UNIVERSITY OF CINCINNATI

Date: 27 September 2004

I, Vinod Chamarty **,**

hereby submit this work as part of the requirements for the degree of:

Master of Science

in:

Computer Engineering

It is entitled:

Investigation of an Information Structure to support the elaboration of simultaneous statements in compile-driven mixed-signal simulation.

This work and its defense approved by:

Chair: Dr. Hal Carter

Dr. Philip Wilsey

Dr. Karen Tomko

Investigation of an Information Structure to support the Elaboration of Simultaneous Statements in Compile-driven Mixed-signal Simulation

A thesis submitted to the

Division of Graduate Studies and Research
of
The University of Cincinnati

in
partial fulfillment
of the
requirements for the degree of

Master of Science

in the
Department of Electrical and Computer Engineering
and
Computer Science
of
The College of Engineering

September 27th 2004

By

Vinod Chamarty

B.Tech., Regional Engineering College,
Calicut, India. 2001

Thesis Advisor and Committee Chair: **Dr. Hal Carter**

Abstract

The issue of performance of compile-driven mixed-signal simulation is a challenging problem with optimization techniques researched to speed-up the various phases of simulation. The matrix load phase of the analog simulation kernel has been found to consume the largest percentage of the total simulation time. It is also known that in the worst case, the matrix load time is a cubic function of the number of equations in the system. Therefore, efforts have been directed towards reducing the matrix load time in a mixed-signal simulation paradigm.

The elaborated set of Characteristic Expressions (CEs) forms the input to the matrix load phase of the analog kernel. The CEs are formed either as a result of elaborating simultaneous statements or because of the association of the quantities and terminals. A reduction in the elaborated set of CEs would result in the reduction of both the matrix load and matrix solve times. The current data structures do not support the reduction of the elaborated set of CEs. This thesis presents the design of a new Information Structure (IS) to support the modification and reduction of CEs and sets of CEs respectively. We exploit this design to improve the performance of compile-driven mixed-signal simulation. A proof of concept has been provided to demonstrate the viability of the designed Information Structure.

To

My loving mom and dad, and my dearest brother

Acknowledgements

I would like to thank my advisor Dr. Hal Carter for his continuous guidance and inspiration all through my research work. I have gained immensely through his motivation and style of working. Without his constant guidance and encouragement, this research work would not have been possible.

I would like to thank Dr. Philip A. Wilsey and Dr. Karen Tomko for taking time out of their busy schedule, and readily accepting to serve on the thesis defense committee.

I would also like to thank my companion researchers in the Distributed Processing Laboratory for their dedicated effort in creating the research platform, SIERRA2. I would like to thank each one of them for their valuable suggestions, and wonderful company during the course of my research work. There were many memorable days spent in the lab trying to build the simulator. I also enjoyed the company of a bunch of friends at UC. My interactions with them have been very entertaining, to say the least. I will always cherish the fun and the good times spent together.

I would like to thank my loving parents for what I am today. Their guidance and inspiration has been my driving force. Their vision and understanding has seen me overcome many difficulties. Thanks are also due, to my dearest brother for standing by me at all times, and helping me take important decisions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Any design environment would require simulation as a tool for analyzing the final behavior of the system being designed. Designing a circuit is a perfect case for simulation. It is an indispensable tool since it allows for verifying the working of the circuit before it is fabricated. With the rising costs of a design re-spin, it is essential to get the design right before submitting for fabrication. However, a few basic factors come into consideration when one chooses to apply simulation to a design cycle, considerations such as the cost, effort, and time. Cost of simulation is mostly counted in terms of the dollars spent on a tool for simulation. The effort can be counted in terms of the total man-hours required for simulating a design. The time taken for a simulation to finish is a very important factor, since it decides the time of availability of the final product in the market. The first two factors are very important to the industry and the last factor is the most important contributor to research in the area of simulation.

From a circuit designer's point of view, simulation can be done in the digital, analog, or mixed-signal domain. Languages like VHDL[1] and Verilog HDL[2] provide circuit designers with the ability to model circuits in the digital domain. Traditionally,

[1] Very High Speed Integrated Circuit Hardware Description Language
[2] Verilog Hardware Description Language

SPICE[1] has been used for the description of circuits in the analog domain. With increasing need for modeling systems with both digital and analog components in the physical domain, hardware description languages like VHDL-AMS[2] and Verilog-AMS have gained increasing importance. Also, with the circuit feature sizes moving into the nanometer scale, digital circuits are best analyzed with an analog description. VHDL-AMS supports a uniform modeling environment that provides a framework for simultaneously representing both continuous time and discrete time descriptions. The applications of this language due to its mixed-signal nature are enormous.

With the evolving need for simulators which support such languages, much effort has been directed towards developing such high performance mixed-signal simulators [1]. This necessitates a need for research into the performance of such systems. This thesis investigates an approach for increasing the performance of a class of mixed-signal simulators with demonstration of the performance increase in a real simulator.

1.1 Motivation

Our primary interest is in compiled mixed-signal simulation, where the repeated solution of a set of Ordinary Differential Algebraic Equations (ODAEs) is obtained for the continuous time component of the model being simulated.

The need for high speed mixed-signal simulation motivated this study of enhancing the performance of mixed-signal simulation. The reduction of the elaborated set[3] ODAEs has been recognized as the major factor influencing the speed of the simulator [2].

---

[1]Simulation Program with Integrated Circuit Emphasis
[2]Very High Speed Integrated Circuit Hardware Description Language – Analog and Mixed Signal
[3]The set of equations formed after elaborating the design hierarchy. It forms the input to the solver

Mixed-signal simulation requires two steps - elaboration to put the model together, and solution, where the solution phase is repetitive over time. The solution phase involves the creation of the matrix and vector values for every time point, and this matrix is then solved.

In general, the total simulation time can be classified into the following three categories:

- Elaboration time
- Matrix build time
- Matrix solution time

It was found that in compiled mixed-signal simulators, the majority of the simulation time is spent in the matrix load phase of the analog kernel. This phase of the simulator would need a differential solver, either an automatic differential solver or a symbolic differential solver. It was observed that this phase takes about 70% of the time taken for the entire simulation [3]. Most of the research has been focused on the matrix build phase since matrix build is a repetitive operation for every time point and is the most time consuming phase of the simulator. This is because the time complexity for finding the solution of the matrix is $O(n^3)$ for dense matrices, where n is the number of equations. However, mixed-signal simulation usually involves very sparse matrices (10% or less density) in which case the execution time complexity is around $O(n^{1.2})$ [4,5]. Research has been directed at reducing the size of the matrix so as to reduce the matrix build and solve time [2]. The reduction of the size of the matrix is achieved by reducing the number of equations or by reducing the number of quantities involved in the simulation.

It has been observed that the means of reducing the size of the matrix is inherently limited by the data structure for the simultaneous statements in a hardware description language for mixed-signal simulation. The current approach only allows for local modification of the set of simultaneous statements in a particular hierarchy.

The following limitations have been identified in the existing method of elaborating simultaneous statements.

- It does not allow for the removal of equations from the elaborated set of ODAEs.
- It does not allow for the addition of new equations into the ODAE set.
- It does not allow for modification of the equations existing in the ODAE set.

This study proposes to remove all of the above limitations and provide an information structure which adequately represents the simultaneous statements in a compile-driven mixed-signal simulation model. We call this the Simple Simultaneous Statement Information Structure, hereafter referred to as the S3IS. The S3IS designed as a part of this study allows for the reduction of the set of ODAEs as it gives scope for addition, removal and modification of an equation.

It can be summarized that the problem of reduction of the matrix size has been reduced to the problem of finding a suitable information structure (IS) to allow for the reduction of the elaborated set of ODAEs. The IS proposed in this document is the S3IS, which has been proved to eliminate all of the above stated limitations of the traditional IS.

1.2 Statement of the Problem

This thesis presents the design of an information structure for *simultaneous statements* in a compile-driven mixed-signal simulation paradigm. The information structure is designed to allow for the reduction of the elaborated set of ODAEs.

We attempt to design an information structure to satisfy these objectives:

- Representation – The information structure (IS) should represent a *simultaneous statement* and be modifiable to allow for the change in the equation description.
- Code publishing – The IS should be published as efficient intermediate code.
- Matrix Load – The IS should enable matrix load operation to be performed.
- Reduction of equation set – The set of equations may be reduced, the method for which may vary.

1.3 Approach to Solution

We need to design an Information Structure which at the minimum satisfies the above criteria. The requirements of the design can be interpreted as follows:

- Design a data structure to represent simultaneous statement.
- Make sure that the data structure can be implemented in a compiled simulator with intermediate published code.
- Ensure a consistent elaboration strategy for simultaneous statements.
- Modify the Matrix Load phase to take the new data structure as an input.
- Design and implement a reduction algorithm to show a proof of concept for the S3IS.

Figure 1.1 shows the flow of this approach. The design of the IS gives scope for the application of a variety of graph algorithms to perform the reduction of the elaborated set of equations. The analysis of the approach taken in this thesis is shown specifically in a VHDL-AMS simulator, SIERRA2, which was developed at the University of Cincinnati. The reduction algorithm is showcased to prove that the elaborated set of equations can be modified and may lead towards improvement of the matrix solve time. The simulator has been designed to run with and without the reduction algorithm so as to help in comparison of the simulation times. It should be noted that the S3IS provides a base for implementation of better reduction algorithms in future.

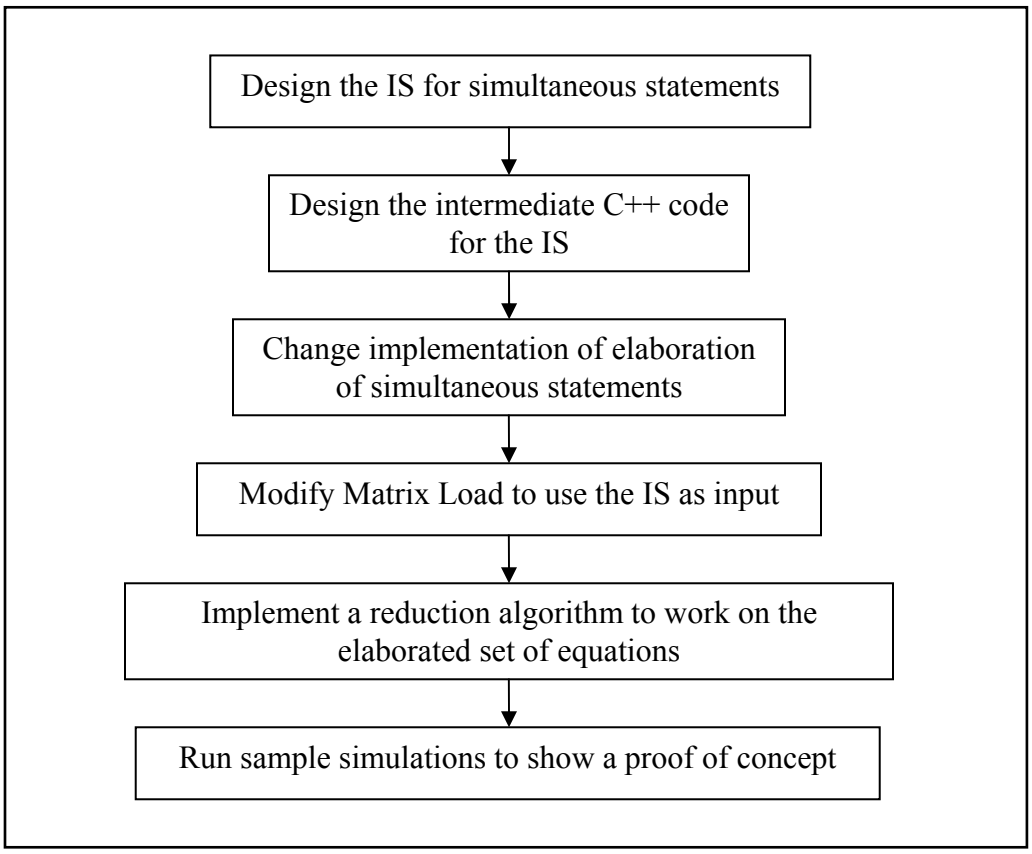

Figure 1.1: Flowchart for the approach to the problem.

1.4 Summary of Results

The results of this effort show a very viable information structure meeting all requirements outlined in the problem statement. In particular, the design allows for the reduction of the set of equations (ODAEs) and the performance improvement is shown to be a function of both the problem size as well as the reduction method applied.

1.5 Overview of the Document

The rest of the thesis is divided into six chapters. A brief outline of each chapter is as follows.

1. Background: This Chapter provides the reader with the basic concept of elaboration. It starts with an overview of compiled mixed-signal simulation followed by a discussion of elaboration as applicable to compiled mixed-signal simulators. We proceed with a discussion of the implementation of the same in a mixed-signal simulator for VHDL-AMS, SIERRA2 at the University of Cincinnati.

2. Problem Statement: This Chapter introduces the problem under consideration and describes why the problem is a compelling one. It details on the scope of the problem and the various provisions that the design seeks to provide.

3. Approach: This chapter enumerates the steps taken to reach the final design of the Information Structure (IS). It details the implementation of the IS using diagrams and illustrations. It also describes the complete elaboration methodology and data structures involved for performing the matrix load operation and reduction of the elaborated set of ODAEs.

4. Optimization Approach: This Chapter describes the optimization approach implemented for reducing the elaborated set of CEs. The algorithm and its features are discussed.

5. Experimental Results: This Chapter presents an analysis of the results obtained from the experiments conducted on SIERRA and SIERRA2. The overhead introduced by the new data structure, as well as the performance of the optimization algorithm are presented.

6. Conclusions and Future Work: This Chapter concludes the thesis and provides some insight into the scope it provides for possible work to be done in the future.

Chapter 2

Background

Hardware is usually described at the system level using a Hardware Description Language (HDL) model. The correct working of the hardware described is verified though simulation. Thus simulation forms a very important step in the modeling and verification of circuits.

A simulator typically allows us to understand the behavior of a circuit through simulation of the hardware described in a HDL model. The implementation of the simulator can vary depending on the needs of the user, as well as the technique underlying the different stages of simulation. The user determines the interface that is to be implemented. This may require that certain data structures be accessible to the end user. Since the simulator itself is programmed using a software language, its implementation may differ based on the various compilation techniques available. The implementation of the simulator also differs based on the type of circuits simulated. Simulators may be capable of simulating, either only digital circuits, analog circuits or sometimes mixed-signal circuits.

This chapter presents the background required to understand the various stages of simulation, especially elaboration. It starts with a description of the various stages of a general simulator. This is followed by a discussion of the common elaboration techniques used in present day simulators. In the later part of the chapter we focus on elaboration as applicable to a mixed-signal hardware description language like VHDL-AMS.

2.1 Introduction to Simulation

Simulation is a technique by which a user visualizes the behavior of a circuit, without actually fabricating it. It is achieved through efficient design of simulators, which are actually programmed using a high-level software language. A simulator is designed to understand a circuit description and evaluate various parameters in a particular time domain.

Figure 2.1 shows a simple block diagram of a generic simulator. The input to the simulator is a model written for a specific hardware, typically using a Hardware Description Language (HDL). HDL based model descriptions are preferred when system level hardware is to be described, since it makes modeling easy. The hierarchical nature of HDL implies that models written using a HDL can be re-used by instantiation in other models. This considerably simplifies the description of complex hardware. The output of the simulator is basically a set of results, which represent the values of the parameters of the circuit described. The output can be in the form of a text file listing the values at various time instants or it can be a graphical visualization of the values obtained.

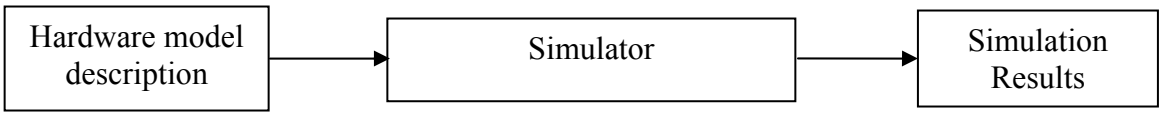

Figure 2.1: Block diagram of a generic simulator.

As discussed previously, the implementation of the simulator may differ based on its model processing technique. It is either compile-driven or interpreter based. Most of the commercial simulators are compiler based because of their efficiency and performance advantage. The block represented by the simulator in Figure 2.1 is actually comprised of various simulation stages as shown in Figure 2.2. The block diagram shown below is an illustration of a typical compile-driven mixed-signal simulator.

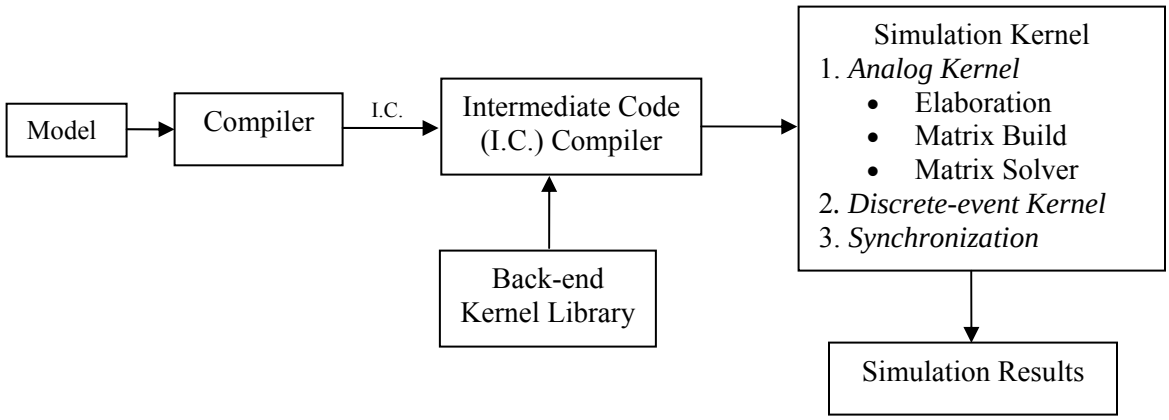

Figure 2.2: Block diagram of a compile-driven mixed-signal simulator.

The following are the different stages of a compile-driven simulator -

1. Front-end Compiler: The front-end compiler is responsible for parsing the model file and generating the Abstract Syntax Tree (AST) [6], which is then converted into an Intermediate Code (I.C.). Parsing starts with the lexical analysis of the model description, which involves identification of the various keywords and identifiers. This is usually done with a standard analyzer like flex. The next stage is the language parser, which creates the Abstract Syntax Tree of the model. An abstract syntax tree is specified for a language in the Language Reference Manual. It is however

constructed based on the model under evaluation. This phase is usually done with parsers like Bison or PCCTS[1].

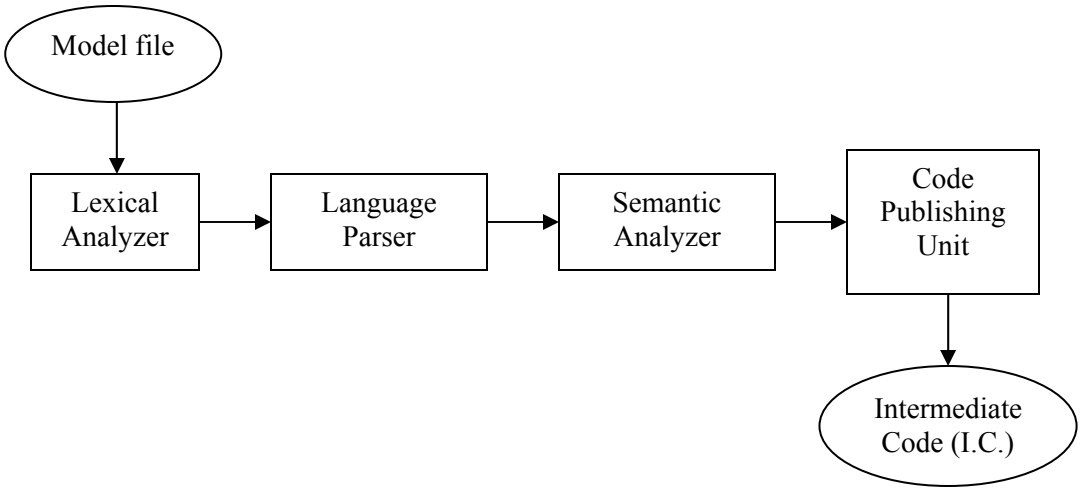

Figure 2.3: Stages of a front-end compiler.

The next step involves the analysis of the Abstract Syntax Tree to determine any deviations from the language's semantic sense. The final stage is the Code Publishing Unit, which walks through the Abstract Syntax Tree and outputs Intermediate Code (I.C.) in a pre-designed format. This stage captures the details of the HDL model described so as to facilitate simulation. The Intermediate Code is typically in a high-level software language so that it can be compiled along with a back-end library to achieve the results of the simulation.

2. Intermediate Code Compiler: After the publishing of the Intermediate Code in the desired format, it is compiled along with kernel libraries specifically designed to understand the Intermediate Code. This stage is basically the usual high-level language compiler.

[1] Purdue Compiler Construction Tool Set

3. Back-end Kernel Library: The kernel library supports the compilation of the Intermediate Code. The final executable includes the kernel libraries as well as the object code of the Intermediate Code. The kernel libraries are designed efficiently to capture all the language features of the Hardware Description Language (HDL) being simulated. To reduce the compilation time of the Intermediate Code it is prudent to capture most of the language features in its back-end library. The library is essentially pre-compiled source code. The Intermediate Code (I.C.) on the other hand, must represent the properties specific to the input model description.

4. Simulation Kernel: The simulation kernel is the heart of the simulator. All the previous stages are meant to provide inputs to the simulation kernel. It is the core of the simulator, which evaluates the circuit description to output the values of the parameters of the circuit.

The simulation kernel of a mixed-signal simulator consists of a discrete-event kernel and an ODAE-based solver. The kernel also provides for communication between the two simulation paradigms. This communication protocol is typically referred to as synchronization [7].

The discrete-event kernel involves execution of the process statements until termination. The elaboration phase creates a set of discrete-event processes from the hierarchical description. The sequential statements within a discrete-event process modify the values of the parameters associated with it. In a distributed simulation paradigm, events are scheduled to propagate the values of signals between the appropriate processes after a time delay. The simulation proceeds in discrete steps depending on when events occur. When an event occurs on a signal to which a process is sensitive, the process

resumes and may schedule transactions on that signal at some later time. Thus the simulation of digital processes takes place in two phases – initialization and repetitive execution of simulation cycle. Initialization phase involves the assigning of initial values to the signals. The simulation cycle is the execution of the events based on the earliest scheduled transaction. The simulation completes when there are no further scheduled transactions. A more detailed explanation of the discrete-event processes can be found in [8,9].

Even though both the kernels of a mixed-signal simulator affect the performance of the simulator, we are more interested in the working of the analog kernel of the simulator as it contributes the largest percentage of the total simulation time. The analog kernel mainly consists of the following three stages –

(a) **Elaboration:** A model written using a HDL describes the hardware of a particular system. This description needs to be directly translated into the simulator. However, since the description is hierarchical, there is a need to collect the information available at different levels of the hierarchy. The method of flattening the entire hierarchical description of a model is called elaboration [10]. This stage of simulation is explained in more detail in the following sections. Elaboration is followed by an initialization phase where all the unknowns in the model are initialized.

(b) Matrix Build: The analog portion of the mixed-signal circuit is evaluated based on the equations which describe the circuit. This set of equations has been previously described and has been referred to as Ordinary Differential Algebraic Equations (ODAEs). The set of ODAEs are loaded into a matrix and then solved

to obtain the values of unknowns. The part of the simulator which assigns memory to the matrix elements and assigns values to all the row and column members of the matrix is referred to as the matrix build stage. Research [3,11] has indicated that this stage is the one of the most critical stages of the simulator in terms of its performance.

(c) **Matrix Solver:** The matrix build phase is followed by a matrix solver. The matrix solver is responsible for finding the values of the unknowns. The matrix solver is actually comprised of three different phases, which convert the possibly non-linear differential algebraic equations to a system of linear equations, which are solved simultaneously. The different phases are shown in Figure 2.4. Many matrix solvers have been implemented and analyzed to speed up the performance of the matrix solve phase.

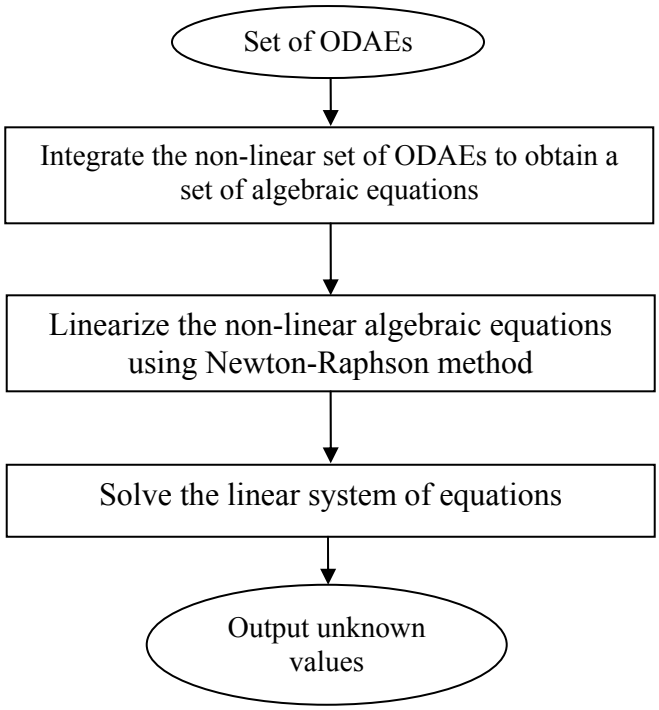

Figure 2.4: Steps in the Matrix Solver stage.

These methods are primarily classified as direct methods and relaxation based methods [12,14,15]. Parallel solver techniques and super nodal approach [13] have also been researched to speed up the performance of the matrix solver.

2.2 Elaboration

A Hardware Description Language (HDL) allows for a hierarchical description of hardware. The hierarchical property implies that models written using a HDL can be re-used by instantiation in other models. Thus, one particular hardware or circuit needs to be described only once. This enables the language user to write complex and higher level models with great ease. However, this feature of the language would make it necessary to collect the information from the sub-circuit when trying to analyze the main circuit. The technique for accomplishing the above mentioned task is called elaboration and has been further described in this section.

2.2.1 Definition of Elaboration

Elaboration is defined as the method of flattening the entire hierarchical description of a HDL model [10]. Usually there is a specific sequence of steps defined to elaborate a model written in a Hardware Description Language. For example, the VHDL-AMS Language Reference manual describes the steps to be taken in the elaboration of the language's various constructs. However, the implementation of elaboration is usually left to the discretion of the developer of the simulator. We would consider one such implementation in the later sections of this document.

Every system or model description involves the use of various language constructs. It is the task of the simulator to provide a means of interpreting every language construct and transferring the information to the simulation kernel. Thus, the elaboration phase ensures that each HDL construct achieves its desired effect. Elaboration has been defined as the process by which a declaration achieves is effect [10]. The implementation of an elaboration strategy would require the creation of proper data structures to ensure faster and functionally correct simulation. Any data lost during elaboration would lead to incorrect analysis of the circuit description. It is also necessary to ensure a unique interpretation of any individual language construct. The construct as such may again require passing different values to the simulation kernel, each time it is invoked. Thus, a proper elaboration strategy would ensure that all the above properties are adhered to.

2.2.2 Implementation of Elaboration

As mentioned previously, the architecture of the implemented elaborator is usually left to the discretion of the developer. This stage of the simulator defines the type of simulator. If the result of elaboration is a program which is interpreted by the simulator, it is called as *interpreted simulation.* On the other hand, if elaboration results in the model description being compiled into object code and linked to the simulation kernel, it is called as *compiled simulation.*

The two major strategies used with compiled simulation are –

(a) Pre-Intermediate Code Elaboration: If the elaboration phase is completed before the publishing of the Intermediate Code (I.C.), it is referred to as pre-I.C.

elaboration. The advantage of this strategy is that elaboration is kept simple and is not extended to the back-end simulation kernel. This type of elaboration finishes elaboration before the compilation time of the published code. The disadvantage of this method is that any new model or circuit, which uses the previously elaborated models or sub-circuits will have to elaborate the sub-circuits again.

