathbf{U}_{\leq K_{BE}}(idle = 0)] \tag{9.8}$$

IF a block becomes idle AND is predicted to be idle for atleast K_{BE} cycles THEN it will become active within K_{BE} cycles with probability less than p_S.

- **P4** (Efficiency of adaptive scheme):

$$(K_{pre} < K_{BE})\&(idle = 1) \Rightarrow P > p_E[true\mathbf{U}_{\leq K_{BE}}(idle = 0)] \tag{9.9}$$

IF a block becomes idle AND is predicted to be idle for less than K_{BE} cycles THEN it will become active within K_{BE} cycles with probability greater than p_E.

9.3.2 Employing statistical model checking on OpenSPARC RTL

We verify the power gating properties in Equations 9.6 - 9.9 by employing statistical model checking (Section 2.4.1) on the OpenSPARC RTL design. In this work, we use Ymer as the statistical model checking tool.

Statistical model checkers such as Ymer require sample execution paths (Definition 3) that are drawn based on the statistics of the design of interest (Section 2.4.1)). Typically, the descriptions of the statistical models (such as DTMCs) for these designs are provided as input to the statistical model checker. The statistical model checker then uses these descriptions to perform Monte Carlo simulations of the design and generate the required set of sample paths.

In order to generate sample paths, the statistical model checker requires descriptions of DTMC models as input. Although we can describe DTMCs for RTL designs (Section 3.4.1), this is not a feasible proposition for the OpenSparc RTL due to the following reasons.

- Firstly, we cannot compute the exact state transition probabilities which are essential for describing the DTMC models. For RTL DTMCs, these state transition probabilities are based on the statistical distribution of the RTL input values. The power profiling of the OpenSPARC depends on typical use cases of the processor. Hence, in order to obtain the input value distributions, we require application runs that simulate typical workloads on the processor. These input value distributions cannot be computed without actually simulating the entire OpenSPARC RTL (i.e., running an application). Hence, we cannot describe the corresponding DTMC models prior to statistical model checking.

- Secondly, even if we can determine the exact distribution of the RTL input values, the DTMC description for OpenSPARC RTL may be prohibitively large [80].

Since we cannot provide the descriptions of the DTMCs for OpenSPARC RTL, the statistical model checker cannot generate the required sample paths. We circumvent this problem by generating the RTL sample paths externally using a commercial RTL simulator such as VCS. We modify the statistical model checker to use these externally generated RTL sample paths and verify the specified properties on OpenSPARC RTL.

We construct a database of RTL sample paths by simulating the entire OpenSPARC RTL design and recording the values of the relevant RTL signals (such as *idle*) in each clock cycle. We simulate the design using RTL input values corresponding to running an application on OpenSPARC. In effect, we are performing Monte Carlo simulations of the RTL DTMC M based on the statistical distribution of the RTL input values.

The desired verification accuracy is specified by the user in terms of α, β and δ (Equation 2.2). The statistical model checker obtains the required RTL samples paths from the cycle-accurate RTL signal values in the database that we create. For RTL DTMCs, each clock cycle corresponds to a DTMC state transition (Section 3.4.1). Therefore, each RTL sample path corresponds to RTL signal values recorded from a sequence of contiguous clock cycles (Definition 3). Statistical model checking uses these sample paths to verify power gating properties (ϕ, listed in Equations 9.6 - 9.9) in the RTL design while guaranteeing the desired verification accuracy.

Ymer is designed for use with continuous-time systems and properties. We make a minor modification to Ymer in order to apply it to discrete-time systems (DTMCs) and properties (pCTL) of our interest.

In future work, we plan to integrate the statistical model checking algorithm with the RTL simulator. Each sample path can then be model checked in situ and need not be recorded in a database. This can avoid any scalability issues that may arise from having to maintain a database of a large number of samples for several RTL blocks.

9.4 Experimental results

We consider the blocks **main**, **div**, **mul** and **vis** (Section 9.3). We build the cycle-accurate database of RTL samples (Section 9.3.2) by simulating the OpenSPARC RTL using an

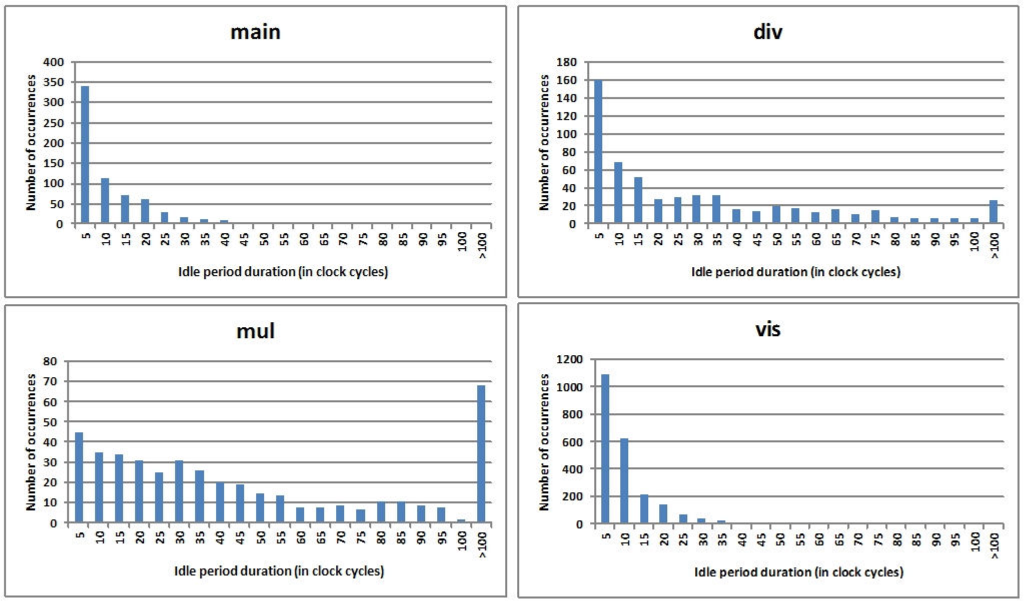


Figure 9.1: Histogram depicting frequency of occurrence of idle periods for **main**, **div**, **mul** and **vis** blocks. Each bin in the histogram corresponds to an interval of 5 idle clock cycles. The x-axis depicts upper bound of these bin intervals. All idle periods exceeding 100 clock cycles are grouped together in the bin denoted ">100". The **div** and **mul** blocks have larger periods of inactivity compared to the **main** and **vis** blocks, and therefore provide more opportunities for saving power.

assembly-language input pattern that is based on running a multi-threaded application on OpenSPARC. We simulate the RTL for 10^5 clock cycles. The verification results that we present here are specific to the statistics of this input pattern. However, our approach can be used with other OpenSPARC applications/benchmarks by generating the corresponding database of RTL samples.

We perform the following experiments:

- We analyze the statistical distribution of the idle periods present in the recorded sample paths.
- We verify whether the power gating schemes satisfy safety/efficiency properties (**P1** to **P4** from Equations 9.6 - 9.9).

Typically, K_{TO} (Section 9.2) depends on the manner in which power gating is implemented [13] and p_S and p_E (Section 9.2) are based on design requirements. Therefore, these values must be specified by the designer. In our experiments, we illustrate our technique by considering several possible values for these parameters.

In our experiments, we focus on demonstrating that statistical model checking adds value to simulation-based analysis since it provides statistical guarantees (Equation 2.2) regarding the verification result. Therefore, we do not report the runtime of the simulation component in our approach, i.e., the time spent in building the database by simulating the RTL. In all our experiments, statistical model checking on the simulation database takes only less than 1s to complete. We perform our experiments on a 3 GHz, 3.25 GB machine (Intel Core2 Duo CPU).

9.4.1 Statistical distribution of idle period durations

In Figure 9.1, we depict the frequencies with which idle periods of different durations occur. We observe that most of the idle periods for the **main** and **vis** blocks are less than 20 cycles long. In the **mul** block, the durations of the idle period are more evenly distributed and there is a significant number of idle periods that exceed 100 clock cycles.

Table 9.1: Values of K_{TO}, computed using property **P2**, such that 50% of the idle periods are less than K_{TO} cycles long. For each block in Column 1, 50% of the idle periods exceed the number of clock cycles in Column 2.

RTL block	Median of idle duration (in clock cycles)
main	5
div	10
mul	20
vis	4

Property **P2** can be used to verify whether an idle period less than K_{TO} clock cycles occurs with probability less than 1-p_E. Therefore, verifying **P2** can be interpreted as verifying the probability distribution of the idle period durations. We wish to determine the value of K_{TO} such that 50% of the idle periods are less than K_{TO} cycles long. In order to do this, we set p_E=0.5 and verify **P2** for different values of K_{TO}. We increment the value of K_{TO} until **P2** is TRUE. Table 9.1 lists these "median" values of K_{TO} for the different blocks.

In Figure 9.2, we show the variation in the number of required sample paths when property **P2** is verified by using different error bounds. The low error tolerance specified by α=β=δ=0.01 requires an order of magnitude more sample paths compared to α=β=δ=0.1. The specification α=β=0.1, δ=0.01 lies in between these two extremes and trades off verification accuracy for the number of required sample paths. We consider α=β=0.1,δ=0.01 for the rest of our experiments.

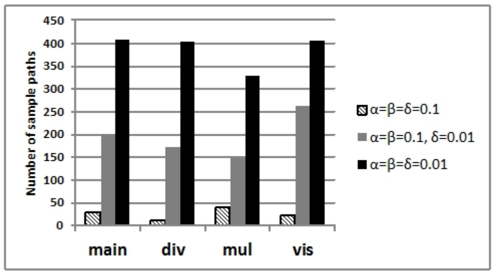


Figure 9.2: Number of sample paths required for verifying property **P2**, as a function of α, β and δ.

We represent all the properties (**P1**, **P2**, **P3** and **P4**) as "bounded-until" pCTL expressions (Equations 9.6 - 9.9). Therefore, the verification procedure is similar for all these properties and in each case, the number of required samples exhibits the same trend as in Figure 9.2.

9.4.2 Safety and efficiency of timeout-based power gating

We consider the breakeven idle period K_{BE} to be 10 cycles. We verify whether the timeout-based power gating scheme meets a safety (Definition 21) requirement of 85%. For this requirement to be satisfied, the safety property **P1** must be TRUE when p_S=0.15. We wish to determine the least value of the timeout threshold K_{TO} such that **P1** is TRUE. We start by setting K_{TO}=5 and increment it in steps of 5 until **P1** is verified to be TRUE. Similarly, for a more stringent safety requirement of 95%, we verify **P1** with p_S=0.05.

