Approximate Computing in Digital Design

Sayandeep Mitra

Approximate Computing in Digital Design

Dissertation submitted in partial fulfillment of the requirements for the degree of

Master of Technology
in
Computer Science

by

Sayandeep Mitra
[Roll No: CS-1413]

under the guidance of

Ansuman Banerjee
Associate Professor
Advanced Computing and Microelectronics Unit

Indian Statistical Institute
Kolkata-700108, India

July 2016

To my family and my supervisor

CERTIFICATE

This is to certify that the dissertation entitled **"Approximate Computing in Digital Designs"** submitted by **Sayandeep Mitra** to Indian Statistical Institute, Kolkata, in partial fulfillment for the award of the degree of **Master of Technology in Computer Science** is a bonafide record of work carried out by him under my supervision and guidance. The dissertation has fulfilled all the requirements as per the regulations of this institute and, in my opinion, has reached the standard needed for submission.

Ansuman Banerjee
Associate Professor,
Advanced Computing and Microelectronics Unit,
Indian Statistical Institute,
Kolkata-700108, INDIA.

Acknowledgments

I would like to show my highest gratitude to my advisor, *Ansuman Banerjee*, Advanced Computing and Microelectronics Unit, Indian Statistical Institute, Kolkata, for his guidance and continuous support and encouragement. He has literally taught me how to do good research, and motivated me with great insights and innovative ideas.

My deepest thanks to all the teachers of Indian Statistical Institute, for their valuable suggestions and discussions which added an important dimension to my research work.

Finally, I am very much thankful to my parents and family for their everlasting support.

Last but not the least, I would like to thank all my friends for their help and support. I thank all those, whom I have missed out from the above list.

Sayandeep Mitra
Indian Statistical Institute
Kolkata - 700108, India.

Abstract

In recent times, approximate computing is being looked at as a viable alternative for reducing the energy consumption of programs, while marginally compromising on the correctness of their computation. The idea behind approximate computing is to introduce approximations at various levels of the execution stack, with an attempt to realize the resource hungry computations on low resource consuming approximate hardware blocks. Approximate computing for program transformation faces a serious challenge of automatically identifying core program areas/statements where approximations can be introduced, with a quantifiable measure of the resulting program correctness compromise. Introducing approximations randomly can cause performance deterioration without much energy advantage, which is undesirable. In this thesis, we introduce a verification-guided method to automatically identify program blocks which lend themselves to easy approximations, while not compromising significantly on program correctness. Our method is based on identifying regions of code which are less influential for the computation of the program outputs and therefore, can be compromised with, however still having a potential of significant resource reduction. We take the help of assertions to quantify the effect of the resulting transformations on program outputs. We show experimental results to support our proposal.

Keywords: *Approximate Computing, Program Slicing, Assertions, Verification, Statement Coverage, Linear Programming.*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Energy efficiency has become an element of paramount concern in design of computing systems. Ongoing technological developments and the Internet of Things mean more aspects of our lives are being computerized and connected, needing ever more processing of data, thereby requiring the computing systems to become increasingly embedded and mobile. Despite advances in reducing the power consumption of devices and enhanced battery technology, today's designs continue to increase their energy use as the amount of computation increases, at a time when energy efficiency is being encouraged and demands on battery life increasingly scrutinized.

Approximate computing is an emerging design paradigm that aims to address the energy utilization problem from a completely different perspective, by using the inherent resilience of applications to perform computations in an in-exact manner. Computing today is not always about producing a precise numerical result at the end. Many applications have intrinsic tolerance of minor to moderate elements of inaccuracy. Applications in domains like computer vision, media processing, machine learning, and sensor data analysis already incorporate imprecision into their design. Large scale data analytics focus on aggregate trends rather than the integrity of individual data elements. In domains such as computer vision and robotics, there are no perfect answers: results can vary in their usefulness, and the output quality is always in tension with the resources that the software needs to produce them. All these applications are approximate programs: a range of possible values can be considered correct outputs for a given input.

The central challenge in approximate computing is forging abstractions that make imprecision controlled and predictable without sacrificing its efficiency benefits. Many applications are often intrinsically resilient to a large part of their computations being performed in an approximate or imprecise manner, enabling us to save computing resources. Approximate computing helps design platforms for which correctness is defined as producing results that are good enough, judged by some metric, departing from the long held belief that computing platforms should be developed by the same strict notion of correctness. A number of research articles have discussed strategies for implementation of approximate computing in both hardware and software. In [30], approximate computing of applications by loop perforations is proposed. Critical versus tunable loops are identified by intentional perforation of loops and observing the respective output. For hardware, [15] presents the design of a system architecture for approximate applications. [20] designs an imprecise adder which consumes low power performing approximate computing.

The technology of approximate computing today is poised at an interesting juncture, which has led researchers to identify the two main subproblems as below.

1. Identification of areas to apply approximation
2. Application of suitable approximations, and verify the correctness and gain of the application.

This dissertation attempts to address both the above problems and create a framework that can enable us harness the full power of approximate computing in the best possible way. We discuss about our specific contributions below.

While existing literature talks mostly about methods for implementation of approximate computing in different layers of the execution stack and approximate designs, the problem of automatic identification of candidate program blocks amenable to approximation has been relatively less addressed. We aim to address this important question, specifically, we aim to develop an automated technique for identification of areas in an application where approximations can be implemented. A few articles in recent past have addressed this question in a semi-automated way. In [12, 25, 29] approximation aware programming frameworks are proposed, which provide language constructs to annotate functions, data or loop as approximate or precise. [13, 14] identify resilient computation kernels by injecting errors and then classifying them based on the output quality metric.

In this dissertation, we present A Verification Guided Approach for selective program transformations for Approximate Computing, that can be used to automatically identify areas in a digital design to apply Approximate Computing. A number of proposals have been made in [13, 14] to identify program blocks based on the kernel computation time of innermost loops. The fundamental difference with this body of work on identification of program blocks for application of approximations and our proposed approach is that existing approaches primarily use a semi-automated way based on the program execution on a specific set of inputs, whereas our work proposes a novel approach of statement identification based on program structure along with the execution traits of the design. Our approach stems from the observation that not all statements in a program execute with same effect, nor influence the computation of the outputs equally. Statements which are less prone to executions, or have minor effects on the program output, can thus be approximated, thereby saving computation resources resulting in improved system resource utilization and efficiency. In our model, we propose a formal framework for statement identification, that can identify the set of statements that are suitable for approximations, thereby reducing resource utilization. Experiments show our proposed framework is capable of saving resources significantly for standard Verilog designs.

1.1 Motivation of this dissertation

The general system stack is shown in Figure 1.1. From the figure it is clear that, approximation provides a lot of gain if applied to the hardware, while it is better if we apply approximation mainly at the user level or at the language level. The latter approach can be taken in two ways.

- Apply approximations directly in the application or program.
- Identify program areas to be computed with approximate hardware.

In both the methods, it is imperative that parts of the program suitable for approximation are identified, in an efficient method. The main issue, as mentioned is that it is difficult to receive a lot of gain by approximating at the language level. To make the best out of the situation, all possible approximating areas of a program needs to identified. To the best of our knowledge although work has been done to identify program statements for approximation in different machine learning programs, no work has yet been done in digital design. This is the main motivation for this dissertation, where we present such an approach for Verilog design codes. The main contribution of this dissertation is highlighted in the following section.

1.2 Contribution of this dissertation

In this dissertation, we propose an automated approach for statement identification for approximation in Verilog programs. We consider a simple overview of our work presented in Fig 1.2, where the three basic steps in our methodology are presented. In this work, initially, we identify statements based on their execution rate and their effects on outputs. After the initial set of statements are identified, we insert random errors (raw approximations) to the programs. This step is interchangeably used along with the correctness and compromise measure which measures the effect of the random error on the program functionality. We use assertions [18] to specify the correctness of the design. After the random error insertion step, we measure the number of assertions that change their truth value. Along with the correctness metric, we also provide the amount of power and circuit area that is reduced due to the random error. Based on these scores, we select the best set of statements to apply our approximations on. In order to avoid applying approximations which result in no

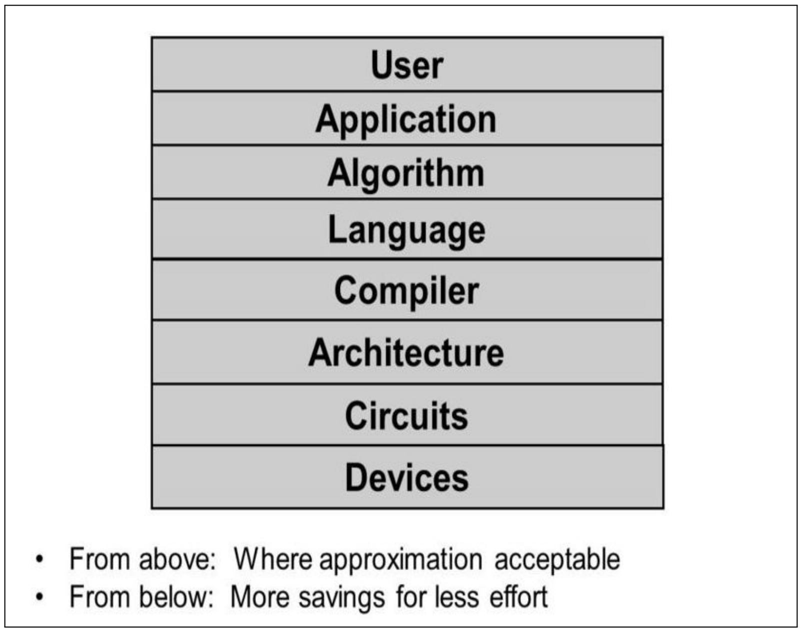
User
Application
Algorithm
Language
Compiler
Architecture
Circuits
Devices
• From above: Where approximation acceptable
• From below: More savings for less effort

Figure 1.1: Stack Showing Layers of Approximate Computing

gain or highly compromised circuits, we propose certain heuristics for candidate statement selection. This helps us to identify a set of statements suitable for approximations in an efficient and scalable way.

