Εθνικο Μετσοβιο Πολυτεχνειο
Τμημα Ηλεκτρολογων Μηχανικων Και Μηχανικων Υπολογιστων

Τομέας Επιστήμης Υπολογιστών
Εργαστηρίο Μικροϋπολογιστών & Ψηφιακών Συστημάτων (MicroLab)

Big Data Techniques Applied on Transient Integrated Circuit Simulations

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Γρηγόριου Ν. Λύρα

Επιβλέπων: Δημήτριος Ι. Σούντρης
Επίκουρος Καθηγητής

Γρηγόριου Ν. Λύρα

Αθήνα, Ιούνιος 2013

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
Τομέας Επιστήμης Υπολογιστών
Εργαστηρίο Μικροϋπολογιστών & Ψηφιακών Συστημάτων (MicroLab)

Big Data Techniques Applied on Transient Integrated Circuit Simulations

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Γρηγόριου Ν. Λύρα

Επιβλέπων: Δημήτριος Ι. Σούντρης
Επίκουρος Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 20/06/2013

....	....	....
Κιαμάλ Πεκμεστζή Καθηγητής	Δημήτριος Ι. Σούντρης Επίκουρος Καθηγητής	Νεκτάριος Κοζύρης Καθηγητής

Αθήνα, Ιούνιος 2013.

....

Γρηγόριος Ν. Λύρας
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Γρηγόριος Ν. Λύρας 2013,
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

List of Figures

2.1 Research Landscape 9
2.2 OpenMP pragma in ngspice BSIM loading routines 12

3.1 Execution Framework 17
3.2 Mod counter signals 18
3.3 Execution Framework Toolchain 21
3.4 Sample netlist 22
3.5 Sapp invocation 23
3.6 Partitioner invocation 24
3.7 Partitioner invocation example 24
3.8 Miner invocation 25
3.9 Miner invocation example 25
3.10 Task Line 25
3.11 Hello world 26
3.12 Hypervisor invocation 26
3.13 Hypervisor invocation example 26
3.14 sapp_commmon.h 27
3.15 Task definition 28
3.16 Signal definition 29

4.1 Inputs and Outputs of a 4x4 multiplier 32
4.2 Subtractor of two signal pairs 33
4.3 Error calculation processj 34
4.4 RMSE error compared to hspice results 34
4.5 Cross platform execution times v1 35
4.6 Comparison of times of v1 against hspice 36
4.7 RMSE error compared to spectre results 38
4.8 Multiple transitions error 39
4.9 Temporal offset error 40
4.10 Cross platform execution times v2 41
4.11 Comparison of times of v2 against Spectre 42

4.12 Comparison of execution times of v1, v2 against hspice and Spectre 43
4.13 Comparison of execution times of v1, v2 44

Abstract

In the recent years, one can observe a shift in the software design principles. Earlier practices focused on the optimization of signle-thread execution. In an era where anyone has multicore systems at their disposal this approach is limiting. Therefore there is a need for flexible and scalable techniques for parallel application development. This project attempts to attack this issue with the use of a versatile development framework. This principle has been verified on the field of transient integrated circuit simulations.

Digital circuits are an integral part of modern world. Ever since the beginning of digital circuit design there has been a need for accurate simulation and verification. This need generated a family of software tools known as SPICE applications. The constant increase of the number of devices per silicon die has pushed this family of applications to the limit. The increasing device inventory of the simulated circuits is also pushing system memory and CPU times to their limit. Especially in systems with a unified memory hierarchy, the extreme demands in main memory causes transient simulations to halt due to bad allocation errors.

The proposed approach redirects this memory load incrementally to the file system. By partitioning the initial simulation to multiple substantially smaller ones we manage to decrease the amount of memory requested during each simulation.

Περίληψη

Τα τελευταία χρόνια, μπορεί κανείς να παρατηρήσει μια στροφή στις αρχές του σχεδιασμού λογισμικού. Παλιότερες πρακτικές εστιάζουν στη βελτιστοποίηση της εκτέλεσης ενός νήματος. Σε μια εποχή όπου ο καθένας έχει στη διάθεσή του συστήματα πολλαπλών πυρήνων, η προσέγγιση αυτή είναι περιοριστική. Ως εκ τούτου, υπάρχει ανάγκη για ευέλικτες και κλιμακούμενες τεχνικές για την ανάπτυξη παράλληλων εφαρμογών. Το έργο αυτό, επιχειρή να αντιμετωπίσει αυτο το θέμα με τη χρήση ενός ευέλικτου πλαισίου ανάπτυξης εφαρμογών. Η αρχή αυτή έχει επαληθευτεί στον τομέα της προσομοίωσης ολοκληρωμένων κυκλωμάτων.

Τα ψηφιακά κυκλώματα αποτελούν αναπόσπαστο μέρος του σύγχρονου κόσμου. Από τις απαρχές της σχεδίασης ψηφιακών κυκλωμάτων υπάρχει έντονη ανάγκη για ακριβή προσομοίωση και επαλήθευση. Αυτή η ανάγκη δημιούργησε μια οικογένεια εφαρμογών γνωστές ως εφαρμογές SPICE. Η συνεχής αύξηση του αριθμού των συσκευών ανά τσιπ πυριτίου έχει ωθήσει αυτή την οικογένεια προγραμματιστικών εργαλείων στα όρια. Η αυξητική αυτή τάση στο μέγεθος των προσομοιούμενων κυκλωμάτων πιέζει τα υπολογιστικά συστήματα που εκτελούν τις προσομοιώσεις, τόσο στο επίπεδο της μνήμης όσο και στο επίπεδο χρόνων εκτέλεσης, στα όριά τους. Ειδικά σε συστήματα με ενιαία ιεραρχεία μνήμης, οι ακραίες απαιτήσεις στην κύρια μνήμη, σταματούν προσομοιώσεις λόγω ανεπάρκειας μνήμης.

Η προτεινόμενη προσέγγιση ανακατευθύνει αυτό το φορτίο σταδιακά στο σύστημα αρχείων. Καταμερίζοντας την αρχική προσομοίωση σε πολλαπλά σημαντικά μικρότερα τμήματα, καταφέρνουμε να μειώσουμε την ποσότητα απαιτούμενης μνήμης κατά τη διάρκεια κάθε προσομοίωσης.

Acknowledgements

The work described in the present thesis has been carried out at the Microprocessors Laboratory and Digital Systems Lab of the School of Electrical and Computer Engineering of NTUA, where I conducted my thesis project, under the supervision of Prof. D.J. Soudris and Dr. D. Rodopoulos.

I am extremely grateful to both of them for their guidance, support and enthusiasm at all stages of this project. I would like to thank them for giving me the opportunity to discover the challenging world of SPICE simulations and Parallel implementations and for all, very generously, they taught me. Their positive attitude and their invaluable advice, whenever I needed it, along with their trust and the freedom they provided me work independently, are very much appreciated.

I would like to thank Dr. Antonis Papanikolaou for his precious work during this work. I thank him for his constant support and innovative ideas he contributed to this project. His experience, guidance and realistic point of view were of great importance to the development of this project.

My special thanks to all my friends in Athens, who contributed in making this story a really worth telling one, Thank you all for the moments we shared, everything we learnt, the fun we had, for being there when needed and for "bringing me back to life" when the work was dragging me down and also the one that gave me that little red book, and all the support they gave me.

Last but not least, I wish to thank my family. The completion of this work wouldn't be possible without their endless love and support. There are no words to express my gratitude... This thesis is dedicated to them.

Γρηγόρης Λύρας

Contents

1 Introduction 6

2 Research Landscape 8
2.1 Introduction 8
2.2 State of the Art 9
2.2.1 SPICE in general 10
2.2.2 SPICE in Specific Hardware 10
2.2.3 SPICE in Generic Hardware 11
2.2.4 Industry Standard Tools 13
2.3 Motivation 14
2.3.1 SPICE in Variability Simulation 14
2.3.2 SPICE in Time-Dependant Verification 14
2.3.3 Summary 15

3 Data Partitioning SPICE 16
3.1 Introduction 16
3.2 Principles 17
3.2.1 Theoretics 17
3.3 Implementation 20
3.3.1 User Manual 20
3.3.2 References to Source Code 26

4 Benchmarks & Results 30
4.1 Introduction 30
4.2 Simulation Description 30
4.3 Results 31
4.3.1 First Benchmarking Session 33
4.3.2 Second Benchmarking Session 36
4.3.3 Results 37

5 Conclusions & Future Work 45

5.1 Conclusions 45
5.2 Future Work 46

Bibliography **48**

6 Appendix **50**
A: DATE 2013 Preprint 50
B: Source Code 57

Chapter 1

Introduction

During our time we have witnessed radical changes in the computing world. In the past decade the processing model has gradually evolved from single-thread execution to multithreaded execution and to massively parallel execution. Nowadays, everyday computing is handled by multiple processing nodes. Even smartphones have multiple asymmetric cores. Thus it can be understood that we need frameworks with great versatility that can adapt and port easily applications in various contexts. There is a constant need for tools that will be able to harness the new processing potential as materialized in multi- and many-core platforms.