In Table 9.2, Column 2 and Column 4 list the least values of the threshold K_{TO} such that the timeout-based scheme meets safety requirements equal to 85% and 95%, respectively. In **main** and **vis**, the majority of the idle periods are less than 25 cycles long. Therefore, the stricter safety requirement of 95% can be met by increasing K_{TO} by only 5 cycles. However, in **div** and **mul** blocks, K_{TO} must be increased significantly to meet the stricter safety requirement.

In Table 9.2, Column 3 and Column 5 list the number of sample paths required for verifying **P1** corresponding to safety requirements of 85% and 95%, respectively. For **vis**, the number of samples required by the statistical model checker to verify safety$\geq$95% is an order of magnitude higher than that required to verify safety$\geq$85%. Typically, the statistical model checker exhibits such behavior when the actual probability of the system is close to the probability threshold specified in the property. In this case, this implies that the probability with which the sample paths of **vis** violate safety is close to p_S=0.05

Table 9.2: Verifying property **P1** to guarantee that timeout-based power gating meets the specified safety requirements. We set α=β=0.1, δ=0.01 for the statistical model checker.

	Safety $\geq$ 85%		Safety $\geq$ 95%	
RTL block	Timeout threshold	Number of sample paths	Timeout threshold	Number of sample paths
main	20	339	25	611
div	15	194	65	571
mul	30	161	60	368
vis	15	254	20	1401

specified in **P1**. Therefore, a large number of samples are required to ensure that the verification is within the desired error bounds (α, β).

The timeout threshold K_{TO} is the number of cycles that are "missed opportunities" with respect to saving power (Section 9.2) . From Figure 9.1, we observe that most of the idle periods are shorter than K_{TO} listed in Table 9.2. In fact, **P2** attests that atleast 50% (Table 9.1) of the idle periods are shorter than the values of K_{TO} for safety$\geq$85% (Column 2, Table 9.2). Therefore, the efficiency (Definition 22) of the the timeout-based scheme is less than 50% for all the blocks.

9.4.3 Safety and efficiency of adaptive power gating

We consider the breakeven idle period K_{BE} to be 10 cycles. We wish to verify whether adaptive power gating meets a specified safety requirement equal to 90%. In order to do this, we set K_{BE}=10 and p_S=0.1 and verify whether property **P4** is satisfied. We find that adaptive power gating does not satisfy the 90% safety requirement for any of the blocks (Table 9.3). However, if we relax the requirement to 80%, we find that **P2** is TRUE for the **mul** and **vis** blocks.

Table 9.3: Verifying property **P3** to guarantee that adaptive power gating meets the specified safety requirements. We set α=β=0.1, δ=0.01 for the statistical model checker.

	Safety $\geq$ 90%		Safety $\geq$ 80%	
RTL block	Verification Result	Number of sample paths	Verification Result	Number of sample paths
main	FALSE	108	FALSE	587
div	FALSE	80	FALSE	212
mul	FALSE	195	TRUE	274
vis	FALSE	106	TRUE	1334

Similarly, we verify property **P4** to guarantee that adaptive power gating meets the specified efficiency requirements. We first check whether Efficiency$\geq$90% by verifying **P4** with p_E=0.9 and K_{BE}=10. From Table 9.4, we see that this requirement is met only for the **mul** block. The **div** and **vis** blocks meet a more relaxed efficiency requirement equal to 80% (p_E=0.8). In the **main** block, property **P4** is TRUE only when the requirement is relaxed to 70% (p_E=0.7).

Table 9.4: Verifying property **P4** to guarantee that adaptive power gating meets the specified efficiency requirements. We set α=β=0.1, δ=0.01 for the statistical model checker.

	Efficiency $\geq$ 90%		Efficiency $\geq$ 80%	
RTL block	Verification Result	Number of sample paths	Verification Result	Number of sample paths
main	FALSE	89	FALSE	161
div	FALSE	502	TRUE	173
mul	TRUE	220	TRUE	n/a
vis	FALSE	189	TRUE	444

9.5 Chapter summary

In this chapter, we apply statistical model checking to validate complex DPM schemes that present known design challenges. We formally specify safety and efficiency requirements of power gating schemes as properties in pCTL [22]. We extend Ymer and create a framework for performing statistical model checking on RTL designs. Although we consider only properties of DPM schemes in this work, this framework can potentially be used to provide guarantees regarding any statistical metric in RTL designs.

CHAPTER 10

ACCELERATING STATISTICAL MODEL CHECKING IN RARE-EVENT SCENARIOS

10.1 Introduction

Simulation-based techniques are known to be inefficient and time-consuming in *rare-event scenarios*, i.e., scenarios that pertain to events which occur with very low probability ($<10^{-4}$). In such scenarios, a very large number of samples need to be generated in order to gather the statistical evidence required by the statistical model checking engine. For example, while verifying whether failure rate $<10^{-5}$, an average of 10^5 samples need to be generated before even a single failure is witnessed.

Low failure rate requirements are typical in the design of reliable SRAM cells [50]. Therefore, conventional statistical model checking (Section 2.4.1) is inefficient for verifying the reliability of an SRAM cell. In this chapter, we analyze a technique to accelerate statistical model checking in rare-event scenarios for an SRAM cell.

10.1.1 Accelerating statistical model checking of an SRAM cell

Statistical model checking for rare-event scenarios can be accelerated by increasing the frequency with which failures (i.e., the rare events in this work) are generated. This can be achieved by carefully modifying the statistical distribution of the design to sample the failure region more frequently. In probability estimation, this technique is referred to as *importance sampling* [47],[48],[49].

In this chapter, we demonstrate an *importance sampling approach* for accelerating statistical model checking in rare-event scenarios for SRAM cells. Let M be an SRAM cell design with random variables V that model the effect of process variations on the transistors. These variations are modeled as a Gaussian distribution $D(V)$ defined over the space spanned by V. We wish to verify that M satisfies a reliability property Φ, where Φ deals with a rare-event scenario. We modify the distribution $D(V)$ and construct a new distribution $D'(V)$ that samples the failure region more frequently. We employ statistical model checking by drawing samples according to the modified distribution $D'(V)$ instead of $D(V)$. In order to preserve the correctness of the verification result, we adjust for the statistical bias that we introduce in each sample.

In rare-event scenarios, we find that our importance sampling approach reduces the number of samples required for statistical model checking to arrive at the verification result with the desired level of accuracy (Section 2.4.1). Therefore, our approach provides significant performance benefits over regular statistical model checking.

In [51], the authors employ an importance sampling strategy for SRAM cells. The authors use empirical evidence to validate the speedup provided by their approach. In this chapter, we derive analytical upper bounds for estimation variance and present a more rigorous analysis for the speedup provided by our approach over regular statistical model checking.

10.1.2 Chapter organization

The rest of this chapter is organized as follows. In Section 10.2, we present preliminaries regarding the SRAM cell design and the statistical model checking algorithm that we use in this work. In Section 10.3, we present the broad intuition behind our importance sampling approach to accelerate statistical model checking. In Section 10.4, we describe all the steps of our approach in detail. In Section 10.5, we provide an analysis regarding the correctness and speedup of our approach. In Section 10.6, we present empirical evidence that demonstrates the correctness and speedup approach while verifying the reliability of an SRAM cell.

10.2 Verifying reliability of an SRAM cell

We now present some preliminaries regarding the source of variations in an SRAM cell. We describe the reliability property that we wish to verify on an SRAM cell in the presence of these variations. We also provide a brief background for the statistical model checking technique that we employ on the SRAM cell.

10.2.1 Variations in an SRAM cell

We consider an SRAM cell design that comprises six transistors [50]. Due to variations arising from the manufacturing process, the threshold voltages of these six transistors are typically modeled as independent, Gaussian random variables [50]. Therefore, the SRAM cell can be viewed as a statistical entity with six random variables.

Let $V=\{v_1, v_2, v_3, v_4, v_5, v_6\}$ be the set of six random variables that model the varying threshold voltages of the transistors in the SRAM cell. Each variable v_j is real-valued and can be assigned a value in the range $[v_j^{min}\ v_j^{max}]$.

Definition 23. The *sample space* $\mathcal{S}$ of the SRAM cell is the 6-dimensional Euclidean space $[v_1^{min}\ v_1^{max}] \times [v_2^{min}\ v_2^{max}] \times [v_3^{min}\ v_3^{max}] \times [v_4^{min}\ v_4^{max}] \times [v_5^{min}\ v_5^{max}] \times [v_6^{min}\ v_6^{max}]$ spanned by the set of threshold voltages V.

Each point in the sample space $\mathcal{S}$ corresponds to a unique assignment of concrete, real values to variables V. We use the 6-tuple $\{v_1, v_2, v_3, v_4, v_5, v_6\}$ to denote a point in the sample space of the SRAM cell.

Let $g_j(v)$ denote the Gaussian probability density function (pdf) for variable v_j. The mean and variance of the distribution $g_j(v)$ can be obtained from the specification of the transistor in the process technology library [50].

Definition 24. The statistical distribution $D(V)$ of the SRAM cell is given by the joint pdf of the threshold voltages V. Since the threshold voltages V are modeled to be statistically independent variables, the joint pdf of V can be computed as a product of the individual pdfs $g_j(v)$ of the variables v_j (j= 1 to 6).

Let $V^i=\{v_1^i, v_2^i, v_3^i, v_4^i, v_5^i, v_6^i\}$ be a sample that is drawn from the space $\mathcal{S}$ (Definition 23) of the SRAM cell. During Monte Carlo simulation of the SRAM cell according to the distribution $D(V)$, the probability density $D(V^i)$ of the sample V^i is given by

$$D(V^i) = \prod_{j=1}^{6} g_j(v_j^i) \tag{10.1}$$

10.2.2 Reliability of an SRAM cell

The delay of the SRAM cell depends on the threshold voltages of the transistors. An SRAM cell is said to *fail* if its delay exceeds a pre-defined timing constraint. We wish to verify that the probability with which an SRAM cell fails is less than a threshold θ. We express this reliability requirement as a property

$$\Phi = P_{\leq\theta}[fail] \tag{10.2}$$

where *fail* refers to the delay exceeding a timing constraint.

If θ is very small, we consider that Φ deals with a *rare-event scenario*. In reliable hardware designs, the failure rate $P[fail]$ of an SRAM cell is required to be very low. Therefore, the reliability property of an SRAM cell deals with a rare-event scenario.

Definition 25. A *failing sample* of an SRAM cell is a sample where the cell delay exceeds a user-specified timing constraint.

For each sample V^i drawn from the space $\mathcal{S}$, the delay of the SRAM cell can be measured by simulating the SRAM circuit using the corresponding values v_j^i assigned to the threshold voltages v_j (j= 1 to 6). The measured delay can be compared against the timing constraint in order to check whether the sample is failing or not.