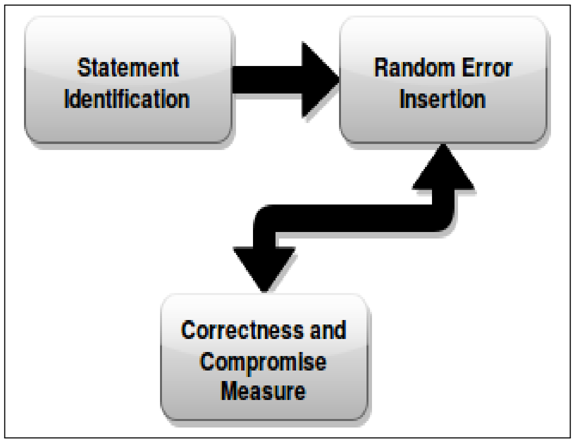

Figure 1.2: Overview of our Approach

1.3 Organization of the dissertation

The rest of the dissertation is organized into 6 chapters. A summary of the contents of the chapters is as follows:

Chapter 2: A detailed study of relevant research is presented here.

Chapter 3: This chapter describes the statement identification step of our approach.

Chapter 4: This chapter describes the random error insertion and correctness step of our approach.

Chapter 5: This chapter presents methods for candidate statement selection.

Chapter 6: This chapter describes the detailed case study of our work, implementation and results.

Chapter 7: We summarize with conclusions on the contributions of this dissertation.

Chapter 2

Background and related work

In this chapter, we first present a few background concepts needed for developing the foundation of our framework. We also present an overview of different schemes proposed in literature for approximate computing.

2.1 Background

In this section, we discuss a few background concepts.

2.1.1 Assertions

Assertions are primarily used to validate the behavior of a design (”Is it working correctly?”). They may also be used to provide functional coverage information for a design (”How good is the test?”). Assertions can be checked dynamically by simulation, or statically by a separate property checker tool i.e. a formal verification tool that proves whether or not a design meets its specification. Such tools may require certain assumptions about the design behavior to be specified.

Some of the popular assertion languages used in the industry are :

- PSL (Property Specification Language) based on IBM Sugar [4]
- Synopsys OVA (Open Vera Assertions) and OVL (Open Vera Library) [7]
- Assertions in Specman [6]
- 0-In (0In Assertions)

- SystemC Verification (SCV)
- SVA (SystemVerilog Assertions)

In this section, we introduce the popular type of assertions System Verilog Assertions (SVA) [18] and describe its functionality.

System Verilog Assertions

SystemVerilog assertions are built from sequences and properties. Properties are a superset of sequences; any sequence may be used as if it were a property, although this is not typically useful. In SystemVerilog there are two kinds of assertion: **immediate** (*assert*) and **concurrent** (*assert property*). Coverage statements (*cover property*) are concurrent and have the same syntax as concurrent assertions, as do *assume property* statements, which are primarily used by formal tools. Finally, expect is a procedural statement that checks that some specified activity (*property*) occurs. The three types of concurrent assertion statement and the expect statement make use of sequences that describe the design's " temporal behavior " i.e. behavior over time, as defined by one or more clocks.

Immediate Assertions

Immediate assertions are procedural statements and are mainly used in simulation. An assertion is basically a statement that something must be true, similar to the if statement. The difference is that an if statement does not assert that an expression should be true, it simply checks that it is true.

Example 2.1 *if (A == B) ... // Simply checks if A equals B*
assert (A == B); // Asserts that A equals B; if not, an error is generated

If the conditional expression of the immediate assert evaluates to X, Z or 0, then the assertion fails and the verification tool writes an error message. An immediate assertion may include a pass statement and/or a fail statement. The following example shows a case with an action specified if the assertion evaluates to true.

Example 2.2 *assert (A == B) $display ("OK. A equals B");*

It is executed immediately after the evaluation of the assert expression. The statement associated with an else is called a fail statement and is executed if the assertion fails:

Example 2.3 *assert (A == B) $display ("OK. A equals B"); else $error("It's gone wrong");*

We may omit the pass statement yet still include a fail statement:

Example 2.4 *assert (A == B) else $error("It's gone wrong");*

Concurrent Assertions

The behavior of a design may be specified using statements similar to these:

"The Read and Write signals should never be asserted together."

"A Request should be followed by an Acknowledge occurring no more than two clocks after the Request is asserted."

Concurrent assertions are used to check behavior such as these. These are statements that assert that specified properties must be true.

Example 2.5 *assert property (!(Read && Write));*

asserts that the expression *Read && Write* is never true at any point in the design.

Properties are often built using sequences.

Example 2.6 *assert property (@(posedge Clock) Req |−> ##[1:2] Ack);*

where Req is a simple sequence (it's just a Boolean expression) and ##[1:2] Ack is a more complex sequence expression, meaning that Ack is true on the next clock, or on the one following (or both). |−> is the implication operator, so this assertion checks that whenever Req is asserted, Ack must be asserted on the next clock, or the following clock.

Concurrent assertions like these are checked throughout simulation or formal verification. They usually appear outside any initial or always blocks in modules, interfaces and programs. Concurrent assertions may also be used as statements in initial or always blocks. A concurrent assertion in an initial block is only tested on the first clock tick.

The first assertion example above does not contain a clock. Therefore it is checked at every point in the simulation. The second assertion is only checked when a rising clock edge has occurred; the values of *Req* and *Ack* are sampled on the rising edge of Clock.

Implication

The implication construct ($|- >$) allows a user to monitor sequences based on satisfying some criteria, e.g. attach a precondition to a sequence and evaluate the sequence only if the condition is successful. The left-hand side operand of the implication is called the antecedent sequence expression, while the right-hand side is called the consequent sequence expression. If there is no match of the antecedent sequence expression, implication succeeds vacuously by returning true. If there is a match, for each successful match of the antecedent sequence expression, the consequent sequence expression is separately evaluated, beginning at the end point of the match.

There are two forms of implication: overlapped using operator $|- >$, and non-overlapped using operator $| =>$. For overlapped implication, if there is a match for the antecedent sequence expression, then the first element of the consequent sequence expression is evaluated on the same clock tick.

s1 $|- >$ s2;

In the example above, if the sequence s1 matches, then sequence s2 must also match. If sequence s1 does not match, then the result is true. For non-overlapped implication, the first element of the consequent sequence expression is evaluated on the next clock tick.

s1 $| =>$ s2;

where true is a boolean expression, used for visual clarity, that always evaluates to true.

Assertion Clocking

Concurrent assertions (assert property and cover property statements) use a generalized model of a clock and are only evaluated when a clock tick occurs. In fact, the values of the variables in the property are sampled right at the end of the previous time step. Everything in between clock ticks is ignored.

A clock tick is an atomic moment in time and a clock ticks only once at any simulation time. The clock can actually be a single signal, a gated clock (e.g. (clk && GatingSig)) or other more complex expression. When monitoring asynchronous signals, a simulation time step corresponds to a clock tick.

Example 2.7 *property p;*

@(posedge clk) a ##1 b;

endproperty

assert property (p);

Putting It All Together

We look at couple of complete examples for System Verilog Assertions.

Example 2.8 *"A request (req high for one or more cycles then returning to zero) is followed after a period of one or more cycles by an acknowledge (ack high for one or more cycles before returning to zero). ack must be zero in the cycle in which req returns to zero."*

assert property (@(posedge clk) disable iff reset

*!req ##1 req[*1:$] ##1 !req*

|− >

*!ack[*1:$] ##1 ack[*1:$] ##1 !ack);*

Example 2.9 *"After a request, ack must remain high until the cycle before grant is high. If grant goes high one cycle after req goes high then ack need not be asserted."*

assert property (@(posedge clk) disable iff reset $rose(req) | => ack[*0:$] ##1 grant);

where $rose(req) is true if req has changed from 0 to 1.

2.2 Related Work on Approximate Computing

In this section, we discuss some related work that has been done so far. In the first part of this section, we discuss various work done on automatic identification for approximate computing in literature. This is followed by a brief analysis of work done on ensuring quality of approximate computing.