(b) **Post-Intermediate Code Elaboration:** If the elaboration phase is performed after the publishing of the Intermediate Code (I.C.), it is referred to as post-I.C. elaboration. This strategy postpones the elaboration phase to the run-time. The published code carries all the information pertaining to each sub-circuit modeled in the system. This code, along with the back-end simulation kernel, is compiled into a final executable. The elaboration of the model occurs only when the final executable is run. The advantage of this approach is that any number of sub-circuits may be analyzed, but, elaboration occurs only with the final configuration of the system.

2.3 Language Effect on Elaboration

The semantic meaning of a language construct is specific to the language. Hence, even though an elaboration strategy may be generic, it is necessary to define the elaboration of each construct in a new language domain. This section of the document introduces the basic language constructs in VHDL and VHDL-AMS and then proceeds with a discussion of elaboration as applicable to VHDL-AMS.

2.3.1 Introduction

VHDL introduces the concept of entities and architectures for the digital domain where simulation occurs in discrete time. An *entity* represents a portion of a hardware design that has well-defined inputs and outputs and performs a well-defined function. An *architecture* body specifies the relationships between inputs and outputs of a design entity [10]. Each entity may be instantiated multiple times to form higher level and complex circuits. The inputs and outputs to the system are defined as *signals*. The relationships between signals are defined using *concurrent statements*. The concurrent statements provide the necessary constructs to describe the behavior of hardware.

VHDL-AMS extends VHDL to include the analog and mixed signal extensions. This enables users of the HDL to simulate analog circuits in the continuous time domain and also perform simulation of mixed-signal circuits, which involve interaction between the analog and digital domains. The inputs and outputs in an analog domain occur between nodes called *terminals* and their values are represented by *quantities*. ODAEs are expressed using *simultaneous statements*, which together with implicit equations generated from the definition of terminals help evaluate the values of quantities.

2.3.2 Elaboration as applicable to VHDL-AMS

A design hierarchy is represented by a design entity. Hence, elaboration of a design hierarchy is achieved through elaboration of each statement within the design entity. The VHDL-AMS Language Reference Manual (LRM) defines elaboration for design hierarchies, declarative parts, statement parts, simultaneous statements and

concurrent statements. We limit our discussion to elaboration of a design hierarchy since it is most relevant to this document.

The elaboration of a design hierarchy creates a collection of processes interconnected by nets and quantities whose values are defined by Characteristic Expressions (CEs). The run-time elaboration strategy of a design hierarchy in VHDL-AMS consists of two phases –

(1) Phase 1: The first phase takes place in three steps, the *instantiation*, the *signal net-list update*, and the *connection*.

- The instantiation takes place in a top-down approach, where the objects represented in the model are created first for the top-most design entity. Creation of objects of a model implies creation of the components and processes representing the models. Objects are also created for all VHDL-AMS constructs which represent declarations.
- The signal net-list update, as the name suggests stores all the information related to the signals in the system. A signal is a (Value, Time) tuple and the information required to update it is acquired from the model description. The fanout is also updated and drivers are created for each of the signals. This step takes place in a bottom-up fashion.
- The connection phase passes the information about the signals on to the instantiated components and processes. This is necessary to create the signal source tree (for multiple drivers of signals) and also to evaluate the type conversion functions.

(2) Phase 2: This phase is necessary to form the characteristic expressions so as to simulate the system in the continuous time domain.

- The first step involves the identification of all the unknown quantities across all design entities. Solvability checks are applied using the *characteristic number* of each external block. The *characteristic number* is equal to the number of characteristic expressions formed for that block.
- In the second step, the association of the formal and actual terminals and quantities is performed. A top-down approach is taken for this step.
- The third step involves identification of *break statements* to form the discontinuity augmentation sets. This helps in detecting discontinuities during mixed-signal simulation [2,10,16,19].
- The last step involves creation of characteristic expressions[1] from the *simultaneous statements*, as per the rules defined in the VHDL-AMS LRM [10]. The characteristic expressions (CEs) created from simultaneous statements are referred to as explicit CEs. Another set of CEs are created from the declarations of quantities with respect to their terminals. These CEs are referred to as implicit CEs. The *conditional simultaneous statements* in VHDL-AMS imply that the elaboration phase maintain a data structure to control the selection of CEs during simulation.

2.4 Summary of the Chapter

The working of a general mixed-signal simulator has been introduced to the reader in this Chapter. This was followed by a description of the various stages of simulation with special importance being given to the elaboration phase. Elaboration was described with reference to VHDL-AMS in order to form the foundation for the problem statement and the approach taken to solving the problem defined earlier in this document.

[1] A characteristic expression either represents a simple simultaneous statement or is an implicit consequence of the declaration and association of quantities and terminals.

Chapter 3

Problem Statement

It has been noted in the previous sections of this document that improving the performance of a mixed-signal simulator has been a major research area [14,17]. Research has been targeted at improving the various stages of a compiled simulator. Methods using pre-processing [2], selective matrix update [3] and parallel solver [13] have been applied to speed up mixed-signal simulators. This document presents a new, novel elaboration approach which provides the infrastructure to speed up mixed-signal simulation in a compiled simulator.

The research documented here has been particularly guided towards removing the performance bottleneck in compiled simulators. The study of the problem has enabled us to come up with a set of requirements to solve the problem. The results from this study have been applied in a research simulator at the University of Cincinnati and the performance of the simulator has been analyzed. This Chapter presents an in-depth description of the problem, a discussion on how compelling it is, the requirements of a solution to the problem, and finally the broad scope that it provides in terms of speed enhancement of a compiled mixed-signal simulator.

3.1 Introduction

Research has come up with a wide variety of proposals to enhance the performance of compiled mixed-signal simulation [18,20]. Earlier research in this area has used profiling techniques to identify the regions where the simulator spends the maximum amount of time.

As has been identified before, the analog kernel basically consists of three phases. They are -

- Elaboration phase
- Matrix Build phase
- Matrix Solve phase

Further, it has been identified that the analog kernel of a mixed-signal simulator takes the largest percentage of the total simulation time [3]. Efforts have been directed at reducing the total simulation time of the analog kernel [2,3,11,14]. Research has been concentrated on the various techniques to improve the performance of each phase of the analog kernel. In this section, we present some of the optimization approaches that have been considered so far to identify the problem considered by this document, and highlight the need for an approach to solve it.

3.1.1 Optimization Approaches to Speed up Analog Kernel

The most important metric for the performance of a mixed-signal simulator is its simulation time. Since the analog kernel of a mixed-signal simulator is the most critical part of the kernel, the time spent by the simulator in the analog kernel is a very important performance metric of a compiled mixed-signal simulator.

It has also been identified in earlier research [3,13] that matrix build phase and the matrix solver phase contribute a greater percentage of the analog solver time. This is because these phases are iterative and hence have a greater effect on the total simulation time of the analog kernel.

The following are some of the techniques which have been applied to improve the simulation time of the analog kernel.

- **Equation Set Optimization (ESO):** The ESO technique attempts to speed up the set up phase of the matrix build operation [3]. The set up phase involves the allocation of the row where the equation will be entered in the matrix. It also takes care of allocation of pointers to the positions in the matrix where the contribution of this equation will be entered. The use of *simultaneous if statements* in VHDL-AMS implies that the set of equations loaded into the matrix keeps changing with progress in time. These equations have been classified as the *Base Set* and the *Conditional Set* [3]. The technique improves the matrix build phase by loading the *Base Set* at the beginning of the simulation time and, loading the *Conditional Set* at every time point in the simulation. This approach considerably improves the simulation time.
- **Conservative Equations Optimization (CEO):** The CEO technique aims to eliminate the need to load the conservative equations of a circuit description repeatedly for every simulation time point. The conservative equations in a circuit are basically the equations formed by applying Kirchoff's Current Law (KCL) to the circuit. Assuming that the *simultaneous if statements* do not modify the circuit net list, it is noted that the KCL equations are essentially the same with progress in the simulation time. Hence this optimization creates a stamp of the matrix and reproduces

it with the contribution KCL equations at every time point. The time saved by avoiding the re-evaluation of the contributions of the conservative equations has been shown to be considerable [3]. This method is also directed at improving the matrix build times.

- **Multiple Solution Methods (MSM):** This method proven to be effective by Vasudevan [21] and later improvised by [3], attempts to reduce the time spent in the matrix solver phase of the analog kernel. The method separated the elaborated set of equations at each time point into the linear set of DAEs and non-linear set of DAEs. The generic solver has a linearization step where the Jacobian matrix is calculated for the set of non-linear equations. The MSM as shown in Figure 3.1, identifies the set of linear DAEs at the current simulation timepoint and passes them on to a faster linear solver. This method has been shown to improve the performance of the matrix solver.

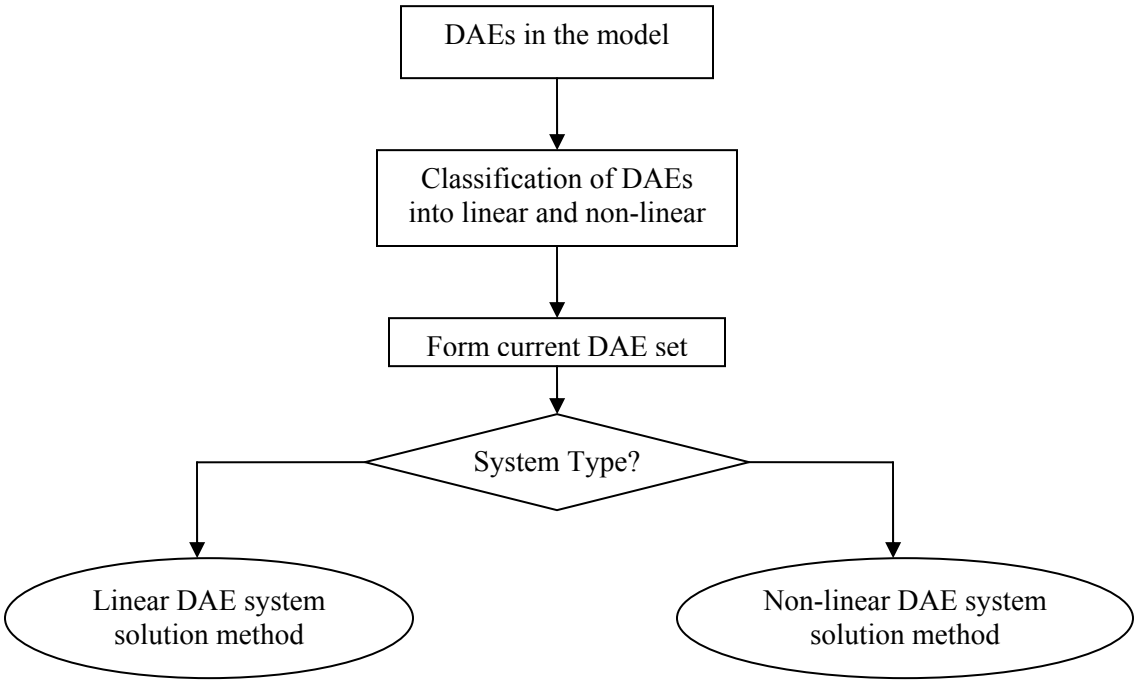

Figure 3.1: DAE system solution process [3].

- **Pre-processing Method for Reduction of Equations:** Our research has been largely motivated by the elaboration support needed by this optimization, namely reducing the number of equations to be simulated. Research has identified that reducing the number of equations in a model description will ultimately lead to a reduction in the size of the matrix [2]. The pre-processing method is applied on the VHDL-AMS description of a model to identify certain properties and change the model description to a more efficient one. The output of the applied algorithm is again a VHDL-AMS model which is then analyzed and simulated. Figure 3.2 gives the general flow of the pre-processing method.

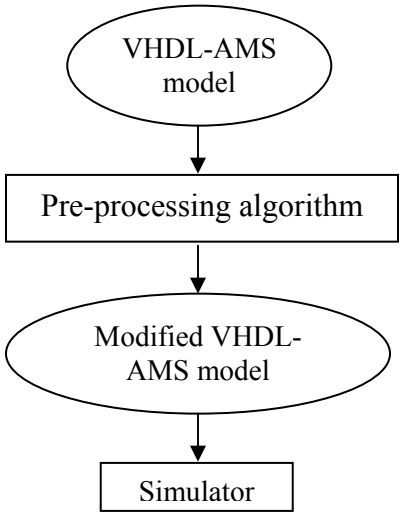

Figure 3.2: Flow graph for the pre-processing method for reduction of equations.

Two different approaches have been described for the conserved models and the non-conserved models. Both the approaches aimed at reducing the number of equations in the model description. The advantages, disadvantages and limitations of this technique have been described in detail in the following sections as we introduce the problem that this document attempts to address.

3.1.2 Identification of the Problem

It is clear to the reader that to improve the performance of compiled, mixed-signal simulators, it is necessary to improve the performance of the analog kernel of the simulator. In addition, research in improving mixed-signal simulation speed has concentrated primarily on the matrix build and solver phases of the analog kernel since they contribute the largest percentage of the of the total simulation time.

The explanation for this is that the matrix build and solver phases are iterative and are repeated at every simulation timepoint. Therefore, we can broadly split the optimization approaches to two different classes. They are -

1. Class A – In this approach, the input to the matrix build phase (elaborated set of ODAEs) is optimized.
2. Class B – In this approach, the matrix build and solver phases themselves are optimized.

The ESO, CEO and MSM methods described above are Class B approaches. In this document we are more interested in Class A methods, like the pre-processing method for the reduction of equations.

The input to the matrix build phase is the elaborated set of equations. Note that any optimization on the elaborated set of equations is a one time activity, assuming that *simultaneous if statements* are not used in the description of the VHDL-AMS model. Thus, this should be our preferred approach in comparison with optimizing the matrix build and solver phases themselves. Even if the *simultaneous if statements* are used in the model description, the elaborated set of equations would not necessarily change for each and every timepoint in the simulation. Therefore, the problem of improving the

performance of the analog kernel has been reduced to the problem of optimizing the elaborated set of equations.

It has been pointed out in previous research that the data structure used for the representation of *simultaneous statements* is a limiting factor in optimizing the elaborated set of equations [2]. This is exactly the problem that this thesis attempts to address. We designed an efficient information structure for the simultaneous statements in a compile driven mixed-signal simulation paradigm. The information structure attempts to provide a data structure for the simultaneous statements, as well as provide a mechanism for producing the Intermediate Code, that enables efficient and functionally correct elaboration of simultaneous statements.

3.2 Analysis of the Problem

The representation of a *simultaneous statement* in a mixed-signal simulator can have an impact on the performance of the simulator. When we refer to simultaneous statements, we are indirectly referring to the representation of the Characteristic Expressions (CEs) in the simulator. In general, this not only effects the loading of the matrix, but also the capability to apply reduction algorithms on the elaborated set of CEs.

The discussion in this section is presented with reference to VHDL-AMS and an implementation of a specific research simulator for VHDL-AMS. However, since the generality of the discussion is maintained, we can confidently state that the findings of this study are applicable to the entire class of compiled mixed-signal simulators. We start with an analysis of the current implementation of the simulator, called SIERRA developed in the Distributed Processing Laboratory at the University of Cincinnati. Then,

we present a set of requirements for the proposed information structure for the *simple simultaneous statements* in order to overcome the limitations imposed by the current data structure.

3.2.1 Discussion of the Current Approach

The research simulator, called SIERRA at the University of Cincinnati, uses a sparse matrix solver in its analog kernel [22,23]. It is essential that the published Intermediate Code (I.C.) is in a format that can be used by the sparse matrix solver. Figure 3.3 presents a block diagram of how simultaneous statements are elaborated and loaded into the matrix. It shows the details of the composition of the published code w.r.t. *simple simultaneous statements*, as an example.

SIERRA publishes its Intermediate Code (I.C.) in C++. This C++ code adequately represents the VHDL-AMS model. The creation of the elaborated set of CEs requires that all the information about the *simultaneous statements* be collected from the entities at every level in the model hierarchy. The Intermediate Code in C++ creates a class for every *architecture* - *entity* combination. Instantiations of a component in the VHDL-AMS model correspond to creation of objects of this class.

The Intermediate Code generator needs to perform two tasks for every *simple simultaneous statement*. They are -

- Create component objects in the published code for every CE to be created in the back-end.
- Publish a function which represents the *simple simultaneous statement*. This function is called in the matrix build phase of the analog kernel, and is in a format

compatible with the sparse matrix solver. This function is also associated with the corresponding component object created as mentioned above.

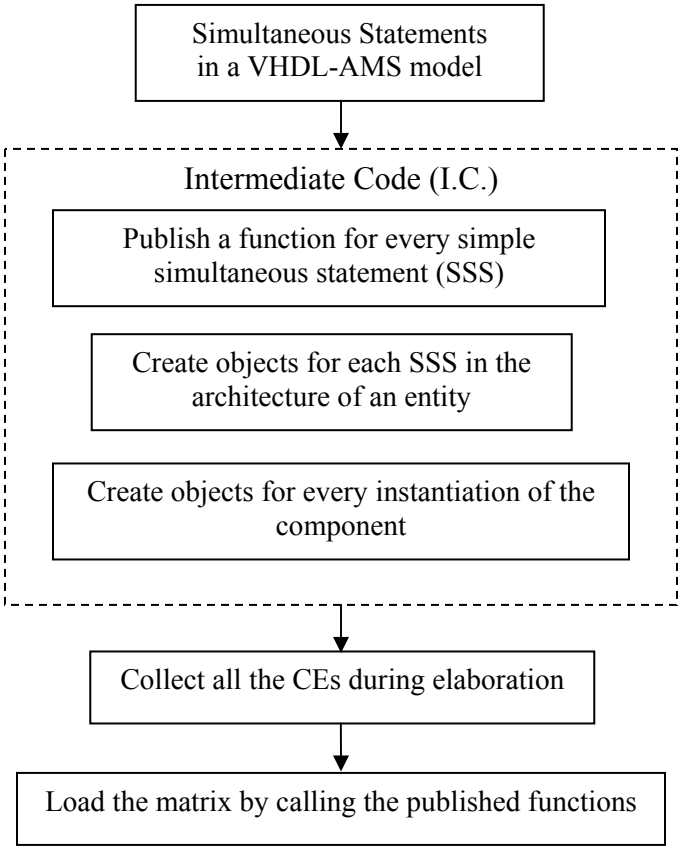

Figure 3.3: Description of elaboration and matrix load of SSS.

An analysis of the current approach shows that all the CEs created in the Intermediate Code have to be loaded into the matrix. Moreover, the matrix load operation takes place by directly calling the functions which have been published based on the VHDL-AMS model. This approach does not allow us to modify the elaborated set of Characteristic Expressions (CEs). It also does not allow us to modify the composition of the CE itself. Thus it creates major limitations in applying equation reduction techniques before loading the matrix. Sanjiv has recorded in his thesis that his pre-processing

approach, which we henceforth refer to as the current method of reduction of CEs, was limited in its application by the data structure of the Characteristic Expressions [10].

We can summarize the limitations or disadvantages of the current methods as follows -

- The method does not allow for the removal of equations from the elaborated set of ODAEs.
- The method also does not allow for the addition of new equations into the ODAE set.
- The current approach does not allow for modification of the equations existing in the ODAE set.
- Any reduction approach may only be performed on the simple simultaneous statements in the *architecture* of an *entity* and hence is not global in its approach. This means that any reduction algorithm may be applied only before the code generation phase.

Our purpose is to develop a data structure that overcomes the above stated limitations.

3.2.2 Requirements of the Proposed Design

As a part of the study conducted to support this thesis, we have analyzed the published Intermediate Code (I.C.), the data structure of the Characteristic Expressions (CEs) and the matrix load operation. A set of requirements for the new and improvised approach have emerged as a result of this thesis. The goal of the improvised approach is to support the reduction of the elaborated set of ODAEs.

The requirements have been summarized below –

- Representation – The information structure (IS) should be able to represent a *simple simultaneous statement* and be modifiable to allow for the change in the equation description.
- Code publishing – The data structure should be published as efficient intermediate code and must be in a format compatible with the sparse matrix solver.
- Matrix Load – The IS should enable matrix load operation to be performed.
- Reduction of equation set – The set of equations may be reduced, the method for which may vary.

3.3 Summary of the Chapter

This Chapter has provided the reader with a description of the problem statement. A detailed analysis of the current approach has been conducted, and its limitations have been highlighted against the backdrop of improving the performance of a mixed-signal simulator. The limitations of the traditional approach and the ever-growing need to improve the performance of the mixed-signal simulator make our problem a compelling one. To conclude, we have provided a set of requirements for the design of a new Information Structure (IS), which would be conducive to the idea of reducing the elaborated set of ODAEs.

Chapter 4

Approach

In this document we have so far presented the concepts of mixed-signal simulation and its various stages with a particular emphasis on elaboration. We have also discussed the nature and validity of the problem statement. We are convinced of the importance of this problem, and propose an approach to the design of an Information Structure (IS) for efficient elaboration from the problem of improving the performance of the mixed-signal simulator.

In the previous Chapter we have identified the requirements of a design which would support the reduction of the elaborated set of ODAEs. We have implemented one such Information Structure (IS) in the analog kernel of a mixed-signal simulator, called SIERRA2 developed at the University of Cincinnati. The design is generic and is applicable to any compiled mixed-signal simulator.

This Chapter presents the design of the Information Structure (IS) for *simple simultaneous statements*. It also describes the design of the necessary Intermediate Code. We, then proceed to describe how the elaboration methodology has been implemented.

This is followed by a discussion of the implementation of the matrix load operation using the data structure.

4.1 Information Structure (S3IS)

The design of an Information Structure (IS) requires careful analysis of the problem. The IS basically comprises of the data structure for the *simple simultaneous statements*, the code generator to achieve the publishing of the necessary Intermediate Code, and the classes in the back-end kernel to support elaboration and the matrix load operation.

Since the basic requirements of the IS have been laid out, we start this section with the concepts behind the design. As we proceed we describe the classification of various elements of the data structure and discuss its implementation in detail. Flow graphs have been provided wherever necessary.

4.1.1 Introduction

In the previous Chapter, we have described the current approach of elaborating *simple simultaneous statements*. Figure 3.3 illustrates the various components of the published Intermediate Code (I.C.) and the subsequent matrix load operation.

It has been observed that the matrix load operation has been performed by directly calling a function in the published Intermediate Code (I.C.) (Section 3.2.1). Even though this function completely describes the Characteristic Expression to be loaded into the matrix, with current elaboration methods CEs and sets of CEs can not be modified once published. The current approach publishes a static function, the terms and operators of

which are not accessible to the developer of the analog kernel. Since we have no access to the description of the CE itself, we are unable to identify situations which are conducive to reduction of CEs, much less actually perform the reduction on the CE or sets of CEs. Thus no equation reductions are possible, or, equivalently, we are forced to load the equations as translated from the VHDL-AMS model.

Figure 4.1 presents a flow graph for the new approach. It illustrates the creation of an *equation tree* for every *simple simultaneous statement* in the model. The following is an ordered description of the elaboration in the new approach -

1. The code generator of the front-end compiler publishes a function which creates an *equation tree* at the run-time for every *simple simultaneous statement* (SSS).
2. A component object is created for every SSS and its corresponding function is associated with the object.
3. The respective *equation tree* function is called during the creation of the component object and this returns the root of the *equation tree*.
4. The elaboration phase involves the formation of the set of Characteristic Expressions (CEs) by traversing the entire hierarchy of the model description. This translates into traversing every object instantiation for the various entities in the model and forming a list of all the CEs encountered.
5. Reduction algorithms are applied to reduce the elaborated set of Characteristic Expressions (CEs).
6. Each *equation tree* is traversed recursively starting from its *root node* to load the matrix.

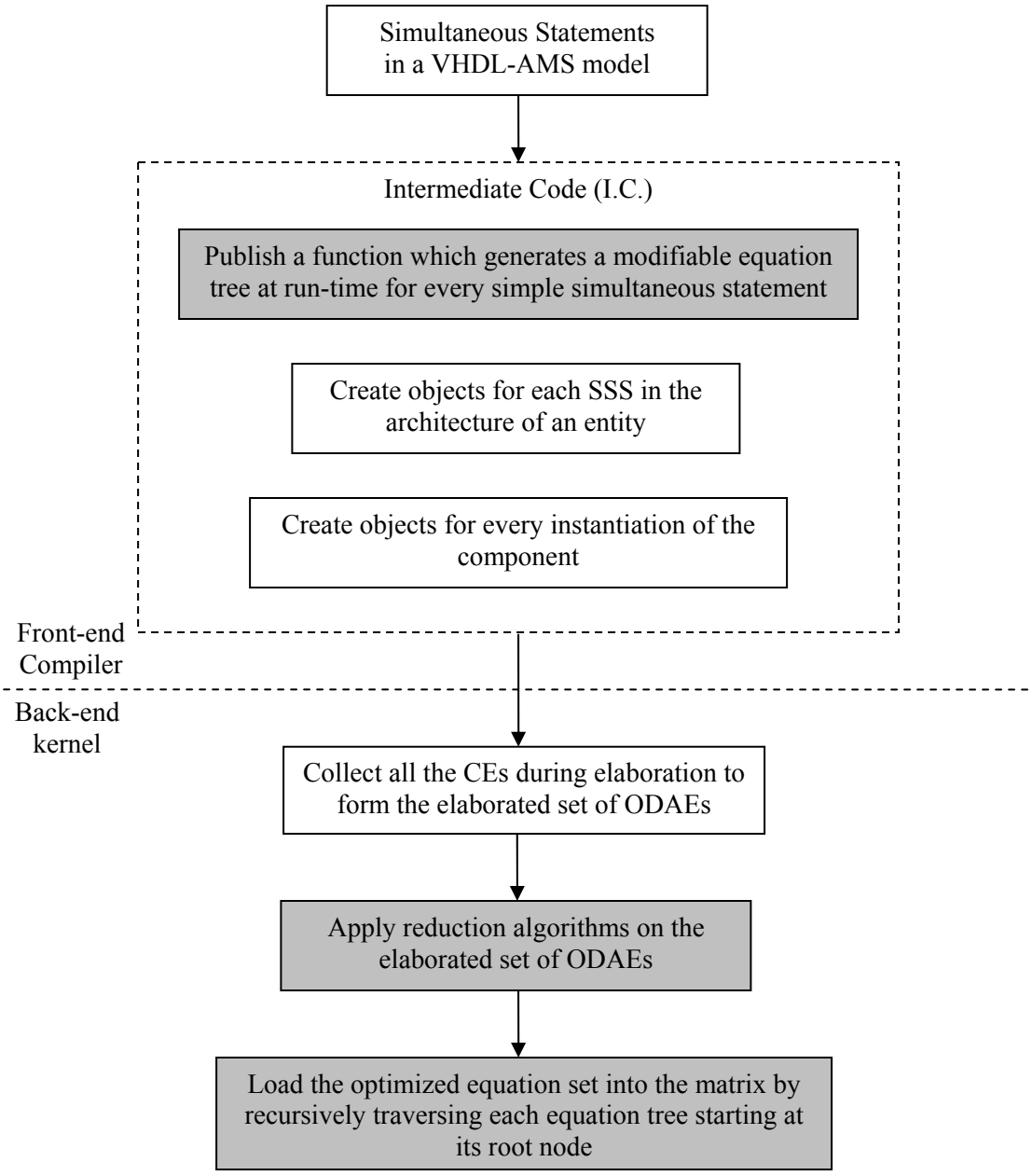

Figure 4.1: Description of elaboration and matrix load of SSS in the new approach. Shaded boxes indicate changed tasks relative those in the current approach (Figure 3.3).