Definition 26. The set of all failing samples in $\mathcal{S}$ constitute the *failure region* $\mathcal{S}_F$ ($\mathcal{S}_F \subseteq \mathcal{S}$) of the SRAM cell.

For a given timing constraint, the SRAM cell fails when the threshold voltages exceed a certain value. Let $v_j^F \in [v_j^{min}\ v_j^{max}]$ be the smallest value of v_j for which an SRAM cell fails for a given timing constraint. Therefore, in all failing samples of the SRAM cell, the value of v_j lies in the range $[v_j^F\ v_j^{max}]$.

Definition 27. For a given timing constraint, $\mathcal{R}_F$ ($\mathcal{R}_F \subseteq \mathcal{S}$) is the smallest 6-dimensional hyper-rectangle in which the failure region $\mathcal{S}_F$ of the SRAM cell is completely contained. $\mathcal{R}_F$ can be viewed as a "box" that bounds the failure region $\mathcal{S}_F$ and is given by $[v_1^F\ v_1^{max}] \times [v_2^F\ v_2^{max}] \times [v_3^F\ v_3^{max}] \times [v_4^F\ v_4^{max}] \times [v_5^F\ v_5^{max}] \times [v_6^F\ v_6^{max}]$.

Definition 28. For a given timing constraint, $\mathcal{C}_F$ ($\mathcal{C}_F \subseteq \mathcal{S}$) is the smallest 6-dimensional hyper-cube in which the failure region $\mathcal{S}_F$ of the SRAM cell is completely contained. $\mathcal{C}_F$ is a box with equal-sized edges and is given by $[v_1^{max}\text{-}c\ v_1^{max}] \times [v_2^F\text{-}c\ v_2^{max}] \times [v_3^F\text{-}c\ v_3^{max}] \times [v_4^{max}\text{-}c\ v_4^{max}] \times [v_5^{max}\text{-}c\ v_5^{max}] \times [v_6^{max}\text{-}c\ v_6^{max}]$, where c is the size of each edge. The hyper-rectangle $\mathcal{R}_F$ (Definition 27) is contained in the hyper-cube $\mathcal{C}_F$, i.e., $\mathcal{R}_F \subseteq \mathcal{C}_F$.

10.2.3 Statistical model checking of an SRAM cell

We wish to verify that an SRAM cell design M verifies a reliability property Φ (Equation 10.2), denoted by $M \models \Phi$. We briefly describe the statistical model checking technique that we employ in order to verify $M \models \Phi$.

Let p_F denote the actual failure rate of M. If p_F is less than the threshold θ specified in Φ, then $M \models \Phi$. Statistical model checking obtains an estimate of the failure rate by performing Monte Carlo simulations of M. $M \models \Phi$ is verified by comparing this estimated failure rate against θ.

Let V^i denote the i^{th} sample drawn according to the statistical distribution $D(V)$ (Definition 24) of M. We define $\mathbb{I}(V^i)$ to be an *indicator function* [21] that is equal to 1 if the sample V^i is failing (Definition 25) and 0 otherwise.

$$\mathbb{I}(V^i) = \begin{cases} 1, & \text{if } V^i \text{ is a failing sample} \\ 0, & \text{if } V^i \text{ is not failing} \end{cases} \tag{10.3}$$

After N_S samples have been generated, the expected (average) failure rate can be estimated as

$$\widehat{p_F} = \frac{1}{N_S}\sum_{i=1}^{N_S}\mathbb{I}(V^i) \tag{10.4}$$

However, in a different sampling run, another set of N_S samples could be used instead to estimate the failure rate. Therefore, the estimate is itself a random variable. For large N_S, the estimate is typically modeled as a Gaussian random variable (Figure 10.1) with mean $\widehat{p_F}$. The variance $\sigma^2_{p_F}$ of the estimate is given by

$$\sigma^2_{p_F} = \frac{\sum_{i=1}^{N_S}[\mathbb{I}(V^i) - \widehat{p_F}]^2}{N_S(N_S-1)} \tag{10.5}$$

The Gaussian distribution represents how well $\widehat{p_F}$ estimates the actual failure rate p_F. p_F is more likely to be near the mean $\widehat{p_F}$ of the distribution and less likely to be in the tail regions.

Statistical model checking verifies $M \models \Phi$ by comparing $\widehat{p_F}$ against the threshold θ. Since $\widehat{p_F}$ is only an estimate obtained using a limited set of simulations, the verification result may be inaccurate. Statistical model checking draws sufficient samples until the verification results are within the specified bounds of error α and β (Equation 2.2).

Figure 10.1 depicts the scenario where $\widehat{p_F} < \theta$. In this scenario, the verification result is incorrect if the actual failure rate p_F is greater than θ. Therefore, the probability of error is equal to the area of the shaded region in the figure. We require this probability to be less than the bound α (Equation 2.2). Similarly, if $\widehat{p_F} > \theta$, we require the probability of error to be less than β.

Verification errors arise when the actual failure rate p_F and the estimate $\widehat{p_F}$ lie on different sides of the threshold θ. As the number of samples N_S increases, the variance of the estimate (Equation 10.5) reduces and the Gaussian curve becomes "narrower". As a

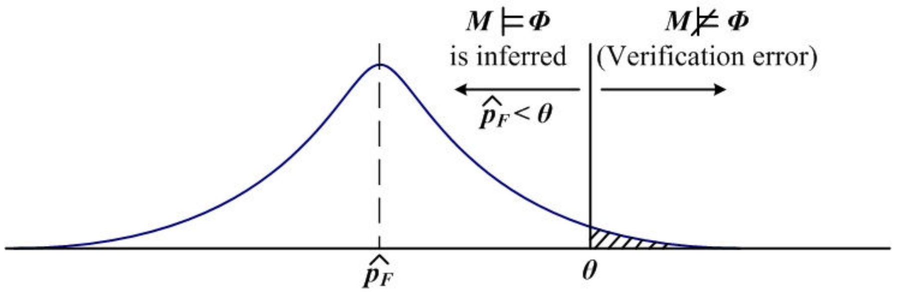


Figure 10.1: Gaussian distribution of failure rate estimates, with mean $\widehat{p_F}$ and variance $\sigma^2_{p_F}$. The area of the shaded region is the probability of error in the verification result.

result, the probability of occurrence of a verification error also reduces.

10.3 Speeding up statistical model checking for rare-event scenarios of an SRAM cell

In this section, we define and establish the criteria for accelerating statistical model checking in rare-event scenarios of an SRAM cell. We consider M to be the SRAM cell model with the Gaussian distribution $D(V)$ (Definition 24). Let Φ (Equation 10.2) denote the reliability property of our interest.

We wish to verify whether $M \models \Phi$. The conventional statistical model checking engine (Section 10.2.3) infers whether $M \models \Phi$ by performing Monte Carlo simulations according to the Gaussian distribution $D(V)$ (Definition 24) on the space $\mathcal{S}$ (Definition 23). In rare-event scenarios, the failure region $\mathcal{S}_F$ lies in the tail region of the Gaussian distribution $D(V)$. Therefore, the failing samples (Definition 25) are generated with very low probability (Equation 10.1). As a result, a very large number of samples need to be generated in order to estimate the failure rate with high confidence (i.e., low estimation variance). This makes statistical model checking extremely time-consuming for rare-event scenarios.

In this chapter, we analyze an importance sampling [47] approach to reduce the number of samples that need to be generated by statistical model checking in rare-event scenarios for SRAM cells. We achieve this by modifying the distribution over the space $\mathcal{S}$ to generate more failing samples and reduce the variance of the failure rate estimate. Let $D'(V)$ be the modified joint distribution of the variables V. We replace the statistical distribution $D(V)$ of M (Definition 24) with the distribution $D'(V)$. We verify $M \models \Phi$ by performing Monte Carlo simulations according to the distribution $D'(V)$ instead of $D(V)$. We show that a low-variance failure rate estimate can now be obtained by generating fewer samples thereby accelerating statistical model checking. This forms the basis of our importance sampling approach.

Our modified distribution $D'(V)$ introduces a statistical "bias" towards the failure region $\mathcal{S}_F$ (Definition 26). Therefore, in effect, we artificially increase the probability densities of failing samples. In order to account for the increased failure rate, we will need to adjust for the statistical bias while verifying $M \models \Phi$ using samples drawn from $D'(V)$. Since we know the closed-form analytical expressions for both $D(V)$ and $D(V')$, we can compute the exact extent to which each generated sample is biased. While verifying whether $M \models \Phi$, we exactly adjust for the statistical bias that we introduce in $D'(V)$. Therefore, verifying $M \models \Phi$ using samples based on the distribution $D'(V)$ is now provably equivalent to regular statistical model checking (Section 10.2.3) that uses samples based on $D(V)$.

The crux of importance sampling lies in the choice of the modified distribution $D'(V)$. We wish to choose $D'(V)$ such that, for a fixed number of samples, the failure rate estimate that we obtain using $D'(V)$ has a lower variance compared to that of the estimate obtained using $D(V)$. In this chapter, we choose $D'(V)$ to be a uniform distribution over the space $\mathcal{S}$. We show that our choice, although simple, results in reduction of the estimation variance which in turn accelerates statistical model checking.

10.4 Our importance sampling approach for statistical model checking of an SRAM cell

We wish to check whether an SRAM cell M satisfies a reliability property Φ (Equation 10.2). We focus on the case where Φ deals with a rare-event scenario (Section 10.2.2). We now describe all the steps in our importance sampling approach (Figure 10.2).

10.4.1 Modifying the distribution on the sample space

In order to increase the frequency of failing samples during Monte Carlo simulation, the failure region $\mathcal{S}_F$ (Definition 26) needs to be sampled more frequently. We achieve this by modifying the distribution $D(V)$ of the SRAM cell (Definition 24) and creating a new distribution $D'(V)$ over the sample space $\mathcal{S}$.