One field that can greatly benefit from the multi and many core architectures is the field of Electronic Design Automation (EDA) and especially transient integrated circuit (IC) simulations. With the integration level race constantly increasing the devices per silicon dye circuit simulations become more complex and consume a lot more CPU time and system memory, thus increasing the development cost in multiple levels.

Another reason for resource hungry circuit simulations comes as an indirect result of aggressively decreased device feature sizes. With devices reaching lengths at the decananometer order, they also display

a stochastic behaviour. Thus designers need to be able to perform a more comprehensive circuit verification. This is usually achieved by Monte Carlo simulations for time zero device variability and through extensive simulations over large workloads for time dependent phenomena. As a result, it is highly desirable to explore the potential of High Performance Computing techniques (e.g. for big data) on the field of EDA applications, such as transient IC simulations.

When we refer to circuit simulations we refer to SPICE applications. There have been attempts to optimise SPICE ever since it was introduced. Novel ideas such as mapping SPICE on custom hardware, such as Field Programmable Gate Arrays (FPGAs) or Graphics Processing Units (GPUs) have been explored, each with its advantages and disadvantages. Open source tools as well as enterprise solutions attempt to improve SPICE performance with the use of multiple threads of execution over a common memory hierarchy. However such approaches cannot overcome the memory wall imposed by the executing system.

In the context of this thesis we propose a framework that can provide a solution to the above problem through partitioning techniques. Since design houses don't have the source code of the commercial tools they use, the framework should remain agnostic with regard to the processing kernel per simulation using the black box paradigm. Instead of running a long massive simulation we propose the execution of multiple smaller ones and forwarding the signals among the simulations. The framework is able to handle multiple simulations and execute them in parallel using a graph of dependencies to determine when the simulations can be executed. Apart from the partitioning in the circuit axis, later referred to as *Node Tearing*, we propose temporal partitioning, referred to as *Workload Tearing* . The latter enables even greater parallelism provided the circuit at hand can be simulated in such a manner.

During our experiments we compared our results with two state of the art commercial tools, **hspice** by *Synopsys* and **spectre** by *Cadence*. For our simulations we used the open source **ngspice**. The runtime environment consisted of three different platforms, an Intel Xeon X3470 powered server, the Single Chip Cloud Computer prototype by Intel Labs and a virtual machine hosted on okeanos cloud service by GRNET. We conducted two benchmarking sessions using a multiplier as the circuit to be simulated. During the first benchmarking session we managed to overcome the memory wall imposed by the executing system and we verified the accuracy of the framework. In the second benchmarking session we improved the performance of the framework using more advanced partitioning techniques.

In conclusion we managed to overcome the limitations imposed by the runtime system. The framework we propose can produce accurate results for circuits far larger than the industry standard tools used, with significant speedups. A subset of this work has been accepted in the proceedings of the Design Automation and Testing in Europe Conference, held in Grenoble, France. A preprint of the respective paper can be found in Appendix A.

Chapter 2

Research Landscape

2.1 Introduction

In this chapter of the thesis we present the current landscape and the research areas regarding optimisations in SPICE applications. In order to achieve this we use the graph shown in Figure 2.1. This graph shows binary splits of the field of study. Each path in the tree structure above, describes a series of design choices one has to make when optimising SPICE applications.

As shown in Figure 2.1 we can divide the field based on the hardware type used, the level of the performed optimisations and the memory hierarchy. Regarding the level of the performed optimisations, we use the term *In Source* to designate optimisations that take place in the application source and the term *Source Agnostic* for optimisations that take place without interacting with the SPICE Application source code. Each category is a binary split of it's parent and the both siblings are complementary. For each leaf we have a path from the root to the leaf that designates the design choices made. For example if one chooses the last leaf on the right the path would be *Generic Hardware->Source Agnostic->Distributed Memory*, describing the equivalent design choices in the development of their application.

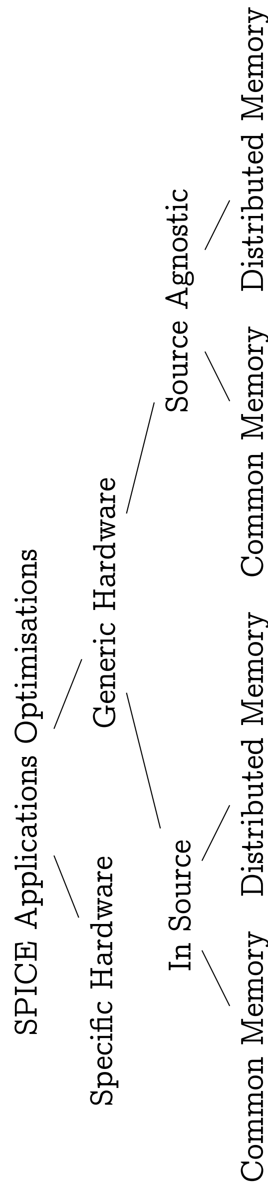


Figure 2.1: Research Landscape

2.2 State of the Art

The need for viable transient circuit simulations has played a significant role in shaping the progress of the EDA industry. This is apparent if one considers the number of EDA tools developed. An area where a lot of work has been delivered is integrated circuits (IC) simulation. For the IC

domain, the tools that have been produced belong to a family of software tools known as SPICE applications.

The "original" SPICE was developed at the Electronics Research Laboratory of the University of California, Berkeley by Laurence Nagel [12]. It changed the skyline of the circuit design world and many research teams started working on derivatives of this work. A lot of work was put in improving the performance of the initial approach with the use of new numerical methods and more efficient code [11]. Compiler optimisations and smarter computer architectures combined with the increasing clock speeds of CPUs provided the EDA industry with enough processing power for the increasing simulation demands. However this approach has reached a saturation point, since CPU frequencies are no longer increasing at the same rate, in comparison to previous years. This leads to the conclusion that we need to explore other means in order to make the ever demanding IC simulations viable for the years to come.

Producing results faster and with less resource consumption along with exploiting parallel architectures are goals well set in the HiPEAC roadmap for the next 10 years [16, 3].

2.2.1 SPICE in general

Since it's first release in 1973 SPICE derivatives have reached a significant level of maturity. The past decade has shown that single thread execution has it's limits. Thus there is a need for SPICE to evolve to the next level. Mapping SPICE to parallel architectures is a multidisciplinary problem and there are quite a few attempts each with advantages and disadvantages.

2.2.2 SPICE in Specific Hardware

One of the software design constraints when developing SPICE is the running platform. Running SPICE on specialized hardware can provide significant speedups. A fine example of this concept is an implementation of SPICE running on Field Programmable Gate Arrays (FPGAs) [6]. This implementation parallelized the different tasks of the SPICE simulator giving maximum speedup up to eleven ($\times 11$). The idea was to map each of the three demanding tasks to a separate solver. So SPICE was broken down to the Iteration Controller, Model Evaluation and Sparse Matrix Solver components that could run independently. However due to the limitations

of the hardware it could produce results of up to thirty thousand (30000) transistors.

Another attempt to run SPICE in specific hardware took advantage of the processing potential of NVIDIA GPUs as described in [5].

In this project the developers mapped *device model evaluation* to the GPU cores enabling parallelism at a device level. However it required significant effort in order to rewrite the kernels for the CUDA architecture. The speedup achieved by this approach was around three (3) with simulation capacity of about eight thousand (8000) transistors.

A few years later *OmegaSim* was introduced, based on this approach. It was a massively parallel commercial SPICE implementation, designed to run on GPUs by Nascentric. The product was based on the NVIDIA's Tesla hardware platform and claimed that "Using NVIDIA's Tesla platform we can perform circuit simulations in minutes to hours that would previously have taken hours, days and weeks," [1]. The company is now defunct and assets have been purchased by a customer [13].

These implementations hold significant results and insight on the steps that can be taken to parallelize SPICE using this kind of hardware. Using hardware specially designed for the task at hand can give tremendous speeds at the cost of performing platform-specific alterations to the applications, in order to enable efficient mapping. This can be interpreted both as cost of configuring and maintaining the hardware as well as the cost of porting the software from one platform to another which can be a task of significant effort. On the other hand, working with generic hardware allows great versatility and portability which seem to be key concepts for application design.