Figure 4.2 presents a more detailed description of the data structures for SSS in the new approach. This perspective of the elaboration process, shows how the front-end

compiler, code generator, and the back-end kernel work in synchronism to achieve elaboration.

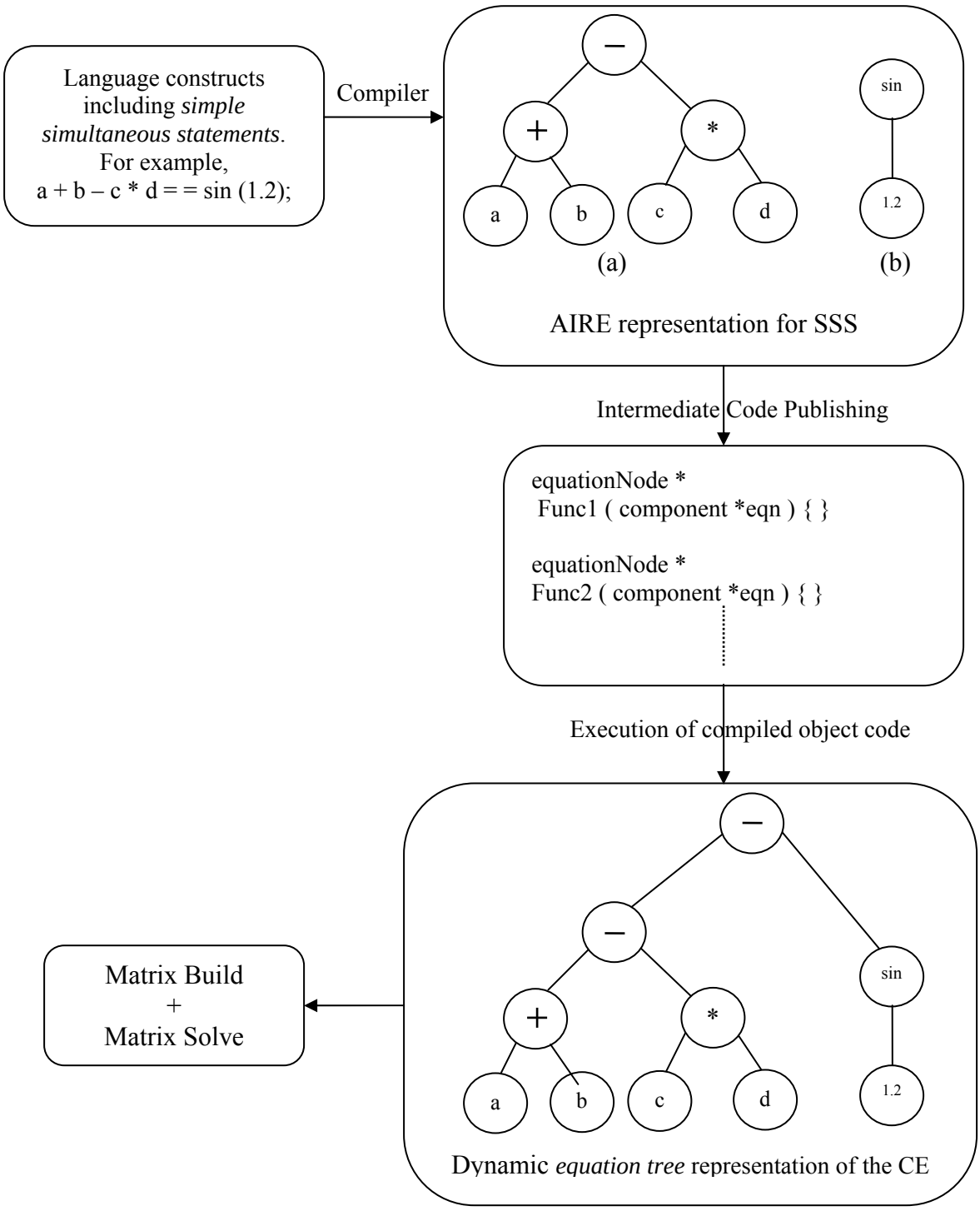

Figure 4.2: Flowchart showing the internal data structures during the elaboration of a SSS.

The front-end compiler is responsible for creating the Abstract Syntax Tree (AST) for the entire model description. The AIRE (Advanced Intermediate Representation with Extensibility) document describes the AST for all the constructs in VHDL-AMS. A *simple simultaneous statement* is stored as two expression trees, ('a' and 'b' in figure) as per the AIRE document [24]. The code generator publishes a function for each SSS which would generate the *equation tree* during execution of the compiled code. Note that the SSS is converted into its Characteristic Expression format during the code generation phase itself. This is described in more detail in the next section.

The Intermediate Code (I.C.) is compiled along with the back-end kernel libraries to form a final executable object (Section 2.1). The *equation tree* is creating dynamically during the execution of this final executable object. The creation of the equation tree occurs during the elaboration phase of the simulation.

The advantage of this new approach is that we have achieved a separation between the elaboration and the matrix load phase of the analog kernel. The matrix is not loaded based on the published Intermediate Code (I.C.), but is loaded using the root node of the *equation tree*. This novel approach gives us the capability to dynamically change the equation description. Thus, we name this new approach *dynamic elaboration*, and the Information Structure to achieve the same is called S3IS.

Since every component object keeps track of the root of its *equation tree*, we are able to modify the description of the *equation tree* itself. We describe the *equation tree* in further detail in the later section, when it would be clear as to how the root of the *equation tree* would remain the same, even when the contents of the equation are modified. The *equation tree* function in the Intermediate Code (I.C.) creates each of its

nodes dynamically on the heap. Therefore they are accessible at a later point of time, when they can be modified.

The elaborated set of CEs is designed as a *list* of characteristic expressions. This *list* of CEs is traversed during the matrix load phase. Thus, removing a CE from the *list* gives the effect of not loading the particular characteristic expression. Since the *equation tree* is created dynamically on a heap, we can evaluate conditions to create a complete *equation tree* or a sub-tree of the *equation tree*. Once the *equation tree* is created and its root node known, we can add the CE to the *list* of CEs. The above mentioned capabilities were simply not achievable with the previous Information Structure.

In this section, we have shown how the new approach satisfies the basic requirement of our problem statement – it supports the reduction of the elaborated set of Characteristic Expressions (CEs). We have also illustrated the modified elaboration methodology to take advantage of the new Information Structure. In the following section, we discuss the implementation of the data structure for *simple simultaneous statements*.

4.1.2 Implementation of the Data Structure

The requirements for the design of the data structure for *simple simultaneous statements* have been discussed in the earlier sections. We present the design in complete detail in this section.

The VHDL-AMS LRM defines a *simple simultaneous statement* as follows -

simple_simultaneous_statement ::=
[label :] simple_expression == simple_expression [tolerance_aspect] ;

where, the *label* and *tolerance_aspect* are optional fields. A *simple simultaneous statement* basically consists of a "left hand side" expression and a "right hand side" expression. We usually represent it as

$$lhs == rhs; \tag{4.1}$$

We realize that this translates into an equation which can be represented as

$$lhs - rhs == 0; \tag{4.2}$$

We call this equation the restructured *simple simultaneous statement*.

Thus, any *simple simultaneous statement* can be converted into the format as shown above. The advantage of using the above notation is that the equation has been converted into a format which always assumes that the effective right hand side expression is "0". The "*lhs – rhs*" in specific, is referred to as the Characteristic Expression (CE) for the particular SSS. The AIRE representation of the *simple simultaneous statement* (SSS) allows the code generator of the front-end compiler, SAVANT[1] to publish the SSS in its Characteristic Expression (CE) format [28,29]. Therefore, we are left with the problem of finding a data structure which allows us to describe the effective "left hand side" expression or Characteristic Expression (CE) in the back-end of the simulator. This task also involves the design of an efficient Intermediate Code (I.C.) to create the data structure during run-time. As described previously, the run-time in a compiled mixed-signal simulator refers to the execution of the final executable object.

We introduce the design of an *equation tree* to solve the above stated problem. An *equation tree* allows us to effectively describe a SSS, and also to create a CE which can be traversed by the back-end kernel to load the matrix.

[1]Standard Analyzer of VHDL Applications for Next Generation Technology

The *equation tree* consists of *nodes*, each of which is completely described by a (*attribute*, *value*, *parent node*) tuple. A *node* in an *equation tree* represents either an operator or a term in the effective "left hand side" expression of the Equation 4.2. The *attribute* of a *node* describes the type of the *node*. A *node* can be described with any one of the following *attributes* -

- Quantity
- Number
- Operator
- Function
- Time

Each *node* is referred to based on its *attribute*. For example, a *node* whose *attribute* is 'Quantity' is called a quantity node. The *value* of a *node* derives its meaning based on the *attribute*. The following definitions are valid for the *value* of a node -

- The *value* of a quantity node is its index in the particular CE.
- The *value* of a number node is the real value of the number itself.
- The *value* of an operator node indicates the type of operator. For example, a *value* '1' indicates the addition operator.
- The *value* of a function node is null.
- The *value* of a time node is null.

The construction of such an *equation tree* requires that each *node* identify its *parent node*. The top-most *node* in such an *equation tree* would be the subtraction operator as defined in equation 4.2. This top-most node is defined to as the *root node* of

the equation. The *equation tree* can be now defined as a tree whose recursive traversal starting from the *root node* leads to the unfolding of the Characteristic Expression (CE).

Consider, for example, the VHDL-AMS *simple simultaneous statement* (SSS) -

$$x + y = = (16.0 – z) / 4.0; \tag{4.3}$$

where, x, y, and z are real type quantities. Assume we desire to construct the *equation tree* for the above statement. The following sequence of steps are taken to ensure the creation of the equation tree for every SSS -

1. Convert the SSS into its CE format during the code generation phase of the front-end compiler. Thus, at the beginning of code generation phase, the SSS is in the CE format as shown below.

$$(x + y) – (16.0 – z) / 4.0$$

2. Convert the CE into an *equation tree* by efficient publishing of Intermediate Code (I.C.). The I.C. is designed so as to generate the *equation tree* during simulation. This is done by creating each *node* in the *equation tree* and associating it with its *parent node*. The *equation tree* for the above CE is shown in Figure 4.3.

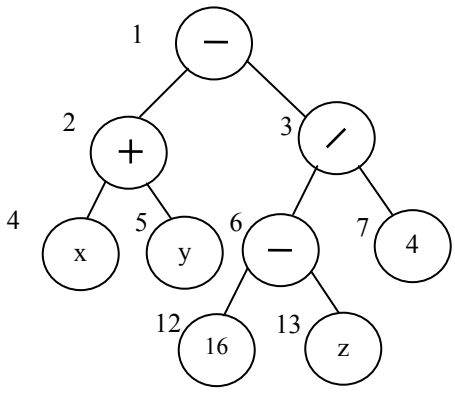

Figure 4.3: *Equation tree* for a simple simultaneous statement.

Nodes numbered 1, 2, 3 and 6 are operator nodes as and *nodes* numbered 4, 5, and 13 are quantity nodes with *values* of 1, 2 and 3 respectively (not shown in the figure). We obtain the *values* of quantity nodes based on their order in the equation. For example, Equation 4.3 can be represented as

$$(q[0] + q[1]) – (16.0 – q[2]) / 4.0$$

where, q is a *vector* of all the quantities occurring in the equation. *Nodes* numbered 7 and 12 in Figure 4.3 are number nodes as per our definition. Finally, *node* numbered 1 is the *root node* of the *equation tree* and is associated with its component object.

Our design of the equation tree is generic and extensible. We consider another example to further our explanation of the equation tree. Consider a SSS as shown below -

$$x * y = = 8.0 + pow (5.0, 10.0); \quad (4.4)$$

where, x and y are real quantities and ‘pow’ is a standard function. The above SSS is transformed during the beginning of the code generator phase into its CE format as shown below -

$$x * y – (8.0 + pow (5.0, 10.0))$$

The *equation tree* for equation 4.4 is shown below. *Nodes* numbered 1, 2, and 3 are operator nodes, *nodes* numbered 4 and 5 are quantity nodes, *nodes* numbered 6, 14, and 15 are number nodes, and *node* numbered 7 is a function node. During the recursive traversal of the *equation tree*, the sub-tree with *node* 7 as the root evaluates to a real number.

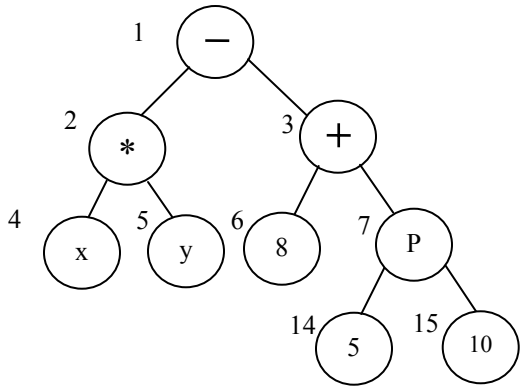

Figure 4.4: *Equation tree* for SSS in equation 4.4.

4.2 Elaboration Methodology

The S3IS Information Structure is designed to support elaboration of VHDL-AMS constructs. It provides a consistent methodology to perform efficient and functionally correct elaboration. SEAMS[1], a mixed-signal simulator developed at the University of Cincinnati implemented an elaboration methodology called RTEMS (Run-time Elaboration for VHDL-AMS) [11,25]. Our elaboration methodology is largely based on the techniques derived from RTEMS.

4.2.1 Concept of Islands

One of the novel approaches implemented in SEAMS to improve the performance of a distributed, compiled mixed-signal simulator is the notion of islands [11]. The idea behind the formation of analog islands is to reduce the size of the matrix to be solved. Disjoint sets of Characteristic Expressions (CEs), which are not dependent on the other set of CEs for the values of their unknowns, are created. Each set of such Characteristic

[1] Simulation Environment for VHDL-AMS

Expressions constitute an analog island. Each of these analog islands are solved using a separate matrix and hence are conducive to a distributed simulation paradigm.

Analysis of the formation of analog islands has led us to the observation that any reduction algorithm for the elaborated set of Characteristic Expressions needs to applied after the formation of islands. This is because any algorithm to reduce the complete set of CEs, would not be able to use an 'unconnected' CE in its algorithm. In other words, independent sets of CEs need independent reduction techniques to be applied. Thus, the elaborated set of CEs is separated into sets of CEs at the end of analog island formation. Each set of such CEs is a separate analog process handled by a separate processor in a distributed simulation paradigm.

4.2.2 Elaboration of Declarative Statements

An entity declaration in VHDL-AMS defines an interface to a component. Every entity declaration is represented as a C++ class derived from the pre-defined elaboration kernel class. The generic constants and signals in the port list of the entity become member objects of the entity class. Elaboration of architecture of an entity involves the elaboration of the declarative parts, and the concurrent and simultaneous statements in its body.

A detailed discussion of elaboration of the various statements in the architecture and entity of a VHDL-AMS model can be found in [9]. We limit our discussion here to the elaboration of terminal and quantity declarations as they may result in the creation of Characteristic Expressions (CEs). The quantity and terminal declarations result in creation of data members which store their attributes and any valid initialization values.

A terminal declaration defines the nature of a terminal. A quantity declaration is either a 'free quantity declaration' or a 'branch quantity declaration'. A branch quantity declaration is used to define quantities which either have an across aspect or through aspect associated with them. A 'free quantity declaration' does not associate the quantity with terminals. However, a 'branch quantity declaration' defines the across or through aspect of the quantity w.r.t its terminals.

Since the solver of our analog kernel uses Modified Nodal Analysis (MNA) to solve the matrix, we must include the equations formed by the application of Kirchoff's Current Law (KCL) [26]. These equations are actually a result of branch quantity declarations, and are referred to as the structural set of Characteristic Expressions.

Thus, the elaboration of branch quantity declarations also results in the creation of CEs, which constitute the structural set of CEs. We use a data structure to identify the CEs based on their declarations. We create contribution nodes for every terminal in the system. Each contribution node is associated with a *list* of through quantities. The following rules define the contributions of the through quantities -

- The contribution is set to +1 if the terminal is the positive terminal of the through quantity.
- The contribution is set to -1 if the terminal is the negative terminal of the through quantity.

The contribution stamp for each contribution node (terminal) represents a CE in the structural set of CEs. These stamps are loaded into the matrix at the beginning of each matrix load operation. Section 3.1.1 has presented a method to optimize the matrix build phase for the structural set of CEs.

4.2.3 Elaboration of Simultaneous Statements

Systems described using VHDL-AMS can be broadly classified into conserved and non-conserved systems. Conserved systems are those which follow the conservation laws. For example, circuit systems follow the Kirchoff's Current Law (KCL). Non-conserved systems are those that do not follow any laws of conservation. Their behavior is completely described using the *simultaneous statements*, contrary to the conserved systems which use both *simultaneous statements* and the association of terminals and quantities to completely define the system behavior.

The elaboration technique designed and implemented in SIERRA2 is valid for both conserved and non-conserved systems. The different ODAEs as recognized by SIERRA2 analog simulation kernel are as follows –

- **Free equation** – If a *simple simultaneous statement* consists of at least one free quantity (e.g. distance), then the equation is classified as free equation.
- **Branch equation** – If the equation is not a free equation, it is classified as a branch equation. The *simultaneous statement* would include one or more branch quantities (e.g. voltage) in its description.

The simulation kernel needs to evaluate each characteristic expression during every iteration in the simulation cycle and as many times as needed. Thus, elaboration must support such evaluations in a simple and efficient manner. All Characteristic Expressions (CEs) formed from the *simple simultaneous statements* constitute a *list* of CEs called as the explicit set of CEs.

Support for Differential Quantities:

In VHDL-AMS *Q'dot* represents the derivate of any quantity Q with respect to time. Numerical integration is performed to convert a differential equation into an algebraic equation. Before performing the numerical integration, the ODAE is transformed into an equivalent set of simultaneous statements. "Every quantity in a simultaneous statement of the form *Q'dot* is replaced by a new quantity IQ (Implicit Quantity) in that statement and a new simultaneous statement of the form IQ = = Q'dot is generated implicitly" [2,11]. Thus, the simultaneous statement with the *Q'dot* replaced, is like any other *simple simultaneous statement* and hence may be implemented with our data structure.

The conversion of the implicit equation into an algebraic equation can now be done using the trapezoidal numerical integration method. The application of the method on the implicit equation gives us a template equation which is loaded at every timepoint during the simulation of the model. But, note that the differential quantity cannot be modified to support optimization. This is a limitation to our design. In other words, the implementation of the numerical method prevents us from modifying the implicit equation. Thus the complete set of CEs includes a *list* of implicit equations.

Support for Conditional Simultaneous Statements:

The design of the elaboration methodology includes the elaboration of *conditional simultaneous statements* (CSS). *Simultaneous if statements* and *simultaneous case statements* together constitute *conditional simultaneous statements* (CSS). The implementation of CSS requires maintaining a control structure for the activation of one

or more *simple simultaneous statements* depending on what the condition evaluates to at the run-time [3]. This would need us to maintain a *true list* and a *false list* for every CSS in the elaborated set of CEs. The designed data structure gives scope for the implementation of *conditional simultaneous statements*. Thus, we create a *tree* of nodes which includes the explicit and implicit CEs along with conditions which branch off into a true and false *list* of nodes. Each node is either a CE or a boolean condition which evaluates to either *true* or *false*.

Figure 4.5 illustrates this elaboration data structure which facilitates the matrix load operation at every timepoint in the simulation cycle. Each node in the data structure is either a CE or a condition. This *tree* is traversed to find the transient set of CEs to be loaded in the matrix to solve for the unknowns.

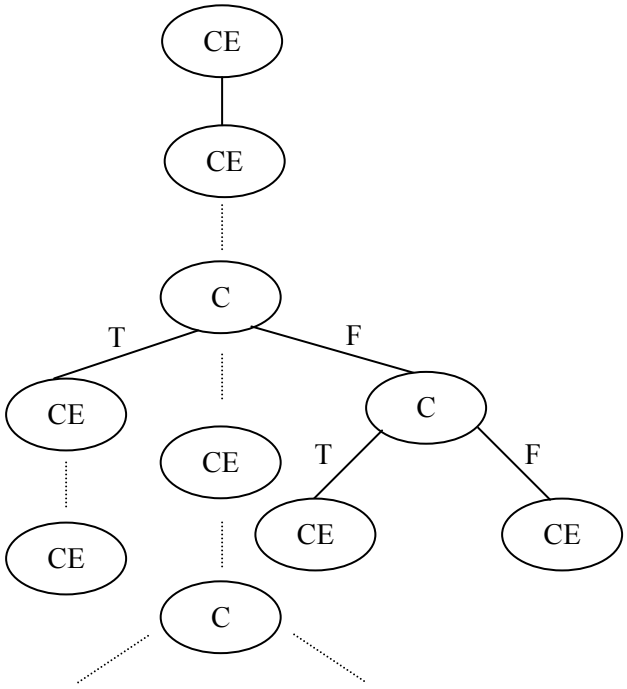

Figure 4.5: Elaboration data structure to support conditional simultaneous statements. In the figure, CE indicates Characteristic Expression, C indicates Condition, and T and F represent *true* and *false* conditions respectively.

Support for Discontinuities in the model:

Break statements are used in VHDL-AMS to model discontinuities. They cause a break in the transient simulation which results in finding a new DC operating point for the set of current set of equations at each discontinuity. A break statement specifies the condition which results in a discontinuity. It also contains the quantities and the new values that must be assigned to them when a discontinuity occurs. The break set consists of a linear *list* of quantities and the expressions which result in a new value for the quantity in the event of a discontinuity.

The elaboration of a break statement results in the creation of a boolean signal corresponding to the break condition and this signal is placed in the sensitivity list of the analog process. During simulation, when the condition for a particular break statement becomes true, discontinuity is reported to the analog kernel by the activation of the break signal. Each time a discontinuity occurs, the structural set of CEs and the current explicit set of CEs are loaded into the matrix and the DC operating point is evaluated. Transient simulation continues after the evaluation of the DC operating point. Thus, the designed Information Structure provides support for the evaluation of discontinuities.

4.3 Matrix Load Operation

In this section, we describe the use of the elaboration Information Structure for the matrix load operation. We describe a new approach to load a characteristic expression into the matrix of a sparse matrix solver. This is followed by a description of the entire elaboration Information Structure to enable the matrix load of all the characteristic expressions of a model.

4.3.1 Matrix Load of a Characteristic Expression

The matrix load operation is performed for the entire elaborated set of Characteristic Expressions (CEs) at each timepoint in the simulation cycle. Each CE is represented by its component object, and each component object calls its associated *equation tree* function to dynamically create the *equation tree* during elaboration.

The flow graph for this methodology is shown in Figure 4.1. The *equation tree* function returns the *root node* of the *equation tree* which is then stored as a member of the component object. During the matrix load operation, each CE is recursively traversed starting from its *root node* and the matrix is loaded. This recursive traversal re-creates the characteristic expression. The use of an *equation tree* also ensures that the rules of precedence are followed in the evaluation of the characteristic expression.

As has been mentioned in the previous sections, the primary advantage of our new approach is that the elaborated set of CEs may be reduced before the matrix load operation. This is now evident from the fact that we do not change the association of the *root node* of the *equation tree* with the component object for the equation.

4.3.2 Elaboration Information Structure for Matrix Load

The matrix load operation is said to complete when all the current set of Characteristic Expressions (CEs) have been loaded into the matrix of a sparse matrix solver. This would require the right set of CEs to be loaded after selection from the complete elaborated set of CEs.

Figure 4.6 presents a summary of the data structures, which support the matrix load operation. It primarily consists of -

- A list of pointers to Quantity objects, which completely describe the attributes of the quantities occurring in the system.
- A list of pointers to Terminal objects, which completely described the attributes of the terminals occurring in the system.
- A data structure to support the structural set of CEs. These represent the conservative equations of the system. A *list* of Contribution Nodes (CN) is created, one contribution node per Terminal. Each CN is associated with a *list* of Quantity Nodes (QN) as described in Section 4.2.2. Each QN also holds a value 'v', where v belongs to the set {-1, +1}. Thus, each CN represents a CE whose stamp is directly loaded into the matrix by using the 'values' in its *list* of QNs.
- A data structure to support the explicit set of CEs. This data structure has already been described in Section 4.3.3. The *tree* representing all explicit CEs is traversed and each CE is loaded into the matrix. Some of the nodes in the *tree* are condition nodes and would lead to a different *sub-tree* based on the evaluation of the condition at run-time.
- A *list* of pointers to implicit equations which constitute the implicit set of CEs. Each pointer refers to an object of 'differential equation' (*IQ = IQ'dot*), which is created for every *'dot* quantity. The stamp for the 'differential equation' is loaded into the matrix as has been discussed in Section 4.3.3.
- A *list* of break sets which constitute the discontinuity augmentation set. Each break set comprises of one or more break elements. A break element consists of quantities and the expressions which result in a new value for the quantity in the event of a discontinuity.

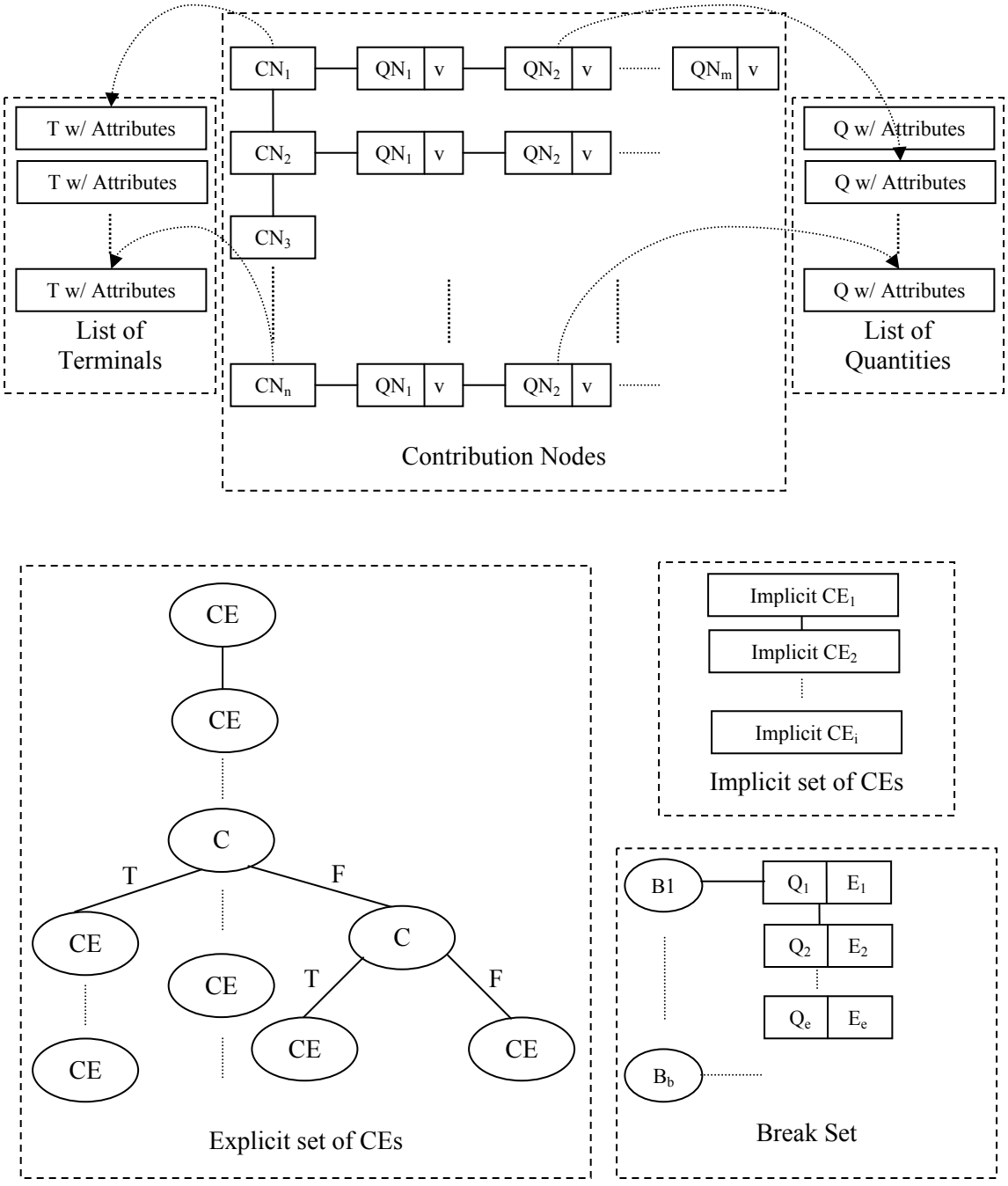

Figure 4.6: The elaboration Information Structure showing the different data structures which support matrix load operation. The following notations have been used in the above diagram.
CN-Contribution Node; QN-Quantity Node; v-value; Q-Quantity; T-Terminal; CE-Characteristic Expression; C-Condition; B-Break set; E-Expression

4.4 Merits and Limitations of the Information Structure

The following are the merits of the above described design for the Information Structure -

- The Information Structure supports our elaboration methodology and is conducive to performing the matrix load operation as described above.
- The Information Structure allows for the addition and removal of ODAEs from the elaborated set of ODAEs, thus allowing the use reduction algorithms.
- The *equation tree* may be modified easily by creating new *nodes* dynamically. Modification operation includes, addition of a sub-tree, deletion of a sub-tree, re-arranging the existing *equation tree*, and compaction of a sub-tree, as required by the equation reduction algorithm.
- The association of the *root node* of the *equation tree* with the component object does not change.
- The *value* of a particular *node* can be changed as necessary. This is most required during modification of a Characteristic Expression (CE), when the index of quantities in the CE may change due to the removal or insertion of a new quantity into the quantities *vector*.
- The data structure is scalable and does not restrict the number of terms or operators in the SSS.
- Intermediate Code may be designed and published based on the SSS, to create the required data structure.