We create a modified distribution $D'(V)$ that uniformly samples the space $\mathcal{S}$. We achieve this by treating the set of threshold voltages V (Section 10.2.1) as a set of independent, uniformly distributed random variables. Let $u_j(v)$ denote the uniform pdf of

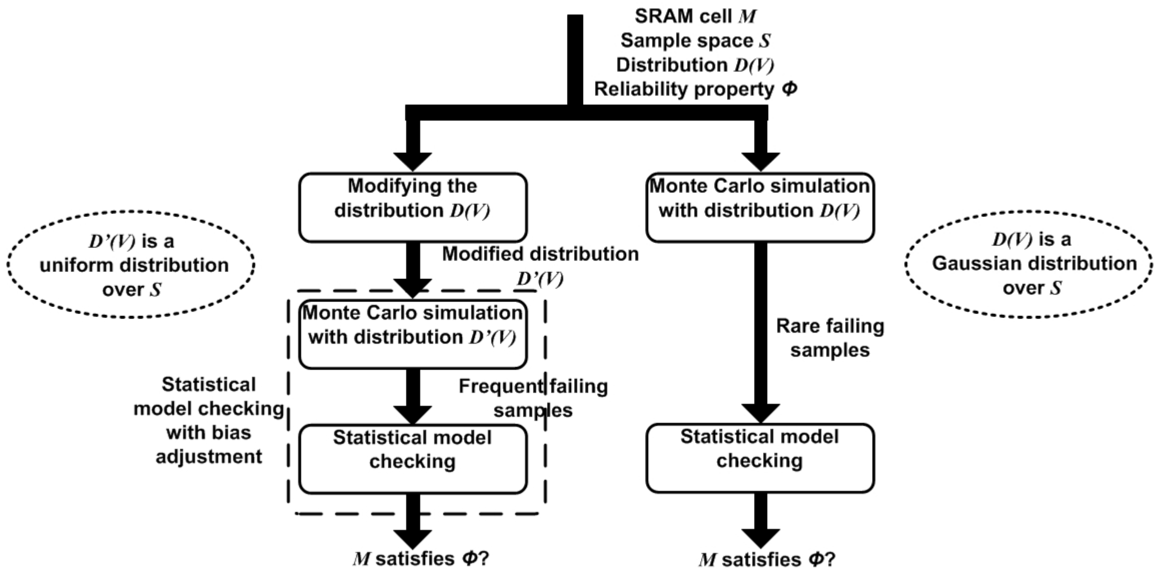

SRAM cell M
Sample space S
Distribution $D(V)$
Reliability property Φ
Modifying the distribution $D(V)$
Monte Carlo simulation with distribution $D(V)$
$D'(V)$ is a uniform distribution over S
Modified distribution $D'(V)$
$D(V)$ is a Gaussian distribution over S
Monte Carlo simulation with distribution $D'(V)$
Rare failing samples
Statistical model checking with bias adjustment
Frequent failing samples
Statistical model checking
Statistical model checking
M satisfies Φ?
M satisfies Φ?

Figure 10.2: Block diagram comparing the steps in our importance sampling approach (left branch) with those in regular statistical model checking (right branch).

the threshold voltage v_j in the range $[v_j^{min}\ v_j^{max}]$. The pdf $u_j(v)$ is given as

$$u_j(v) = \frac{1}{(v_j^{max} - v_j^{min})} \tag{10.6}$$

The modified distribution $D'(V)$ is the joint distribution of the variables v_j with the uniform pdfs $u_j(v)$.

In Equation 10.1, we describe the probability density of a sample V^i=$\{v_1^i, v_2^i, v_3^i, v_4^i, v_5^i, v_6^i\}$ according to the distribution $D(V)$. If samples are drawn according to the distribution $D'(V)$ instead of $D(V)$, the probability density of the sample V^i is given by

$$\begin{aligned} D'(V^i) &= \prod_{j=1}^{6} u_j(v_j^i) \\ &= \prod_{j=1}^{6} \frac{1}{(v_j^{max} - v_j^{min})} \end{aligned} \tag{10.7}$$

We modify M by substituting the original Gaussian distribution $D(V)$ with the uniform distribution $D'(V)$. If Monte Carlo simulation is performed using the distribution $D'(V)$, a larger number of failing samples are generated in comparison to using the original distribution $D(V)$ (Figure 10.2).

10.4.2 Statistical model checking with bias adjustment

We now perform Monte Carlo simulations and uniformly sample the space $\mathcal{S}$ of the SRAM cell M based on its modified distribution $D'(V)$ (Equation 10.7). We use the generated samples to perform statistical model checking and verify whether $M \models \Phi$. In order to maintain the accuracy of the statistical model checking results, we adjust for the statistical bias in each sample.

Let $L(V^i)$ denote the extent to which a sample V^i is biased. We define $L(V^i)$ as

$$\begin{aligned} L(V^i) &= \frac{D(V^i)}{D'(V^i)} \\ &= \prod_{j=1}^{6} \frac{g_j(v_j^i)}{u_j(v_j^i)} \end{aligned} \tag{10.8}$$

where $D(V^i)$ is the probability density of V^i according to the original distribution $D(V)$ (Equation 10.1). Similarly, $D'(V^i)$ is the probability density of V^i based on the modified distribution $D'(V)$ (Equation 10.7). In importance sampling, $L(V^i)$ is typically referred to as the *inverse likelihood ratio* [47].

After N'_S biased samples have been generated, the estimate $\widehat{p_F}'$ of the actual failure rate can be computed [47] as

$$\widehat{p_F}' = \frac{\sum_{i=1}^{N'_S} L(V^i)\mathbb{I}(V^i)}{N'_S} \tag{10.9}$$

where $\mathbb{I}(V^i)$ is the indicator function described in Equation 10.3. The use of $L(V^i)$ adjusts the estimate for the statistical bias in each sample. The variance ${\sigma'}^2_{p_F}$ of the estimate, in the presence of importance sampling, is given by

$${\sigma'}^2_{p_F} = \frac{\sum_{i=1}^{N'_S} [L(V^i)\mathbb{I}(V^i) - \widehat{p_F}']^2}{N'_S(N'_S - 1)} \tag{10.10}$$

As described in Section 10.2.3, the statistical model checking engine draws samples based on the modified distribution $D'(V)$ until the error bounds specified by α and β (Equation 2.2) are satisfied. We find that the number of samples N'_S required to meet the error bounds using our approach is much smaller than the number of samples N_S required while using regular statistical model checking (Section 10.2.3).

10.5 Analysis of our importance sampling approach

We now briefly outline the correctness and speedup of our approach. We also derive analytical upper bounds for the variance of our failure rate estimates.

10.5.1 Correctness and speedup

In order to verify whether an SRAM cell M satisfies a reliability property Φ, statistical model checking estimates the probability of failure p_F and checks whether it is less than the specified probability threshold θ (Section 10.2.3). In our approach, we estimate the probability of failure by drawing samples according to the uniform distribution $D'(V)$ instead of the original Gaussian distribution $D(V)$. If the estimation accuracy is preserved in the presence of importance sampling, the verification result would remain the same.

For probability estimation, the use of Equation 10.9 has been proven to be correct [47] when modifying a distribution to frequently generate rare events. Therefore, the verification result that we obtain with our importance sampling approach is consistent with what we obtain with regular statistical model checking.

A metric for speedup in statistical model checking is the reduction in the number of samples required to provide the verification result. Statistical model checking terminates when the estimation error is low enough to meet error bounds specified by α and β (Section 10.2.3). If the error bounds can be met with fewer samples, statistical model checking can be sped up.

From Figure 10.1, we observe that the error in verification can be decreased by reducing the variance of the failure rate estimate (Section 10.2.3). In other words, with variance reduction, a given error bound can be met with fewer samples. Importance sampling, with a careful choice of the modified distribution, is known to reduce estimation variance. As a result, importance sampling can result in a speedup for statistical model checking. As the failure rate p_F becomes smaller, the extent of such speedup typically becomes larger.

In Section 10.5.2, we derive analytical upper bounds for variance and show that our importance sampling approach guarantees a reduction in variance over regular statistical model checking. In Section 10.6, we also present empirical evidence for the variance reduction provided by our approach in comparison to regular statistical model checking.

10.5.2 Analytical bounds for variance of the estimates

In [47], the authors present analytical equations for computing the asymptotic variance of the failure rate estimates that are obtained with or without importance sampling. We now use these equations to derive upper bounds for the estimation variance.

In regular statistical model checking (Section 10.2.3), without importance sampling, the asymptotic variance AVar of the failure rate estimate (Equation 10.4) is given as

$$\mathrm{AVar} = \int\limits_{\mathcal{S}} [\mathbb{I}(V)]^2 D(V) dV - {p_F}^2 \tag{10.11}$$

where the integration is performed over the sample space $\mathcal{S}$ of the SRAM cell (Definition 23). In other words, $\mathcal{S}$ is the domain of the 6-D integration variable V.

Since $\mathbb{I}(V)$ is equal to 1 inside the failure region $\mathcal{S}_F$ and 0 elsewhere (Equation 10.3), the above equation for AVar can be written as

$$\mathrm{AVar} = \int\limits_{\mathcal{S}_F} D(V) dV - {p_F}^2 \tag{10.12}$$

In an SRAM cell, we know that $\mathcal{S}_F$ is contained in the 6-D hyper-rectangle $\mathcal{R}_F$ (Definition 27). Therefore, in geometric terms, the volume of $\mathcal{R}_F$ is greater than or equal to the volume of $\mathcal{S}_F$. Since the integrand in Equation 10.12 is strictly positive everywhere in $\mathcal{S}$, AVar can be bounded as

$$\begin{aligned} \mathrm{AVar} &\leq \int\limits_{\mathcal{R}_F} D(V) dV - {p_F}^2 \\ &\leq \int\limits_{\mathcal{R}_F} D(V) dV \end{aligned} \tag{10.13}$$

The tightness of the upper bound described in Equation 10.13 depend on the tightness with which $\mathcal{R}_F$ bounds the failure region $\mathcal{S}_F$. We approximate the upper bound for AVar as

$$\mathrm{AVar}_{\mathrm{UB}} = \frac{\mathrm{Volume}(\mathcal{S}_F)}{\mathrm{Volume}(\mathcal{R}_F)} \times \int\limits_{\mathcal{R}_F} D(V) dV \tag{10.14}$$

where $\frac{\mathrm{Volume}(\mathcal{S}_F)}{\mathrm{Volume}(\mathcal{R}_F)}$ denotes the fraction of $\mathcal{R}_F$ occupied by $\mathcal{S}_F$.

In terms of the individual variables v_j, we express the upper bound AVar_{UB} as

$$\text{AVar}_{\text{UB}} = \frac{\text{Volume}(\mathcal{S}_F)}{\text{Volume}(\mathcal{R}_F)} \times \prod_{j=1}^{6} \int\limits_{v_j^F}^{v_j^{max}} g_j(v)dv \tag{10.15}$$

In the presence of importance sampling, the asymptotic variance AVar' of the failure rate estimate is given as

$$\text{AVar'} = \int\limits_{\mathcal{S}} [L(V)\mathbb{I}(V)]^2 D'(V)dV - {p_F}^2 \tag{10.16}$$

where $L(V){=}\frac{D(V)}{D'(V)}$ as described in Equation 10.8.