2.2.3 SPICE in Generic Hardware

When one wants to parallelize SPICE for generic hardware it's common to work in the SPICE source. This enables the developers to have fine-grained control over the exact structures to run in parallel. **Ngspice** has a series of OpenMP pragmas for this exact purpose as shown in Figure 2.2. Commercial products have also provided this functionality which can take advantage of the multiple cores of the running platform. That way the developers improve the application speed without compromising its versatility. Moreover, using industry standard architectures and setups, the end users can run the SPICE application of their preference on the

```
#pragma omp parallel for num_threads(nthreads) private(here)
for (idx = 0; idx < model->BSIM4InstCount; idx++) {
    here = InstArray[idx];
    good = BSIM4LoadOMP(here, ckt);
}
```

Figure 2.2: OpenMP pragma in ngspice BSIM loading routines

hardware they already own.

This approach seems to apply on the new multiprocessor context towards which the computer systems market has shifted [4, 3, 16, 7]. This assumes that the runtime environment has enough resources to finalise the simulation at hand. However this is not always the case. In unified memory hierarchies the requests for memory allocation consume a common resource. Thus many researchers and circuit designers may reach the memory wall, when running transient simulations with many devices and using complex device models. In order to solve this problem many design houses invest large amounts of money for computer systems with massive resources [10]. Another angle to attack this issue would be to design the SPICE application on a multiple machine context. This would mean that one would use a Message Passing Interface (MPI) framework over a computer cluster context or a grid.

The aforementioned approaches require a lot of development time, specifically for each runtime system in order to achieve optimality. This means that one has a solid understanding of the target platform and can work on the source code of the SPICE application. But when EDA software companies sell their products they obviously don't release the relative source code. The design houses rely on the EDA software companies. Features of a computer system such as cache size and memory size vary greatly among processors even if they belong in the same family (x86,x86_64). Thus it is an extremely difficult task to build software that runs optimally on a range of platforms even if this range is restricted by the underlying architectures. So relying on the EDA software companies cannot always produce optimal results since the tools provided are not flexible enough. From the above observations, it is highly pragmatic to treat SPICE applications as *black boxes*.

The monolithic *black box* paradigm doesn't provide the end users with the required flexibility one would need from an EDA tool. So instead of attempting to parallelize the EDA tool by performing alterations in its code base, we can treat it as a *black box* and perform development on top of

an existing SPICE executable. Running multiple smaller instances of the monolithic simulator, while breaking the initial simulation into multiple smaller ones, we can execute the partial simulations independently. That way we are able to parallelize an application without really knowing how it operates internally.

2.2.4 Industry Standard Tools

The work of this thesis is going to be compared to industry standard SPICE applications. Parallel implementation of SPICE is not a new idea. Significant work has been done on part of the EDA industry. Many leading companies in the field have put effort into building parallel versions of their software. Two products that have such capabilities are **hspice** by *Synopsys* and **spectre** by *Cadence*. In the context of this thesis we used those commercial tools in order to have an objective and realistic estimation of the correctness and the computational viability of our proposal.

Spectre features

Specte is a Circuit Simulator software developed by *Cadence*. It claims to provide improved capacity over other simulators due to effective convergence algorithms for large circuits. Dynamic memory allocation, according to the manual, allows the Spectre circuit simulator to use less than half as much memory as SPICE for large circuits. The developers also claim that it runs two to three and two to five times faster than SPICE for small and large circuits respectively. Models and accuracy are also improved.

HSpice features

HSpice is a Circuit Simulator by *Synopsys*. It claims to be a performance leader in the field on both single and multi core computers. It features a client-server model that boosts the performance and improved integration. A significant speedup for large netlists is also claimed.

Summary

Both of those tools are well known and industry standard. They enable the utilisation of threads and numerical methods to exploit the capabilities of

the multi core architectures available in modern processing environments. For our implementation we used the open source alternative **ngspice** and compared the results and resource consumption with the commercial tools.

2.3 Motivation

2.3.1 SPICE in Variability Simulation

The past 10 years in digital circuit design have been a race on the integration levels and the number of transistors per silicon die. This trend has led manufacturers to very high integration levels up to a point. Latest technology has reached a significant bounding factor, device variability.

In other words the smaller the gate lengths of the transistors, the more they exhibit a stochastic behaviour [15]. The greater the integration factor of a circuit the greater the variability it presents. Thus there is a need for variability verification for very large circuits [14]. This has lead to the conception of a new field where circuit simulation can provide insight, that of *Device Variability*.

In order to verify the behaviour of a circuit, designers often have to resort to extensive simulations. Particularly for time zero device variability, engineers often have to resort to Monte Carlo simulations which are immensely intensive [9]. This methodology involves iteration of the same simulation, while sampling various random variables that represent the degrees of circuit variability. The pool of statistical samples has to be large enough to cover all of the n-stochastic-variable space. These simulations are very intense with regard to computational power and also have great need for system memory due to the size of the circuits.

2.3.2 SPICE in Time-Dependant Verification

In circuit design it is often necessary to study phenomena that evolve in the axis of time. This requires very long transient simulations. Such simulations require large amount of computational power in order to complete and are often brought to a halt due to insufficient resources. This kind of Time-Dependent Verification in circuit simulations requires viable simulations over extensive workloads.

As stated in [15] digital circuits behaviour varies, not only at time zero (immediately after manufacturing), but also as the lifetime of circuit progresses. As argued in [14], in order to account for workload dependent circuit variability, engineers need to perform transient simulations of long workloads. These simulations require large amounts of time and computational resources.

2.3.3 Summary

In the context of this thesis, we claim that we can benefit greatly by providing viable and relatively fast SPICE simulations. As circuits continuously grow to larger inventories it is understood that simulations require continuously greater amounts of resources. The memory wall is a bound that we need to overcome in order to be prepared for the circuits to be simulated for the years to come. Both Variability Verification and the study of Time-Dependant phenomena provide a target group that requires simulations that evolve over a large range of time for large circuits. Thus we conclude that researchers, designers and developers in these fields can greatly benefit from faster and less resource hungry simulations.

Chapter 3

Data Partitioning SPICE

3.1 Introduction

In our implementation we used a series of techniques that are applied when handling big data, *data partitioning* in particular. This chapter describes the tools and techniques we used to apply the data partitioning principles in circuits simulations.

The phases of *data partitioning* consist of the *Domain Decomposition*, a process when the data is split into subsets, the *Execution*, when the application runs performing the actual calculation, and the *Data Recollection* when the distributed data is recollected and reconciled in order to produce the final results.

The suggested framework consists of an input signal **partitioner**, a generic **hypervisor**, a SPICE kernel **wrapper** and a data **miner**. This set of tools enables the handling of workload tearing, task dependencies, output forwarding and data mining. An abstract visualisation of the framework is shown in Figure 3.1.

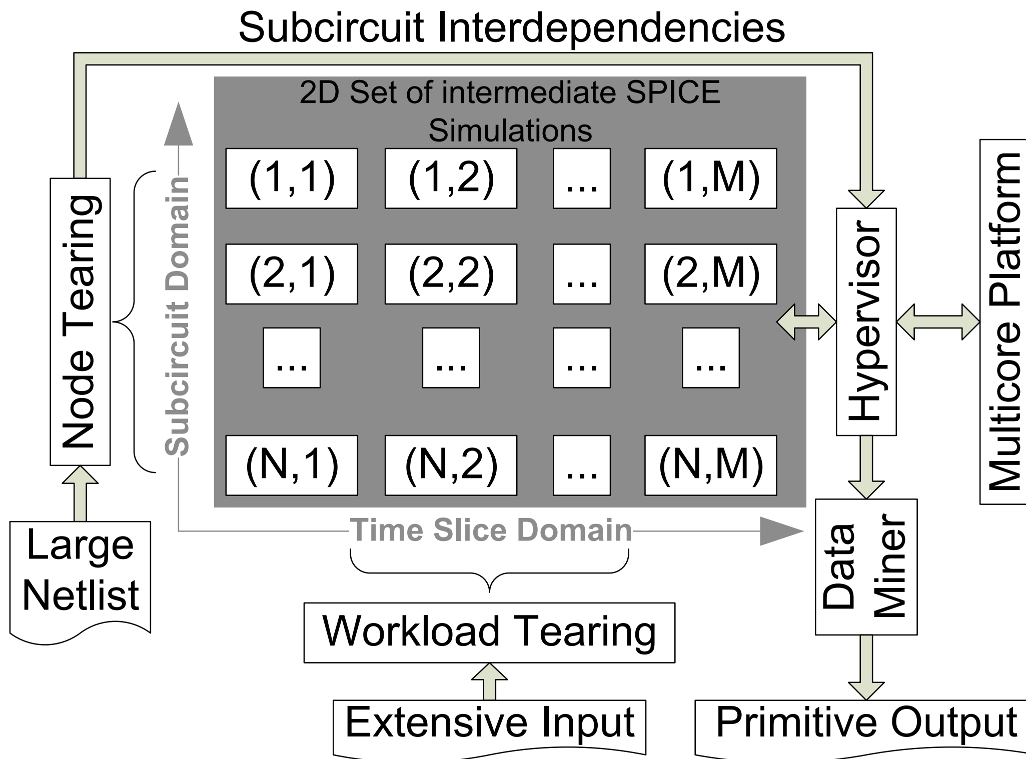


Figure 3.1: Execution Framework

3.2 Principles

3.2.1 Theoretics

The leading principle behind this project is *data partitioning*. We apply two concepts of partitioning, on the time axis which will be further referred to as *workload tearing* and on the circuit level which will be referred to as *node tearing*.