Limitations of the Design:

The following are the limitations of the design presented above -

- The differential quantity node may not be reduced or substituted in the current implementation of the simulator. This is because of the approach adopted in the implementation of the numerical integration method in the simulator.
- The data structure does not support user defined functions. This is not considered to be a major limitation as any user defined function may be expanded in the *simple simultaneous statement* itself.

4.5 Summary of the Chapter

This Chapter has introduced a new data structure for elaborating *simple simultaneous statements*. We have shown that this data structure is most useful in enabling reduction of the elaborated set of CEs. We have also described in detail the entire Information Structure to support dynamic elaboration and subsequent matrix load operation. We have thus satisfied all the requirements of the problem statement.

Chapter 5

Optimization Approach

The previous Chapter has described a new data structure for *simple simultaneous statements*, and a dynamic elaboration methodology. The unique advantage of this approach is that it enables reduction of the elaborated set of CEs. This was hitherto not possible due to the limitations of the previous data structure.

In this Chapter, we present an approach to reduce the elaborated set of CEs. This is presented as a proof of concept for the design of our Information Structure. We present the algorithm used to perform the reduction of the elaborated set of CEs.

5.1 Introduction

The optimization approaches that have been implemented so far have not attempted to reduce the elaborated set of CEs as this was a limitation of the previous data structure. The reduction algorithm for the elaborated set of CEs is meant to improve the performance of the simulator, the most important performance metric being the total simulation time.

The use of the data structure in reducing conserved equations is the subject of a companion research effort [27]. Thus, we illustrate the use of the data structure using only non-conserved equations. However, no rigor is lost by restricting our attention to non-conserved equations since the algorithms and results exercise the primary components of the structure.

5.1.1 Optimization of Conserved Systems

Conserved systems include Characteristic Expressions (CEs) derived from *simple simultaneous statements* as well as the association of terminals and quantities. In other words, the elaborated set of CEs consists of the explicit set of CEs, and the structural set of CEs.

The optimization approaches used in such systems can be classified as follows -

- **Type 1** – Elimination of Terminals in the model description. For example, a circuit or sub-circuit with a series of linear elements (like resistors, capacitors, etc.) can be reduced to a single linear element with equivalent value. This translates to reduction of the internal nodes or Terminals in a system. In terms of the input to the matrix, we are attempting to reduce the structural set of CEs (contributions of each Terminal) as well as the explicit set of CEs (branch equations).
- **Type 2** – Elimination of Quantities in the model description. For example, a circuit or sub-circuit with linear elements in parallel can be reduced to a single linear element with equivalent value. This translates into eliminating the branch quantities of the parallel branches in the model. This reduces the explicit set of

CEs (branch equations) and modifies the structural set of CEs (contribution equations).

The above two reduction approaches make use of the new S3IS Information Structure to reduce and modify the elaborated set of CEs. The algorithms and results of the optimization approach will be discussed by [27].

5.1.2 Optimization of Non-conserved Systems

In this section, we present substitution as a method of reduction of the elaborated set of CEs in a non-conserved system. We contend that substitution can be an effective optimization approach to reduce the elaborated set of CEs generated from a model description.

This approach attempts to reduce the total number of CEs that are loaded into the matrix. Since the reduction algorithm runs only once before the start of the matrix load operation and the matrix load and solve operations are iterative, we anticipate that we would achieve a net reduction in the total simulation time.

Figure 5.1 presents the reduction algorithm. The algorithm contains a number of terms which are described here-

- **Donor Expression:** This is the characteristic expression which is used for substitution.
- **Acceptor Expression:** This is the characteristic expression into which substitution takes place.

- **Donor Quantity:** This is the Quantity 'y' in a characteristic expression of the form "y – f(x)". This quantity is substituted in the rest of the characteristic expressions.
- **Donor copy node:** This refers to the root of the sub-tree which needs to be copied or substituted, into an acceptor expression tree.
- **Quantity index:** This is the index of the quantity in the characteristic expression as defined in Section 4.1.2.

Description of the Algorithm:

The idea behind the use of substitution as a method of reduction is that it is an effective method to reduce the number of unknowns in the system. The algorithm called '*reduce*' is shown in Figure 5.1. The input to the algorithm is the explicit set of CEs, and the output is a reduced set of CEs. The algorithm starts with the search for a donor quantity 'y' and the respective donor expression as defined above. The donor expression is then removed from the basic set of CEs. Substitution is achieved by walking the *equation tree* of every CE and replacing one or more *nodes* representing the donor quantity with its equivalent sub-tree. Thus each donor quantity *node* is substituted with a sub-tree representing the expression for the donor quantity in the donor expression. To ensure the correctness of the CE, we also need to modify the *vector* of quantities for each characteristic expression by removing the donor quantity and changing the *value* of the other quantity *nodes* in the modified *equation tree*. The algorithm '*reduce*' continues as long as the elaborated set of CEs consists of equations of the form "y = f(x)", which can be used as donor equations. The complexity analysis for the algorithm is provided in Appendix A.

```
Algorithm reduce
{
Inputs: The explicit set of characteristic expressions
Outputs: Reduced set of characteristic expressions

Process:
   Find a Donor Qty 'y';
   Remove the Donor Expression from basic set of CEs;
   for ∀ e ∈ CE Set
     if (donorQty ∈ e )
         Remove donor quantity from the vector of quantities;
         call WalkEquationTree (root node, Qty index);
     end if
   end for
   Remove Donor Qty from the list of quantities;
}
```

Figure 5.1: Algorithm for reducing the elaborated set of CEs in a non-conserved system.

The two important methods used in the above algorithm are described below-

- **WalkEquationTree:** The algorithm for this method is shown in Figure 5.2. This method is responsible for recursively traversing each acceptor *equation tree* to locate *nodes* which represent the donor quantity. Each of these donor quantity *nodes* is replaced by the sub-tree starting with the donor copy *node*. The creation of the sub-tree is achieved by the 'CreateSubTree' method. Once the sub-tree is dynamically created, it is connected to the acceptor *equation tree*. This method is also responsible for changing the *values* of the quantity *nodes* in the *equation tree* for the acceptor expression. The function of this algorithm can be summarized as follows – it recursively traverses the acceptor expression to achieve substitution by creating sub-trees wherever necessary.

```
Algorithm WalkEquationTree
{
Inputs: The current equation node during the traversal
        Value of the donor quantity node
Outputs: Modified acceptor equation tree

Process:
   if (root node != NULL)
       call WalkEquationTree ( current equation node → leftChild, Qty index);
       if ( current equation node is a 'quantity node' )
          if ( current equation node → value = Qty index )
             call CreateSubTree ( Donor copy node);
             Make parent and child connection for the root of this sub-tree;
          else if (current equation node → value > Qty index )
             Decrement the node's value;
          end if
       end if
       call WalkEquationTree(current equation node → rightChild, Qty index);
   end if
}
```

Figure 5.2: Algorithm for traversing and modifying the acceptor *equation tree*.

- **CreateSubTree:** Figure 5.3 describes the algorithm behind the implementation of this method. This method is responsible for creating a copy of the donor sub-tree to enable substitution of the donor quantity. We replace each occurrence of the donor quantity in the acceptor *equation tree* with a sub-tree starting from the donor copy *node*. This method allows us to dynamically create *nodes* using their (*attribute*, *value*, *parent node)* tuples. Thus the donor *equation tree* is recursively traversed starting from the donor copy *node* and new *nodes* are created dynamically. Any new quantities occurring in the acceptor expression as a result of substitution are added to the *vector* of quantities for that acceptor expression.

```
Algorithm CreateSubTree
{
Inputs: The current copy node
Outputs: A dynamic sub-tree representing the equivalent expression of the donor
         quantity

Process:
   if ( copy node != NULL)
      if (copy node is a quantity node )
         flag = Check if the quantity already exists in the acceptor equation;
         if ( flag = true )
           Assign new node value;
         else
           Add new quantity to the vector of quantities;
         end if
       end if
      Create new equation node dynamically;
      call CreateSubTree ( copy node → leftChild);
      call CreateSubTree ( copy node → rightChild);
   end if
}
```

Figure 5.3: Algorithm for dynamically creating a sub-tree for the substituted quantity.

Comparison with Compiler Optimization Techniques:

The optimization approach presented above is very similar to 'copy propagation', an optimization approach used in compiler code optimization. Copy propagation is a transformation that uses an assignment of the form 'y ← x', where 'x' and 'y' are some variables, and replaces all the following uses of the variable 'y' with uses of 'x', as long as there are no intermediate assignments for the variable 'y' [30]. However, our approach differs from copy propagation in the following ways-

- Since we are working with *simultaneous statements* we are not restricted by the redefinition of the variable 'y'. All occurrences of 'y' may be replaced by the *rhs* expression since our aim is to reduce a quantity.

- Our reduction approach does not restrict the substitution to just copy statements of the form "y = = x". Any *simultaneous statement* of the form "$y = = f(x_1,x_2,\ldots x_i)$", where y and x_i are quantities in the continuous time domain may be used for substitution as long as y is not used in its derivative form in any of the *simultaneous statements*.

Chapter 6

Experimental Results

This Chapter presents the data obtained from the various tests that were conducted on the current and previous mixed-signal simulators developed at the University of Cincinnati. We analyze the overhead introduced with the new approach and also compare the performance metrics with and without the use of the reduction algorithm.

We begin with a description of the various performance metrics used in the evaluation of the data structure and the optimization algorithm. This is followed by a description of the experimental set up. We present the results obtained using various plots, and conclude with a summary of the performance of the new approach.

6.1 Introduction

In the previous Chapter, we have presented substitution as a method of reduction for the elaborated set of CEs. In this section, we define the metrics necessary for evaluating the overhead of the data structure, as well as the performance of the reduction algorithm. The following metrics are most relevant to the results presented in the following sections-

- **Total Simulation Time (T_{sim})**: This is the total time taken for the completion of simulation.

$$T_{sim} = T_{elab} + T_{opt} + T_{ml} + T_{ms} + T_o \quad (5.1)$$

where, T_{elab} is the total elaboration time

T_{opt} is the time taken for the optimization algorithm

T_{ml} is the time taken matrix load operation

T_{ms} is the time taken for the matrix solve phase

T_o is the total output time

- **Elaboration time (T_{elab})**: This includes the time taken for elaboration of the language constructs and also the creation of necessary data structures to enable matrix load operation and reduction algorithms.
- **Optimization time (T_{opt})**: This is the time for which the optimization algorithm runs. An important metric is the percentage contribution of the optimization algorithm to the total simulation time.
- **Matrix Load time (T_{ml})**: This is the total time spent by the simulator in the matrix load phase of the analog kernel.
- **Matrix Solve time (T_{ms})**: This is the total time spent by the simulator in the matrix solve phase of the analog kernel.
- **Output time (T_o)**: Simulation involves the file output of the values of unknowns at every timepoint in the simulation. The output time of a simulator is the time spent in writing this output file.
- **Non-IO Simulation Time ($T_{n\text{-}sim}$)**: A reduction algorithm involves a reduction of unknowns in the system. Since fewer output values need to be recorded, it affects

the file output time of the simulator. Therefore, a more relevant performance metric is the total simulation time without the contribution of the output time.

$$T_{n\text{-}sim} = T_{sim} - T_o \tag{5.2}$$

- **Intermediate Code Size**: This metric measured in bytes represents the amount of the Intermediate Code published. Although the absolute value of the Intermediate Code size is not of any interest to us, the difference in the Intermediate Code size with and without the use of the new data structure is of interest to us.
- **Percentage Improvement**: Percentage improvement is used to measure the gain in performance for any metric. It is defined as follows-

$$\%improvement = \frac{\left(T_{normal} - T_{optimized}\right)}{T_{normal}} * 100 \tag{5.3}$$

where, T_{normal} is the time without the application of the optimization algorithm

$T_{optimized}$ is the time with the application of the optimization algorithm

6.2 Reduction of Performance Factors

In this section, we present statistical analysis to identify the significant factors relevant to the new data structure that affect the performance of compiled mixed-signal simulation. This analysis is commonly referred to as 2^k factor analysis for the identification of significant factors.

The factors which influence the performance include-

1. **Matrix size**: As the matrix size increases, there will be a significant increase in the time taken for the matrix load and solve phases of the continuous time kernel.

Our aim is to determine the significance of this factor after defining the levels for the factor.

2. **Average equation size:** We define the average equation size as the total number of nodes used to represent all the Characteristic Expressions (CEs) in the continuous time domain, divided by the number of CEs in the system. We expect an increase in the matrix load time with an increase in the number of nodes to be traversed in the matrix load operation. We define the levels for this factor and then determine its significance.

6.2.1 Model description

Since we are studying the effect of the new data structure, we consider systems which are represented using only free equations. In order to use real models, we have chosen to use circuit descriptions of a network of resistors, with the conservative equations written explicitly as free equations.

We consider two levels for each of the factors listed above. We consider matrix sizes of 20 and 400 as the two levels. To generate models of different average equation sizes, we consider a completely series circuit and a completely parallel circuit as shown in Figure 6.1. Circuits of the form shown in Figure 6.1 were generated keeping the number of unknowns same. The series circuit has a low average equation size and the parallel circuit has a high average equation size. Typically, in electronic circuits, the average number of nodes per equation ranges approximately between 5 and 10. For the model used here with 20 unknowns, the average equation size of 5.65 and 9.05 nodes[1]. Thus the two levels for the average equation size were chosen as 5.65 and 9.05.

[1] The average equation size range is an estimate only. Further study is needed.

VHDL-AMS model descriptions were written for the two circuits shown in Figure 6.1. A model generator was used to create multiple instantiations of the two base circuits. This enables us to increase the matrix size. Thus we have four models for running the 2^k factor experiments. Each of the four models was simulated five times so as to account for the variances introduced due to available computing power.

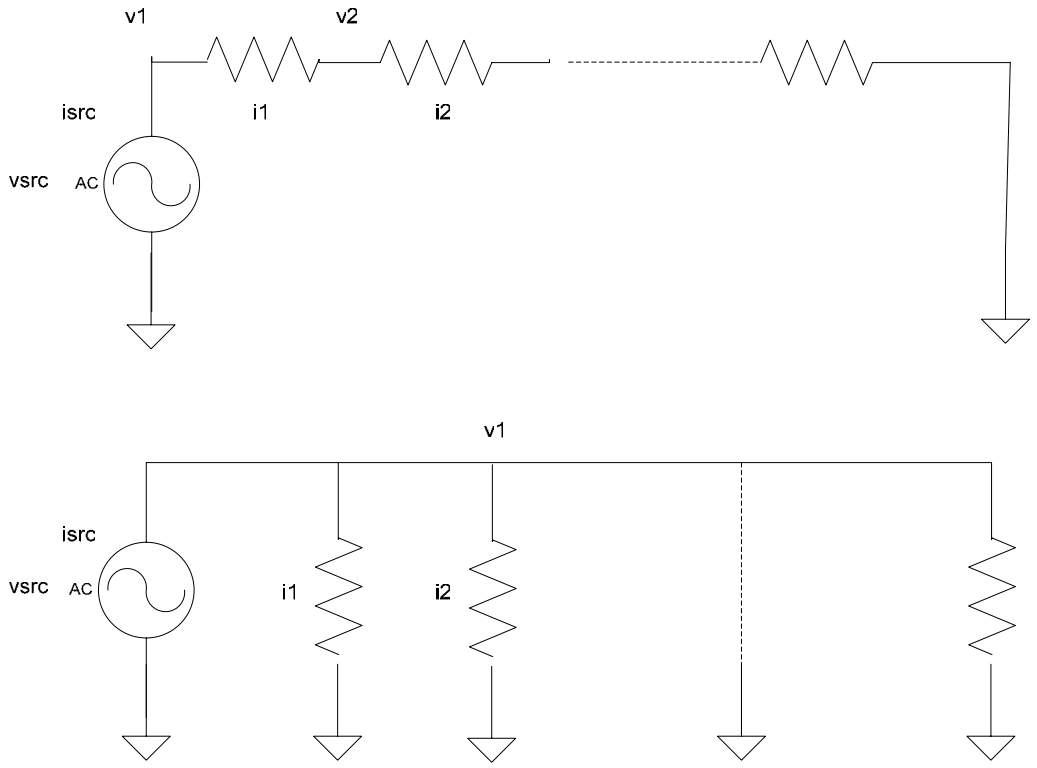

Figure 6.1: Series and parallel resistor circuits used in 2^k factor experiments.

6.2.2 Results

Figure 6.2 shows the 2^k factorial table obtained by collecting data from the simulations as described above. Next, we perform significance analysis of the data. We use the ANOVA process in the SAS system. ANOVA results for the three responses are shown in Figure 6.3, Figure 6.4 and Figure 6.5.

Simulation Run	Factors		Responses		
	Matrix Size	Average Equation Size	Matrix Load Time (s)	Matrix Solve Time (s)	Total Simulation Time (s)
1	-1	-1	20.435	0.446992	23.6931
2	-1	-1	20.5344	0.446294	23.7609
3	-1	-1	20.4527	0.444041	23.6782
4	-1	-1	20.4332	0.456495	23.6979
5	-1	-1	20.4056	0.440096	23.6462
6	-1	+1	24.6695	0.457713	27.9449
7	-1	+1	24.7685	0.450417	28.0057
8	-1	+1	24.7239	0.450692	27.9576
9	-1	+1	24.7487	0.454693	27.9901
10	-1	+1	24.7005	0.451381	27.9629
11	+1	-1	388.034	33.2913	507.219
12	+1	-1	388.109	33.39	507.155
13	+1	-1	389.267	33.4137	508.416
14	+1	-1	389.176	33.3666	508.474
15	+1	-1	392.801	33.2313	512.067
16	+1	+1	526.765	33.7411	653.588
17	+1	+1	525.954	33.7866	652.733
18	+1	+1	521.936	33.7856	648.882
19	+1	+1	534.374	34.0461	661.492
20	+1	+1	525.103	33.7058	652.003

Matrix Size
Level -1 20x20 matrix
Level +1 400x400 matrix

Average Equation Size
Level -1 5.65
Level +1 9.05

Figure 6.2: Raw results of 2^k factorial experiments with two factors

The ANOVA Procedure

Dependent Variable: MLT Matrix Load Time

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	2	973652.9800	486826.4900	372.17	<.0001
Error	17	22237.2823	1308.0754		
Corrected Total	19	995890.2623			

R-Square	Coeff Var	Root MSE	MLT Mean
0.977671	15.04655	36.16733	240.3696

Source	DF	Anova SS	Mean Square	F Value	Pr > F
MatSize	1	948583.0394	948583.0394	725.17	<.0001
AvgEqnSize	1	25069.9406	25069.9406	19.17	0.0004

Figure 6.3: ANOVA results for 2^k factor design for Matrix Load Time.

The ANOVA Procedure

Dependent Variable: MST Matrix Solve Time

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	2	5486.924515	2743.462258	126209	<.0001
Error	17	0.369537	0.021737		
Corrected Total	19	5487.294053			

R-Square	Coeff Var	Root MSE	MST Mean
0.999933	0.866618	0.147436	17.01285

Source	DF	Anova SS	Mean Square	F Value	Pr > F
MatSize	1	5486.635728	5486.635728	252404	<.0001
AvgEqnSize	1	0.288787	0.288787	13.29	0.0020

Figure 6.4: ANOVA results for 2^k factor design for Matrix Solve Time.

The ANOVA Procedure

Dependent Variable: ST Total Simulation Time

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	2	1570056.384	785028.192	536.32	<.0001
Error	17	24883.340	1463.726		
Corrected Total	19	1594939.724			

R-Square	Coeff Var	Root MSE	ST Mean
0.984399	12.60506	38.25867	303.5183

Source	DF	Anova SS	Mean Square	F Value	Pr > F
MatSize	1	1542174.464	1542174.464	1053.60	<.0001
AvgEqnSize	1	27881.920	27881.920	19.05	0.0004

Figure 6.5: ANOVA results for 2^k factor design for Total Simulation Time.

6.2.3 Summary of the 2^k Factor Analysis

As seen from the F-variates of the analysis, matrix size has the largest impact on matrix load time, matrix solve time and the total simulation time as it is 97.42%, 99.99%, and 98.22% significant respectively. The average equation size has minimal impact on the matrix load time. The dependence on average equation size is an expected behavior

due to the recursive traversal of the *equation tree*. However, the dependence is very low. It has almost no impact on the matrix solve time. Note that the total variation for total simulation time response is 1542174 + 27881 = 1570055. Thus, the variation of matrix size as a percent of total variation is 98.23%, whereas average equation size variation is only 1.77%. Since change in the average equation size has almost no affect on simulation time changes, we need not consider this factor further.

6.3 Analysis of the Overhead of the Data Structure

The new data structure and the reduction algorithm were implemented in SIERRA2, a compiled mixed-signal simulator developed at the University of Cincinnati. This section describes the tests that were conducted to measure the overhead introduced with the use of the new data structure.

We compare the relevant test data obtained from SIERRA2 with the respective data obtained from a previous compiled, mixed-signal simulator called SIERRA developed at the University of Cincinnati. The test environment for the same is shown in Figure 6.6.

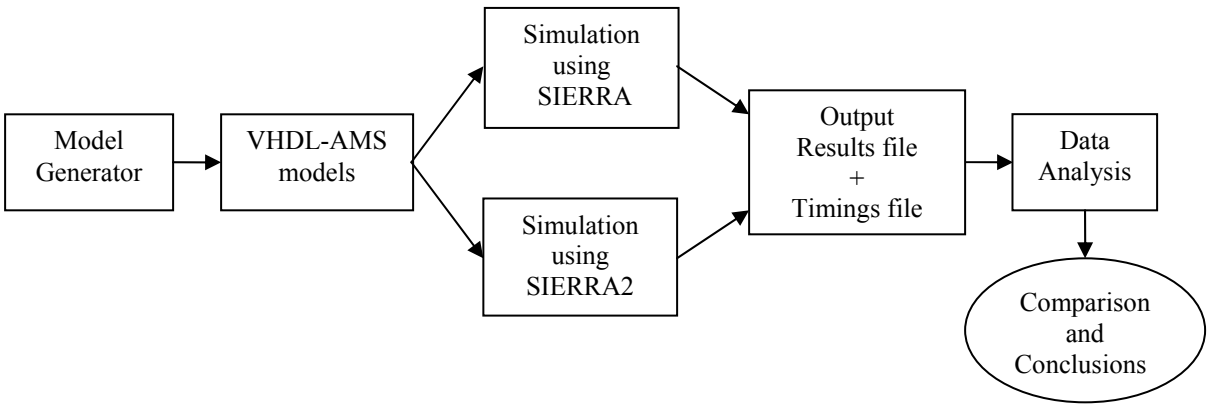

Figure 6.6: Test environment for evaluating the overhead of the new data structure.

The overhead incurred as a result of the introduction of the new data structure can be summarized as follows-

- Intermediate Code (I.C.) Size – Intermediate Code, which results in the creation of a dynamic *equation tre*e, uses high-level language constructs. This is expected to consume more memory in bytes when compared to the previous approach of publishing the equation itself.
- Elaboration Time – The *equation tree* is actually created at run-time during the elaboration phase of the analog kernel. This introduces an overhead in the elaboration time in the new methodology.
- Matrix Load Time – The recursive traversal of the *equation tree* during the matrix load is expected to be more time consuming than the previous approach. We analyze the overhead introduced in the matrix load phase as a result of the new data structure.
- Run-time memory requirement – Since the new elaboration methodology involves the creation of dynamic *equation trees*, we have a greater run-time memory usage. We present an analysis of the additional run-time memory used.

6.3.1 Model Description

The models used for this set of simulations were again generated using the model generator described in the previous section. The circuit shown in Figure 6.7 representing a network of resistors with 13 unknowns is taken as the base model. Multiple instances of the base circuit give us models of higher matrix sizes. Since SIERRA only accepts simultaneous statements of the form "*y* = = *rh*s", we generate models in the respective

format. The same model description is used as the input to both the simulators. Each model was simulated five times. The maximum variance for the data collected was 0.6%. Therefore, we have used average values in the plots provided below.

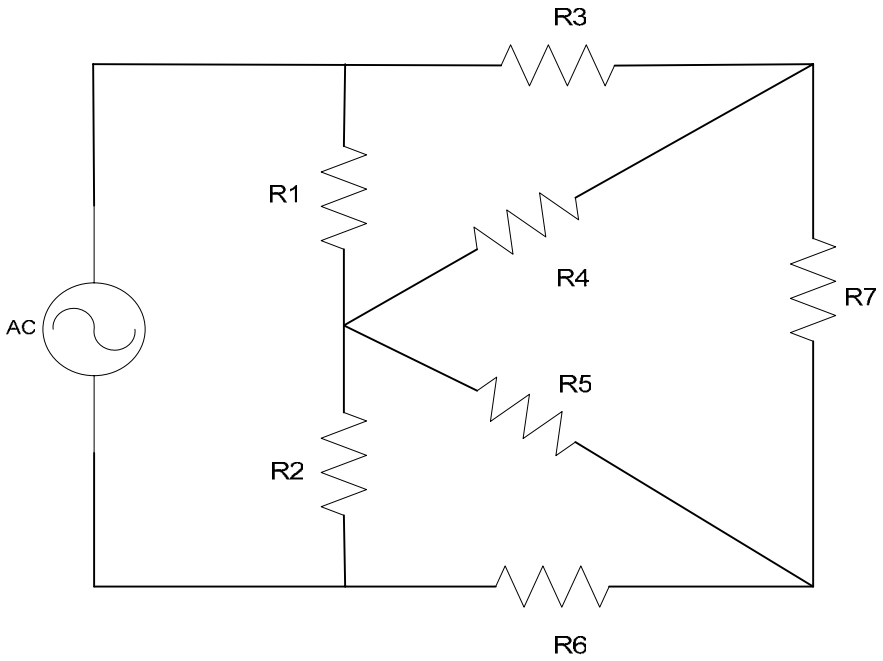

Figure 6.7: A network of resistors model (Model 1).