We employ the same line of reasoning described above and derive the upper bound of AVar' as

$$\begin{aligned}\text{AVar'} &= \int\limits_{\mathcal{S}_F} L(V)^2 D'(V)dV - {p_F}^2 \\ &= \int\limits_{\mathcal{S}_F} \frac{D(V)^2}{D'(V)}dV - {p_F}^2 \\ &\leq \int\limits_{\mathcal{R}_F} \frac{D(V)^2}{D'(V)}dV \end{aligned} \tag{10.17}$$

As in Equation 10.15, we approximate the upper bound AVar'_{UB} as

$$\begin{aligned}\text{AVar'}_{\text{UB}} &= \frac{\text{Volume}(\mathcal{S}_F)}{\text{Volume}(\mathcal{R}_F)} \times \int\limits_{\mathcal{R}_F} \frac{D(V)^2}{D'(V)}dV \\ &= \frac{\text{Volume}(\mathcal{S}_F)}{\text{Volume}(\mathcal{R}_F)} \times \prod_{j=1}^{6} \int\limits_{v_j^F}^{v_j^{max}} \frac{(g_j(v))^2}{u_j(v)}dv \end{aligned} \tag{10.18}$$

If AVar'_{UB} (Equation 10.18) is less than AVar_{UB} (Equation 10.15), our importance sampling approach can provide a speedup over regular statistical model checking. We define the variance reduction factor as

$$\frac{\text{AVar}_{\text{UB}}}{\text{AVar'}_{\text{UB}}} \quad = \quad \prod_{j=1}^{6} \frac{\int\limits_{v_j^F}^{v_j^{max}} g_j(v)dv}{\int\limits_{v_j^F}^{v_j^{max}} \frac{(g_j(v))^2}{u_j(v)}dv} \tag{10.19}$$

which, due to the approximation that we make for the upper bounds, is independent of the fraction $\frac{\text{Volume}(\mathcal{S}_F)}{\text{Volume}(\mathcal{R}_F)}$.

In Figure 10.3, we plot the variance reduction factor as a function of the size of the hyper-rectangle that bounds the failure region $\mathcal{S}_F$. For ease of illustration, we consider the hyper-cube $\mathcal{C}_F$ (Definition 28) instead of the hyper-rectangle $\mathcal{R}_F$ since the size of $\mathcal{C}_F$ can be controlled by varying a single parameter c (the size of each edge of $\mathcal{C}_F$). A small hyper-cube implies a small failure region $\mathcal{S}_F$ which in turn implies a small failure rate p_F. We observe that the variance reduction factor increases with decrease in size of the hyper-cube. Therefore, the speedup provided by our importance sampling approach increases with decrease in p_F.

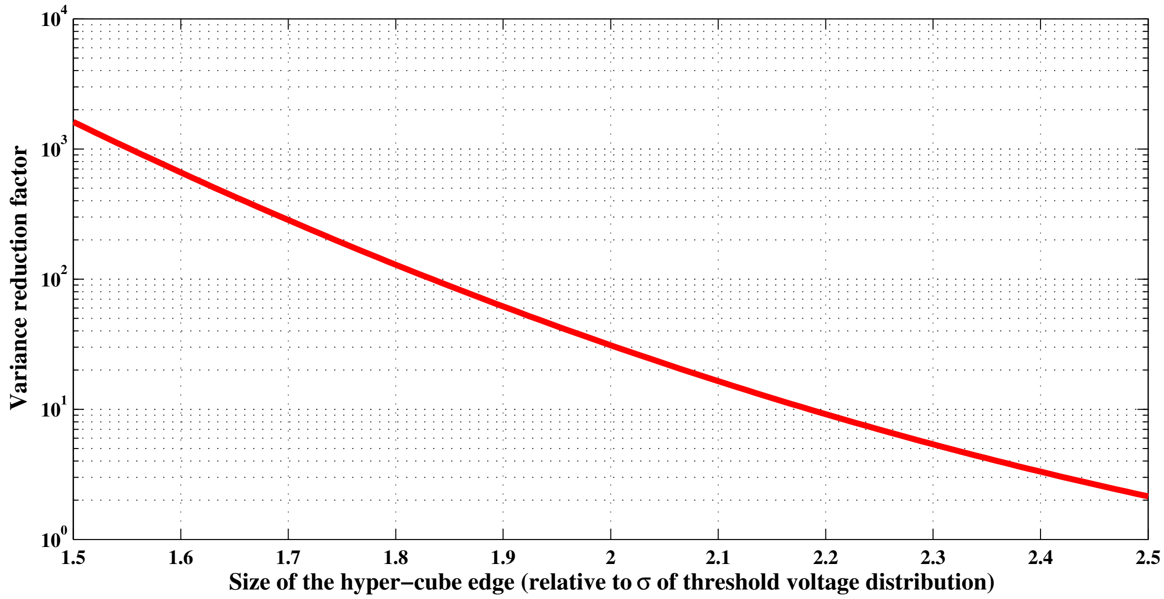


Figure 10.3: A plot depicting the variance reduction achieved by importance sampling as a function of the size of the hyper-cube $\mathcal{C}_F$ that bounds the failure region of the SRAM cell. On the x-axis, the size of the hyper-cube edge is expressed in terms of the standard deviation σ of the Gaussian distribution of the threshold voltage values.

10.6 Experimental results

In Table 10.1, we provide empirical evidence for the variance reduction provided by our importance sampling approach (Column 5) over regular statistical model checking. We choose three different timing constraints for the SRAM cell. For each timing constraint, statistical model checking estimates the corresponding failure rate (Column 1) of the SRAM cell. For a fixed number of samples (=10^5), the variance in the estimate computed using our approach (Column 4) is significantly less than that when importance sampling is not employed (Column 3).

We consider the failing samples that we generate using importance sampling and approximately determine the boundary of the 6-D hyper-rectangle $\mathcal{R}_F$ (Definition 27) in each of the six dimensions. We estimate the boundary v_j^F in the j^{th} dimension ($j = 1$ to 6) of the hyper-rectangle by computing the minimum value of the j^{th} variable v_j among all the failing samples that we generate. In Table 10.2, we list the variance reduction factors that we compute using Equation 10.19. Although the boundaries that we estimate only coarsely approximate the actual hyper-rectangle $\mathcal{R}_F$, the reduction factors that we compute in both Table 10.1 and Table 10.2 are comparable in magnitude.

In Table 10.3, we provide evidence for the correctness and speedup provided by our approach. For the three different failure rates that we consider in Table 10.1, we verify Φ (Equation 10.2) for low values of θ. The variance reduction provided by our approach results in a significant reduction in the number of samples required by the statistical model checking engine to arrive at a result within the specified error bounds (Section 10.2.3). Therefore, our approach provides considerable speedup (Column 8) over regular statistical model checking. Moreover, our verification results (Column 7) are consistent with those obtained using regular statistical model checking (Column 4).

Table 10.1: Demonstrating variance reduction using our importance sampling approach on an SRAM cell. We use 10^5 samples for both approaches.

Regular SMC		Our approach		
Failure rate estimate $\widehat{p_F}$	Variance $\sigma^2_{p_F}$	Failure rate estimate $\widehat{p_F}'$	Variance ${\sigma'}^2_{p_F}$	Variance reduction $\frac{\sigma^2_{p_F}}{{\sigma'}^2_{p_F}}$
2.20x10^{-3}	2.20x10^{-8}	2.29x10^{-3}	1.21x10^{-8}	1.81x
5.22x10^{-4}	5.22x10^{-9}	5.14x10^{-4}	1.51x10^{-9}	3.46x
8.76x10^{-5}	8.76x10^{-10}	6.57x10^{-5}	9.61x10^{-11}	9.12x

Table 10.2: Demonstrating that our importance sampling approach provides a reduction in the analytical upper bound of the estimation variance.

Failure rate estimate	Variance reduction factor $\frac{\text{AVar}_{\text{UB}}}{\text{AVar'}_{\text{UB}}}$
$2.20\text{x}10^{-3}$	2.14x
$5.22\text{x}10^{-4}$	5.38x
$8.76\text{x}10^{-5}$	16.45x

Table 10.3: Demonstrating speedup of our importance sampling approach on an SRAM cell. We set the error bounds α=β=0.01.

		Regular SMC		Our approach		
Failure rate estimate	Threshold θ	Number of samples	Result	Number of samples	Result	Speedup
$2.20\text{x}10^{-3}$	$2.5\text{x}10^{-3}$	44254	TRUE	11928	TRUE	3.71x
$5.22\text{x}10^{-4}$	$5\text{x}10^{-4}$	181720	FALSE	26881	FALSE	6.76x
$8.76\text{x}10^{-5}$	10^{-4}	322556	TRUE	31044	TRUE	10.39x

10.7 Chapter summary

In this chapter, we analyzed an importance sampling based approach for accelerating statistical model checking in rare-event scenarios for SRAM cells. We achieve this modifying the distribution of the design such that the failure region is sampled more frequently. We demonstrate that our approach is sound and provides significant speedup while verifying the reliability of SRAM cells.

CHAPTER 11

RELATED WORK

We briefly describe our contributions in this thesis in the context of related work.

11.1 Formal probabilistic analysis of hardware

For hardware designs, Markov chains have frequently been used to compute high level system performance and power [60]. These models do not represent details of the hardware implementation that are required to compute bit error-related performance. Markov chains have also been used at a circuit level, to design circuits with high error tolerance [100] and to analyze stability [101]. However, these models provide an excess of detail. Therefore, they restrict the size of the systems that can be analyzed.

In [102], the authors use the probabilistic model checking tool PRISM to evaluate the reliability of defect-tolerant systems. However, the evaluation is restricted to gate-level descriptions of the systems. The defects in gate functionality are considered to be stochastic in nature. The authors illustrate their technique using a NAND multiplexing example. The state-space of the DTMCs that are used to represent the gate-level descriptions depends on the number of gates in the system. RTL designs map to gate-level descriptions that may have hundreds of thousands of gates. Therefore, it is infeasible to use such gate-level analysis techniques to evaluate reliability of RTL designs.

The authors in [103] obtain analytical expressions for the errors introduced in RTL due to internal quantization of data. However, this approach is intractable for complex MIMO designs. Moreover, the analytical expressions do not model the probabilistic nature of errors that are caused by external data corruption.

The Mobius tool [104] provides a flexible formalism that can be used to model and formally analyze probabilistic systems. However, we find that several extraneous variables need to be introduced into the model to represent correct RTL functionality. Therefore, the scalability afforded by this tool is limited.