A software framework for automatically discovering approximable data in a program by using statistical methods is presented in [28]. Their technique first collects the variables of the program and the range of values that they can take. Then, using binary instrumentation, the values of the variables are perturbed and the new output is measured. By comparing this against the correct output, which fulfills the acceptable QoS threshold, the contribution of each variable in the program output is measured. The variables are marked as approximable or nonapproximable based on the above score. Thus, their framework obviates the need of a programmers involvement or source code annotations for Approximate Computing. They compared this to a baseline with type-qualifier annotations by the programmer [29], their approach achieves nearly 85% accuracy in determining the approximable data. Their limitation is that some variables that are marked as nonapproximable in the programmer-annotated version may be marked as approximable by their technique, which can lead to errors. A technique was presented in [13][14] for automatic resilience characterization of applications. The method has two parts. The resilience identification part, considers innermost loops that occupy more than 1% of application execution time as atomic kernels. The application executes with input datasets, then random errors are introduced into the output variables of a kernel using the Valgrind DBI tool. If the output quality is not upto the mark or if the application crashes, the kernel is marked as sensitive; otherwise, it is potentially resilient. In the resilience characterization step, potentially resilient kernels are further explored to see the applicability of various approximation strategies. In this step, errors are introduced in the kernels using Valgrind based on the approximation models. To quantify resilience, they propose an Approximation Computing Technique(ACT)-independent model and an Approximate Computing Technique specific model for approximation. The ACT-independent approximation model studies the errors introduced due to ACT using a statistical distribution that shows the probability, magnitude, and predictability of errors. The ACT-specific model may use different ACTs, such as precision scaling, inexact arithmetic circuits, and loop perforation. The experimental results show that several applications show high resilience to errors, and many parameters such as the scale of input data, granularity of approximation have a significant impact on application resilience. Two techniques were presented in [27] for selecting approximable computations for a reduce and rank kernel. A reduce and rank kernel performs reduction between an input vector and each reference vector, the outputs are then ranked to find the subset of top reference vectors for that input. Their first technique decomposes vector reductions into multiple partial reductions and interleaves them with the rank computation. The next step identifies whether a particular reference vector is expected to appear in the final subset. Based on this, future computations that have little impact on the output after relaxation are selected. The second technique leverages the temporal or spatial correlation of inputs. Depending on the similarity between current and previous input, this technique approximates or entirely skips processing parts of the current inputs. Approximation is achieved using precision scaling and loop perforation strategies. Language extensions and an accuracy-aware compiler for facilitating writing of configurable-accuracy programs has been presented in [9]. The compiler performs auto tuning using a genetic algorithm to explore the search space of possible algorithms and accuracy levels for dealing with recursion and sub calls to other configurable-accuracy code. Initially, the population of candidate algorithms is maintained, which is expanded using mutators and later pruned to allow more optimal algorithms to evolve. Thus, the user needs to specify only accuracy requirements and does not need to understand algorithm specific parameters, while the library writer can write a portable code by simply specifying ways to search the space of parameter and algorithmic choices. To limit computation time, the number of tests performed for evaluating possible algorithms needs to be restricted. This can lead to the choice of suboptimal algorithms and errors, hence the number of tests performed needs to be carefully chosen. A programming language, named Rely was proposed in [12], that allows programmers to determine the quantitative reliability of a program. In the Rely language, quantitative reliability can be specified for function results; for example, in int$< 0.99 * R(arg) >$ FUNC(int arg, int x) code, 0.99*R(arg) specifies that the reliability of return value of FUNC must be at least 99% of reliability of arg when the function was invoked. Rely programs can run on a processor with potentially unreliable memory and unreliable logical/arithmetic operations. The programmer can specify that a variable can be stored in unreliable memory and/or an unreliable operation can be performed on the variables. Integrity of memory access and control flow are maintained by ensuring reliable computations for the corresponding data. By running both error-tolerant programs and checkable programs (those for which an efficient checker can be used for dynamically verifying result correctness), they show that Rely allows determination of integrity (i.e., correctness of execution and validity of results) and QoR of the programs.

It was noted in [19] that, for several computation intensive applications, although finding a solution may incur high overheads, checking the solution quality may be easy. They proposed decoupling error analysis of approximate accelerators from application quality analysis by using application specific metrics called light weight checks (LWCs). LWCs are directly integrated into the application, which enables compatibility with any ACT. By virtue of being lightweight, LWCs can be used dynamically for analyzing and adapting application-level errors. Only when testing with LWCs indicates quality loss below a set standard, exact computation needs to be performed for recovery. Otherwise, the approximation is considered acceptable. This saves energy without compromising reliability. Their approach guarantees bounding worst-case error and obviates the need of statically designed error models. A quality control technique for inexact accelerator based platforms was proposed in [24]. They note that an accelerator may not always provide acceptable results; thus, blindly invoking the accelerator in all cases will lead to quality loss and waste of energy and execution time. They proposed a predictor that guessed whether the invocation of accelerator will lead to quality degradation below a threshold. If yes, their technique instead invokes the precise code. A novel product program construction for differential assertion checking is presented in [23] that permits procedural programs, and allows leveraging off-the-shelf program verifiers and invariant inference engines. This work shows that mutual summaries naturally express many relaxed specifications for approximations, and SMT-based checking and invariant inference can substantially automate the verification of such specifications and provides us with an insight that assertions can be used as a proper metric for approximation quality.

To the best of our knowledge, our proposed model is the first work of its kind. We use automated techniques to identify approximable regions of digital design code and measure the resulting correctness compromise using assertions. This is the major limelight of this dissertation.

Chapter 3

Statement Identification for Approximate Computing

In this chapter, we formally explain our approaches for identifying statements which can be suitable for approximations. In the subsequent chapter, we elaborate the next two steps of our complete methodology.

3.1 Definition

The main idea of the statement identification step is to segregate the statements of a digital design, into two parts.

- Possibly Approximable Statements : These are statements which can be executed in a manner so that they produce a *good enough* result.
- Sensitive Statements : These are the statements which are extremely *important* to the digital design. We can say that these statements needs to be executed with exact accuracy.

There are two approaches we have pursued to identify possibly approximable statements.

- Dynamic Method : The dynamic method uses coverage to separate between the statements. Low covered statements are judged to be possibly approximable whereas highly covered statements are marked as sensitive statements.
- Static Method : We use the program's dependency graph to find out the number of output variables each statement affects. Statement which affect lower number of

output variables are marked as potentially approximable statements, while the rest are marked as sensitive statements.

We explain in detail both the methods in the discussion below.

3.2 Dynamic Method

Detailed Methodology

The dynamic approach is based on the calculation of statement coverage $C_{statement}$, of the Verilog code based on a large number of given test cases, similar to the approach used in [10].

Statement Coverage : Statement coverage, also known as line coverage is the easiest understandable type of coverage. From N lines of code and according to the applied stimulus how many statements (lines) are covered in the simulation is measured by statement coverage. If a DUT is 10 lines long and 8 lines out of them were exercised in a test run, then the DUT has line coverage of 80%. Line coverage includes continuous assignment statements, Individual procedural statements, Procedural statement blocks, Procedural statement block types, Conditional statement and Branches for conditional statements.

For a particular test case, not all statements of the design are executed. For a large set of test cases, a particular statement in the design will be executed for a subset of given test set. To understand this approach consider Figure 3.1. There are two paths based on the condition *C1*. One which contains statement *S2* and another which contains statement *S3*. *S1* is common to both the paths. For say around t number of test cases, it is obvious that *S1* will be executed for all. However *S2* will be executed for some of the test cases among t, and the rest shall cause *S3* to execute.

We can intuitively see that the set of statements can be segregated into two sets. One, Cov_{high} with a very high coverage value, greater than a given threshold t_c, which can be set depending upon the level of approximation we want to perform. This signifies that majority of the test cases had the statement on their execution path. We claim these sentences to be very important to the design, which need to be executed with exact precision and thus, are marked as sensitive. Second, Cov_{low} the rest of the statements which have a low coverage value, as they are executed for a very less number of test cases. We claim these statements to be potentially approximable statements, which can be approximated to reduce the gate count or power consumption of the design. Since they are not executed very frequently, we can deal with their precision being slightly inexact, having significant resource optimization in return. The value of threshold t_c can be decided by the user. It acts as the controlling parameter of the level of approximation we want to perform.