6.3.2 Results

This section presents the results obtained from the tests run on SIERRA and SIERRA2 to analyze the overhead introduced with the use of the new data structure.

Intermediate Code size:

Figure 6.8 shows a linear increase in the additional bytes of memory required as the matrix size increases. The plot shows the difference in the bytes of memory required by SIERRA and SIERRA2. As expected, SIERRA2 requires additional amount of memory in the Intermediate Code to have instructions which create the *equation tree*.

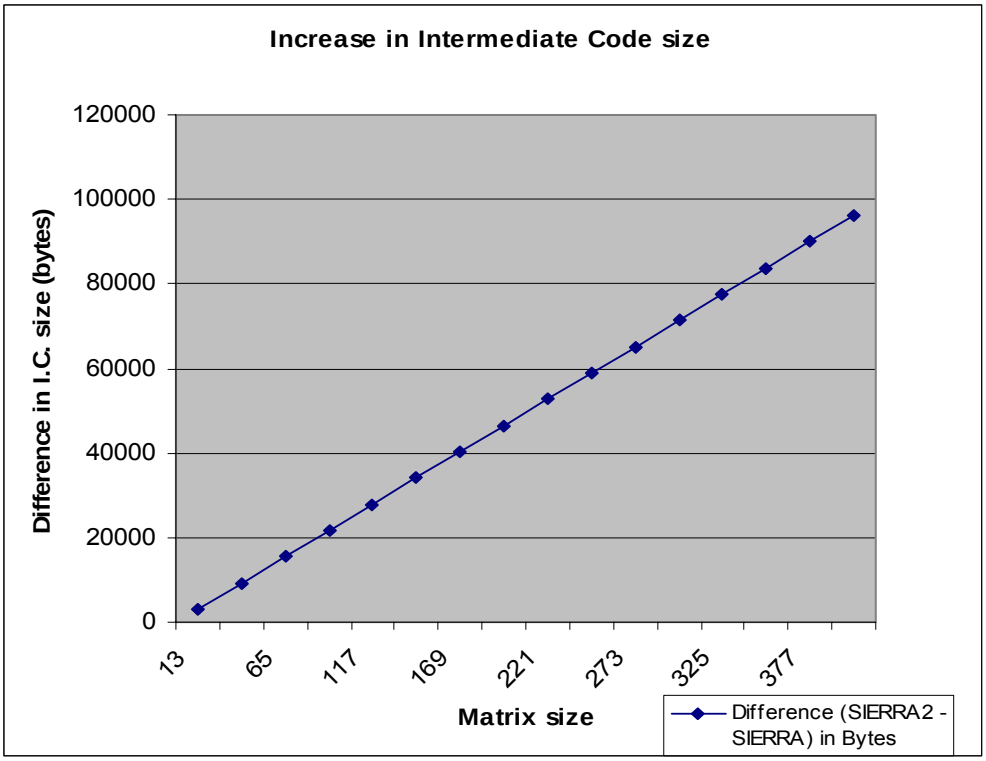

Figure 6.8: Overhead in Intermediate Code size as matrix size increases.

Elaboration Time:

Figure 6.9 shows an increase in the elaboration time as matrix size increases. With increasing size of matrix, we have more constructs to be elaborated, hence the greater elaboration time. We also observe that the elaboration time in SIERRA2 is greater than the elaboration time in SIERRA.

The percentage increase in the elaboration time is shown in Figure 6.10. It is observed that the percentage increase in the elaboration time varies from about 127% to 145% as matrix size is varied from 39 to 403. The plot also seems to indicate that elaboration time increases by a particular factor. This is because of the combination of two reasons-

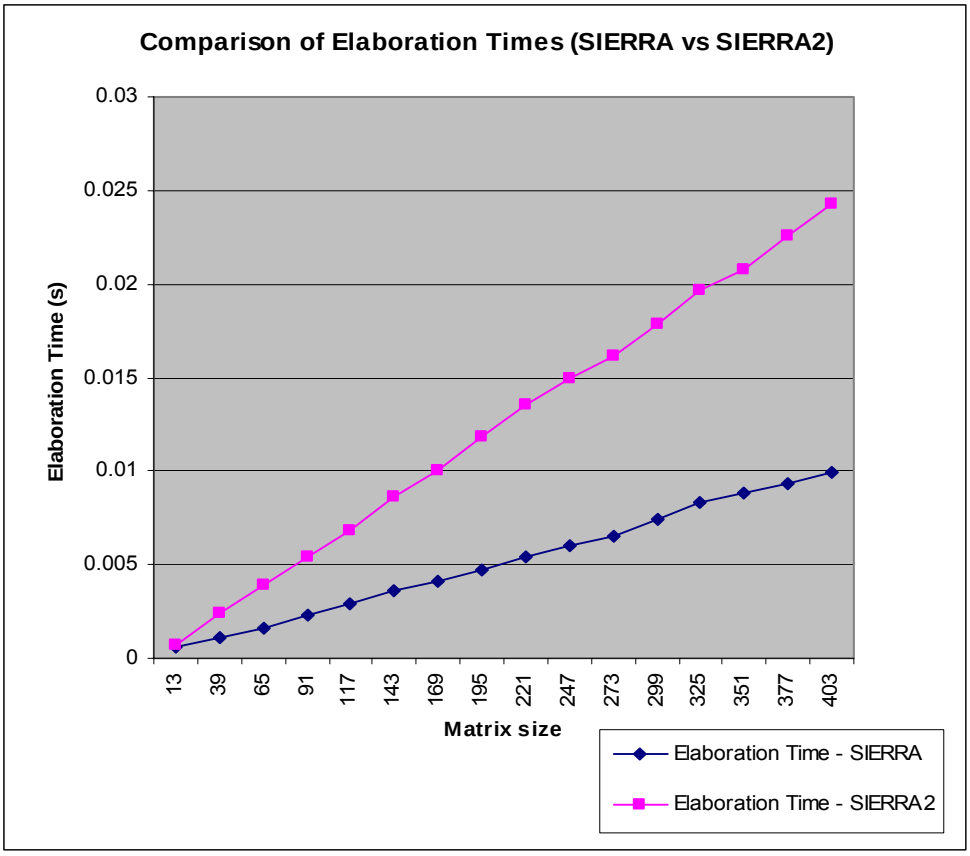

Figure 6.9: Comparison of Elaboration Time (SIERRA vs. SIERRA2).

- SIERRA2 involves the creation of dynamic *equations trees* during elaboration which increases the elaboration time for a particular matrix size.
- Changes in processes like class structure between SIERRA and SIERRA2 effects a factor of times increase in the elaboration time. This is more evident because of the low absolute values of elaboration time.

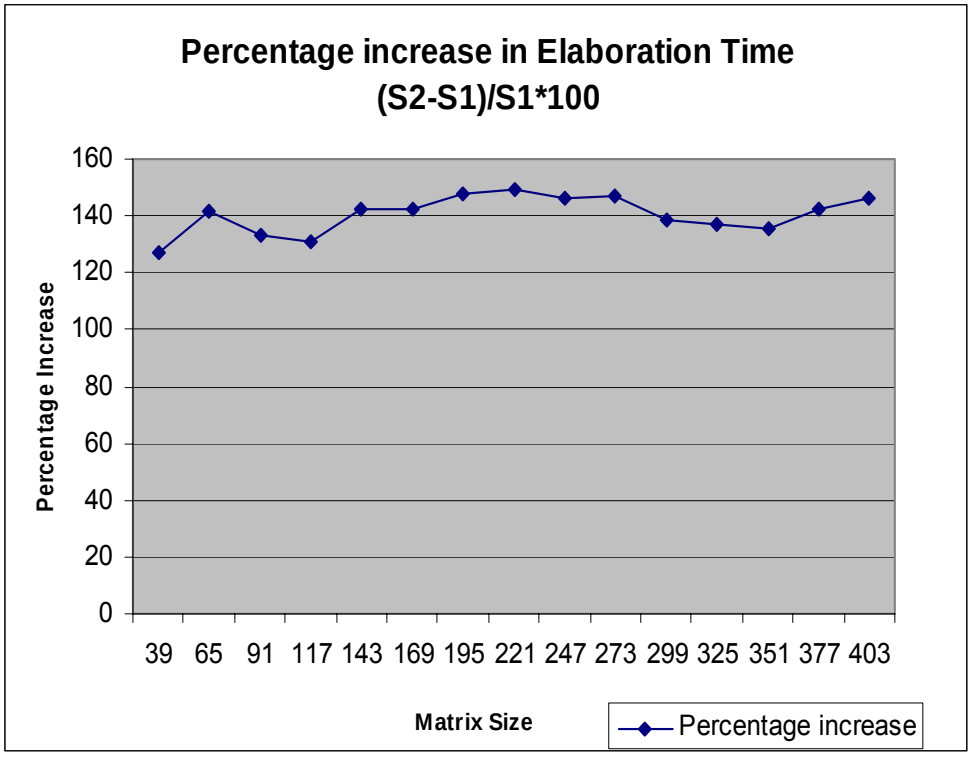

Figure 6.10: Percentage increase in Elaboration Time (SIERRA vs. SIERRA2).

Matrix Load Time:

We observe from Figure 6.11 that matrix load time in SIERRA2 is greater than the matrix load time for SIERRA. This is because of two reasons-

- SIERRA2 involves the recursive traversal of the *equation trees* for each iteration. SIERRA, however just calls a function describing an equation in the published I.C. to load the matrix.
- SIERRA2 has a provision to solve generalized equations of the form *"lhs = = rhs"*, whereas SIERRA can only solve equations of the form *"y = = rhs"*, where *lhs* and *rhs* are expressions and y is a quantity. There is an additional

cost in the matrix load operation due to the scope provided for loading the generalized equations.

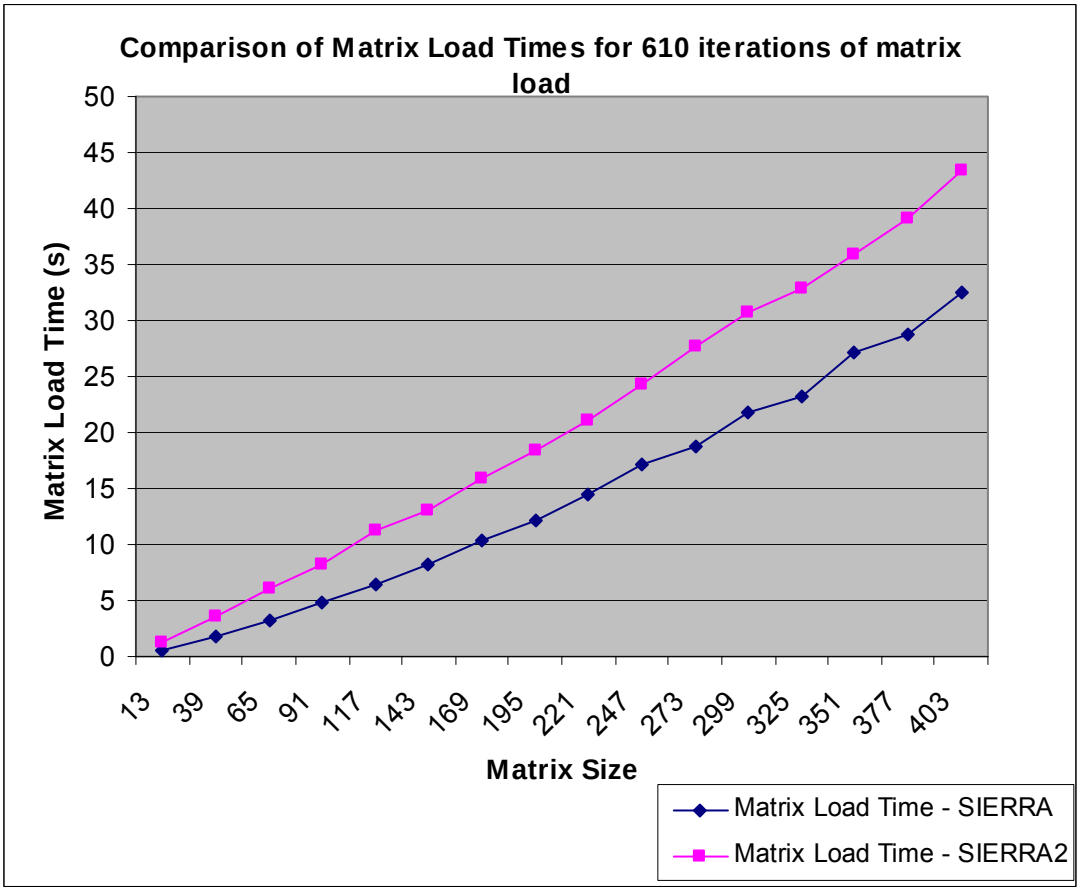

Figure 6.11: Comparison of Matrix Load Time (SIERRA vs. SIERRA2).

Figure 6.12 shows the percentage increase in the matrix load time as the matrix size increases. It is observed that the percentage increase in the matrix load time decreases as the matrix size increases. This indicates that as the matrix size increases, the recursive traversal of the *equation tree* is not the dominant factor in the matrix load time. This is a positive point as it indicates a reduced overhead with increasing matrix size.

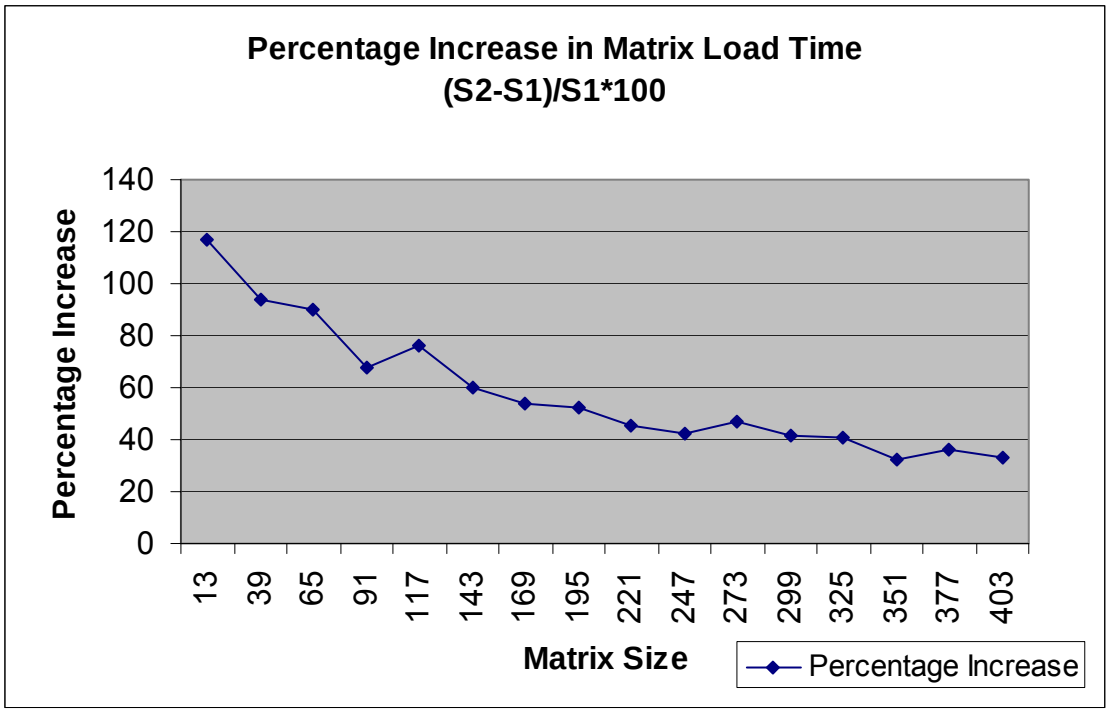

Figure 6.12: Percentage increase in Matrix Load Time (SIERRA vs. SIERRA2).

Run-time memory requirement:

The new approach involves the creation of *equation trees* during run-time for all the CEs occurring in the continuous time domain. This adds to the run-time memory requirement of the application. In this section, we present an analysis of the memory requirement of the new data structure.

Let,

$p_i \leftarrow$ number of operands equation 'i'.

$r_i \leftarrow$ number of operators in equation 'i'.

$n \leftarrow$ total number of explicit equations in the system.

$S_i \leftarrow$ total number of nodes in the Characteristic Expression for equation 'i'.

then,

$S_i = p_i + r_i + 1$ (the additional node is for the '-' in the CE)

The kernel uses the objects of the "equationNode" class to create operator and operand nodes. All our simulations have been run on a Sun Sparc machine on which

Sizeof (equationNode) = 36 bytes

Sizeof (equationNode *) = 4 bytes

Since *equation trees* are created dynamically, we have memory space allocated for the object as well as the pointers used to alias the object.

Therefore,

Additional memory required for equation 'i' = $[(p_i + r_i + 1)*36 + (x_i)*4]$

where, x_i is the number of pointers aliasing to the root of the *equation tree*. It is a constant for any equation.

Therefore,

Total additional run-time memory required = $36 * \sum_{i=1}^{n}(1 + p_i + r_i) + 4 * \sum_{i=1}^{n} x_i$

It should be noted that both the terms in the above equation are of the *O(n)*. Thus the memory requirements for the data structure scale linearly with increasing matrix size.

6.3.3 Summary of the Data Structure Overhead Analysis

We conclude that the new data structure introduces an overhead in the elaboration time, matrix load time and the run-time memory requirement. The matrix solve time is not dependent on the data structure and hence it is not analyzed. The overhead in the size of the published Intermediate Code in form of additional bytes is not a concern in current day computers. We hope that the reduction approach implemented as a result of the

introduction of the new data structure would give us enough performance improvement to substantiate the overhead incurred in the simulation time.

6.4 Analysis of the Performance of the Data Structure

The test environment for evaluating the performance of the data structure is shown in Figure 6.13. An example reduction algorithm which makes use of the new data structure has been presented in Section 5.1.2. The models are simulated using a command-line switch to include or not include the reduction algorithm during execution. The simulator has embedded code to time the various stages of simulation. This timing information is output by the simulator along with the results file. In this section, we present the data obtained from the above described analysis.

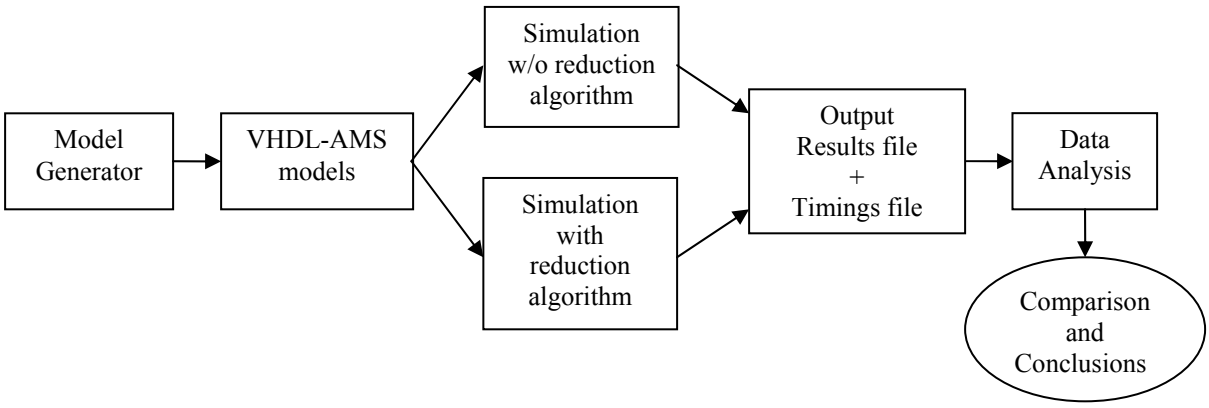

Figure 6.13: Test environment for evaluating the reduction algorithm.

6.4.1 Model Description:

Three different circuit models have been used to analyze the performance of the optimization algorithm. The first model is the network of resistors circuit driven by a time-varying source as shown in Figure 6.7. The second circuit used to validate the

performance of the optimization algorithm is the phono pre-amplifier circuit shown in Figure 6.14. The third model is that of an active high-pass filter as shown in Figure 6.15. The model generator is responsible for generating models of increasing matrix size. In order to compare the various performance metrics of the simulator for increasing matrix sizes, we keep the reduction factor of a particular model constant. In other words, a constant percentage of CEs are reduced for all matrix sizes. Since equations of the form *"y = = rhs"* are reduced, we generate a constant percentage of simultaneous statements in the particular form.

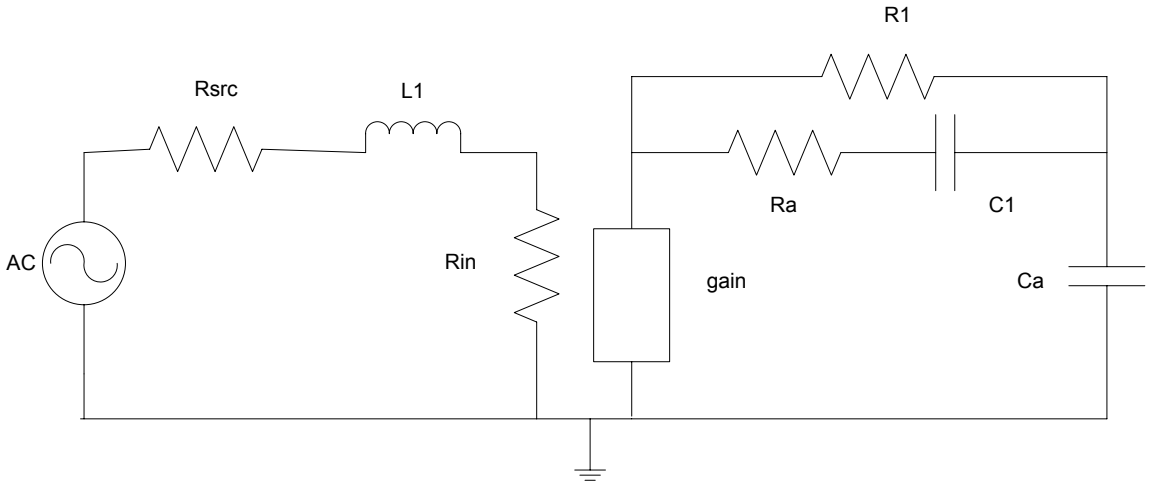

Figure 6.14: A phono pre-amplifier circuit (Model 2).

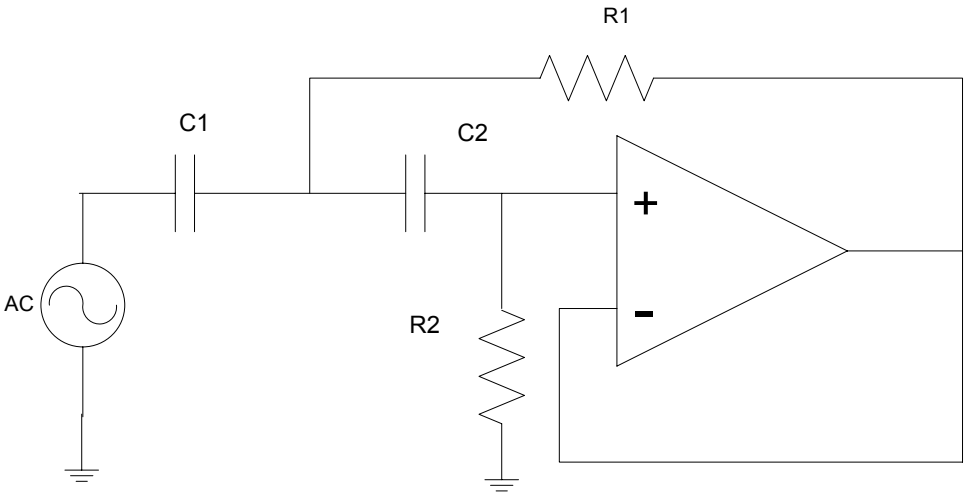

Figure 6.15: Active high-pass filter circuit (Model 3).

6.4.2 Results

In this section, we present the plots to illustrate the effectiveness of the reduction algorithm. The results presented below were obtained with a simulation time of 1μs. We analyze the important performance metrics of the simulator.

I. Results for Model 1

The results obtained from the network of resistors model shown in Figure 6.7 are presented below. Three unknowns were reduced from a total of 13 unknowns in the base model. Thus, the reduction factor for the model is maintained 23%.

Non-IO Simulation Time:

Figure 6.16 shows that we have indeed achieved a reduction in the Non-IO simulation time for models of various matrix sizes. Figure 6.17 shows an increasing percentage improvement in the Non-IO simulation time. Table 6.1 shows the confidence intervals for the difference of the means of the Non-IO simulation time in the normal and optimized modes. It must be noted that the intervals do not include zero at 99% confidence level and thus, for all 'n', percentage improvement is non-zero (e.g., real differences occur at 99% confidence).

It must be noted that the improvement in Non-IO simulation time is mainly due to the reduced matrix load and solve times after optimization. At higher matrix sizes, we have greater percentage improvement since for a constant reduction factor, there is a greater gain at higher matrix sizes.

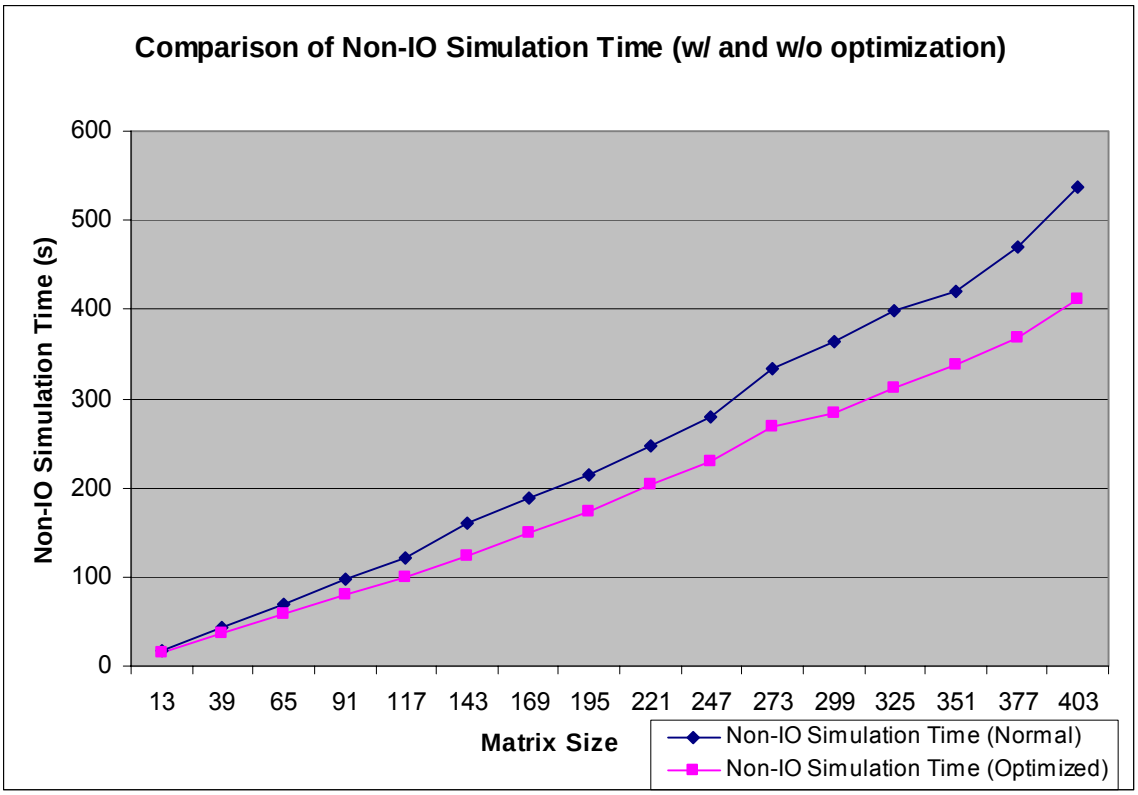

Figure 6.16: Comparison of Non-IO Simulation Time for Normal and Optimized modes.