11.2 Macromodeling

Macromodels [18],[19],[105] propagate information from the lower levels of hardware to the higher levels of design. High-level analyses use macromodels as plugins to provide performance (e.g., timing and power) estimates early in the design flow. Typically, macromodels provide estimates in RTL that are within 20% of the actual measurements obtained at the gate level.

Statistical timing estimates in RTL can be obtained with commercial CAD tools that use delay macromodels [18],[19]. However, such variation-aware RTL timing analysis tools almost exclusively consider only process variations and cannot be used in the context of input variations. In this work, we consider RTL delay macromodels for both input variations and for process variations.

To the best of our knowledge, ours is the first delay macromodeling strategy in the context of input variations. Our macromodels are constructed offline and can be made more accurate by including more features from the later stages of the design flow. For example, optimizations during logic synthesis can modify the design delay significantly. In this work, we show that our macromodels can faithfully capture delay changes resulting due to downstream synthesis optimizations. In future work, we could refine our macromodels to model other downstream features such as parasitics, interconnect delay and crosstalk. These refined macromodels can then be plugged into our SHARPE methodology to achieve the desired level of estimation accuracy. Since the macromodels are but plugins, refining them will not require departure from the fundamental SHARPE methodology that we propose in this thesis.

11.3 Performance analysis of MIMO systems

Conventionally, performance estimation is done by performing Monte Carlo simulations [61] of MIMO RTL using random input vectors. Estimates that are reasonably accurate can be obtained by simulating the MIMO systems [25] over many cycles. This technique is time-consuming and incomplete. FPGA implementations [106] and ASIC prototypes [107] provide accelerated simulations, thereby speeding up performance estimation. However, both these methods involve significant overheads in terms of cost.

The performance of high level systems can be computed formally using probabilistic model checking [108] and Markov chains [60]. Markov chains have also been used at a circuit level, to design circuits with high error tolerance [100] and to analyze stability [101]. In [62], we present a novel methodology that uses probabilistic model checking at RTL in order to estimate error-related performance. However, none of the above techniques model faults that may be present in the physical hardware implementation. Therefore, they cannot be used to formally analyze the vulnerability of performance to physical faults.

Several simulation-based techniques exist that study the effects of physical faults by injecting them into RTL designs [8],[67]. In [70], the authors propose a formal verification methodology in order to determine whether a fault that is present in the interior of an RTL design can propagate to an output of interest. However, we are interested in computing the average probability with which such propagations can occur rather than checking for a single instance of their occurrence.

Although several techniques exist that perform a probabilistic analysis of the effects of hardware faults [9],[109],[110] they are mostly simulation-based, and therefore not rigorous. In [102], the authors use probabilistic model checking to formally evaluate the reliability of defect-tolerant systems. However, the evaluation is restricted to gate-level descriptions of the systems. An exact probabilistic analysis of faults for RTL designs is presented in [111]. However, each bit of an RTL variable needs to be represented individually. Therefore, it is infeasible to use the techniques in [102],[111] for complex RTL designs.

To the best of our knowledge, ours is the first work that provides a unified framework which incorporates the effects of physical faults while formally analyzing BER performance from the RTL description.

11.4 Probabilistic timing analysis

At the lower levels of design (e.g., gate-level), statistical static timing analysis (SSTA) is expected to provide highly accurate estimates. Gate-level SSTA is a well-established research topic that has matured over the past few decades. During this time period, SSTA has evolved to use sophisticated delay models in order to provide statistical timing estimates that are highly accurate. At the gate-level, timing verification methods include SSTA techniques such as [5],[75],[76],[77]. Some circuit design techniques like [72] use such gate-level timing analyses to enable better design goals than pessimistic worst case design in the presence of process variations. For input-dependent timing variations, the performance of better-than-worst-case designs [2],[3],[73],[74] can be verified using gate-level probabilistic timing analysis described in [26]. However, such gate-level techniques do not offer a scalable solution for statistical timing analysis at the higher levels of design.

In recent work, SSTA has been adapted to be employed early in the design flow, during high-level synthesis [7],[78],[79]. These *high-level SSTA* techniques use relatively simple delay macromodels to introduce variation-awareness early in the design flow. The authors demonstrate that such variation-awareness can improve high-level design exploration. Such high-level approaches emphasize on predictability and are not intended to be highly accurate in comparison to downstream analysis.

Unlike existing high-level SSTA, our SHARPE methodology is applied to RTL. Moreover, our methodology can be applied in the context of both input variations and process variations. Therefore, a direct comparison of our SHARPE methodology with existing high-level SSTA is not possible.

11.5 Aging analysis in hardware

Commercial tools like RelXpert [112] perform extensive simulations at the transistor-level in order to estimate the delay degradation of the circuit. In [30],[31],[32],[33],[34] aging effects are analyzed at the gate-level. Our methodology is more scalable than these techniques since we perform analysis at a higher level of abstraction. In [32], the delay degradation of microarchitectural components is estimated by synthesizing them into gate-level netlists. Our methodology operates at a finer granularity since the degradation is estimated for each RTL statement.

11.6 Compositional reasoning for formal verification

Compositional techniques have been used before to improve the scalability of formal hardware verification [38]. A form of circular assume-guarantee reasoning is used while model checking the individual components. Additionally, a *case-splitting* technique is used while verifying properties of a component over a set of different data values. However, the decomposition strategy is not automatic and is specifically intended for non-probabilistic model checking.

Automatic decomposition has been proposed for systems with Boolean variables [90]. The dependence between components is expressed through relations that are obtained through *learning-based* techniques [90],[113]. However, this approach considers only non-probabilistic systems, and therefore cannot be extended to probabilistic model checking of hardware designs.

Several compositional reasoning approaches have been presented in the context of probabilistic model checking [37],[39],[40],[41],[42]. However, these approaches rely on the ability of the designer to identify each M_i and the corresponding ϕ_i. These approaches do not describe any automatic methodology to derive the components and their corresponding properties. Therefore, a large amount of manual intervention is demanded while employing such techniques. Moreover, these approaches are not intended specifically for hardware designs, and therefore cannot exploit the characteristics of hardware systems.

11.7 Abstraction for formal verification

In the realm of software verification, there exist several techniques [114],[115] for predicate abstraction. Properties regarding program correctness/safety can be expressed using a set of predicates, that are either specified or automatically inferred. These predicates can be used to abstract a program and convert it into a Boolean program on which the properties can be easily verified. More generally, *abstract interpretation* [20] is the theory of reasoning with the approximate semantics of a large program rather than the set of all possible concrete behaviors. However, unlike predicate abstraction, all such abstractions are not necessarily property-specific. In all these abstractions, the concrete numeric values of data can either be completely abstracted out of the program or can be restricted to finite intervals [116].

Data abstraction techniques have been applied even in the context of hardware verification [92]. These techniques employ predicate abstraction in order to focus on the verification of Boolean control logic for which the exact numeric values of datapath variables are inconsequential. In [44], RTL designs are verified by restricting data values to intervals that are imposed by the execution of the RTL program. Therefore, these intervals are not property-specific.

Abstraction techniques have been employed in the context of probabilistic systems as well [42],[43],[117],[118]. In [118], the abstraction is performed on the source code itself. However, this technique is intended for probabilistic software and cannot be extended to RTL designs.

CHAPTER 12

CONCLUSION

We have presented SHARPE, a methodology for formally estimating statistical invariants regarding RTL performance. The key advantages of our SHARPE methodology is two-fold. Firstly, the use of formal probabilistic analysis makes SHARPE analysis rigorous compared to conventional simulated-based techniques for statistical analysis. Secondly, with the use of macromodels, our SHARPE methodology provides estimates of performance as early as the RTL stage. Therefore, our SHARPE methodology is an effective CAD technique that provides quick, accurate RTL estimates of performance which a designer can use to efficiently explore the RTL design space.

Formal verification has most success stories in the hardware domain. Traditionally, hardware verification has been limited to checking functional correctness. In order to facilitate widespread adoption of formal verification, it must remain relevant and practical even in new contexts such as variation-aware timing verification. Our work represents a strategic step in this direction. With the SHARPE methodology, we define a probabilistic notion of hardware correctness into the RTL verification paradigm.

We have demonstrated the breadth of the scope of the SHARPE methodology by applying it to estimate a diverse range of statistical performance metrics in the presence of different sources of variations. We apply our methodology to verify statistical properties of both combinational and sequential designs. In Chapter 4, we apply our SHARPE methodology in order to compute the BER performance of MIMO RTL designs. We show that our methodology can also provide realistic estimates of BER by modeling the presence of physical hardware faults. In Chapter 5, we apply our methodology in order to compute probabilistic delay distributions at the outputs of RTL modules. We show that our SHARPE methodology can consider either input variations or process variations as the primary source of statistics in delay. In Chapter 6, we apply our techniques to provide RTL estimates of delay degradation in the presence of aging effects. In both applications, we show that the RTL estimates provided by our methodology closely track those obtained from the gate level.

The scalability of our methodology is limited by the feasibility of the formal probabilistic analysis at the core of the SHARPE methodology. In this thesis, we have explored a slew of approaches to significantly improve the scalability of our methodology. In Chapter 4, we scale our SHARPE methodology by exploiting the symmetry inherently exhibited by large classes of MIMO designs. In Chapter 7, we present a novel approach for automatic compositional reasoning for probabilistic verification of hardware designs. In Chapter 8, we present our technique for property-specific data abstraction that employs static analysis of RTL source code. We demonstrate the soundness and effectiveness of our approaches by considering several case studies.

We find that, even with our techniques for scalability improvement, the SHARPE methodology is still not feasible for large RTL designs such as those pertaining to multicore systems. For such massive RTL designs, we employ statistical model checking, a simulation-based, scalable alternative to probabilistic model checking. We demonstrate this scalable framework by verifying the performance of dynamic power management schemes in OpenSPARC, an industry-strength, eight-core processor. In this thesis, we also demonstrate an importance sampling approach to accelerate the statistical model checking engine in rare-event scenarios of SRAM cell.

With growing sources of variations in hardware, we expect statistical design techniques such as "better-than-worst-case" design to become more widespread. In such a scenario, the notion of hardware correctness being a statistical metric will become the norm rather than the exception. Therefore, the groundwork laid by our SHARPE methodology and the other approaches presented in this thesis will be essential for developing more sophisticated statistical analyses to provide performance guarantees for hardware.

REFERENCES

[1] K. A. Bowman, M. Orshansky, and S. S. Sapatnekar, “Tutorial ii: Variability and its impact on design,” in *Proceedings of ISQED’06*, 2006, p. 5.

[2] D. Ernst, N. S. Kim, S. Das, S. Pant, T. Pham, R. Rao, C. Ziesler, D. Blaauw, T. Austin, and T. Mudge, “Razor: A low-power pipeline based on circuit-level timing speculation,” in *Proceedings of MICRO’03*, 2003.