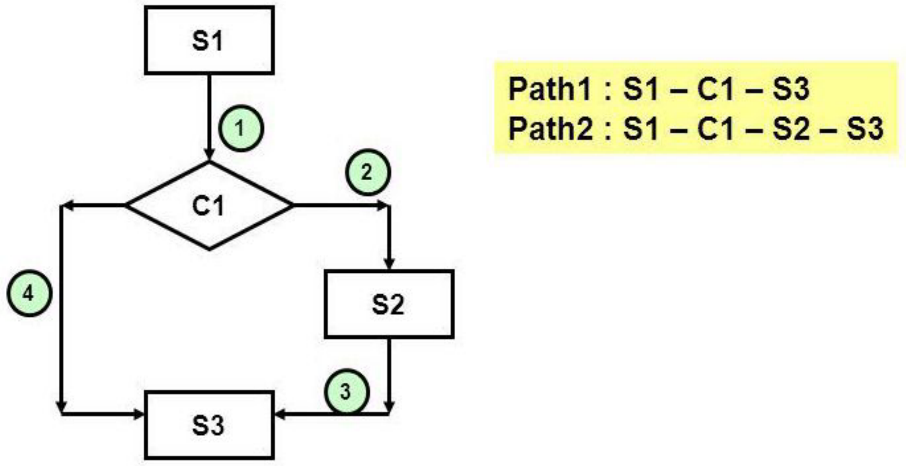

Figure 3.1: Example of Statement Coverage

Example 3.1 *If t_c is set to be 20%, it means we shall consider only those statements in our next step, who have a coverage score of less than 20%,i.e they have been executed for less than 20% of the test cases.*

Figure 3.2 shows the steps of the above approach.

Formally, the dynamic approach can be stated as follows: We define **S** to be the set of all statements of the given digital design code and $C_{statement_s}$, the coverage score of a statement $s \in \mathbf{S}$. s is used to segregate statements as:

$$s \in \begin{cases} Cov_{high}, & \text{if } C_{statement_s} > t_c \\ Cov_{low}, & \text{otherwise} \end{cases} \tag{3.1}$$

At the end of the dynamic approach, we can generate a matrix A of the form *Statements* x *Test Cases*. A 1 in the position a_{ij} signifies that statement i has been executed by test case j, 0 signifies otherwise. This will be used in the upcoming sections for the purpose of ranked aggregation.

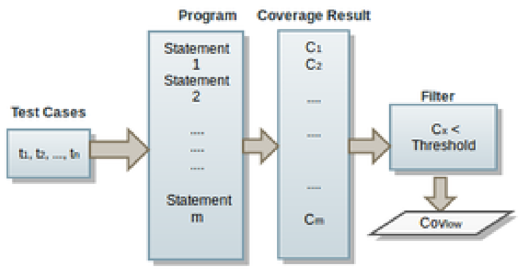

Figure 3.2: Dynamic method of Statement Identification

Example 3.2 *The matrix below shows the matrix A for* 30 *sentences against* 5 *test cases. Statement 1 is executed for all five of the test cases, whereas statement 2 is executed for only two of the test cases, t_2 and t_5.*

$$
A = \begin{array}{c} \\ s_1 \\ s_2 \\ \\ s_{30} \end{array}
\begin{array}{c} \begin{array}{ccccc} t_1 & t_2 & t_3 & t_4 & t_5 \end{array} \\
\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & 1 & 0 & 1 \end{bmatrix} \end{array}
$$

3.3 Static Approach for Statement Identification

3.3.1 Reason for a Different Approach

The dynamic approach of statement identification, brings forward the old question of *Are there enough test cases?*. It is quite possible, that for a larger number of test cases, some of the statements, which had low coverage score previously might end up having a high coverage score. Also, to get a large number of test cases, one has to use a random test generator, which without considering the design shall provide us test cases. It is quite possible that, the test cases may be focused on some particular branches more than the rest. So it is evident that blindly increasing the number of test cases is not enough. However the dynamic approach gives us a rough acceptance, that our intuition of segregating statements in a Verilog design is possible and its initial results were quite supportive as well.

As a second method, we decided to look at an approach that does not depend on the behavior of the test cases, or the number of test cases. We decided to utilize the program structure to shortlist our statements, rather than depend on any other separate inputs. We present the *static* approach, which utilizes the dependency graph for a module to shortlist the statements, based on the number of output variables it affects.

3.3.2 Approach

As mentioned in the previous section, we now aim to develop a statement identification approach, which shall use the program structure to identify statements possible for approximations. We present the static approach which uses the concept of the effect a statement has on a particular output, to mark statements which are *sensitive* and *possibly approximable.*

In a Verilog program, each module has multiple outputs. Each statement in the program, does not take part in the decision for the value of every output variable. This is the major motivation behind the static approach, where we aim to find out the number of output variables affected by a particular statement, and then use that score to segregate between the statements. To understand this concept let us look at the following example.

Example 3.3 *Consider four statements s_1, s_2, s_3 and s_4 in a module. There are three outputs in the module, o_1, o_2 and o_3.*

The value of o_1 is affected by s_1 and s_3.

The value of o_2 is affected by s_1, s_3 and s_4.

The value of o_3 is affected by s_2.

From the above example, we see that there is a score on the basis of which we can segregate the statements. Clearly, statement s_2 has a lot less affect on the total functionality of the module. Similar to the dynamic approach, we can separate the statements into two categories. One, in which the statements affect a large number of output variables, i.e, above a given threshold, and the other in which the statements affects a less number of output variables.

Formally, the static approach can be stated as follows: We define $\mathbf{S}$ to be the set of all statements of the function and $OutScore_{statement_s}$ to be the score of a statement, which specifies the number of output variables it affects $s \in \mathbf{S}$. t_{os} is the given threshold. s is segregated as:

$$s \in \begin{cases} OutScore_{high}, & \text{if } OutScore_{statement_s} > t_{os} \\ OutScore_{low}, & \text{otherwise} \end{cases} \tag{3.2}$$

Similar to the dynamic approach, we claim that the statements in the set $OutScore_{high}$, are very important to the program, as they affect a large number of output variables. These statements needs to be performed possibly with exact accuracy, and are thus said to be *sensitive.* For the statements in $OutScore_{low}$, the program can deal with their results being a little inexact from the correct value, as they affect very less number of output variables. These statements are claimed as *possibly approximable.* The value of threshold t_{os} can be decided by the user. It acts as the controlling parameter of the level of approximation we want to perform.

Example 3.4 *If t_{os} is set to be 40%, it means we shall consider only those statements in our next step, who influence the value of less than 40% of the total output variables.*

At the end of the static approach, we can generate a matrix B of the form *Statements* x *Output Variables.* A 1 in the position b_{ij} signifies that statement i has affected output variable j, 0 signifies otherwise. This will be used in the upcoming sections for the purpose of ranked aggregation.

Example 3.5 *The matrix below shows the matrix B for* 30 *sentences against* 4 *output variables. Statement 2 is responsible for the value of* 4 *of the output variables, whereas statement* 30 *affects only two output variables, o_1 and o_3.*

$$\text{B} = \begin{array}{c} \\ s_1 \\ s_2 \\ \\ s_{30} \end{array} \begin{array}{c} \begin{array}{cccc} o_1 & o_2 & o_3 & o_4 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

```verilog
module farm_control(clk, car_present, enable_farm, short_timer, long_timer, farm_light,
     farm_start_timer, enable_hwy);

input clk;
.
.
output farm_light;
output farm_start_timer;
output enable_hwy;
.
.
initial farm_light = RED; /*farm_start_timer,
               enable_hwy, farm_light*/

assign farm_start_timer = (((farm_light == GREEN) && ((car_present == NO) || long_timer)) ||
     (farm_light == RED) && enable_farm); //farm_start_timer

assign enable_hwy = ((farm_light == YELLOW) && short_timer);   /*enable_hwy*/

always @(posedge clk) begin
  case (farm_light)             /*farm_start_timer,
                       enable_hwy, farm_light*/
  GREEN:
    if ((car_present == NO) || long_timer)      /*farm_start_timer, enable_hwy, farm_light*/

         farm_light = YELLOW;        /*farm_start_timer,
                       enable_hwy, farm_light*/
  YELLOW:
    if (short_timer)            /*farm_start_timer,
                       enable_hwy, farm_light*/
         farm_light = RED;           /*farm_start_timer,
                       enable_hwy, farm_light*/
  RED:
    if (enable_farm)            /*farm_start_timer,
                       enable_hwy, farm_light*/
         farm_light = GREEN;         /*farm_start_timer,
                       enable_hwy, farm_light*/
  endcase
end
always@(posedge clk)
begin
 gm0 : assert property(!((farm_light == GREEN) && (hwy_light == GREEN)));
 gm1 : assert property (((car_present == YES) ##0 (!(farm_light == GREEN))[*0:$]));
 gm2 : assert property ((hwy_light == GREEN)[*1:$]);
end
endmodule
```

Figure 3.3: Working example of a module of traffic light controller

3.3.3 Motivating Example

To implement the static approach, we implement dependency slicing, which traverses the control dependency flow graph of the module, and identifies statements which changes the values of the variables. A number of dependencies arise in this approach, namely transitive, conditional and direct dependencies which need to be taken care of. We present an example which presents a clear explanation of the static approach for statement identification.