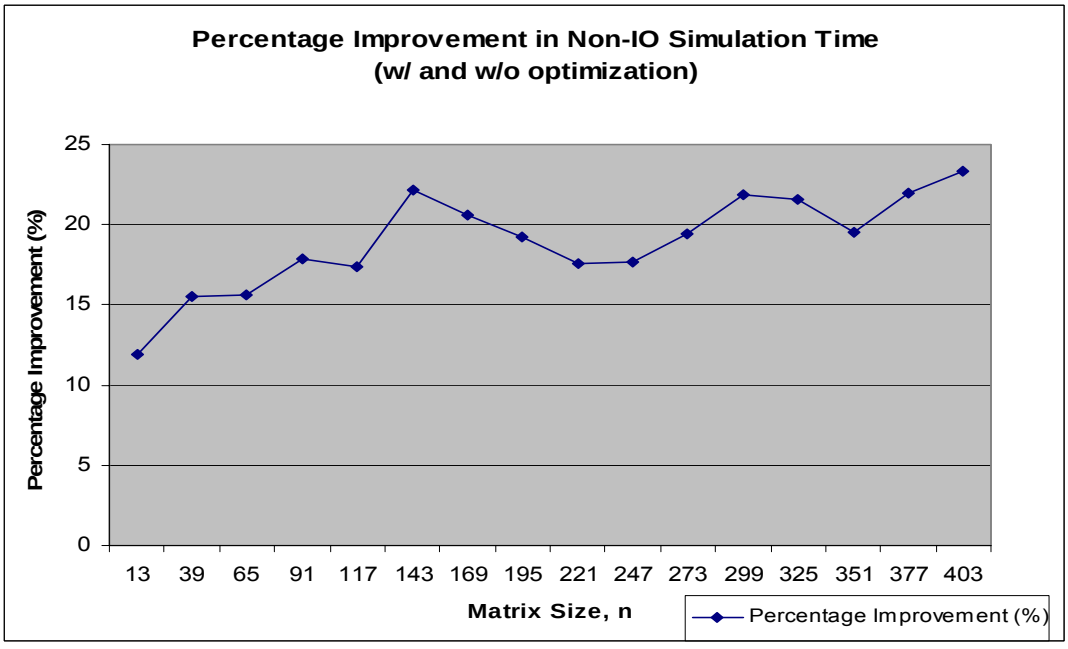

Figure 6.17: Percentage improvement in Non-IO Simulation Time.

Matrix Size	Mean Difference of Non-IO Simulation Time	Confidence Interval for the Mean	Confidence Level %
13	2.039	(1.958,2.119)	99
39	6.609	(6.548,6.670)	99
65	10.951	(10.483,11.418)	99
91	17.378	(17.085,17.669)	99
117	21.046	(20.906,21.185)	99
143	35.463	(34.909,36.016)	99
169	38.684	(38.184,39.184)	99
195	41.225	(40.827,41.623)	99
221	43.536	(42.597,44.474)	99
247	49.584	(49.317,49.850)	99
273	65.022	(63.904,66.140)	99
299	79.825	(79.458,80.191)	99
325	86.093	(85.708,86.477)	99
351	82.074	(81.111,83.037)	99
377	103.244	(100.310,106.176)	99
403	125.252	(121.451,129.052)	99

Table 6.1: Confidence Interval for mean difference of the Non-IO Simulation Time.

Matrix Load Time:

The data collected shows that matrix load time constitutes a very large percentage of the total simulation time. Figure 6.18 shows the absolute improvement achieved in the matrix load time as matrix size increases.

It is observed that the matrix load time after the application of the optimization algorithm is always less than the matrix load time without the optimization algorithm. This proves our initial assumption that the size of the elaborated set of CEs has a definite influence on the time spent in the matrix load phase of the simulator. Figure 6.19 shows a plot of the percentage improvement in matrix load time with increasing matrix size. The percentage improvement is higher at a greater matrix size due to the effect of constant reduction factor. Keeping the reduction factor constant implies that the higher order matrices are reduced by a greater number of unknowns. The plot proves that we have achieved a reduction in matrix load time by reducing the elaborated set of CEs.

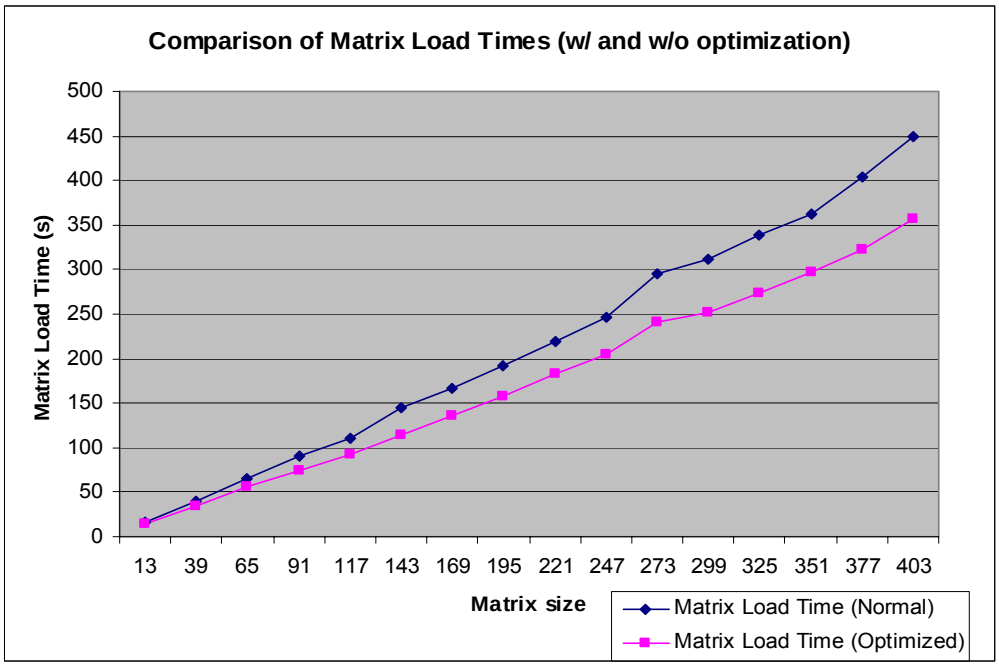

Figure 6.18: Comparison of Matrix Load Time for Normal and Optimized modes.

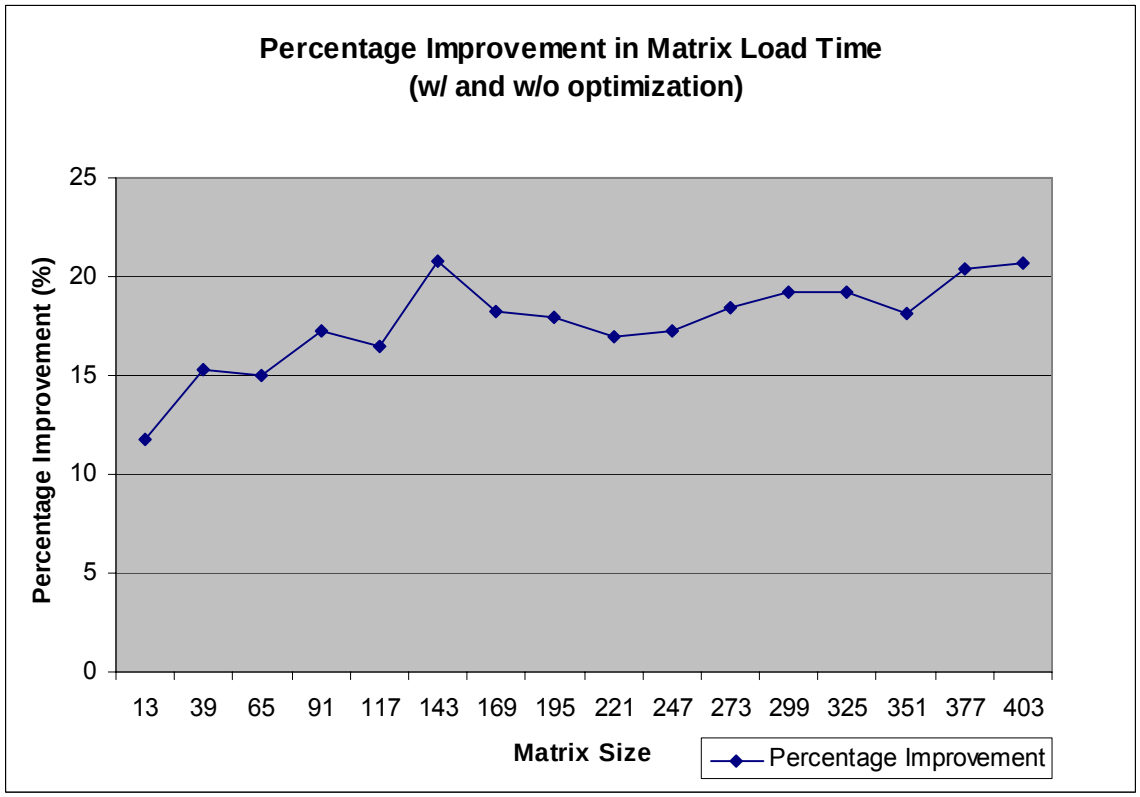

Figure 6.19: Percentage improvement in Matrix Load time.

Table 6.2 shows the confidence intervals for the difference of the means of matrix load time in the normal and optimized modes. Again, the intervals do not include zero at 99% confidence level and thus, for all 'n', percentage improvement is non-zero. In other words, real differences occur at 99% confidence.

Matrix Size	Mean Difference of Matrix Load Time	Confidence Interval for the Mean	Confidence Level %
13	1.887	(1.807,1.966)	99
39	6.114	(6.031,6.196)	99
65	9.781	(9.306,10.256)	99
91	15.570	(15.259,15.881)	99
117	18.180	(18.032,18.327)	99
143	29.911	(29.311,30.511)	99
169	30.382	(29.921,30.841)	99
195	34.350	(33.974,34.725)	99
221	37.152	(36.201,38.103)	99
247	42.609	(42.327,42.890)	99
273	54.291	(53.131,55.449)	99
299	59.882	(59.560,60.203)	99
325	65.073	(64.652,65.494)	99
351	65.785	(64.668,66.901)	99
377	82.748	(79.736,85.759)	99
403	92.748	(88.909,96.585)	99

Table 6.2: Confidence Interval for mean difference of the Matrix Load Time.

Matrix Solve Time:

Figure 6.20 compares the matrix solve times in the optimized and non-optimized (normal) simulators. It is seen that the matrix solve time is consistently lower for the optimized approach when compared to the un-optimized approach. This speaks in favor of a reduction algorithm, and supports our initial assumption that a one time reduction would give us a performance advantage since the matrix solve phase is iterative.

Figure 6.21 shows the percentage improvement in matrix solve time as the matrix size increases. We observe a greater percentage improvement at higher matrix size due to

the constant reduction factor. Table 6.3 shows the confidence intervals for the difference of the means of matrix solve times for the normal and optimized modes of the simulator. The data shows a confidence level of 99% that the difference is significant.

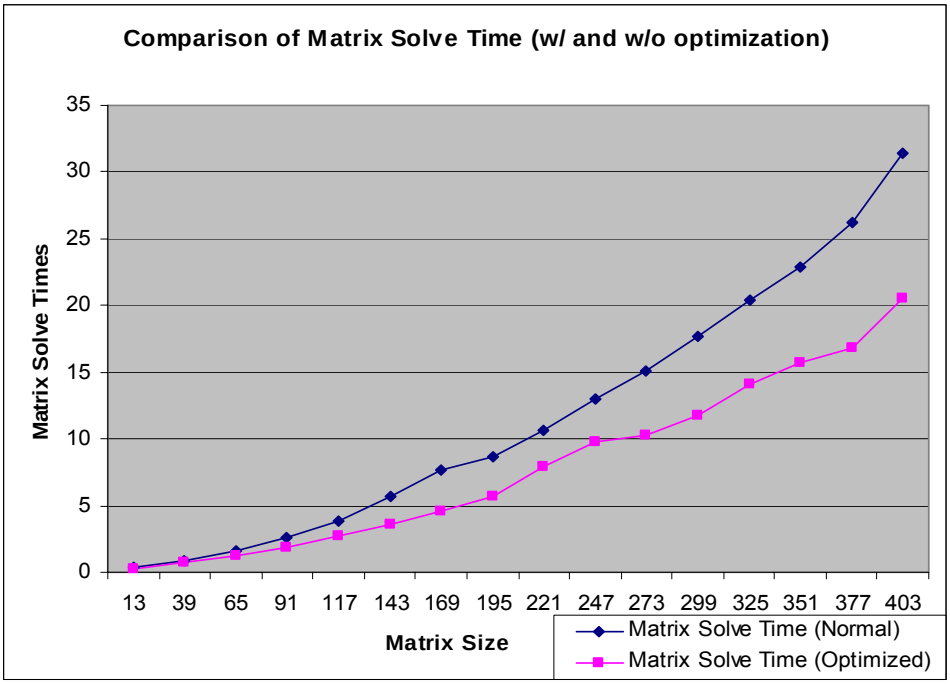

Figure 6.20: Comparison of matrix solve times for the normal and optimized modes.

Matrix Size	Mean Difference of Matrix Solve Time	Confidence Interval for the Mean	Confidence Level %
13	0.051	(0.048,0.054)	99
39	0.181	(0.171,0.190)	99
65	0.457	(0.447,0.466)	99
91	0.740	(0.698,0.781)	99
117	1.148	(1.094,1.200)	99
143	2.132	(2.117,2.147)	99
169	2.985	(2.927,3.041)	99
195	2.981	(2.909,3.052)	99
221	2.749	(2.680,2.817)	99
247	3.219	(3.137,3.301)	99
273	4.811	(4.716,4.904)	99
299	5.857	(5.761,5.952)	99
325	6.275	(6.125,6.423)	99
351	7.279	(7.109,7.447)	99
377	9.361	(9.207,9.514)	99
403	10.860	(14.743,14.977)	99

Table 6.3: Confidence Interval for mean difference of the Matrix Solve Time.

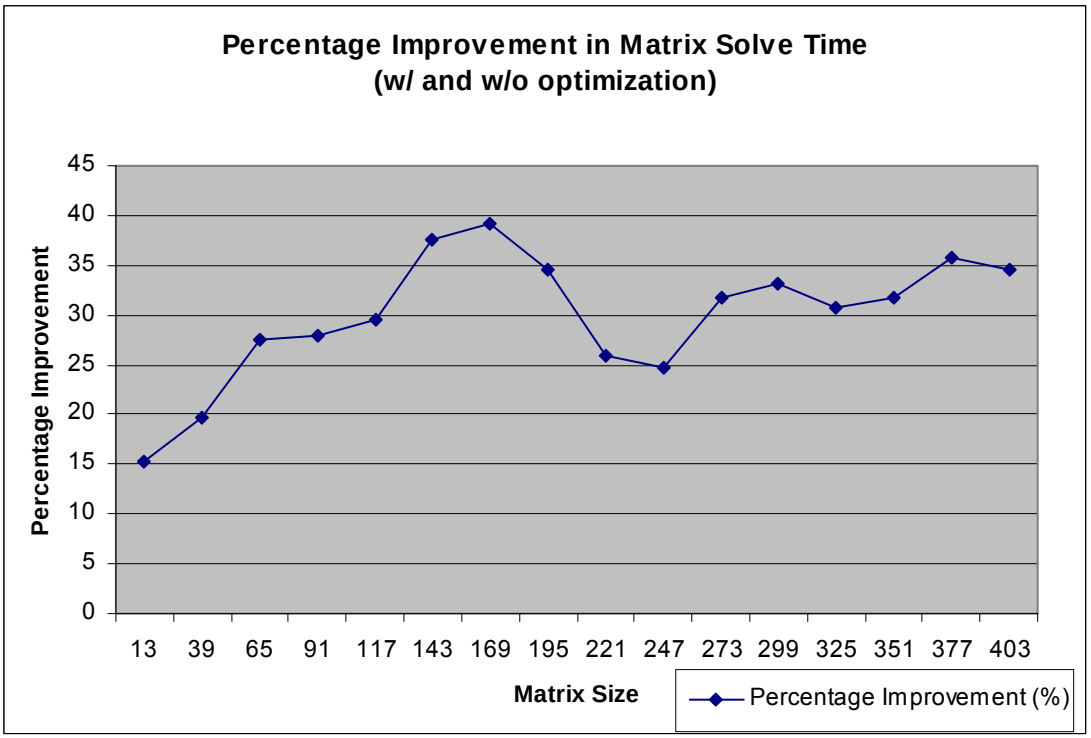

Figure 6.21: Percentage improvement in Matrix Solve time.

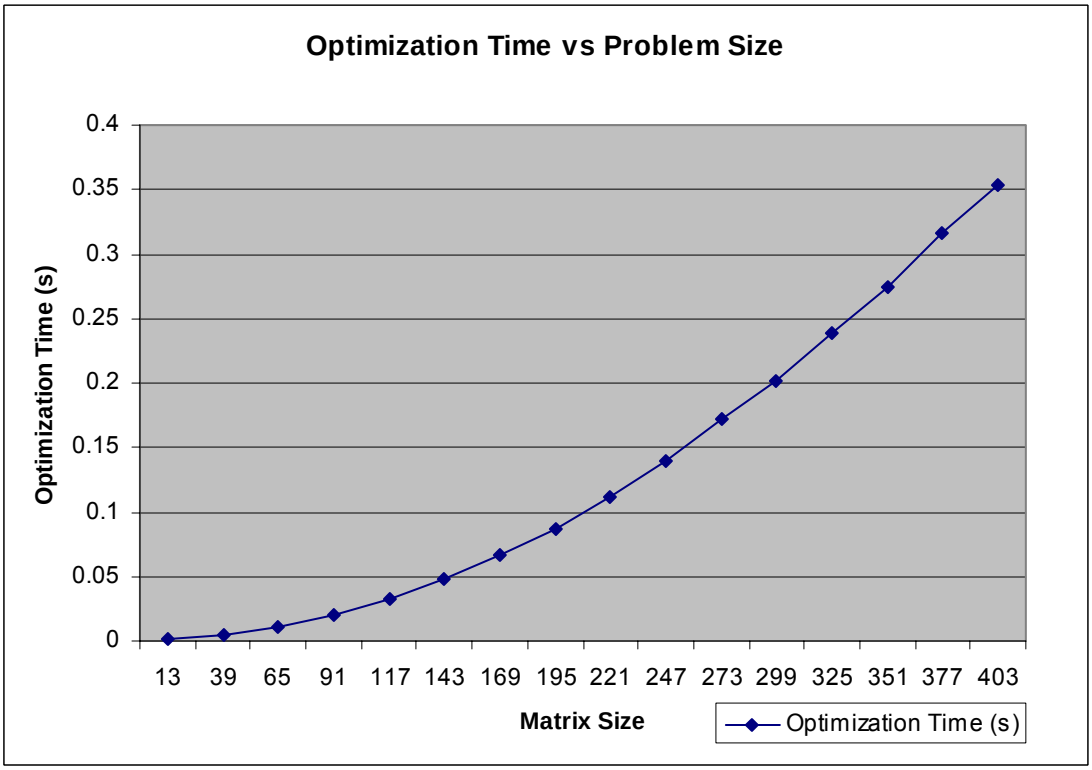

Figure 6.22: Optimization Time as a function of matrix size.

Optimization Time:

It is observed from Figure 6.22 that the time consumed by the optimization algorithm increases with the increase in matrix size. However, since the percentage contribution of the optimization time to the total simulation time is very low, we obtain performance improvement with the use of the reduction algorithm.

II. Results for Model 2

The results obtained from the phono pre-amplifier model shown in Figure 6.14 are presented below. Four unknowns were reduced from a total of 17 unknowns in the base model. Thus, the reduction factor for the model is maintained 23.5%.

Non-IO Simulation Time:

Figure 6.23 shows that the optimization algorithm has resulted in a lower value for the Non-IO simulation time for all the matrix sizes considered.

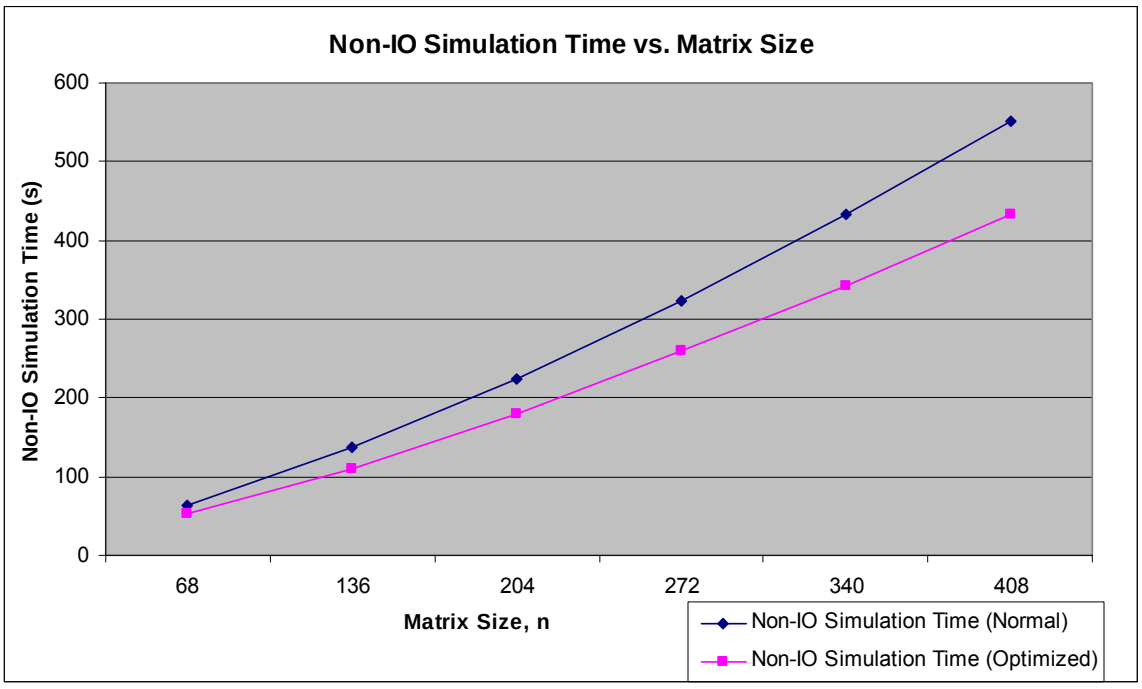

Figure 6.23: Comparison of Non-IO Simulation Time (model 2).

We observe from Figure 6.24 that the percentage improvement in the Non-IO simulation time increases as the matrix size increases. As pointed out for Model 1, this is due to the constant reduction factor. Reducing the matrix size by a constant percentage, results in increased gains at higher matrix sizes.

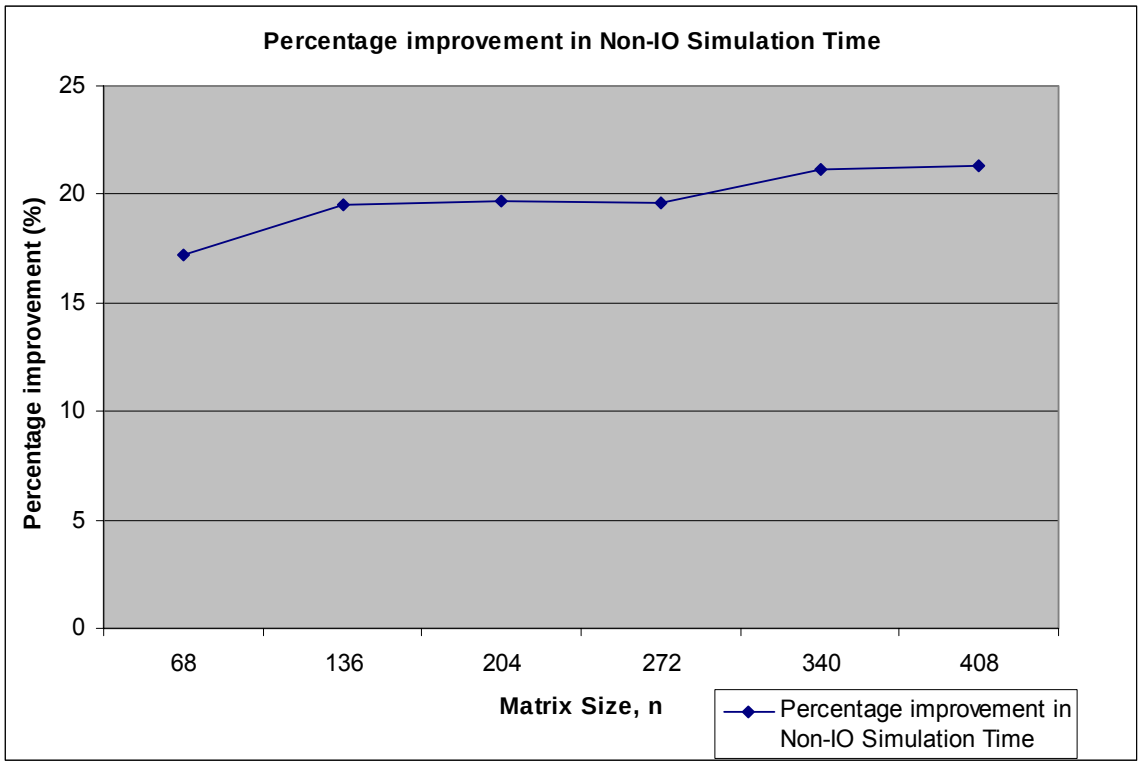

Figure 6.24: Percentage improvement in Non-IO Simulation time (model 2).

Matrix Size	Mean Difference of Non-IO Simulation Time	Confidence Interval for the Mean	Confidence Level %
68	10.84912	(10.674,10.916)	99
136	26.623368	(26.294,26.750)	99
204	44.14764	(43.119,44.546)	99
272	63.39334	(63.247,63.449)	99
340	91.69692	(90.609,92.118)	99
408	117.3832	(116.729,117.636)	99

Table 6.4: Confidence Interval for mean difference of the Non-IO Simulation Time (model 2).

Table 6.4 shows the confidence intervals for the difference of the means of Non-IO simulation time in the normal and optimized modes. Since the intervals do not include zero at 99% confidence level, we can state that for all matrix sizes percentage improvement is non-zero.

Matrix Load Time:

We observe from Figure 6.25 that the matrix load time is lower when the optimization algorithm is used. The gains in matrix load time contribute towards the gain in the Non-IO simulation time.

Figure 6.26 shows the percentage improvement in matrix load time as a function of the matrix size. We observe an increasing percentage improvement due to the constant reduction factor.