Workload Tearing

Workload Tearing consists of breaking the input signals on appropriate time slices. One of the challenges was to identify those time slices, and the overall timeslice duration. Input signals are in the form of PieceWise Linear (PWL) sources described in text files. This form is easily parsed and can be used directly as input following specific naming conventions described in the next section.

For slicing the input signals we used two methods. For the first version we generated PWL signals that had a specific form. The pulses were in a regular pattern and each pulse contained four points as shown in Figure 3.2. When the input signals have this form, one can easily partition the signals with the use of a mod counter every four points and generate the partitioned output signals. However, in the general case that may not be applicable. Input signals might not be that regular due to the characteristics of the simulation at hand. Thus a new more advanced partitioning scheme needs to be applied.

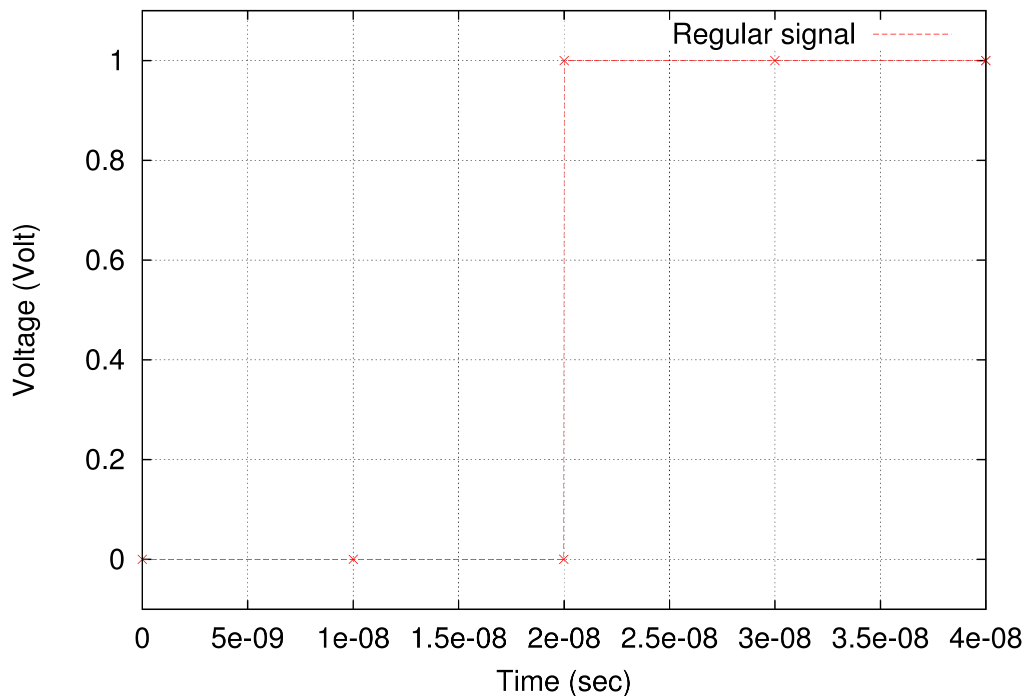


Figure 3.2: Mod counter signals

In order to tear the input signals we need to specify timeslice durations which will be used for the partitioning proceed. Each timeslice has to contain whole pulses of the input signals meaning that there has to be a minimum amount of time after a transition otherwise the accuracy of the simulation will be compromised. In order to achieve that we use a simple but effective method. Since we only perform digital simulations we can safely partition in times where all signals are constant and in a stable state. By stable state, we define one of the two states: logical 0 or logical 1. This way all the transitions will be contained in the partitioned signals.

Additionally we define a minimum time after each transition. We identify the regions where all signals are constant simultaneously and we perform the slicing at these parts.

One should note that we are dealing with digital signals. In case of analog signals that are not latched to constant voltage levels, one would have to interact with the DC operating point solver of SPICE. This task falls beyond the current thesis.

Node tearing

Node tearing is a well-established technique and many approaches have targeted efficient implementations. In the context of this thesis, we do not elaborate on efficient node tearing algorithms but exploit the modular syntax of a SPICE netlist with a specific subcircuit that is replicated. This requires the user to define the components of the circuit at hand and the dependencies amongst them. After this analysis is complete then the end user can easily compose the configuration file that will be parsed by the **hypervisor** of the framework. This file will describe all the requested input files and all the output files for the simulations along with the dependencies. The user has to follow the same naming conventions described in the next section.

Signal filtering

When performing digital simulations we comprehend that we don't need multiple points when the signal is in a stable state. Where stable state is defined by the logical 0 and 1 values. SPICE uses adaptive step sizing to stride over such intervals. So while a signal is in one region or the other we only need two points to identify its value, since it's essentially constant. However, the adaptive step sizing, as implemented in ngspice, is highly suboptimal and produces many points. As a result, signal filtering is required, in order to avoid storage of useless time,voltage points between intermediate simulations.

Keeping the above in mind we can devise a filtering function that replaces all the points in a constant signal region with only two points. This will apply only in the logical 0 and logical 1 states of the signal while leaving the intermediate points intact. This procedure can greatly decrease the signal size. In some cases even an order of magnitude, with no observable loss in data. That we can greatly reduce the number of points kept in the signals

and thus improving performance and reducing the memory footprint of the output PWL files of each simulation.

While having this piece of information we can apply an even more crude filtering method if it proves crucial to increase drastically the performance. If in our simulation we claim that we do not care about any intermediate points, we can simply ignore them and perform filtering based on that assumption. This will ensure the preservation of transition information for digital simulations while minimising the signals' size.

Execution Grid

Instead of task oriented parallelization we implement a data oriented parallel framework. The two axis of partitioning, *Workload Tearing* and *Node Tearing* constitute a 2-D space of simulations as shown in Figure 3.1. These simulations will be handled by the framework and run on the given platform.

3.3 Implementation

In Figure 3.3 we observe the set of tools implemented, namely the hypervisor (hyper), the main execution kernel (sapp), the partitioner (party) and the data miner (aspirin). The figure also shows the workflow in order to use the framework. The process is rather simple. Initially one has to partition the input signals with the use of **party**. Afterwards one will generate the execution graph and spawn **hyper**. **Hyper** will parse the execution graph and spawn as many instances of **sapp** are needed to complete the simulation. Finally the user will spawn **aspirin** in order to reconcile the output signals.

3.3.1 User Manual

Netlist generation

The netlist generation process is simple. The user has to define the circuit as a number of independent components that can be simulated separately. These subcircuits have a number of inputs and outputs. These have to be enumerated in order to be substituted properly by the forwarded input signals.

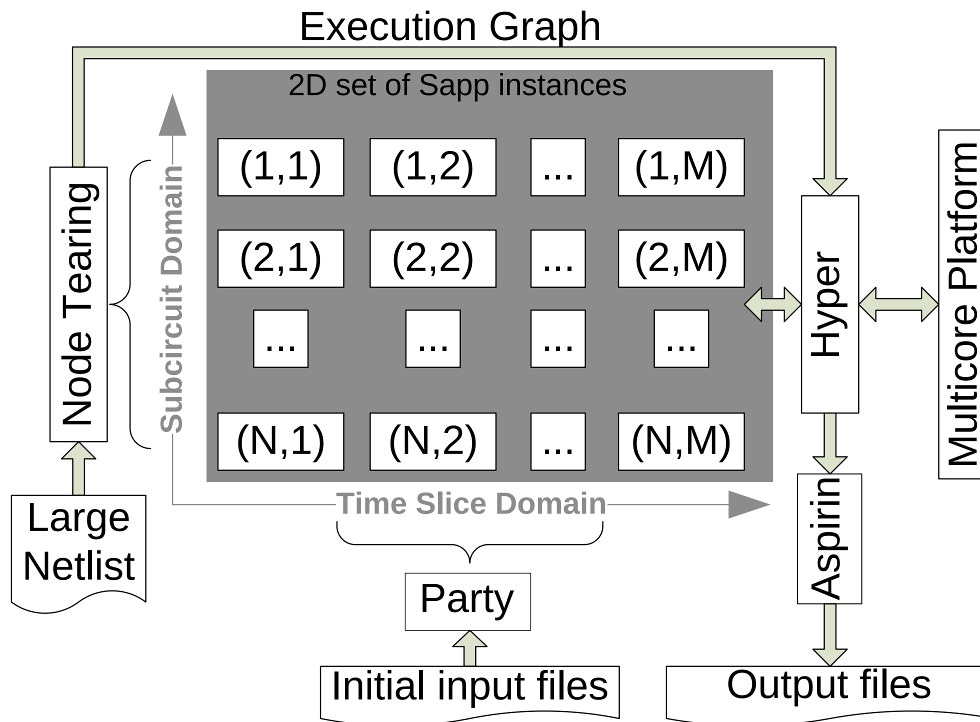


Figure 3.3: Execution Framework Toolchain

The inputs will be handled by the wrapper provided that the netlist of the subsircuit uses a simple naming convention. The names of the input signals in the netlist wiring instantiation need to be in the form *"inX"* where X is the number of the input starting from 0. What's more, in the same netlist there need to be a series of include lines in the format *".include INPUT%03d"*.