Example 3.6 *Consider Figure 3.3, which shows a simple Verilog design module of a light controller, which controls the light on a crossing (details not shown in figure), depending on the various inputs received (e.g., car present, timer duration, etc). Our algorithm begins by examining each output variable in turn. The variables in green beside the statements show that the statement modifies this particular variable. Consider the output* $farm\_light$ *which is modified in three statements,* 24, 29, *and* 34. *Each of these is control dependent on an if condition. The variables in the if conditions on line* 22, 27, *and* 32 *are* $enable\_farm, short\_timer, car\_present$ *and* $long\_timer$. *All these statements lie nested under the case statement at line* 19 *and are therefore, control dependent on it. Further, statement* 11 *belongs to the dependency slice of* $farm\_light$ *since it assigns a value to it. For enable_hwy, statement* 16 *belongs to its dependency slice due to the direct data dependency. The variables on the right hand side of the assign statement are* $farm\_light$ *and* $short\_timer$. *The concurrency semantics of Verilog language makes the other two outputs* $farm\_start\_timer$, *enable_hwy affected by statements 22 to 34 as well. In particular, enable_hwy is assigned at statement 16, which checks for a condition on* $farm\_light$. *Hence, statements 11, 19, 22, 24, 27, 29, 32, and 34 which belong to the dependency slice of* $farm\_light$ *end up indirectly affecting the logic of computation for enable_hwy. A similar reasoning holds for the other output variable as well.* ■

3.3.4 Algorithm for Static Approach

The dependency slicing [11][26] step takes in the following input: (a) The program P and (b) A list of module output variables. The output of the method is a list of statements $\phi \subseteq P$ that influence the computation of the output variables. The dependency slice, Dep_i, for variable i computes a chain of static data and control dependencies. In the slicing algorithm, for each output variable *out*, we traverse the program and mark the data and control dependencies, both the direct ones and the transitive ones that propagate through other variables. In other words, we end up computing the transitive closure of the data and control dependencies for the variable *out*. Finally, we mark all such statements as influencing the variable *out*. The slicing is done on the control and data flow graph (CDFG) [11] constructed from the module code. Algorithm 1 presents the dependency slice computation strategy. We take each module output *out*, and compute the dependency slice chain starting from the first unmarked statement where *out* is assigned. The algorithm terminates when we reach a fix point, in other words, no new statements are added and no new variables are encountered.

Algorithm 1: Dependency Slice Computation

```
Input: $D$ : Design to be approximated
$Out$ : Outputs of $D$
$S$ : Set of all statements of $D$
Output: $Dep_{Out}$ : Dependency slice of module outputs.
begin
    for all $out \in Out$ do
        for all unmarked $s \in S$ do
            if out modified in $s$ then
                Add $s$ to $Dep_{out}$ and mark $s$
                $B[s][out] \longleftarrow 1$
                for for all variables $x_1, x_2, ..x_p$ in the RHS of $s$ do
                    Compute dependency slice $Dep_{x_i}$
                if $s$ depends on conditional statement $c$ then
                    for all variables $z_1, z_2, ..z_q$ in the RHS of $s$ do
                        Compute dependency slice $Dep_{z_i}$
            else
                $B[s][out] \longleftarrow 0$
        Add $Dep_{out}$ to $Dep_{Out}$
```

3.4 Merging both the Approaches

In the earlier sections we have introduced the dynamic and static approach for statement identification suitable for approximate computing. Each method segregates statements based on a particular behavior of the program. The dynamic method uses the fact that not all statements are executed with the same frequency, while the static approach uses the behavior of the verilog design that not all statements affect the value of same number of output variables. Both the approaches have some drawbacks. The limitations of the dynamic approach is mentioned in Section 3.3. The static approach faces with the limitation of dealing with a high complexity during the calculation of the dependency slicing. For our best interests, use of both the approaches should aid us in the best identification of statements suitable for approximation. This is due to the fact because both the methods are so different in their underlying philosophy, they force the enforce that the really approximable sentences are selected with greater priority.

In both the approaches, we generated a matrix at the end. The dynamic method generated a matrix *Statements x Test Cases*, while the static approach generated a matrix of the form *Statements x Output Variables.* From both the matrices, we now get a ranking of the statements of a module.

Example 3.7 *Consider two matrices of the form described above. Matrix A is from the dynamic approach while Matrix B is from the static approach. It shows 5 statements against 3 output variables and 5 test cases.*

$$A = \begin{array}{c} \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{array} \begin{array}{c} \begin{array}{ccccc} t_1 & t_2 & t_3 & t_4 & t_5 \end{array} \\ \begin{bmatrix} 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{bmatrix} \end{array}$$

$$B = \begin{array}{c} \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{array} \begin{array}{c} \begin{array}{ccc} o_1 & o_2 & o_3 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix} \end{array}$$

The ranking for the 5 statements generated from matrix A and B are R1 and R2 respectively, *where the score of the statement is included beside the statement number in braces.*

$$R1 = \begin{bmatrix} s_4(5) \\ s_5(4) \\ s_2(3) \\ s_1(2) \\ s_3(2) \end{bmatrix} R2 = \begin{bmatrix} s_1(3) \\ s_3(2) \\ s_5(2) \\ s_2(1) \\ s_4(1) \end{bmatrix}$$

Now that we have two rankings of the statements, our final aim is to get a common ranking of the statements based on both. We apply Borda's method of ranked aggregation [16] to achieve this.

Definition 3.1 ***Ranked List Aggregation :*** *Given two full lists, sorted in the same order τ_1 and τ_2 generated from matrix A and B respectively, then for each $s \in \boldsymbol{S}$ and list τ_i, Borda's method first assigns a score $B_i(c)$ =the total number of candidates ranked below c in τ_i, and then the total Borda score $B(c)$ is defined as $\sum_{i=1}^{k} B_i(c)$. The candidates are then sorted in decreasing order of total Borda score [17].*

Example 3.8 *For the ranked lists R1 and R2 in Example 3.7, the Borda's score for the statements are given in the form Score(R1), Score(R2) in the following matrix.*

$$\begin{array}{c} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{array} \begin{bmatrix} 1,4 \\ 2,1 \\ 0,3 \\ 4,0 \\ 3,2 \end{bmatrix}$$

The final ranking of the 5 statements is given in the matrix F, where the final Borda score is shown beside the statements,

$$F = \begin{bmatrix} s_1(5) \\ s_5(5) \\ s_4(4) \\ s_2(3) \\ s_3(3) \end{bmatrix}$$

Chapter 4

Approximation Insertion and Correctness Compromise Quantification

In this chapter, we discuss the next step of the work flow. After statement identification, we now have a set of statements which are *possibly approximable.* However we need to quantify the amount of compromise we are making while approximating these statements and the gain we shall acquire as a result of the approximation. The next part of the problem can thus be broken up into two segments.

- *Approximation Insertion :* In this step we enter random approximations (errors) into the statements selected in the previous step.
- *Correctness Compromise Quantification :* After the approximation has been done, we measure the amount of deviation that has occurred from the exact solution, and also the gain in power, circuit area that can be achieved.

In the upcoming sections we describe the logic behind random approximation insertion, how it is used in handshake with correctness compromise quantification. We explain the use of assertions as a correctness metric and then show the different resource gains we have taken into consideration. We also lay down the foundation of the problem which selects the list of possible statements suitable for approximation. This selection is done based on quantified results, which makes the selection all the more stronger.

4.1 Random Approximation Insertion

The random approximation(error) insertion step, aims to crudely approximate the statements. This is a standard approach taken to identify approximable parts of the code. We have based this approach based on [13][14]. The idea is to reduce computation of the statement, so that there is a possible gain in resource utilization. As an example, we use loop perforation to trim down the number of iterations for which a loop runs, condition modification (e.g., replacing the condition of a conditional if/case statement with simpler conditions or constants), modifying assignment statements with random values, modifying n-bit arithmetic operations by truncating the number of bits, etc.

A snapshot of some examples of random approximations that have been applied are given in Table 4.1.

Original Statement	**Possible Modification**	**Description**
if(x)	if(1)	If condition to be always true
assign x = b[2:0], c[3:0]	assign x = b[2:0], c[3:2]	Making the last two bits zero
for(..)	Loop Perforation	Modify the loop to execute in reduced count
assign x = (a & b)..(z & d) & (a |z)	assign x = (a & b)..(z & d)	Drop part of a large computation

Table 4.1: Snapshot of possible modifications

A lot of these approximations have been stated in the literature for approximating computing. Some examples are [10][21][30].

The step of random error insertion is used interchangeably with the correctness and compromise measure step. This can be understood from the fact that once a statement s has been transformed to s', we need to quantify the amount by which the correct result has deviated.

4.2 Correctness and Compromise Measure

Use of Assertions as a Metric

In the previous section we have discussed about random approximation insertion in a statement. The next step is to find out how much the approximation has affected the output or the correctness of the program. We propose the use of assertions as the metric, following the approach in [23] to judge the amount of compromise we are making in the correctness of the program while inserting approximation. Modern designs have a large number of assertions, which specify the way a design should behave.

We have, for every design a set of assertions *Assert* along with their expected valuation (*true/false*) on execution for the original design. An assertion is typically evaluated by a model checker considering all possible feasible execution paths of a design. Thus, introduction of an approximate transformation at a statement s may or may not change the truth value (*true* on original, $false$ on modified or vice versa) of an assertion, depending on whether the transformation affects the truth value computation of the assertion. Based on this idea, for each statement in Res_{pos} a candidate approximation transformation is introduced and the number of assertions changing truth value is measured. For a particular statement s in Res_{pos}, let α_s be the number of assertions changing truth value on transforming s to s'. We thus can quantitatively measure the amount of correctness of the program we have compromised.