[3] B. Greskamp and J. Torrellas, “Paceline: Improving single-thread performance in nanoscale cmps through core overclocking,” in *Proceedings of PACT 2007*, 2007.

[4] T. Austin, V. Bertacco, D. Blaauw, and T. Mudge, “Opportunities and challenges for better than worst-case design,” in *Proceedings of ASP-DAC’05*, 2005, pp. 2–7.

[5] M. Berkelaar, “Statistical delay calculation,” Workshop Notes of the International Workshop on Logic Synthesis, 1997.

[6] M. Orshansky and A. Bandyopadhyay, “Fast statistical timing analysis handling arbitrary delay correlations,” in *Proceedings of DAC’04*, 2004, pp. 337–342.

[7] G. Lucas, S. Cromar, and D. Chen, “Fastyield: variation-aware, layout-driven simultaneous binding and module selection for performance yield optimization,” in *Proceedings of ASP-DAC’09*, 2009, pp. 61–66.

[8] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. patel, “Characterizing the effects of transient faults on a high-performance proceedingsssor pipeline,” in *Proceedings of DSN’04*, 2004, p. 61.

[9] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, “A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor,” in *Proceedings of MICRO’03*, 2003, pp. 29–40.

[10] A. Lungu, P. Bose, A. Buyuktosunoglu, and D. J. Sorin, “Dynamic power gating with quality guarantees,” in *Proceedings of ISLPED’09*, 2009, pp. 377–382.

[11] K. K. Rangan, G.-Y. Wei, and D. Brooks, “Thread motion: fine-grained power management for multi-core systems,” in *Proceedings of ISCA’09*, 2009, pp. 302–313.

[12] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *Proceedings of MICRO'06*, 2006, pp. 347–358.

[13] L. Benini, A. Bogliolo, and G. De Micheli, "A survey of design techniques for system-level dynamic power management," *IEEE TVLSI*, vol. 8, pp. 299–316, June 2000.

[14] J. G. Proakis and M. Salehi, *Communication systems engineering.* Prentice-Hall, Inc., 1994.

[15] R. Teodorescu and J. Torrellas, "Variation-aware application scheduling and power management for chip multiprocessors," in *Proceedings of ISCA'08*, 2008.

[16] T. S. Hoang, Z. Jin, K. Robinson, A. McIver, and C. Morgan, "Probabilistic invariants for probabilistic machines," in *Proceedings of ZB'03.* Springer, 2003, pp. 240–259.

[17] S. Sambamurthy, J. Abraham, R. Tupuri, and S. Raghuram, "A robust top-down dynamic power estimation methodology for delay constrained register transfer level sequential circuits," in *Proceedings of VLSID'08*, 2008, pp. 521–526.

[18] T. Koyagi, M. Fukui, and R. Saleh, "Delay macromodeling and estimation for RTL," in *Proceedings of ISCAS'08*, May 2008, pp. 2430–2433.

[19] M. Nourani and C. A. Papachristou, "False path exclusion in delay analysis of RTL structures," *IEEE TVLSI*, vol. 10, no. 1, pp. 30–43, 2002.

[20] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points," in *Proceedings of POPL'77*, 1977, pp. 238–252.

[21] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press, 2005.

[22] H. Hansson and B. Jonsson, "A logic for reasoning about time and reliability," *Formal Aspects of Computing*, vol. 6, pp. 102–111, 1994.

[23] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 2.0: A tool for probabilistic model checking," in *Proceedings of QEST'04*, 2004, pp. 322–323.

[24] D. Tse and P. Viswanath, *Fundamentals of wireless communication.* Cambridge University Press, 2005.

[25] J. H. Han, A. T. Erdogan, and T. Arslan, "A low power pipelined maximum likelihood detector for 4x4 QPSK MIMO wireless communication systems," in *Proceedings of ISVLSI'06*, 2006, p. 185.

[26] L. Wan and D. Chen, "DynaTune: circuit-level optimization for timing speculation considering dynamic path behavior," in *Proceedings of ICCAD'09*, 2009, pp. 172–179.

[27] M. A. Alam and S. Mahapatra, "A comprehensive model of PMOS NBTI degradation," *Microelectronics Reliability*, vol. 45, no. 1, pp. 71 – 81, 2005.

[28] B. C. Paul, K. Kang, H. Kufluoglu, M. A. Alam, and K. Roy, "Temporal performance degradation under NBTI: estimation and design for improved reliability of nanoscale circuits," in *Proceedings of DATE'06*, 2006, pp. 780–785.

[29] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, pp. 10–16, Nov. 2005.

[30] S. V. Kumar, C. H. Kim, and S. S. Sapatnekar, "An analytical model for negative bias temperature instability," in *Proceedings of ICCAD'06*, 2006, pp. 493–496.

[31] W. Wang, S. Yang, S. Bhardwaj, R. Vattikonda, S. Vrudhula, F. Liu, and Y. Cao, "The impact of NBTI on the performance of combinational and sequential circuits," in *Proceedings of DAC'07*, 2007, pp. 364–369.

[32] M. DeBole, K. Ramakrishnan, V. Balakrishnan, W. Wang, H. Luo, Y. Wang, Y. Xie, Y. Cao, and N. Vijaykrishnan, "A framework for estimating NBTI degradation of microarchitectural components," in *Proceedings of ASP-DAC'09*, 2009, pp. 455–460.

[33] S. V. Kumar, C. H. Kim, and S. S. Sapatnekar, "NBTI-aware synthesis of digital circuits," in *Proceedings of DAC'07*, 2007, pp. 370–375.

[34] J. Abella, X. Vera, and A. Gonzalez, "Penelope: The NBTI-aware proceedingsssor," in *Proceedings of MICRO'07*, 2007, pp. 85–96.

[35] W. D. Obal II, M. G. McQuinn, and W. H. Sanders, "Detecting and exploiting symmetry in discrete-state Markov models," in *Proceedings of PRDC'06*, 2006, pp. 26–38.

[36] M. Kwiatkowska, G. Norman, and D. Parker, "Symmetry reduction for probabilistic model checking," in *Proceedings of CAV'06*, 2006, pp. 234–248.

[37] L. de Alfaro, T. A. Henzinger, and R. Jhala, "Compositional methods for probabilistic systems," in *Proceedings of CONCUR'01*, 2001, pp. 351–365.

[38] R. Jhala and K. L. McMillan, "Microarchitecture verification by compositional model checking," in *Proceedings of CAV'01*, 2001, pp. 396–410.

[39] N. Lynch, R. Segala, and F. Vaandrager, "Compositionality for probabilistic automata," in *Proceedings of CONCUR'03*. Springer-Verlag, 2003, pp. 208–221.

[40] J. Hillston, "A compositional approach to performance modelling," PhD Thesis, The University of Edinburgh, 1996.

[41] R. Segala, "Compositional verification of randomized distributed algorithms," in *Proceedings of COMPOS'97*, 1998, pp. 515–540.

[42] M. Kattenbelt, M. Kwiatkowska, G. Norman, and D. Parker, "Game-based probabilistic predicate abstraction in PRISM," *ENTCS*, vol. 220, no. 3, pp. 5–21, 2008.

[43] L. D. Alfaro and P. Roy, "Magnifying-lens abstraction for Markov decision proceedingssses," 2006.

[44] V. P. Nazanin, N. Mansouri, and R. Vemuri, "Automatic data path abstraction for verification of large scale designs," in *Proceedings of the ICCD*, 1998, pp. 192–194.

[45] H. L. S. Younes and R. G. Simmons, "Probabilistic Verification of Discrete Event Systems using Acceptance Sampling," in *Proceedings of CAV'02*, 2002, pp. 223–235.

[46] A. Wald, "Sequential Tests of Statistical Hypotheses," *The Annals of Mathematical Statistics*, vol. 16, no. 2, pp. 117–186, 1945.

[47] T. C. Hesterberg, "Advances in importance sampling," PhD Dissertation, Stanford University, Palo Alto, CA, 1988.

[48] F. Southey, D. Schuurmans, and A. Ghodsi, "Regularized greedy importance sampling," in *Proceedings of NIPS'02*, 2002, pp. 753–760.

[49] D. P. Reijsbergen, P. T. de Boer, W. R. W. Scheinhardt, and B. R. H. M. Haverkort, "Rare event simulation for highly dependable systems with fast repairs," in *Proceedings of QEST'10*, 2010, pp. 251–260.

[50] A. Singhee and R. A. Rutenbar, "Statistical blockade: very fast statistical simulation and modeling of rare circuit events and its application to memory design," *IEEE TCAD*, vol. 28, pp. 1176–1189, Aug. 2009.

[51] R. Kanj, R. Joshi, and S. Nassif, "Mixture importance sampling and its application to the analysis of sram designs in the presence of rare failure events," in *Proceedings of DAC'06*, 2006, pp. 69–72.

[52] G. D. Forney, Jr., "Maximum-likelihood sequence estimation of digital sequences in the presence of intersymbol interference," *IEEE Transactions on Information Theory*, vol. 18, no. 3, pp. 363–378, May 1972.

[53] J. R. Norris, *Markov Chains.* Cambridge University Press, 1997.

[54] E. M. Clarke, M. Fujita, S. P. Rajan, T. W. Reps, S. Shankar, and T. Teitelbaum, "Program slicing of hardware description languages," in *Proceedings of CHARME'99*, 1999, pp. 298–312.

[55] Y.-C. Wang, A. Komuravelli, P. Zuliani, and E. M. Clarke, "Analog circuit verification by statistical model checking," in *Proceedings of ASP-DAC'11*, 2011.

[56] K. Sen, M. Viswanathan, and G. Agha, "VESTA: A statistical model-checker and analyzer for probabilistic systems," in *Proceedings of QEST'05*, Sept. 2005, pp. 251 – 252.

[57] A. Legay, B. Delahaye, and S. Bensalem, "Statistical model checking: An overview," in *Proceedings of RV'10*, 2010, pp. 122–135.

[58] H. Younes, "Ymer: A statistical model checker," pp. 429–433, 2005.

[59] D. N. Jansen, J.-P. Katoen, M. Oldenkamp, M. Stoelinga, and I. Zapreev, "How fast and fat is your probabilistic model checker? An experimental performance comparison," in *Proceedings of HVC'07*, 2008, pp. 69–85.

[60] G. Norman, D. Parker, M. Kwiatkowska, S. K. Shukla, and R. K. Gupta, "Formal analysis and validation of continuous-time Markov chain based system level power management strategies," in *Proceedings of HLDVT'02*, 2002, p. 45.