The simulation duration can either be constant and common for all timeslices or be different due to custom partitioning. If the first is the case, then the user must ensure that the runtime folder does not contain a timeslice log file. The timeslice log file is named always *"tslices.log"* so it is easy to recognise and remove if necessary.

In the case of custom timeslice duration the timeslice log file is of utmost importance. This special file is part of the framework's partitioning mechanism and provides the start times and durations for each of the timeslices. This file has always the same name *"tslices.log"* and should be handled with care. It is in essence a two column data file in the format "<start time> <duration>". In the general case this file should be generated by the **partitioner** and used by the wrapper and the miner. When this file

is present in the directory running the simulation, the wrapper reads it's contents and replaces the ".tran" line in the netlist filling in the proper duration for each execution. So one should be careful not to delete it by mistake since the wrapper as it is at the moment will simply ignore the duration parameters leading to possibly erroneous results.

Output handling follows a similar approach to the input signal handling. The ".print" line in the netlist needs to hold the signals to be forwarded to the next simulation before any other signals of interest. These do not need to follow a specific naming convention but the same numbering as in inputs should be applied to avoid confusion. Below (Figure 3.4) is a sample netlist following the aforementioned conventions.

```
** A 2 x 2 multiplier based on the FA* module **

** Subcircuits & Modelcards **
.include modelcard.nmos
.include modelcard.pmos
.include subcircuits.cir

** DC Sources **
Vvdd vdd 0 1.0
Vvss vss 0 0

** Multiplier Instantiation (per FA* Module) **
X0 vdd vss vss   in0 in2 vss   product0 wire0    module
X1 vdd vss vss   in1 in2 wire0 wire1    wire2     module
X2 vdd vss wire1 in0 in3 vss   product1 wire3    module
X3 vdd vss wire2 in1 in3 wire3 product2 product3 module

** Inputs **
.include INPUT001
.include INPUT003
.include INPUT002
.include INPUT004

**Simulation Definition **
.tran 800.000000ns 800.000000ns

** Output File Definition **
.print tran v(product0) v(product1) v(product2) v(product3) i(Vvdd) i(Vvss)
```

Figure 3.4: Sample netlist

Wrapper (Sapp)

The wrapper we used for *ngspice* is a pure *ANSI C* tool that was developed for this project. Its role is to prepare and execute an *ngspice* simulation and harvest the outputs. **Sapp** (the wrapper name) is statically linked against the *microsig* signal library developed for this project. As it can be seen in Figure 3.5, **sapp** parses all the information it needs as command line arguments. It copies the contents of the input files to its private namespace as designated by the timeslice id and circuit id and generates the PWL signals that will be used for the particular simulation. It also copies the netlist module and then inserts the PWL input filenames in its personal netlist module. If a timeslice log file is available it reads it and inserts the appropriate ".tran" line in its netlist. Otherwise it ignores this step.

At this point the simulation is ready to start. So it is executed with the use of the `system` command. If the simulation finishes successfully the wrapper is going to parse the simulation output. While in this step the wrapper will also apply a simple 0-1 high-Z filter on the output signals in order to decrease its size. Afterwards it writes each output as designated by the command line arguments <output 1> <output 2> and so on. After that step the wrapper will delete all intermediate results from the folder to save space. If the simulation finishes with an error status code the wrapper will not try to parse the erroneous output nor delete the files to enable debugging.

```
sapp <timeslice id> <circuit id> <number of inputs> <input 1> <input 2>
        .. <number of outputs> <output 1> <output 2> .. <netlist module>
```

Figure 3.5: Sapp invocation

Input signals generation

The input signals are initially waveforms represented in the PWL format. This is a well known format used by all typical SPICE applications and is used to great extent. Signals in PWL format are relatively easy to parse and generate. The framework can effectively use any set of signals already generated for simulations and does not require any specific handling. When the initial signals are prepared all the end user has to do is invoke the **partitioner** with the PWL files.

The **partitioner** is a simple tool written in *C* using the microsig library which was developed for this project. The partitioning process for the

user is as simple as running one shell command (Figure 3.6). Beware of the double quotes " in the command. It is important to add the quotes so that the shell will not expand **.in* to all the files in the folder. It is often needed to partition a large number of files and passing them all as command line arguments is impractical. The file pattern is used instead and the **partitioner** searches the current directory for all the files that satisfy the pattern.

For example if we use the command in Figure 3.7 the **partitioner** will search the current directory for all the files that have the suffix ".in". Thus files such as "a0.in", "a1.in" and so on will be partitioned in at most 10 timeslices. Due to the partitioning theory limitations at the moment, the **partitioner** might be unable to provide the requested timeslices. However it will attempt to provide as many as possible. For reasonable slicing the process will provide the requested number of slices.

It is important to note that the initial files are not erased written over or otherwise tampered with. The **partitioner** will use the filename as a base and create a series of files in the format "%04d_base.prty". This ensures that no matter what the initial files are the output files will not overwrite or corrupt the input. The output files are simple two column raw signals that are handled in the runtime by the wrapper.

```
partitioner <requested timeslices> "<input file pattern>"
```

Figure 3.6: Partitioner invocation

```
partitioner 10 "*in"
```

Figure 3.7: Partitioner invocation example

Data mining - Output signals recollection

When the simulation is complete multiple output files will be generated. For every signal there will be n output files containing part of the full output where n is the number of timeslices. These output files will be zero aligned and have to be collected in order to produce the complete output signal. In order to achieve this we created a miner that takes into consideration the number of timeslices the timeslice log in order to generate the final signal. As shown in Figure 3.8 the miner also takes the signal base name as input. The output signals are named in a specific manner with the timeslice index as a prefix (000_<base name>, 0002_<base name>..). Thus the miner

will gather all the signals that have the same <base name> as a suffix and merge them to create the final output. In Figure 3.9 we call the miner for 10 timeslices using the file tslices.log to collect all the 01_vsout bit signals.

```
miner <slices> <timeslice log> <base name>
```

Figure 3.8: Miner invocation

```
miner 10 tslices.log 01_vsout
```

Figure 3.9: Miner invocation example

Task graph generation

One of the most complex tasks one has to complete in order to use the framework is the task graph generation. This constitutes the core of the implementation. The task graph will be parsed by the **hypervisor** and provide all the information that is needed to run the entire simulation.

The *execution graph* or *task graph* is a text file. The first line contains the program to be executed, which in our case is the **ngspice** distribution installed in the executing platform. Each of the following lines contain a task. A task line is as shown in Figure 3.10. The arguments are the command arguments that will be passed to the executing program. For example we can have an execution graph as shown in Figure 3.11. The **hypervisor** understands that the task WORLD depends on the task HELLO. In that context it will execute "/bin/echo Hello" which will print "Hello" and afterwards it will execute "/bin/echo world" so in the end it will print "Hello world".

```
<task name> <number of dependencies> <dep 1> <dep 2> .. <arguments>
```

Figure 3.10: Task Line

Hypervisor

The **hypervisor**, as shown in the task graph generation process, is a completely agnostic tool with regard to what it is running. The **hypervisor** runs only one program, the one specified on the first line in the configuration file. This design choice may seem a bit rigid but allows great flexibility. The invocation process is again simple. As shown in Figure 3.12 the only

```
/bin/echo
HELLO 0 hello
WORLD 1 HELLO world
```

Figure 3.11: Hello world

options needed are the execution nodes available and the task graph that was created beforehand. The example in 3.13 shows the command that we used to run the **hypervisor** for 48 execution nodes (the SCC).