Example 4.1 *Consider the example in Figure 3.3. Three assertions are provided at the end of the module. Statements* 40 *to* 42 *in the design code in Figure 3.3 contain three assert statements. The approximation transformation done to statement* 14 *leads to the violation of assertion gm2. On the other hand, for statement* 16 *which affects only one output variable as found out by the static approach, if we discard the condition on the right and assign a value true as a candidate approximate transformation, it does not alter the truth value of any assertion.*

Thus for every statement that had been selected as potentially approximable, we now have a score of the number of assertions that have changed their state, i.e., the amount of correctness that has been compromised due to an approximation that has been inserted. As mentioned previously this generates a ranked list of the statements, where the statement at the top has caused the lowest amount of change in the number of assertions. For every statement there can be multiple approximations. We deal with this at the end of this chapter.

Resource Utilization Gain

Once we have a measure of the amount of correctness that we have compromised, we need to give a measure of the amount of resource gain that we have achieved due to the approximation. For this, we have considered simulating the design to calculate the power usage reduction and the circuit area reduction due to the approximation inserted. For every transformation, we thus have some gain in the resource utilization in the form of area reduction, Δ_s or reduced power consumption, ω_s. The important fact to note here is that these measures are an estimate of the amount of resource gain we can achieve based on the amount of approximation we perform. As an example, the power reduction is 6.17%, while the area reduction is 5.36%, for statement 16, when we apply the approximation described in the previous step.

For the two metrics, we have now further have two ranked lists of the set of possibly approximable statements, one based on the power gain and the other based on the decrease in circuit area. In the first list, the statement at the top has the largest amount of power gain, while in the second list the statement at the top has the highest decrease in circuit area.

Algorithm 2: Resource Utilization Measure Algorithm

```
Input: $Res_{pos}$ : generated set of statements which are possibly approximable
$Assert$ : given set of assertions
$Approx$ : A given set of possible errors
Output: $\forall s \in Res_{pos}, < \omega_s, \Delta_s > \leftarrow$Resource gain, $< \alpha_s > \leftarrow$ Number of assertions
        changing
begin
    for all $s \in Res_{pos}$ do
        for each candidate approximation $x$ for $s$ do
            Apply $x$ to $s$, converting $s$ to $s'$
            Execute the program and fire $Assert$; $\alpha_s \longleftarrow$Number of assertions flipping
            $\omega_s \longleftarrow$Gain in power
            $\Delta_s \longleftarrow$Gain in area
```

4.3 Final Form of the Problem

We now have generated for every statement s in Res_{pos}, a tuple of the form $< \omega_s, \alpha_s, \Delta_s >$. Thus we have three separate ranked lists, for the set of possibly approximable statements. The first one is arranged for the % of assertions flipping in ascending order. The second and third lists are arranged according to the % of power gained due to approximation and % of circuit area decreased due to the approximation. Both of these lists are arranged in decreasing order of the values.

Our aim is to find statements, approximating which leads to the lowest number of assertions changing state, the highest amount of power gain and the highest amount of circuit area decrease. It is possible that a statement which has the highest power gain may not be the first rank holder in the other two ranked lists. This has the flavor of a multi objective optimization problem. The problem is modeled as a ranked list aggregation problem [17], where we aim to select the best statements which shall give us the maximum optimized value in all the three metrics. We present the optimization problem and its possible solutions in the next chapter.

The problem can become more complex when we consider the fact that there can be multiple approximations possible for a single statement. Thus we also have to select which approximation to select for each statement along with the earlier selection criteria. This added constraint increases the complexity of the problem, and we aim to provide suitable heuristics to overcome this. As an example, let us consider the following situation.

We have three sentences s_1, s_2, s_3. Let the set of possible approximations be a_1, a_2, a_3. In the bipartite graph shown in Figure 4.1, a line between a statement node and an approximation node shows that the particular approximation is applicable to the statement.

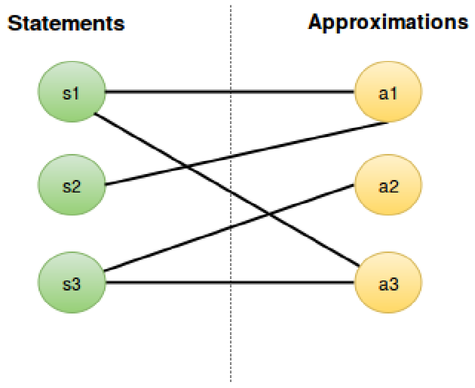

Figure 4.1: Bipartite Graph relationship between statements and approximations

Thus one possible combination which can be applied is $s_1(a_1)$, $s_2(a_1)$, $s_3(a_3)$, where $s_i(a_j)$

means that approximation a_j has been applied to statement s_i, which will have a different value of the tuple consisting of the three correctness and compromise values, from the rest of the combinations. Our problem boils down to selecting the best possible combination among all the possible combinations that can be possible. We deal with the different variations of this problem in the next chapter.

Chapter 5

Statement and Approximation Selection based on Multiple Optimization Criteria

As discussed in the earlier chapter, we are presented with three ranked lists of the set of all possible combinations of possibly approximable statements and the possible approximations. The ranked lists are based on increasing order of the % of assertions flipping, decreasing order of % of power reduction and decreasing order of % of circuit area reduction. This is based on the fact that when one approximation is applied to each statement, the combination of all the approximated statements in the module shall cause certain change in correctness and shall cause some resource optimization. We present below optimization formulations for the problem and scalable heuristics.

5.1 An Integer Linear Programming formulation for Ranked List Aggregation

The motivation behind this method is to generate an aggregate ranked list that minimizes the number of pairwise disagreements between client pairs between the individual rank lists. Intuitively, if a statement S_i is ranked before a statement S_j in most of the individual rank lists, the aggregate list should reflect this.

Let the multi set of the rankings be denoted by Σ. Each ranked list is represented by $\sigma(1)....\sigma(n)$, where $\sigma(i)$ represents the candidate with rank i. Note that $\sigma^{-1}(i)$ is the rank of candidate i, where σ^{-1} denotes the inverse of σ. Let there be m ranked lists and the set of candidates be $\{1, ..., n\}$ denoted as $[n]$.

In *distance-based* rank aggregation, the goal is to find a ranking, called the *aggregate ranking*, that is as "close" as possible to all the votes simultaneously. Closeness is measured via a chosen distance function over $\mathbb{S}_n$. For a given distance d, the aggregate ranking π is formally evaluated to according to

$$\pi^* = \arg\min_{\pi \in \mathbb{S}_n} \sum_{\sigma \in \Sigma} d(\pi, \sigma). \tag{5.1}$$

We have used Kendall distance as our distance measure. The Kendall distance between two permutations π and σ, denoted by $d_K(\pi, \sigma)$ is the number of disagreements between π and σ, i.e., the number of ordered pairs (i, j) such that π ranks i higher than j, and σ ranks j higher than i. Formally, the distance may be defined as

$$d_K(\pi, \sigma) = |\{(i, j) : \pi^{-1}(i) < \pi^{-1}(j), \sigma^{-1}(j) < \sigma^{-1}(i)\}|$$

The solution of (5.2) for the Kendall distance is known as the *Kemeny aggregate.*

For $\sigma \in \mathbb{S}_n$, and $i, j \in [n]$, let

$$\sigma_{ij} \in \begin{cases} 1, & \text{if } \sigma^{-1}(i) < \sigma^{-1}(j) \\ 0, & \text{otherwise} \end{cases} \tag{5.2}$$

Let P be the set of points $x = (x_{ij})$ satisfying,

$$x_{ij} + x_{ji} = 1, \quad \text{for distinct } i, j \in [n] \tag{5.3}$$

$$x_{ij} + x_{jk} + x_{ki} \leqslant 2, \quad \text{for distinct } i, j, k \in [n] \tag{5.4}$$

$$x_{ij} \in \{0, 1\}, \quad \text{for distinct } i, j \in [n] \tag{5.5}$$

$$x_{ii} = 0, \quad \text{for distinct } i \in [n] \tag{5.6}$$

The objective of the Kemeny rank aggregation method is to minimize the number of disagreements with the individual rankings. The Kemeny aggregate is thus a solution of the following integer program,

$$\begin{aligned} \min_x \sum_{\sigma \in \Sigma} \sum_{i,j} x_{ij} \sigma_{ji} \\ \text{subject to} \qquad x_{ij} \in P \end{aligned} \tag{5.7}$$

Constraint 5.3 expresses for any statement pair, S_i , S_j , one of them has to be ranked ahead of the other, thus both the binary variables x_{ij} and x_{ji} cannot be 0 or 1. The second constraint, 5.4 is the transitivity constraint between statement triplets. Unless this constraint is in place, the aggregate ranking may assign values to the binary variables with a cyclic majority, i.e the ranks may be assigned as, $\sigma^{-1}(i)$ ahead of $\sigma^{-1}(j)$, $\sigma^{-1}(j)$ ahead of $\sigma^{-1}(k)$, and $\sigma^{-1}(k)$ ahead of $\sigma^{-1}(i)$. The third constraint, 5.5 expresses the fact that the x_i and x_j variables are binary. The final constraint enforces the fact no statement can be ranked ahead of itself. The output of the optimization is a value (0 / 1) for each binary variable x_{ij}, that leads to the minimum value of the objective, subject to the constraints.