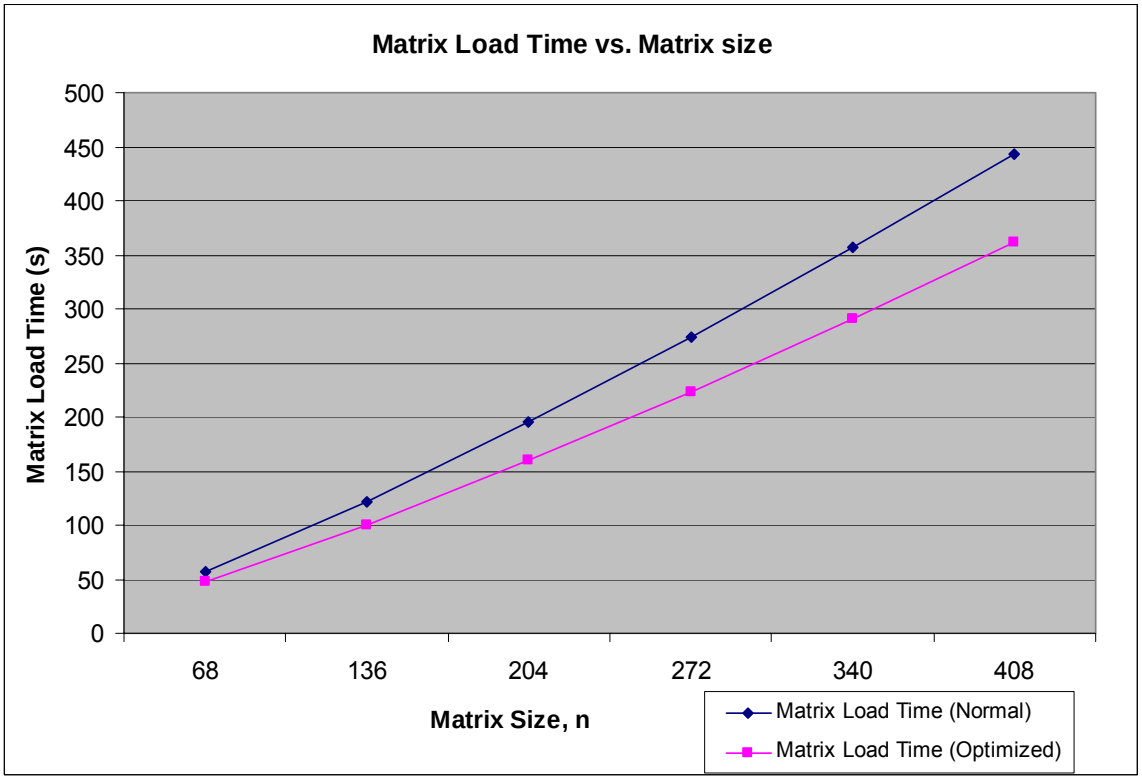

Figure 6.25: Comparison of Matrix Load Time (model 2).

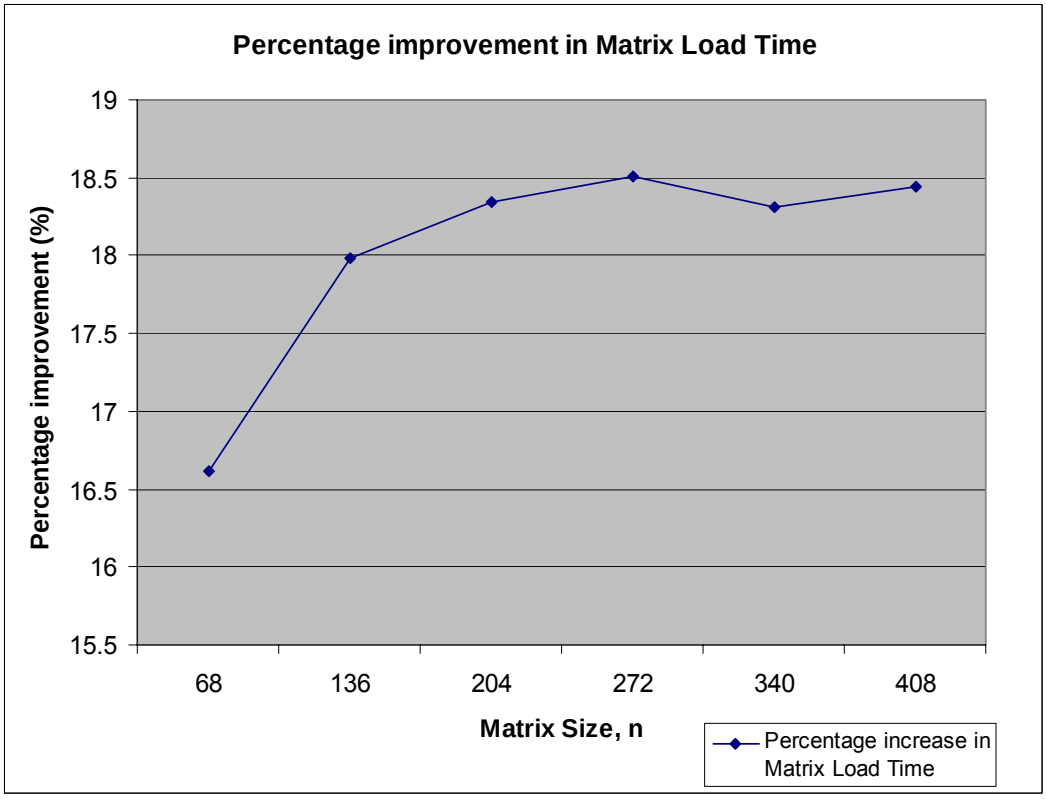

Figure 6.26: Percentage improvement in Matrix Load Time (model 2).

Table 6.5 shows the confidence intervals for the difference of the means of matrix load time in the normal and optimized modes. We observe that a non-zero improvement has been achieved for all 'n' at 99% confidence level.

Matrix Size	Mean Difference of Matrix Load Time	Confidence Interval for the Mean	Confidence Level %
68	9.52824	(9.384,9.671)	99
136	21.77792	(21.450,22.104)	99
204	35.7744	(34.689,36.859)	99
272	50.5896	(49.483,51.695)	99
340	65.3204	(64.856,65.784)	99
408	81.5954	(80.953,82.237)	99

Table 6.5: Confidence Interval for mean difference of the Matrix Load Time (model 2).

Matrix Solve Time:

In Figure 6.27, we observe a lower matrix solve time with the use of the optimization algorithm. This proves that lower matrix solve times can be achieved by the reduction of the matrix size. Based on Table 6.6, we can state that we have achieved a real improvement in matrix solve time at 99% confidence level as a result of the optimization algorithm.

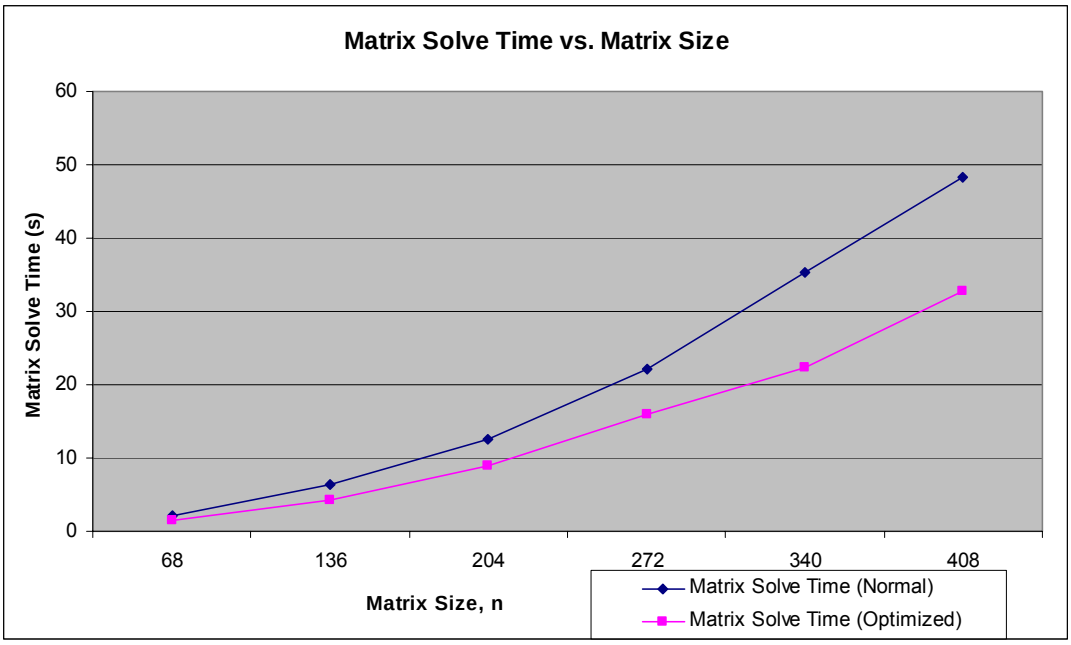

Figure 6.27: Comparison of Matrix Solve Time (model 2).

Matrix Size	Mean Difference of Matrix Solve Time	Confidence Interval for the Mean	Confidence Level %
68	0.520592	(0.456 ,0.584)	99
136	2.083972	(2.035 ,2.132)	99
204	3.7182	(3.664 ,3.771)	99
272	6.12202	(6.008 ,6.235)	99
340	13.06304	(12.961 ,13.164)	99
408	15.6625	(5.481 ,15.843)	99

Table 6.6: Confidence Interval for mean difference of the Matrix Solve Time (model 2).

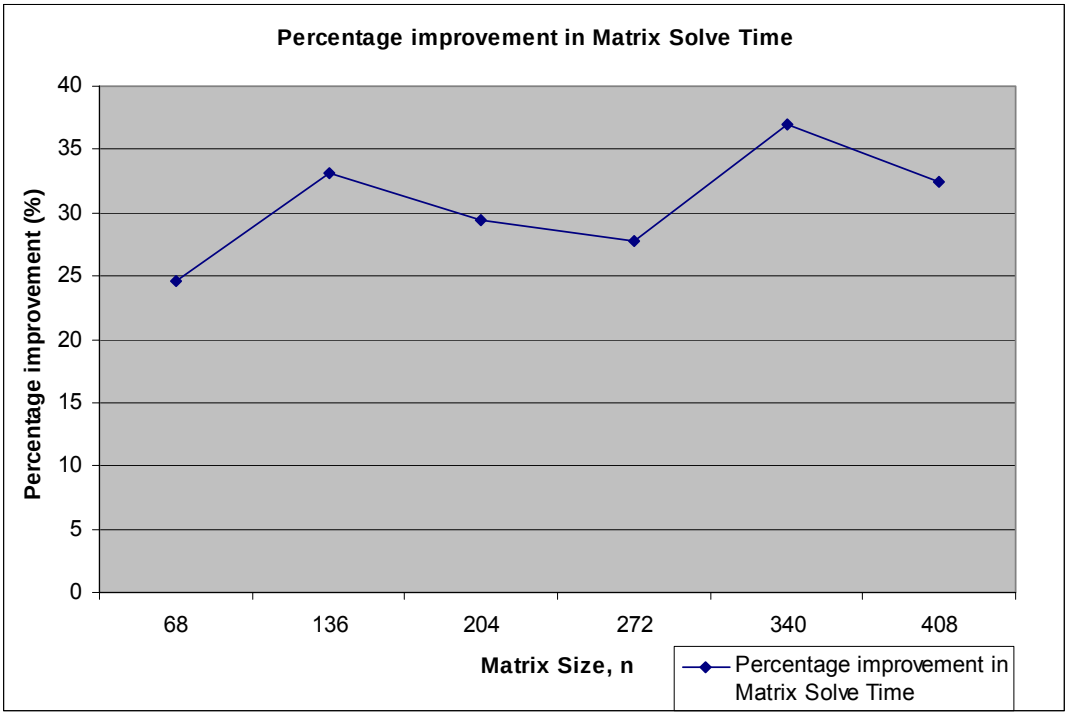

Figure 6.28: Percentage improvement in Matrix Solve Time (model 2).

Figure 6.28 shows that the percentage improvement in matrix solve time increases with increase in matrix size. There is a 24% improvement in matrix solve time for a matrix size of 68 and a 32% improvement for a matrix size of 408. The increase in the percentage improvement is because of the constant reduction factor.

Optimization Time:

It is observed from Figure 6.29 that the time consumed by the optimization algorithm increases with the increase in matrix size. As discussed in the following section, the data obtained from the simulations indicate that the percentage contribution of the optimization time to the total simulation time is very low. Thus it does not constitute a major overhead in the total simulation time.

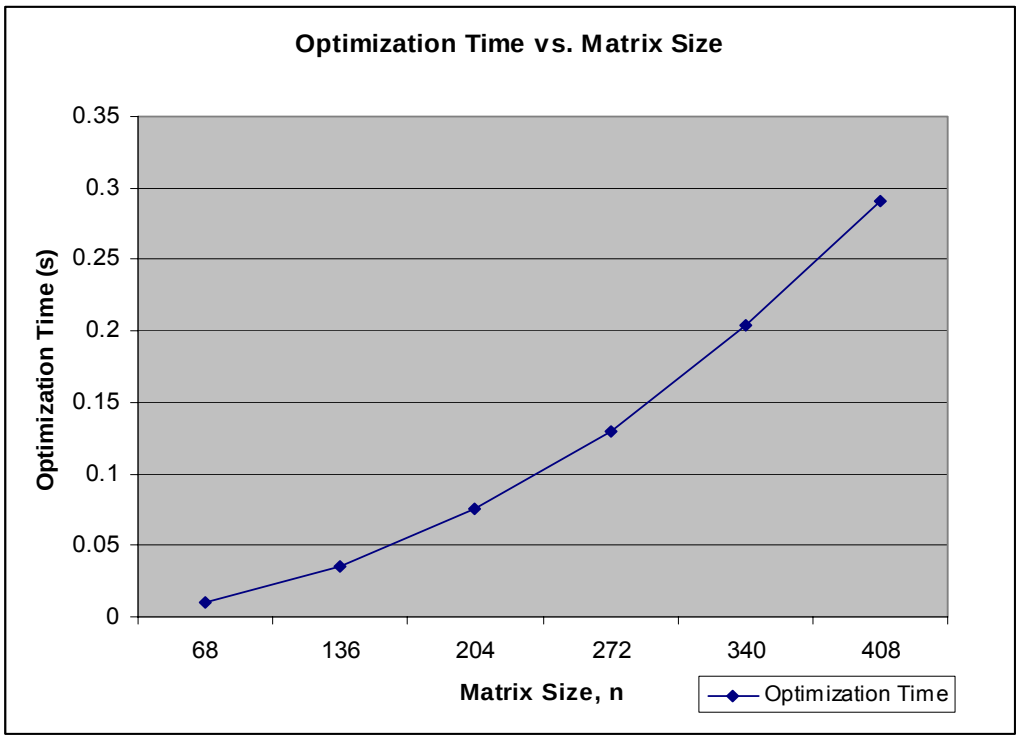

Figure 6.29: Optimization time as a function of matrix size (model 2).

III. Results for Model 3

The results obtained from the active high-pass filter model shown in Figure 6.15 are presented below. Two unknowns were reduced from a total of 16 unknowns in the base model. Thus, the reduction factor for the model is maintained 12.5%.

Non-IO Simulation Time:

Figure 6.30 shows that the optimization algorithm results in a reduction of the Non-IO simulation time for all the matrix sizes considered. We observe from Figure 6.31 that the improvement achieved as a result of optimization is greater at higher matrix sizes. This is because of the constant reduction factor as discussed earlier.

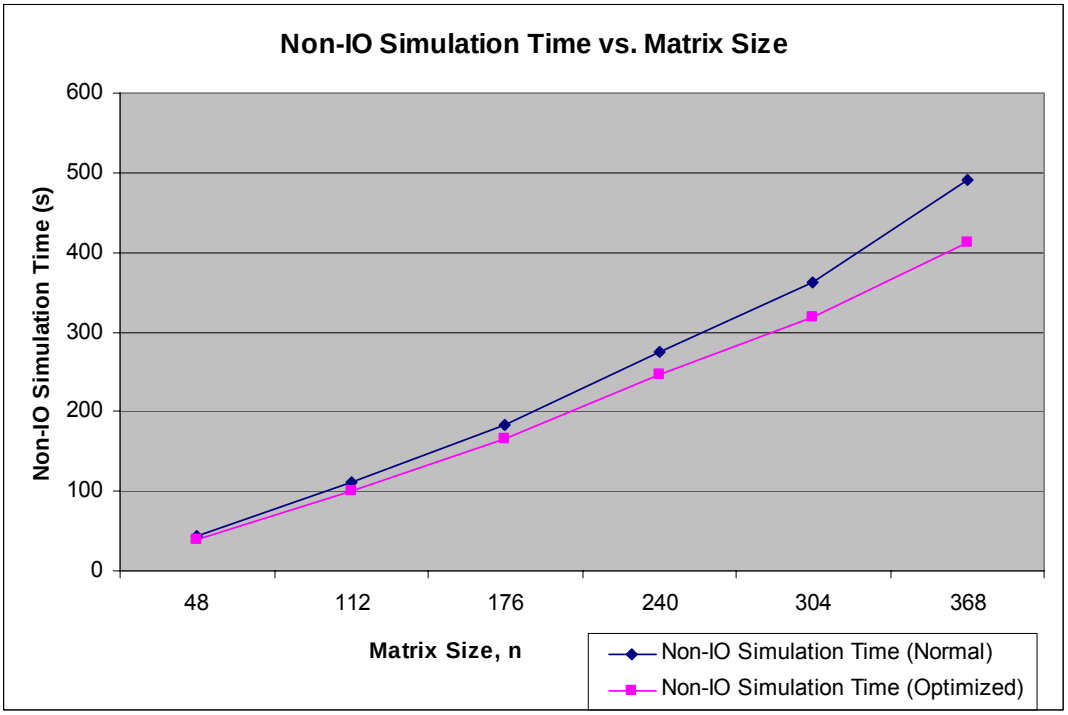

Figure 6.30: Comparison of Non-IO Simulation time (model 3).

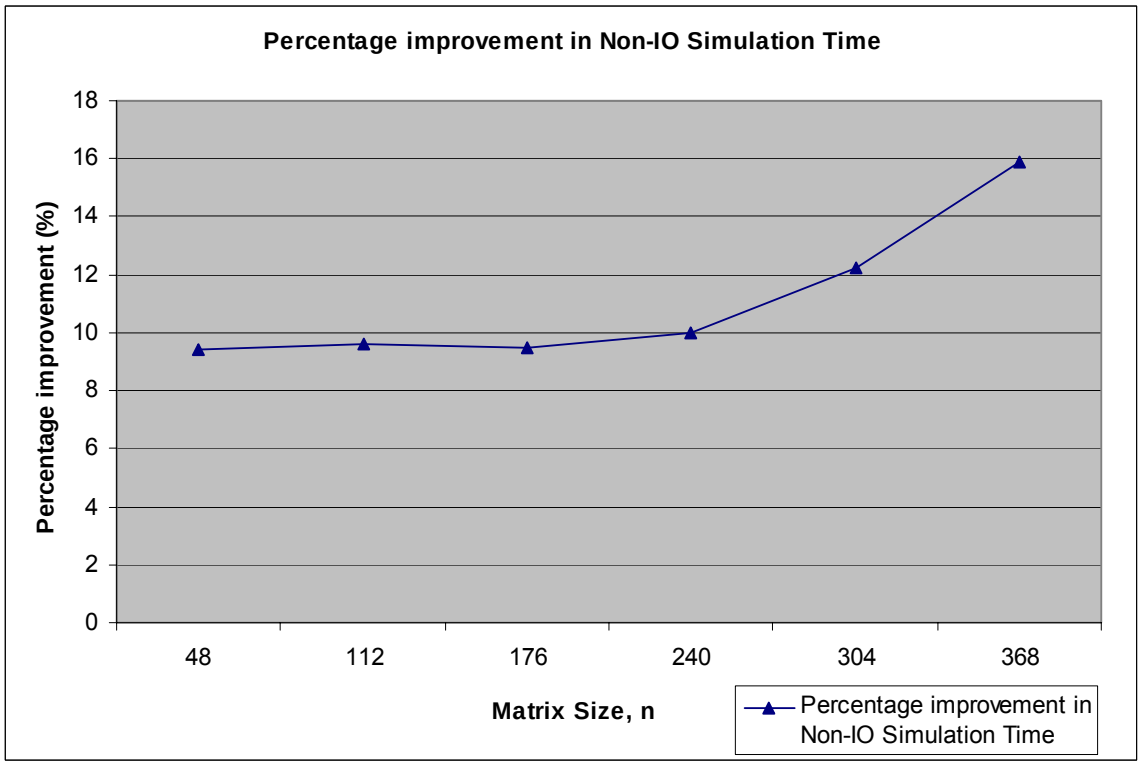

Figure 6.31: Percentage improvement in Non-IO Simulation Time (model 3).

Matrix Size	Mean Difference of Non-IO Simulation Time	Confidence Interval for the Mean	Confidence Level %
48	4.140994	(4.062,4.219)	99
112	10.677854	(9.920,11.435)	99
176	17.31874	(16.294,18.342)	99
240	27.34398	(26.877,27.810)	99
304	44.35556	(43.433,45.277)	99
368	77.94372	(77.404,78.483)	99

Table 6.7: Confidence Interval for mean difference of the Non-IO simulation time (model 3).

Table 6.7 shows the confidence intervals for the difference of means for varying matrix sizes. It gives us a statistical proof that the optimization algorithm results in real, non-zero improvement of the Non-IO simulation time.

Matrix Load Time:

We observe from Figure 6.32 that the optimization algorithm causes considerable reduction in the matrix load time as a result of the reduction in the size of the matrix to be loaded. Table 6.8 further proves that the improvements seen in matrix load time as a result of optimization are non-zero for all matrix sizes at 99% confidence level.

Figure 6.33 shows the percentage improvements achieved in the matrix load time. We observe that the percentage improvement in matrix load time increases from 9.3% to 14.2% as the matrix size increases from 48 to 368. This increase is attributed to the constant reduction factor in the models as matrix size increases. Since there is a greater effect of a constant percentage decrease in matrix size at higher matrix sizes, we find greater percentage improvements in matrix load time. Since matrix load time dominates the total simulation time, we find similar increases in percentage improvement in the Non-IO simulation time plot presented earlier.

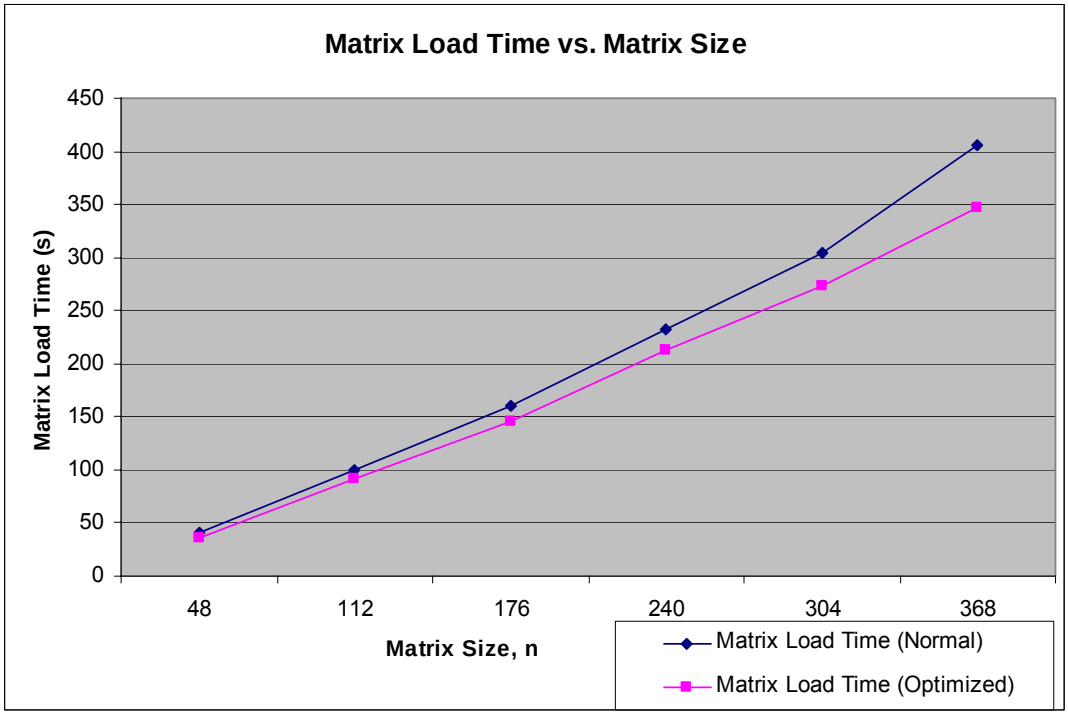

Figure 6.32: Comparison of Matrix Load Time (model 3).

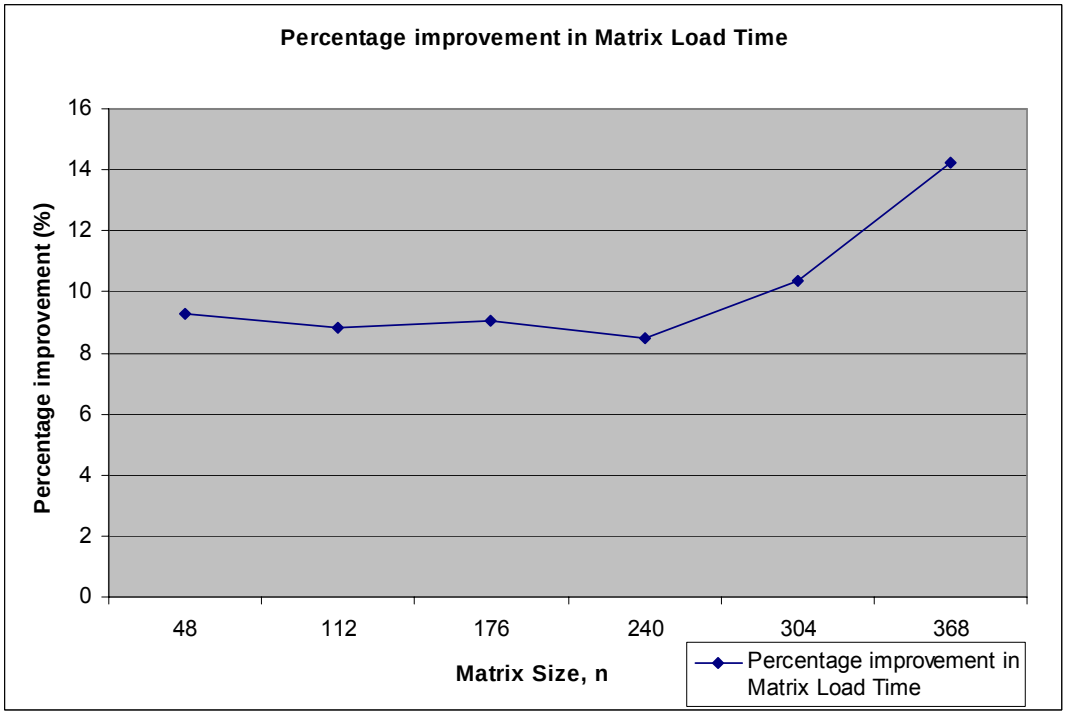

Figure 6.33: Percentage improvement in Matrix Load Time (model 3).

Matrix Size	Mean Difference of Matrix Load Time	Confidence Interval for the Mean	Confidence Level %
48	3.7484	(3.657,3.839)	99
112	8.83438	(8.036,9.631)	99
176	14.5632	(13.505,15.620)	99
240	19.651	(19.273,20.028)	99
304	31.5326	(30.583,32.481)	99
368	57.7214	(57.280,58.162)	99

Table 6.8: Confidence Interval for mean difference of the Matrix Load Time (model 3).

Matrix Solve Time:

Figure 6.34 compares the matrix solve time with and without the application of the reduction algorithm. We observe a definite reduction in the matrix solve time with the use of the reduction algorithm. This is further verified in Table 6.9 which shows that at 99% confidence level, there exist real, non-zero differences between the means for the normal and optimized modes.