When the **hypervisor** parses the file it creates multiple tree structures in the system memory representing the dependencies. All roots are available to be executed since they have no dependencies. Each available task can be executed as long as there are resources (execution nodes) available. When the task completes the resource is released and another task can be executed on that node. This is practically implemented with the abuse of the return values when the processes die. The scheduling algorithm is a simple first come first served algorithm using dual ended queues. Prioritisation could be handled if instead of queues we would have heaps but that is left as future work.

```
hyper <available execution nodes> <input file>
```

Figure 3.12: Hypervisor invocation

```
hyper 48 executionGraph
```

Figure 3.13: Hypervisor invocation example

3.3.2 References to Source Code

Wrapper (sapp)

Sapp is a C application used to prepare and run an ngspice kernel and afterwards break the ngspice output file to simple two column files per output bit. It uses the microsig library developed in the context of this thesis. The source code contains a properly written Makefile for it's compilation. It uses two macros in order to specify the command to be run. When compiling with the RUN_ON_SCC macro defined, the ngspice command is the ngspice compiled for the SCC in a path the SCC can read. This can be extended with the use of multiple RUN_ON directives and equivalent NGSPICE_CMD_LINE macros. The macros are defined in the file

"sapp_common.h" and the specific part regarding the command executed is shown in Figure 3.14.

```
#ifndef NGSPICE_CMD_LINE
#if RUN_ON_MITSOS
#define NGSPICE_CMD_LINE "/shared/master/bin/ngspice -b %s 2>  %04d_%05d.log > %04d_%05d.out"
#elif RUN_ON_SCC
#define NGSPICE_CMD_LINE "/shared/master/bin/ngspice -b %s 2> %04d_%05d.log > %04d_%05d.out"
#endif /* where do you want to run */
#endif /* NGSPICE_CMD_LINE */
```

Figure 3.14: sapp_commmon.h

Hypervisor (hyper)

Hyper is a C++ application used to handle the graph of task dependencies describing the simulations. It parses the execution graph and spawns any task that is ready to run on the available execution nodes.

Run process

The run process is simple. When there are available execution nodes and jobs that are ready to be spawned the hypervisor forks to new process and spawns task on the given platform by executing the run function. The run function is currently defined at compile time. The run function for the platforms implemented currently uses the `system` command in order to spawn the configured application. When the run is complete the forked process exits using the execution node id as an exit value.

When the father process runs out of execution nodes or runs out of processes that can be spawned it performs the `waitpid` system call in order to release an execution node that was previously used. When a process exits the hypervisor identifies the execution node and, with the use of a lookup table, the task that was running. For the task that was running and the hypervisor decreases the reference counter of it's children. If the reference counter reaches zero the child is ready to be put in the available jobs queue.

The exit code conception limits the number of parallel processes to 256. This limitation can easily be overcome with the use of a multi-level spawn process. As an example we can spawn a hypervisor that spawns 256 hypervisors that can in turn handle 256 execution nodes each. Thus the number of parallel processes that can be handled increases exponentially. Extension of the number of parallel processes can also be performed with the use of the pthreads API or any equivalent framework.

Data Structures

In this section we explain some of the basic data structures that were developed and used in the hypervisor.

Task

The task is a simple data structure (Figure 3.15) containing the command to be executed as a string, the number of dependencies and a vector of the indices of it's children. This allows the hypervisor to decrease the dependencies of the children of every task that is completed.

```
class task {
        std::string command;
        int dependencies;
        std::vector<int> children;
    public:
        task(const task &N);
        task(int deps, std::string cmd);
        ~task();
        bool decDeps();
        bool ready();
        void addChild(int i);
        const char *getCmd();
        int childrenN();
        std::vector<int> getChildren();
        void print();

};
```

Figure 3.15: Task definition

Execution Nodes

For the handling of the execution nodes the data structure is a simple dual ended queue. During the initialization stage the number of execution nodes indicated by the command line arguments is inserted in the deque. When one job needs to be started, if there are available execution nodes, the hypervisor pops the front of the queue and spawns the job on that execution node using the run function defined for the given platform.

Partitioner (party)

The **partitioner** is a C application developed for the partitioning of input signals. It handles PWL input signals and outputs the partitioned signals as described in the previous section. For the representation of the signals it uses the microsig library signal datatype as shown in Figure 3.16.

```
struct _signal {
    uint32_t len;
    uint32_t alloc;
    double *time;
    double *value;
};

typedef struct _signal signal;
```

Figure 3.16: Signal definition

Miner (aspirin)

The **miner** is a C application that handles the recollection of signals after the simulation is complete. It uses the same library as the **partitoner** for the signal handling. Specifically it parses the 2-column signals, shifts them according to their respective timeslice offset, and appends them in the right order.

Chapter 4

Benchmarks & Results

4.1 Introduction

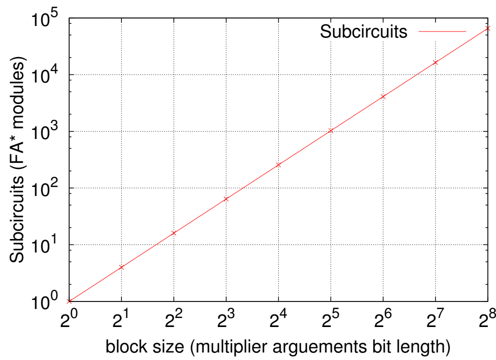
In this chapter of the thesis we provide details regarding the benchmarks we performed and their respective results. The simulation outputs were compared with the reference values and we also compared the execution times of the hypervised version and the reference tool.

During the verification process of this thesis we performed two benchmarking sessions. During the first session we compared the results of the simulations using the results of **hspice** as reference values. For the second benchmarking session we improved the hypervised version and used **spectre** as the reference tools.

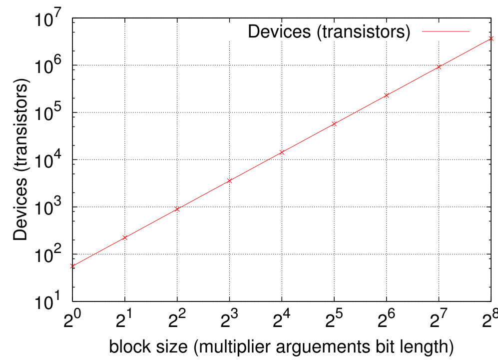
4.2 Simulation Description

To verify the proof of concept of the implementation we run a set of benchmark simulations. The circuit of choice was a typical multiplier described in [17]. This circuit features great modularity and scales easily as shown in Figures 4.1(a) and 4.1(b). This inherent quality of the circuit allows the verification process to explore large device inventories. Obviously, the modularity of the circuit provides great speed and efficiency when it comes to simulations with the use of our tool. However, even circuits that are not characterised by such modularity can use our tool since its benefits come also from the *Workload Tearing* process.

The decisive factor when using our tool was the type of the simulation.



(a) Subcircuits



(b) Transistors

Due to the algorithm of the filtering process and the concept of output forwarding, we limit the application range to digital simulations in order to be able to use the partitioning methods described in Chapter 3 and the filtering methods described in Section 3.2.1. In conclusion, roughly any Digital Circuit Simulation can be handled with the use of our tool.

4.3 Results

One of the objectives of this process was to evaluate the validity of the approach. Thus we extracted the output signals from our output results and compared them to outputs derived from industry standard tools. The output bits of the multiplier are the p_i bits shown in Figure 4.1. For each benchmark run, we had two data sets of the bit outputs of the multiplier. One set was the output of our tool and the second set was the output of the reference tool.

The output of our tool was a set of files, each for one output bit. The files themselves are simply two column data files. The first column being the time and the second the voltage values.

For the first benchmarking session the reference tool was **hspice** by *Synopsys*. The outputs were in `.tr0` format. In order to parse these files we developed a miner for the specific output format.

For the second benchmarking session the reference tool was **spectre** by *Cadence*. The outputs were in `.raw` format. For these files a simple `awk` script was used to extract the useful pieces of information from the data files.