[61] M. Jeruchim, "Techniques for estimating the bit error rate in the simulation of digital communication systems," *IEEE J-SAC'84*, vol. 2, no. 1, pp. 153–170, 1984.

[62] J. A. Kumar and S. Vasudevan, "Statistical guarantees of performance for MIMO designs," in *Proceedings of DSN'10*, 2010, pp. 467 –476.

[63] K. Roy and J. A. Abraham, "A novel approach to accurate timing verification using RTL descriptions," in *Proceedings of DAC '89*, 1989, pp. 638–641.

[64] S. Andova, H. Hermanns, and J.-P. Katoen, "Discrete-time rewards model-checked," pp. 88–104, 2003.

[65] S. Derisavi, H. Hermanns, and W. H. Sanders, "Optimal state-space lumping in Markov chains," *Information Proceedingsssing Letters*, vol. 87, no. 6, pp. 309 – 315, 2003.

[66] "Synopsis formality," [Online]. Available: http://www.synopsys.com/tools/verification/formalequivalence/pages/formality.aspx, 2012.

[67] M.-L. Li, P. Ramachandran, U. Karpuzcu, S. Hari, and S. Adve, "Accurate microarchitecture-level fault modeling for studying hardware faults," in *Proceedings of HPCA'09*, feb. 2009, pp. 105 –116.

[68] M. Zhang and N. Shanbhag, "Soft-error-rate-analysis (SERA) methodology," *IEEE TCAD*, vol. 25, no. 10, pp. 2140 –2155, Oct. 2006.

[69] B. W. Johnson, Ed., *Design & analysis of fault tolerant digital systems.* Addison-Wesley Longman Publishing Co., Inc., 1988.

[70] S. A. Seshia, W. Li, and S. Mitra, "Verification-guided soft error resilience," in *Proceedings of DATE '07*, 2007, pp. 1442–1447.

[71] N. J. Rohrer, "Introduction to statistical variation and techniques for design optimization," Tutorial, ISSCC, 2006.

[72] N. Shanbhag, "Reliable and energy-efficient digital signal processing," in *Proceedings of DAC'02*, 2002, pp. 830–835.

[73] Y.-S. Su, D.-C. Wang, S.-C. Chang, and M. Marek-Sadowska, "An efficient mechanism for performance optimization of variable-latency designs," in *Proceedings of DAC '07*, 2007, pp. 976–981.

[74] L. Benini, E. Macii, M. Poncino, and G. D. Micheli, "Telescopic units: A new paradigm for performance optimization of VLSI designs," *IEEE TCAD*, vol. 17, no. 3, pp. 220–232, 1998.

[75] H. Chang and S. S. Sapatnekar, "Statistical timing analysis considering spatial correlations using a single pert-like traversal," in *Proceedings of ICCAD'03*, 2003.

[76] J.-J. Liou, K.-T. Cheng, S. Kundu, and A. Krstic, "Fast statistical timing analysis by probabilistic event propagation," in *Proceedings of DAC'01*, 2001, pp. 661–666.

[77] M. Orshansky and K. Keutzer, "A general probabilistic framework for worst case timing analysis," in *Proceedings of DAC'02*, 2002, pp. 556–561.

[78] J. Jung and T. Kim, "Timing variation-aware high-level synthesis," in *Proceedings of ICCAD'07*, 2007, pp. 424–428.

[79] W.-L. Hung, X. Wu, and Y. Xie, "Guaranteeing performance yield in high-level synthesis," in *Proceedings of ICCAD'06*, 2006, pp. 303–309.

[80] J. A. Kumar and S. Vasudevan, "Variation-conscious formal timing verification in RTL," in *Proceedings of VLSID'11*, 2011, pp. 58–63.

[81] A. Bogliolo, L. Benini, and G. De Micheli, "Regression-based RTL power modeling," *ACM TODAES*, vol. 5, no. 3, pp. 337–372, 2000.

[82] C. Baier, M. Grer, and F. Ciesinski, "Partial order reduction for probabilistic systems," in *Proceedings of QEST'04*, 2004, pp. 230–239.

[83] S. Nowick, "Design of a low-latency asynchronous adder using speculative completion," *IEE Proceedingsedings - Computers and Digital Techniques*, vol. 143, no. 5, pp. 301–307, Sep 1996.

[84] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," in *Proceedings of DAC'03*, 2003, pp. 338–342.

[85] P. R. Panda and N. D. Dutt, "1995 high level synthesis design repository," in *ISSS*, 1995, pp. 170–174.

[86] "Predictive technology model," Device Group at Arizona State University, [Online]. Available: http://www.eas.asu.edu/~ptm, Nov. 2008.

[87] S. V. Kumar, "Reliability-Aware And Variation-Aware CAD Techniques," Ph.D. dissertation, University of Minnesota, 2009.

[88] Y. Wang, X. Chen, W. Wang, V. Balakrishnan, Y. Cao, Y. Xie, and H. Yang, "On the efficacy of input vector control to mitigate NBTI effects and leakage power," in *Proceedings of ISQED'09*, 2009, pp. 19–26.

[89] A. V. Oppenheim, R. W. Schafer, and J. R. Buck, *Discrete-time signal processing (2nd ed.)*. Prentice-Hall, Inc., 1999.

[90] W. Nam, P. Madhusudan, and R. Alur, "Automatic symbolic compositional verification by learning assumptions," *FMSD*, vol. 32, no. 3, pp. 207–234, 2008.

[91] P. Cousot, "Abstract interpretation," *ACM Computing Surveys*, vol. 28, no. 2, pp. 324–328, 1996.

[92] E. Clarke, O. Grumberg, M. Talupur, and D. Wang, "High level verification of control intensive systems using predicate abstraction," in *Proceedings of MEMOCODE'03*, 2003, pp. 55–65.

[93] L. Liu and S. Vasudevan, "Efficient validation input generation in RTL by hybridized source code analysis," in *Proceedings of DATE'11*, 2011, pp. 1596–1601.

[94] P. Johannsen, "BOOSTER: Speeding up RTL property checking of digital designs by word-level abstraction," in *Proceedings of CAV'01*, 2001, pp. 373–377.

[95] J. C. King, "Symbolic execution and program testing," *Communications of ACM*, vol. 19, pp. 385–394, July 1976.

[96] M. Berkelaar, K. Eikland, and P. Notebaert, "lp_solve 5.5, open source (mixed-integer) linear programming system," May 2004. [Online]. Available: http://lpsolve.sourceforge.net/5.5/

[97] A. Lungu, P. Bose, D. J. Sorin, S. German, and G. Janssen, "Multicore power management: Ensuring robustness via early-stage formal verification," in *Proceedings of MEMOCODE'09*, 2009, pp. 78 –87.

[98] "OpenSPARC T2 core microarchitecture specification," [Online]. Available: https://www.opensparc.net/pubs/t2/docs//OpenSPARCT2_Core_Micro_Arch.pdf, Dec. 2007.

[99] G. Norman, D. Parker, M. Z. Kwiatkowska, S. K. Shukla, and R. Gupta, "Using probabilistic model checking for dynamic power management." *Formal Aspects of Computing*, pp. 160–176, 2005.

[100] K. Nepal, R. I. Bahar, J. L. Mundy, W. R. Patterson, and A. Zaslavsky, "Designing logic circuits for probabilistic computation in the presence of noise," in *Proceedings of DAC'05*, 2005, pp. 485–490.

[101] E. Clarke, A. Donzé, and A. Legay, "Statistical model checking of mixed-analog circuits with an application to a third order $\Delta - \Sigma$ modulator," in *Proceedings of HVC '08*, 2009, pp. 149–163.

[102] G. Norman, D. Parker, M. Kwiatkowska, and S. Shukla, "Evaluating the reliability of NAND multiplexing with PRISM," *IEEE TCAD'05*, vol. 24, no. 10, pp. 1629–1637, 2005.

[103] B. Akbarpour and S. Tahar, "A methodology for the formal verification of fft algorithms in hol," in *Proceedings of FMCAD'04*, 2004, pp. 37–51.

[104] G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. Webster, "The Mobius modeling tool," in *Proceedings of PNPM'01*, 2001, p. 241.

[105] D. Bertozzi, L. Benini, and B. Ricco', "Parametric timing and power macromodels for high level simulation of low-swing interconnects," in *Proceedings of ISLPED'02*, 2002, pp. 307–312.

[106] D. Markovic, C. Chang, B. Richards, H. So, B. Nikolic, and R. Brodersen, "ASIC design and verification in an FPGA environment," in *Proceedings of CICC'07*, Sept. 2007, pp. 737–740.

[107] A. Burg, M. Borgmann, M. Wenk, M. Zellweger, W. Fichtner, and H. Bölcskei, "VLSI implementation of MIMO detection using the sphere decoding algorithm," *IEEE JSSC'05*, vol. 40, no. 7, pp. 1566–1577, July 2005.

[108] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: Probabilistic model checking for performance and reliability analysis," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 4, pp. 40–45, 2009.

[109] S. Hari, M.-L. Li, P. Ramachandran, B. Choi, and S. Adve, "mSWAT: Low-cost hardware fault detection and diagnosis for multicore systems," in *Proceedings of MICRO'09*, dec. 2009, pp. 122 –132.

[110] X. Li and D. Yeung, "Application-level correctness and its impact on fault tolerance," in *Proceedings of HPCA'07*, 2007, pp. 181–192.

[111] J. Fernandes, M. Santos, A. Oliveira, and J. Teixeira, "Probabilistic testability analysis and DFT methods at RTL," in *Proceedings of DDECS'06*, 2006, pp. 214 –215.

[112] "Cadence RelXpert manual," [Online]. Available: http://www.cadence.com, 2008.

[113] A. Gupta, K. L. Mcmillan, and Z. Fu, "Automated assumption generation for compositional verification," *FMSD*, vol. 32, no. 3, pp. 285–301, 2008.

[114] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, "SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft," in *Proceedings of IFM'04*, 2004, pp. 1–20.

[115] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from proofs," *SIGPLAN Notes*, vol. 39, no. 1, pp. 232–244, 2004.

[116] D. Monniaux, "A minimalistic look at widening operators," *Higher Order Symbolic Computation*, vol. 22, no. 2, pp. 145–154, 2009.

[117] P. R. D'argenio, H. E. Jensen, and K. G. Larsen, "Reachability analysis of probabilistic systems by successive refinements," in *Proceedings of PAPM/PROBMIV'01*, 2001, pp. 39–56.

[118] M. Kattenbelt, M. Kwiatkowska, G. Norman, and D. Parker, "Abstraction refinement for probabilistic software," pp. 182–197, 2009.