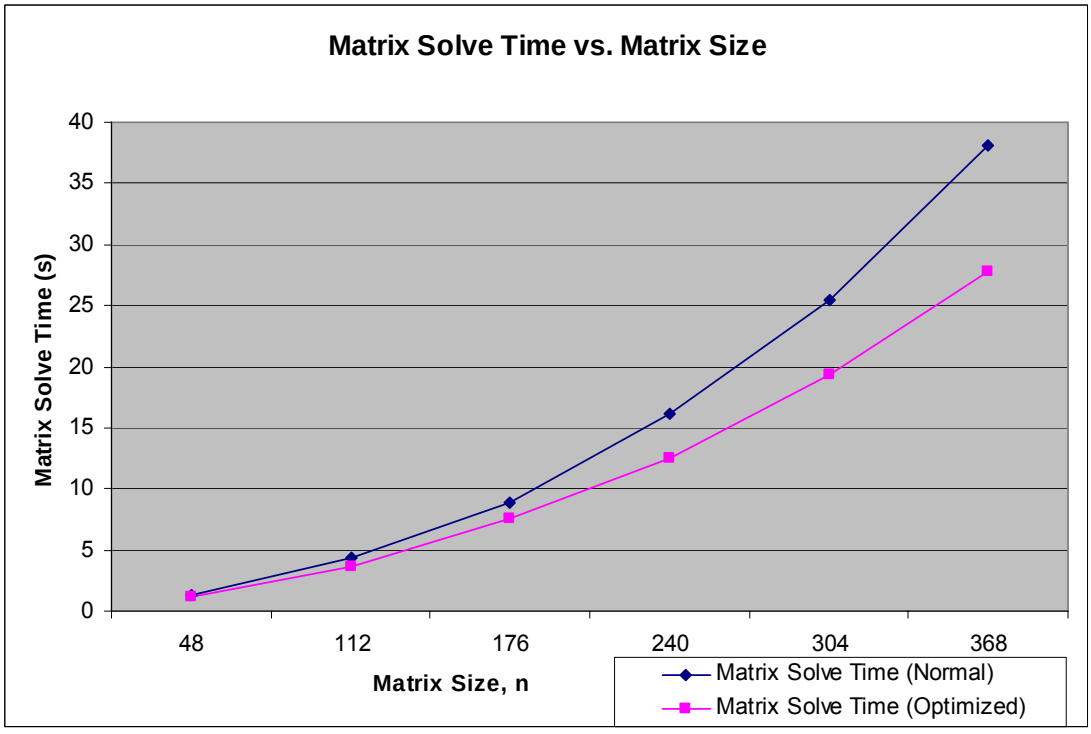

Figure 6.34: Comparison of Matrix Solve Time (model 3).

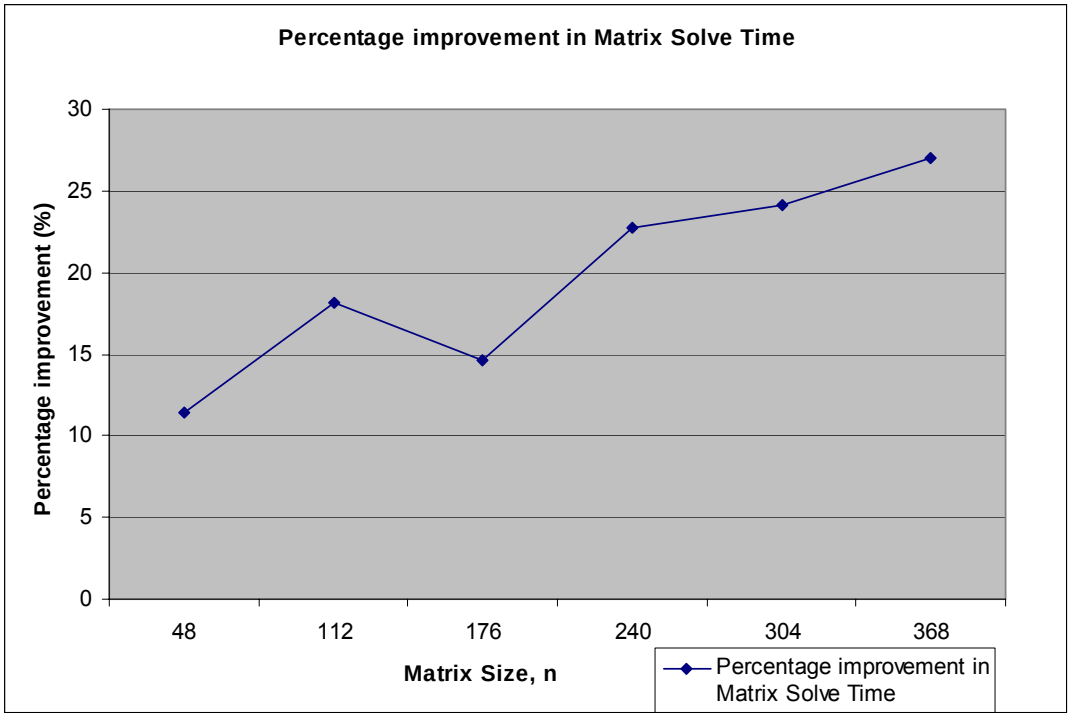

Figure 6.35: Percentage improvement in Matrix Solve Time (model 3).

Matrix Size	Mean Difference of Matrix Solve Time	Confidence Interval for the Mean	Confidence Level %
48	0.14764	(0.138,0.156)	99
112	0.805648	(0.779,0.831)	99
176	1.306636	(1.254,1.358)	99
240	3.68118	(3.579,3.782)	99
304	6.14936	(6.110,6.187)	99
368	10.29806	(10.224,10.371)	99

Table 6.9: Confidence Interval for mean difference of the Matrix Solve Time (model 3).

We observe from Figure 6.35 that percentage improvement in matrix solve time increases with increasing matrix size. This is because, at higher matrix sizes, we have a greater gain in matrix solve time with the same percentage reduction factor.

Optimization Time:

It is observed from Figure 6.36 that the time consumed by the optimization algorithm increases with the increase in matrix size. The data obtained from the simulations indicate that the percentage contribution of the optimization time to the total simulation time is very low. Thus it does not constitute a major overhead in the total simulation time.

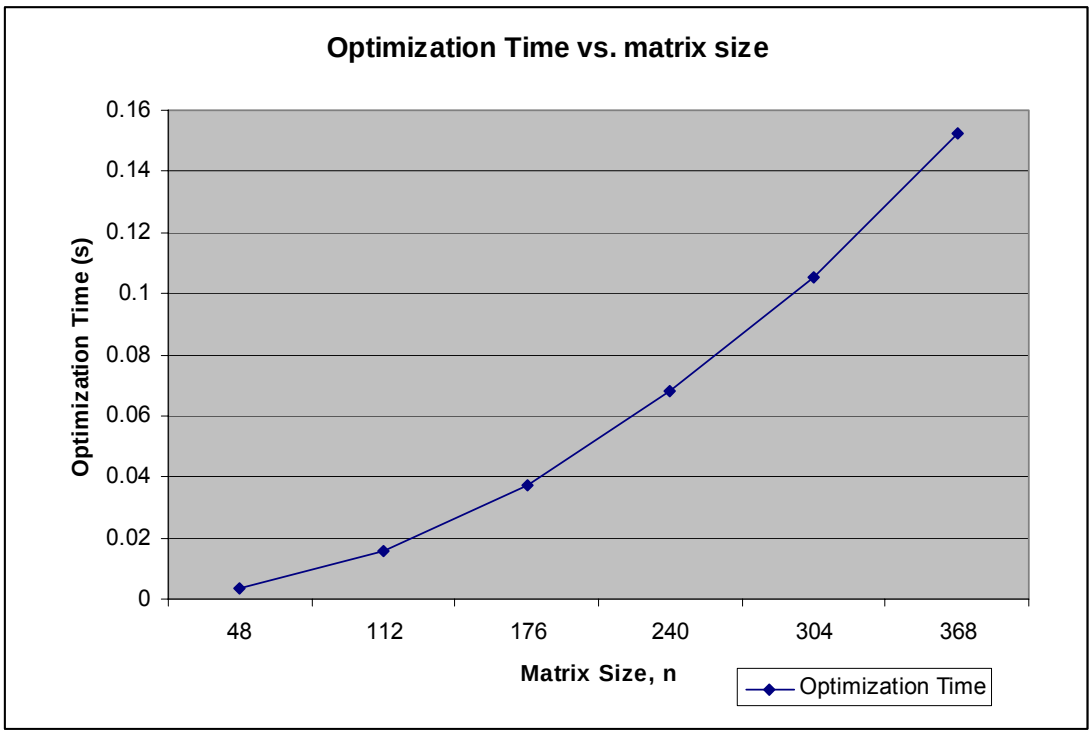

Figure 6.36: Optimization time as a function of matrix size (model 3).

6.4.3 Summary of the Performance of the Data Structure

We conclude that an improvement in the simulation time is indeed possible with the use of reduction algorithms applied on the new data structure to reduce the elaborated set of CEs. This performance improvement is a result of a decrease in matrix load time, as well as the matrix solve time of the simulation kernel.

6.5 Contribution of the various phases of Simulation

In this section, we present pie-charts and plots to show the contribution of the reduction algorithm to the total simulation time. We also illustrate the percentage of time spent by the simulator in the different phases of simulation to prove the domination of the matrix load phase of simulation.

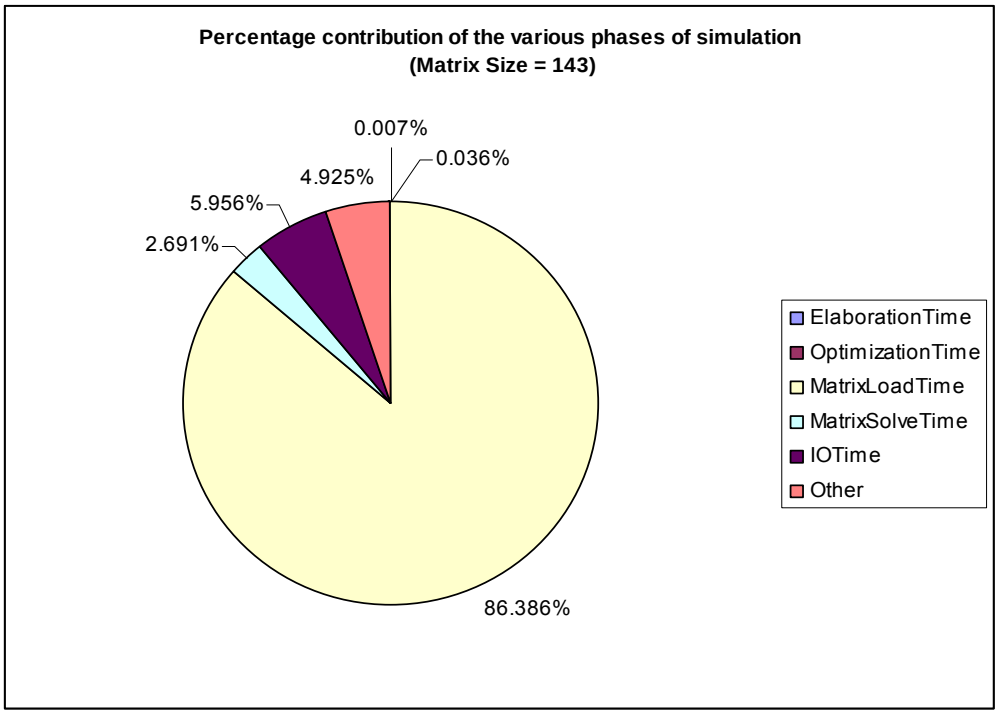

Figure 6.37: Percentage contribution of various phases of simulation kernel (matrix size = 143).

Figure 6.37 shows the percentage contribution of the various phases of the simulation kernel for a matrix size of 143 and a simulation time of 1 μs. It is observed that the reduction algorithm contributes only 0.036% of the total simulation time of 134.214 seconds. Figure 6.38 shows the percentage contributions of the various phases of the kernel for a matrix size of 299 and a simulation time of 1 μs. We observe again that the optimization algorithm has a very low contribution to the total simulation time of

303.792 seconds. It is also observed that as the matrix size increases, the contribution of the matrix load phase of the kernel reduces from 86.386% to 83.405%. This is due to an increased contribution of the matrix solve phase and the IO phase of the kernel.

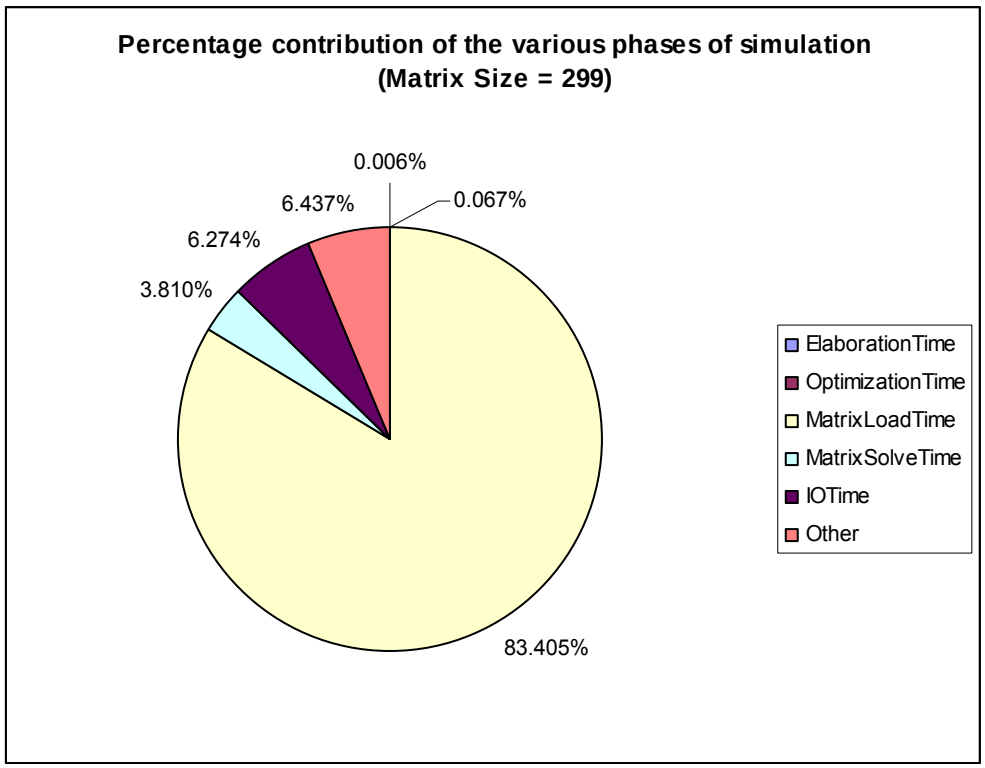

Figure 6.38: Percentage contribution of various phases of simulation kernel (matrix size = 299).

Figure 6.39 shows the percentage contribution of the matrix load and solve times as the internal simulation time of the model is increased. The simulation time was varied from 0.1 µs to 0.1 ms leading to a change in wall-clock simulation time from 4.265 seconds to 4191.2 seconds for a model of matrix size 39. It is observed that as the simulation time increases, there is a slight decrease in the contribution of the matrix load phase of the kernel. The percentage contribution of the matrix solve phase remains almost

a constant around 2.1%. The decrease in the percentage contribution of matrix load time is because of an increase in the IO time as the simulation time increases.

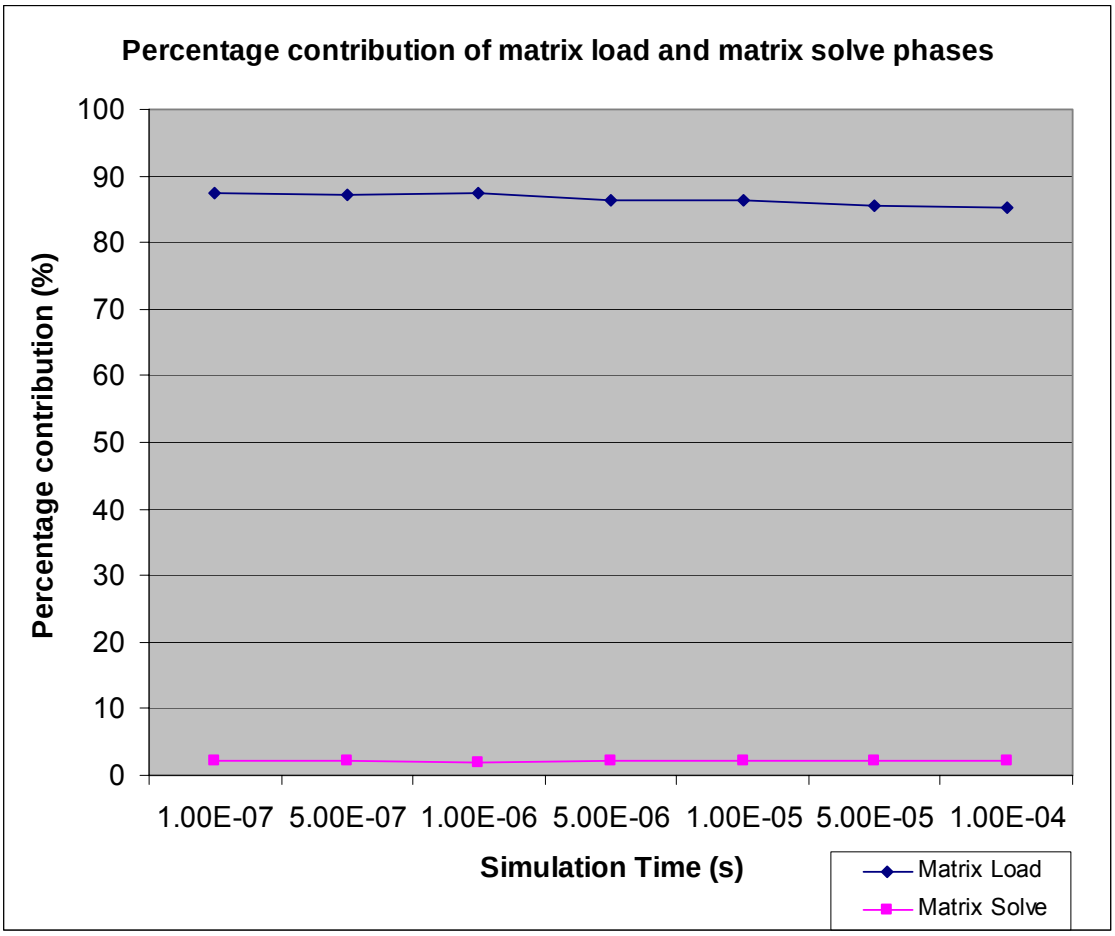

Figure 6.39: Percentage contribution of matrix load and matrix solve phases with increasing internal simulation time.

6.6 Summary of Results

The following is a summary of the results obtained -

- Reduction of the elaborated set of CEs has been shown to be possible with our new information structure.

- Substitution as a method of reduction achieved an improvement in the total simulation time for non-conserved systems.
- Analysis of the non-I/O simulation time is more appropriate than the total simulation time when reduction algorithms are applied.
- The reduction in the elaborated set of CEs leads to a reduction in the matrix load time and the matrix solve time.
- The optimization algorithm contributes a very small percentage of the total simulation time in general and more so at higher problem sizes.

Chapter 7

Conclusions and Future Work

7.1 Summary of Conclusions

This research aimed at improving the performance of compile-driven mixed-signal simulators. Our focus was on improving the performance of the analog kernel of the mixed-signal simulator. The strategy was to improve the matrix build and solve times of the analog kernel. The current approaches showed that optimization of the elaborated set of characteristic expressions could lead to an improvement of the matrix build and solve times, and hence the total simulation time. However the internal data structure for the *simple simultaneous statements* did not allow for the modification or reduction of characteristic expressions or sets of characteristic expressions. A new data structure has been designed and implemented for the *simple simultaneous statements* and an information structure (S3IS) has been designed to support the dynamic elaboration methodology.

Additionally, we have demonstrated a proof of concept for the design by implementing a reduction algorithm which showed performance improvements. Below

we summarize the contributions of this research against the objectives set forth in Section 1.2.

- This thesis has successfully documented the design and implementation of a data structure for *simple simultaneous statements*, which allows run-time modification of the characteristic expression.
- The design has been implemented in a compiled, mixed-signal simulator with Intermediate Code in C++.
- The information structure has been shown to allow the matrix load operation both by design and implementation.
- The S3IS information structure supports a dynamic elaboration strategy which separates the elaboration phase from the matrix load phase, thus enabling a run-time reduction of the elaborated set of characteristic expressions.
- In addition, this document also presents the design and implementation of a reduction algorithm based on substitution for non-conserved systems. The experimental results validate the design of our information structure, and also show performance improvement for smaller problem sizes.

7.2 Future Work

Some directions and suggestions for future work are presented below:

- More complex reduction algorithms may be explored to improve the performance of the mixed-signal simulator. Since the optimization phase of our algorithm contributes very minimally to the total simulation time, we can afford to design and implement more complex reduction algorithms.

- Another suggestion related to the above is the use of graph theory and algorithms to reduce the elaborated set of ODAEs.
- Modify the implementation of the numerical method of integration of the simulator so as to allow the reduction of differential quantities in the *simultaneous statements*.
- Extend the design of the S3IS to include user defined functions.

Bibliography

[1] Kettenis, D. L. An algorithm for parallel combined continuous and discrete-event simulation. *Simulation Practice and Theory* (May 1997), 167-184.

[2] Pandey, S. *Improving performance of mixed-signal simulation by reducing equation-set*, Master's thesis, University of Cincinnati, 2002.

[3] Agrawal, S *Optimization approaches for analog kernel to speed-up VHDL-AMS simulation*, Master's thesis, University of Cincinnati, 2002.

[4] Vlach, J. and Singhal, K. *Computer Methods for Circuit Analysis and Design*. Van Nostrand Reinhold, New York, NY, 1994.

[5] Buchanan, James L. and Turner, Peter R. *Numerical methods and analysis*. McGraw-Hill, Inc., 1992.

[6] Aho, A. V., Sethi R., and Ulman J. D., *Compilers Principles Techniques and Tools.* Addison-Wesley, Reading, MA, 1986.

[7] Frey, P. *Protocols for Optimistic Synchronization of Mixed-Mode Simulation*. Ph.D. dissertation, University of Cincinnati, 1998.

[8] Ashenden, P. J. *The Designer's Guide to VHDL*. Morgan Kaufmann, 1995.

[9] Subramani, K. *The Design of Parallel VHDL Simulation Kernel based on Time Warp*, Master's thesis, University of Cincinnati, February 1998.

[10] IEEE Computer Society, *IEEE Draft Standard VHDL-AMS Language Reference Manual*, (1 August), 1998.

[11] Nellayappan, K. *SEAMS: A Mixed-signal simulation environment for VHDL-AMS with emphasis on run-time elaboration and analog design partitioning*, Master's thesis. University of Cincinnati, 1998.

[12] Young-Hyun Jun; Song-Bai Park. *KMIX: a mixed-mode simulator for analog/digital circuits using event driven waveform relaxation method*. Circuits and Systems, 1989, IEEE International Symposium, 8-11 May 1989, pp. 877 – 880, vol.2.

[13] Subramanian, S. *A Super nodal approach to the linear analog solver in a VHDL-AMS system*. Master's thesis. University of Cincinnati, 2003.

[14] Geeta, B. *Iterative Relaxation Algorithm: An efficient and improved method for circuit simulation used in Sierra: VHDL-AMS simulator*. Master's thesis. University of Cincinnati, 2002.

[15] Chetput, C.L. An analog kernel using the direct method for solving ordinary algebraic differential equations in a mixed-mode simulator. Master's thesis, University of Cincinnati, 1997.

[16] Ashenden, Peter J., Peterson, Gregory D., and Teegarden, Darrel A. *The System Designer's Guide to VHDL-AMS*, Morgan Kaufmann Publishers, San Francisco, September 2002.

[17] Manavalan, JKA. *Performance Evaluation and Speed Improvement of the SEAMS VHDL-AMS Simulator using Dynamic Adjustment of the Analog Simulation Interval*. Master's thesis, University of Cincinnati, 1998.

[18] Antrim Design Systems, Inc. *Advanced Techniques for the Simulation of Mixed-Signal Integrated Circuits*, 1999.

[19] Mayiladuthurai, R. S. *Processing discontinuities and SPICE modeling in VHDL-AMS*. Master's thesis, University of Cincinnati, 1998.

[20] Lightner, M. *Computer Aided Circuit Simulation*. CRC Press, 1993.

[21] Shanmugasundaram, V. *A Dynamic Multiple Solution Approach to Improve the Efficiency of VHDL-AMS Simulation*, Master's thesis, University of Cincinnati, April 1998.

[22] Kundert, K. S. *Sparse User's Guide*, University of California, Berkeley, 1998.

[23] Kundert, K. S. Sparse matrix techniques, In *Circuit Analysis, Simulation and Design*, A. Ruehli, Ed. North-Holland, 1986.

[24] Willis, J., Wilsey, P.A., Peterson, G.D., Hines, J., Zamfirescu, A., Martin, D.A., and Newschutz, R. Advanced intermediate representation with extensibility (**AIRE**). In *VHDL: Multidisciplinary Systems Design and Multimedia* (Oct. 1996), VHDL Internationa User's Forum, pp. 33-39.

[25] Frey, P., Nellayappan, K., Mayiladuthurai, R. S., Chandrashekar, C. L., Carter H. W., SEAMS: Simulation Environment for VHDL-AMS, In *Proceedings of the 1998 Winter Simulation Conference*, (1998), pp. 539-546.

[26] Ho, C. W., Ruehli, A. I., and Brennan, P. A., *The modified nodal approach to network analysis*, IEEE Trans. On Circuits and Systems, 1975, Vol. CAS-22, No. 6, pp. 504-509.

[27] Venkataramani, H. *Optimization of the elaborated set in a conserved system*, Research in progress, Distributed Processing Laboratory, University of Cincinnati, 2004.

[28] Wilsey, P. A., Martin, D.E., and Subramani, K., "SAVANT/TyVIS/WARPED: Components for the Analysis and Simulation of VHDLT," *VHDL Users' Group Spring 1998 Conference*, 195-201, 1998.

[29] Wilsey, P. A. TyVIS: A VHDL Simulation Kernel, 1999. (available on the www at http://www.ececs.uc.edu/~paw/tyvis).

[30] Muchnick, S. S., *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, California, 1997.

Appendix A

Complexity of the Optimization Algorithm

To analyze the complexity of the optimization algorithm presented in Section 5.1.2, we model the time spent by the optimization algorithm in the sub-routine '*reduce*'. The following parameters are used in this discussion-

1. Total number of characteristic expressions in the continuous-time system (*n*).
2. Average equation size as described in Section 6.2 (*m*).

The following are the constants used in the discussion-

1. Percentage of donor expressions (*p*).
2. Percentage of quantity nodes (average) in an acceptor expression (*q*).
3. Number of donor quantities in an acceptor expression (l_{av}, l_w).

   The product of q and m gives us this value.
4. Percentage of characteristic expressions which contain the donor quantity (r).
5. Number of acceptor expressions which are walked to replace the donor quantity (k). Therefore, $k = r * n$

The following are the times taken by various sub-routines discussed in Section 5.1.2-

1. Time taken to find a donor expression (T_{fde}): This is the time taken to check a CE is fit to become a donor expression. This operation is repeated, traversing the list of '*n*' CEs to find a donor expression.

2. Time taken to test if the donor quantity exists in a acceptor expression (T_{tdq}): This is the time taken in traversing the '*l*' *vector* of quantities associated with a CE. We need to perform this action for 'n-1' CEs.
3. Time taken to walk an *equation tree* and replace occurrences of the donor quantity (T_{walk}): This is the time taken in walking an *equation tree* and replacing '*l*' donor quantity *nodes* with a sub-tree of the *donor equation tree*. In total, we walk 'k' such equation trees.

We present the average and worst case time taken by the optimization algorithm for substituting '*p*n*' donor expressions. In the average case,

$$T_{opt} = p * n \left(\frac{n}{2} * T_{fde} + (n\text{-}1)\, T_{tdq} + k\, T_{walk} \right)$$

where, T_{fde} is a constant, T_{tdq} *is* $\frac{q}{2}$, and $T_{walk} = m + l_{av}(m\text{-}2)$. In the worst case,

$$T_{opt} = p * n \left(n * T_{fde} + (n\text{-}1)\, T_{tdq} + k\, T_{walk} \right)$$

where, T_{fde} is a constant, T_{tdq} *is* q, and $T_{walk} = m + l_w(m\text{-}2)$.

From the above two equations, we conclude that the average and the worst case complexity of the optimization algorithm is $O(n^2)$.