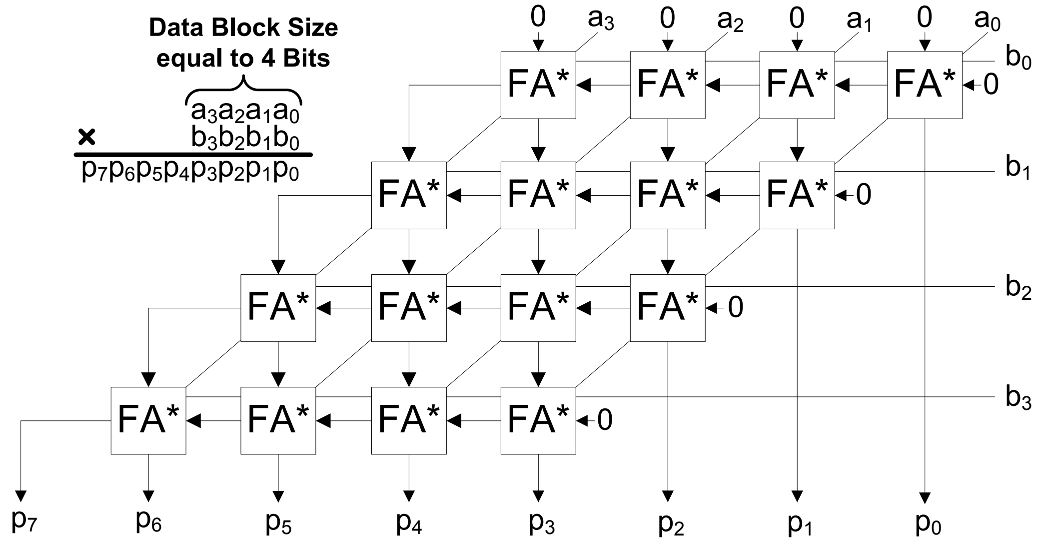


Figure 4.1: Inputs and Outputs of a 4x4 multiplier

The two digital signal sets (our set and the reference values) have different and variable sample rates. Thus in order to be able to calculate their difference we had to interpolate the signals. Instead of interpolating and performing text handling on the signals we calculated the difference between the two signal sets as shown in Equation 4.1 using a different approach. In order to calculate $diff_i$ we used an ideal subtractor as shown in Figure 4.2. This technique was used in order to make the data verification more accurate and less strenuous.

In a later step we extracted the difference signals per output bit. Then we calculated the Root Mean Square Error (RMSE) for each of these difference signals as shown in Equation 4.2. The process is graphically presented in Figure 4.3.

$$diff_i = reference_bit_signal_i - hyper_spice_bit_signal_i \tag{4.1}$$

$$RMSE_i = \sqrt{\sum_{j=0}^{n} \frac{(x_j - mean_i)^2}{n}} \tag{4.2}$$

The development progress had two major steps. The first round of results has been published in [8].

```
** A signal subtractor for 2 pairs of signals **

** VCVS instantiations
E0        output0       0        input0        input1        1
E1        output1       0        input2        input3        1

** Sources **
.include input0.in
.include input1.in
.include input2.in
.include input3.in

** Simulation Definition **
.tran 800.000000ns 800.000000ns

** Output File Definition **
.print tran v(output0) v(output1)
```

Figure 4.2: Subtractor of two signal pairs

4.3.1 First Benchmarking Session

During the first round of results the tool we compared against was **hspice** by *Synopsys*. We extracted the bit signals directly from the .tr0 output files and compared them bit by bit to the outputs of our tool.

As shown in 4.4 it is apparent that the approach we implemented was not far off the target. In fact, the errors values are as low as 10^{-18}. Due to the RMSE process as described above in 4.2 the error is calculated in Volts. Thus we reach the conclusion that the process remains accurate.

Unfortunately that can only be verified up to data block size of 2^5 as the commercial tool failed to produce results further due to bad allocation errors. This fact proves the necessity of our tool since **hspice** cannot produce results for data block sizes greater than a 32 by 32 bit multiplier, given the memory wall of the executing system.

As shown in Figure 4.5 the prototype idea keeps producing results within a viable time frame up to 2^7 with almost linear times with regard to the circuit size on all the platforms tested.

Finally in Figure 4.6 we see a comparison of execution times between the **hspice** and *hypervised spice*. The latter seems to execute at a slower pace for the first smaller circuit sizes, however when the netlist becomes

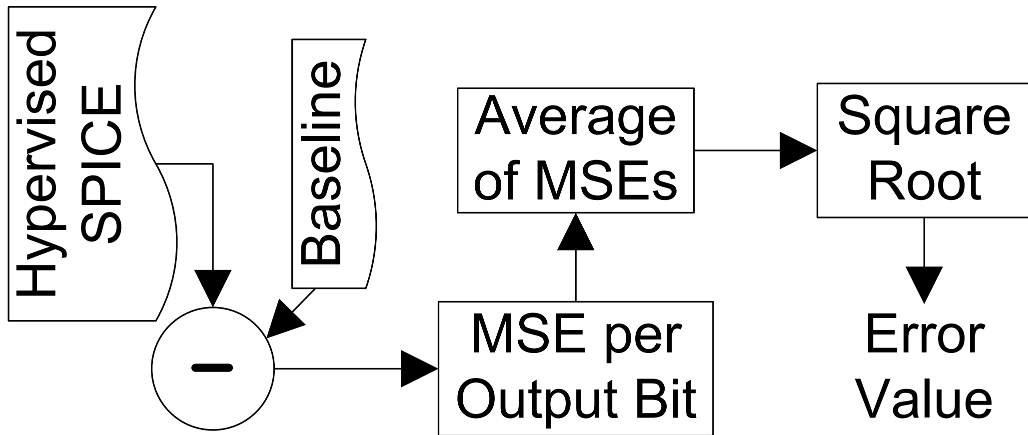


Figure 4.3: Error calculation processj

substancially large the tool provides faster execution times.