5.2 Borda's Score

One simple way to complete the ranked list aggregation problem is to use Borda's score [16]. This is a score based rank aggregation for which we have generated results in the coming sections.

Definition 5.1 ***Borda's Method for Ranked List Aggregation :*** *Given n full lists τ_1 .... τ_n, then for each $c \in \boldsymbol{C}$, where **C** is the set of all possible combinations of approximations and statements, and list τ_i, Borda's method first assigns a score $B_i(c)$ =the total number of candidates ranked below c in τ_i, and then the total Borda score $B(c)$ is defined as $\sum_{i=1}^{k} B_i(c)$. The candidates are then sorted in decreasing/ increasing order of total Borda score [17].*

5.3 Working

Statement ID	**Approximations Applicable**
s_1	a_1, a_3, a_4, a_5
s_2	a_1, a_6
s_3	a_1, a_3, a_4
s_4	a_1, a_3, a_4

Table 5.1: List of Approximations applicable

We show the working of the Integer Linear program formulated earlier and Borda's method. We consider the *pci_rst_in* module of the PCI verilog code [3]. Four statements are selected from the module based on the statement identification approaches described earlier. Let the sentences be names s_1 to s_4. There are 6 possible approximations that can be applied. Let the approximations be numbered as a_1 to a_6. Table 5.1 shows the approximations that can be applied to each statement of the module. a_1 signifies no approximation, i.e., the statement is left in its original form.

The total possible combinations that can arise, as we see from the above table is 72. We show the top ten results that arise from both the methods, i.e Borda's score and Kemeny Aggregation. The symbol $s_i(a_j)$ means that approximation a_j has been applied to s_i.

Result using Borda's method

Combination	**% of Approximations Flipping**	**% of Power Gain**	**% of Circuit Area Gain**
$s_1(a_4)$	16.67	66.695	26.221
$s_1(a_3)$	33.33	66.695	30.892
$s_1(a_5)s_2(a_6)$	16.67	33.31	23.567
$s_1(a_4)s_4(a_3)$	16.67	66.695	26.221
$s_1(a_4)s_2(a_6)$	16.67	66.695	30.892
$s_1(a_5)s_3(a_3)$	16.67	66.695	30.892
$s_1(a_5)s_2(a_6)s_4(a_3)$	33.33	33.39	21.338
$s_1(a_5)s_2(a_6)s_4(a_3)$	33.33	66.695	30.892
$s_1(a_5)s_2(a_6)s_3(a_3)s_4(a_4)$	50	66.695	28.662
$s_1(a_5)s_2(a_6)s_3(a_4)$	33.33	66.695	30.892

Table 5.2: Top Ten Combinations using Borda Score

Result using Kemeny Aggregate

Combination	**% of Approximations Flipping**	**% of Power Gain**	**% of Circuit Area Gain**
$s_1(a_4)$	16.67	66.695	26.221
$s_1(a_3)$	33.33	66.695	30.892
$s_1(a_4)s_4(a_4)$	16.67	66.695	19.002
$s_1(a_5)s_2(a_6)$	16.67	33.31	23.567
$s_1(a_5)s_3(a_3)$	16.67	66.695	30.892
$s_1(a_5)s_2(a_6)s_4(a_3)$	33.33	66.695	30.892
$s_1(a_5)s_2(a_6)s_3(a_4)$	33.33	66.695	30.892
$s_1(a_3)s_2(a_6)$	33.33	66.695	30.892
$s_1(a_5)s_2(a_6)s_4(a_3)$	33.33	33.39	21.338
$s_1(a_3)s_4(a_4)$	50	66.695	26.221

Table 5.3: Top Ten Combinations using Kemeny Aggregate

Chapter 6

Implementation and Results

6.1 Implementation

Figure 6.1 shows an architectural overview of our framework. The slice generator is imple-

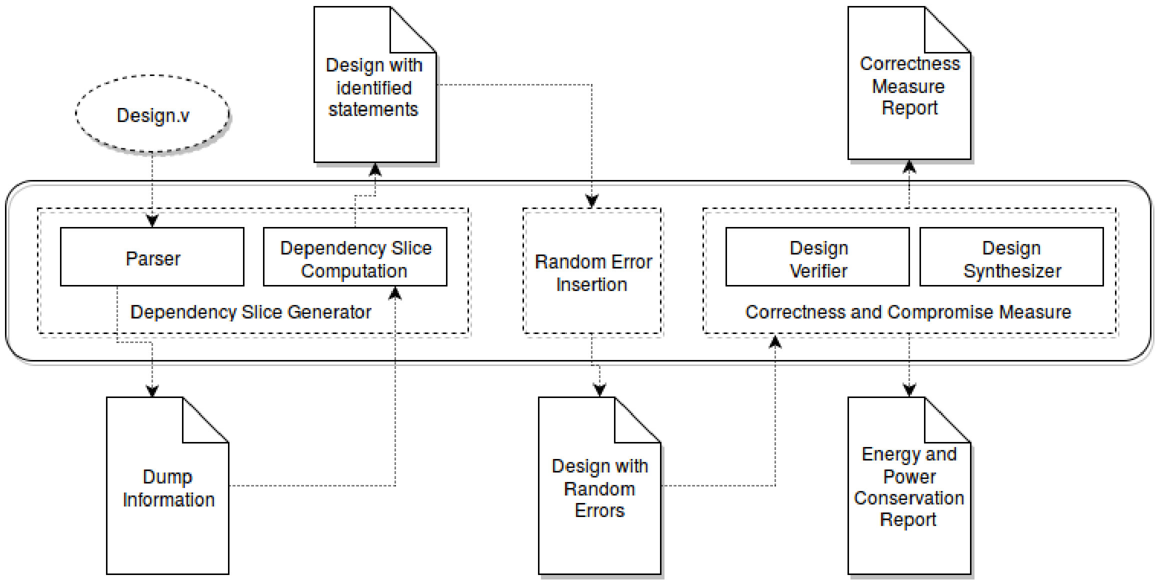
Design.v
Design with identified statements
Correctness Measure Report
Parser
Dependency Slice Computation
Dependency Slice Generator
Random Error Insertion
Design Verifier
Design Synthesizer
Correctness and Compromise Measure
Dump Information
Design with Random Errors
Energy and Power Conservation Report

Figure 6.1: System Architecture

mented in two steps. First, the Verilog design is parsed using a commercial Verilog analyzer, through which an information dump file is generated. The dump file contains details about variable modification, conditional statements and information about the output variables of the module. The information dump is then fed into the dependency slice computation program, which makes use of the provided information to list down each statement's influence on the output variables. For each statement which affects output variables below a certain threshold, determined by the user, we say they can be potentially resilient. In our experiments, we have taken the threshold to be 50%.

Approximation insertion is presently done manually and we identify the change in the program correctness, by calculating the number of design properties changing their original value. We have used SystemVerilog Assertions [18] as the assertion specification language and the formal property checker *Magellan* [1]. The assertions for the experiment setup have been generated using Goldmine [31]. Along with the correctness measure, the reductions in power and area area also calculated, using the *Design Compiler* tool [22] from Synopsys. We considered the total sequential and combinational switching power in our experimental setup.

Once the tuples are generated for a design, consisting of information about the gain-correctness trade off for each statement for each transformation, we use ranked list aggregation [16] to merge the score of the three measures, number of assertions changing state, power reduction and area reduction. We implement Borda's method [16], and the integer linear programing model presented in Chapter 5, to implement the aggregation. We first create three ordered lists from our earlier generated approximation transformation information. List 1 contains an ordered listing of the approximations on the resilient statements based on the power gain achieved, from highest to lowest. Similarly, list 2 contains a similar ordering based on area gain. List 3 orders the approximations based on the number of assertions changing truth value, from lowest (more desirable) to highest (less desirable). Evidently, these three lists will not agree in the usual case, or in other words, the approximation with the best power gain has the best area gain and the lowest assertion change is not typically the case. Hence, choice of the best candidate approximation or even ordering the approximations is not straightforward. We need to therefore aggregate these lists with the approaches mentioned in the previous chapter.

The majority of the results are based on the static approach of statement selection, due to multiple drawbacks of the dynamic approach as discussed earlier. However we have taken into consideration the merging of both the approaches for a few modules, which is described in the next section.

6.2 Evaluation

We applied our framework on three standard Verilog designs with assertions. We describe our experiments below. Results indicate that indeed there are statements which lend themselves to easy approximation with less compromise on correctness, while producing significant gains in resource computation. Our dependency slicing algorithm can efficiently and correctly identify the statements that are suitable for this analysis. Table 6.1 shows the outcome of the implementation of our tool.

6.2.1 Case Study on USB Function Core

We present experimental results of the tool on selected modules from the USB Function Core 2.0 protocol [8] in Verilog. Our framework has been implemented on five modules, *usbf_pa*, *usbf_wb*, *usbf_idma*, *usbf_pd* and *usbf_mem_arb*.

Table 6.1: Evaluation Results of our Framework

Circuit Module	Lines of Code (LOC)	Number of statements selected by the Dependency Slice Generator	Number of Assertions	Average % of Assertions changing state	Average % of Power Reduction	Average % of Area Reduction
USB Function Core						
usbf_pa	240	17	17	7.9585	2.2741	2.6915
usbf_wb	147	20	47	12.234	2.4933	−0.1189
usbf_idma	336	28	45	11.9841	1.0205	1.2382
usbf_pd	270	48	53	6.5252	0.9668	1.6391
usbf_mem_arb	73	12	12	8.3333	4.2358	−5.4488
OR1200						
or1200_du	1278	40	498	4.6976	0.0.0939	−0.0306
PCI IP Core						
pci_rst_in	64	4	6	12.5	25	9.5011
pci_sync_module	57	5	4	30	9.4396	4.251

The columns indicate the lines of code (LOC) in each module, the number of lines selected based on the dependency slice generator's results, the number of assertions describing the properties of each module, average percentage of assertions changing state, power reduction and area reduction. We applied candidate approximations for each resilient statement and recorded the outcome. For all the modules, the average % of assertions changing state is around 7 to 12. On an average, there is power reduction for all the statements due to approximate transformations. Area reduction is also significant for three of the modules, however for two of the modules *usbf_wb* and *usbf_mem_arb*, there is a gain in the average circuit area as well. This suggests that majority of the statements in the two modules *usbf_wb* and *usbf_mem_arb*, do not help in resource optimization. The module *usbf_mem_arb* has the highest gain in power consumption but a significant area compromise. The problem of maximizing this trade off is dealt with using aggregation.

The modified approach of merging the static approach and the dynamic approach has been implemented on the *usbf_pd* module. Figure 6.2 shows a snapshot of the coverage report for the module on 10000 test cases. Each line begins with it's corresponding line number in the Verilog code.

The coverage of line 2470 is 13, i.e., 0.13%, which is a very low coverage score, and thus, is marked as potentially resilient. The coverage report has been generated using Questa Advanced Simulator [5].

```
2469          1          55    always @(posedge clk)
2470          1          13      if(crc16_clr)
2471          1                    crc16_sum <= 16'hffff;
2472                              else
2473          1           2      if(data_valid_d)
2474          1                    crc16_sum <= crc16_out;
```

Figure 6.2: Example of the modified approach

6.2.2 Case Study on OR1200

We tested our approach on the *or*1200*_du* module of the OpenRISC 1200 architecture design [2], whose IP core is implemented in Verilog. Result is shown in Table 6.1. For the module, majority of the statements caused no or very less change to the state of the assertions, suggesting that the statements were quite resilient. Few of the selected statements turned out to be very sensitive, causing the average reduction in power consumption and circuit area to decrease.

6.2.3 Case Study on PCI IP Core

Our framework has been implemented on two modules, *pci_rst_in* and *pci_sync_module* of the PCI IP Core [3], which is a bus bridge device between the WISHBONE SoC bus and the PCI local bus. Table 6.1 shows the results. Module *pci_rst_in*, shows promising results, with a sharp reduction in power consumption and circuit area. Module *pci_sync_module*, however, also has a very high rate of assertions changing state, along with noticeable reduction in both the fields, indicating that the module has more *sensitive* statements than *resilient* statements.

Chapter 7

Conclusion and Future Work

In this work, we have proposed a novel strategy for approximate computing in a digital design. Experimental results show promising improvements in the amount of power gain and reduction in circuit area while compromising extremely less on the program correctness. With approximate computing gaining extreme popularity in varied domains, we believe our work will have interesting applications.

In this work, we have proposed an algorithm for identification of regions suitable for approximation. We have also shown that these statements can function without compromising a lot in terms of correctness and also provide some gain in circuit power and circuit area.

In our future work, we wish to extend our research to two main directions. One one side, we want to work with multiple approximations in the design. We also want to reduce the number of candidates in the ILP for statement selection, by proposing a heuristic solution, as currently we are dealing with all possible combinations of the approximations for all the statements. A separate path that can be followed can be integrating approximate hardware in the statements selected in place of random approximation insertion. The idea also includes providing a probabilistic bound for the same.

Chapter 8

Disseminations out of this work

- S. Mitra, M. Das, A. Banerjee, K. Datta and T. Yi Ho, "A Verification Guided Approach for Selective Program Transformations for Approximate Computing ," under review *Asian Test Symposium (ATS '16)*, 2016.

Bibliography

[1] Magellan. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.227.3881&rep=rep1&type=pdf.

[2] Or1200 openrisc processor. http://opencores.org/openrisc,or1200.

[3] Pci ip core. http://opencores.org/project,pci,home.

[4] Psl. http://www.asic-world.com/verilog/assertions4.html.

[5] Questa advanced simulator. http://www.mentor.com/products/fv/questa/.

[6] Specman. http://www.asic-world.com/specman/.

[7] Synopsys ova. http://www.open-vera.com/.

[8] Usb function core. http://www.opencores.org/cores/usb.

[9] Ansel, J., Wong, Y. L., Chan, C., Olszewski, M., Edelman, A., and Amarasinghe, S. Language and compiler support for auto-tuning variable-accuracy algorithms. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2011), CGO '11, IEEE Computer Society, pp. 85–96.

[10] Baek, W., and Chilimbi, T. M. Green: A framework for supporting energy-conscious programming using controlled approximation. PLDI, pp. 198–209.

[11] Bhattacharjee, D., et al. Evodeb: Debugging evolving hardware designs. VLSID, pp. 481–486.

[12] Carbin, M., et al. Verifying quantitative reliability for programs that execute on unreliable hardware. OOPSLA, pp. 33–52.

[13] Chippa, V. K., et al. Analysis and characterization of inherent application resilience for approximate computing. DAC, pp. 1–9.

[14] Chippa, V. K., et al. Approximate computing: An integrated hardware approach. In *Asilomar Conference on Signals, Systems and Computers* (2013), pp. 111–117.

[15] Cho, H., Leem, L., and Mitra, S. Ersa: Error resilient system architecture for probabilistic applications. *IEEE Trans. on CAD of Integrated Circuits and Systems 31*, 4 (2012), 546–558.

[16] de Borda, J. C. Mémoire sur les élections au scrutin. *Histoire de l'Académie Royale des Sciences* (1784).

[17] Dwork, C., Kumar, R., Naor, M., and Sivakumar, D. Rank aggregation methods for the web. In *Proceedings of the 10th International Conference on World Wide Web* (New York, NY, USA, 2001), WWW '01, ACM, pp. 613–622.

[18] Foster, H. D., et al. *Assertion-based design.* Springer Science & Business Media, 2004.

[19] Grigorian, B., and Reinman, G. Dynamically adaptive and reliable approximate computing using light-weight error analysis. In *Adaptive Hardware and Systems (AHS), 2014 NASA/ESA Conference on* (July 2014), pp. 248–255.

[20] Gupta, et al. Impact: imprecise adders for low-power approximate computing. ISLPED, IEEE Press, pp. 409–414.

[21] Hoffmann, H., et al. Dynamic knobs for responsive power-aware computing. ASPLOS XVI, pp. 199–212.

[22] Kurup, P., and Abbasi, T. *Logic Synthesis Using Synopsys*, 2nd ed. Springer Publishing Company, Incorporated, 2011.

[23] Lahiri, S. K., et al. Automated differential program verification for approximate computing. Tech. rep., Microsoft Research, 2015.

[24] Mahajan, D., Yazdanbakhsh, A., Park, J., Thwaites, B., and Esmaeilzadeh, H. Prediction-based quality control for approximate accelerators.

[25] Park, J., et al. Flexjava: Language support for safe and modular approximate programming. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (2015), pp. 745–757.

[26] Qi, D., et al. Path exploration based on symbolic output. ESEC/FSE, pp. 278–288.

[27] Raha, A., Venkataramani, S., Raghunathan, V., and Raghunathan, A. Quality configurable reduce-and-rank for energy efficient approximate computing. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition* (San Jose, CA, USA, 2015), DATE '15, EDA Consortium, pp. 665–670.

[28] Roy, P., Ray, R., Wang, C., and Wong, W. F. Asac: Automatic sensitivity analysis for approximate computing. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems* (New York, NY, USA, 2014), LCTES '14, ACM, pp. 95–104.

[29] Sampson, A., et al. Enerj: Approximate data types for safe and general low-power computation. PLDI, pp. 164–174.

[30] Sidiroglou-Douskos, et al. Managing performance vs. accuracy trade-offs with loop perforation. ESEC/FSE, pp. 124–134.

[31] Vasudevan, S., et al. Goldmine: Automatic assertion generation using data mining and static analysis. DATE, pp. 626–629.