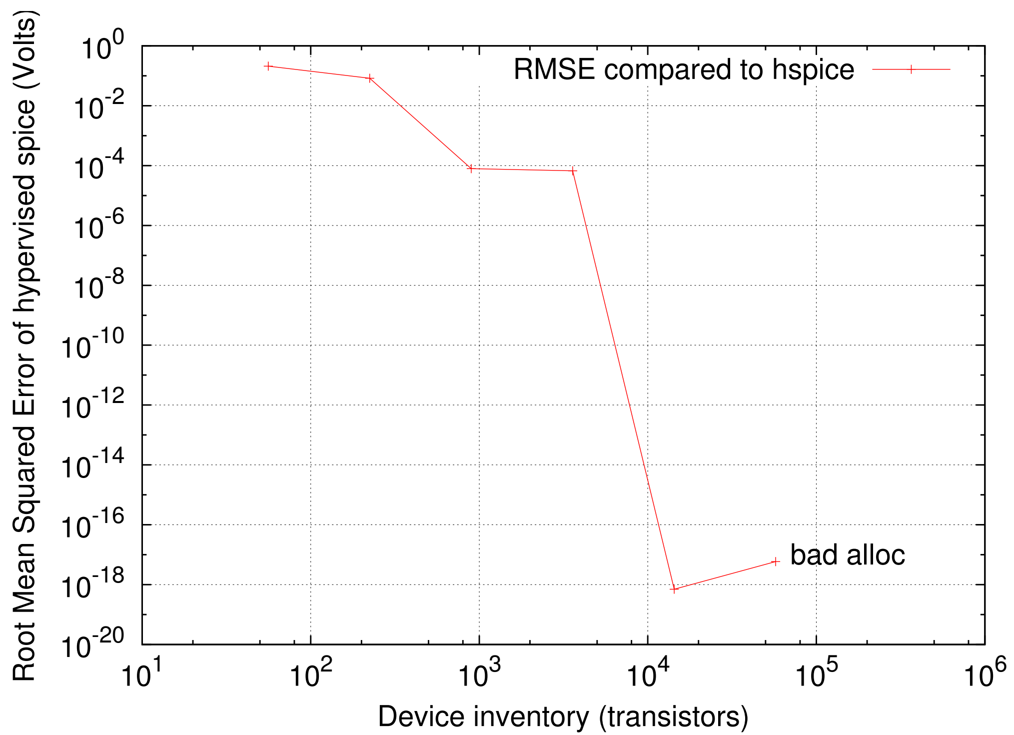


Figure 4.4: RMSE error compared to hspice results

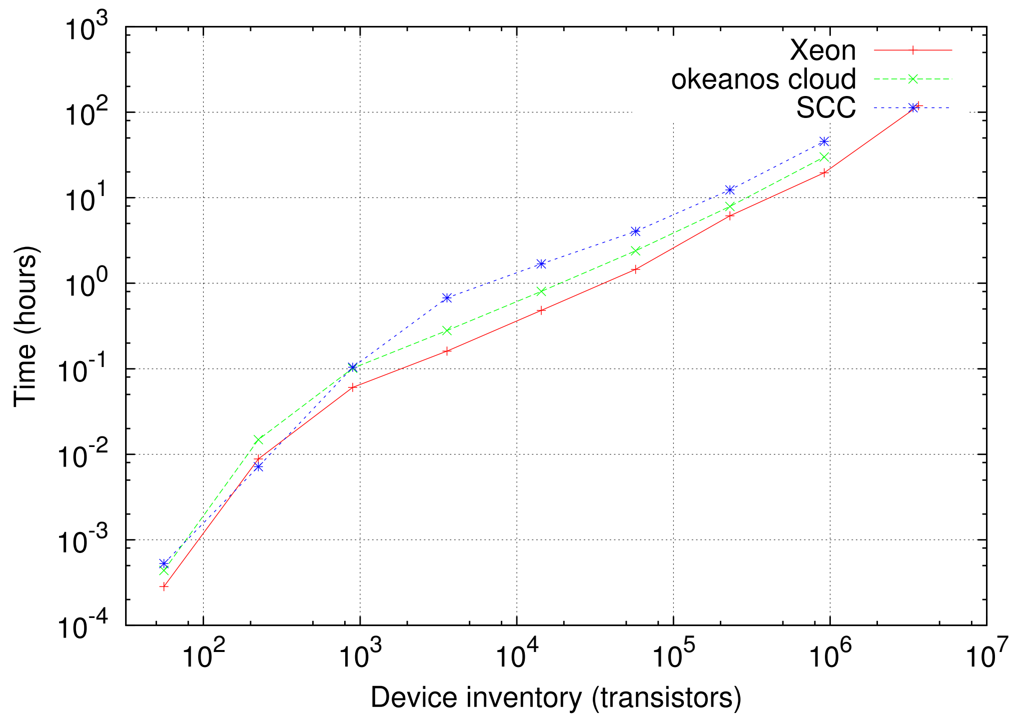


Figure 4.5: Cross platform execution times v1

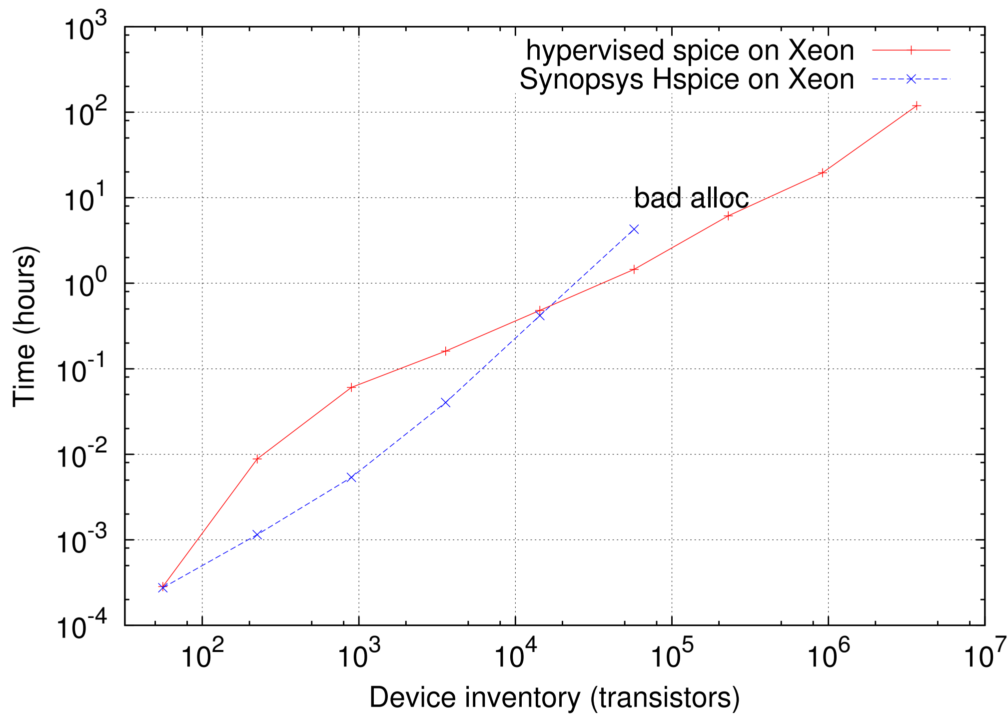


Figure 4.6: Comparison of times of v1 against hspice

The first set of results led to a series of realisations. One of the first design choices that were made during the development of the *hypervised spice* tool was that all the signal forwarding operations will be conducted over the file system. This way we can overcome the memory wall but the strain on the file system is significant. Thus we can understand that the IO of the application, at the level of produced (and later consumed) PWL text files, appears to be the a bottleneck.

4.3.2 Second Benchmarking Session

The *hypervised spice* tool is bound by the file system. So it becomes necesary that one should minimise the IO in order to increase the performance. During our benchmarks we had a series of file systems, one for each tested platform, that were degrading the performance. In order to overcome that barrier we took two simple but significant measures.

As a first step we incresed the timeslice duration. With the more advanced partitioning technique as described in Chapter 3 we could regulate the partitioning size. Since we no longer needed the input signals to be broken by

a mod counter we could regulate in a finer manner the number of timeslices and consequently their duration.

With a smaller number of timeslices for the same total duration it is apparent that the timeslices for the second benchmarking session were larger. With larger timeslices, the number of accesses to the filesystem were reduced by a factor of aproximately 0.75. This is evident if we keep in mind the number of timeslices for the second round of simulations. During the First Benchmarking Session we used 40 timeslices for a lifetime of $20ns$ each. During the Second Benchmarking Session we used 10 timeslices, with an average duration of $80ns$ each. Thus we reduced the number of executed simulations and, as a result, the number of output files by the aforementioned factor. So having only one fourth of the initial load we expected an increase in the performance.

That was not the only action point. During the first set of experiments we observed large signal files that had to be transfered between simulations. When taking a closer look to the matter, one one particular case, the number of data points that were forwarded exceeded the number of 60000 for a simulation duration of merely $20ns$. Thus we realised that we needed a method to decrease the signal size, effectively reducing the number of data points that were frowarded between simulations.

From this process the concept that we explored was the idea of signal filtering. Thus we applied the filtering technique described in Section 3.2.1 in order to decrease the number of data points copied from one simulation to the next.This technique had two significant impacts. The first was of course the reduction of the strain on the file system. Smaller data files are faster to read, copy and process in general. The second impact was the duration of each simulation. With less points to parse, **ngspice** was a lot faster to converge.

4.3.3 Results

As show in Figure 4.7 the results during the second benchmarking session were degraded regarding the errors. The main factor that these errors can be attributed to is the filtering process. This was an area we expected to present such a descrepancy and requires further development.

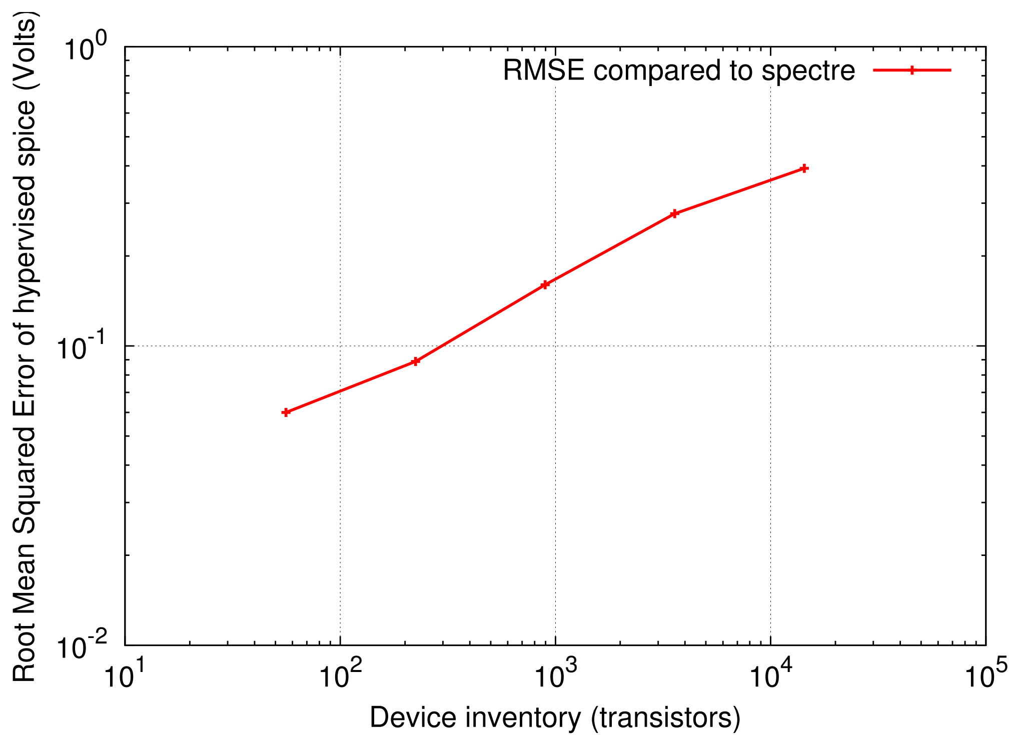


Figure 4.7: RMSE error compared to spectre results

In order to understand why this appears we need to refer back to the filtering process. In the filtering process we search the signal for transitions. When performing digital simulations the transitions are signified by two values. If V_1 is the high voltage level and V_0 is the low voltage level any value below $V_0 + (V_1 - V_0) \times 0.1$ is considered low and any value above $V_0 + (V_1 - V_0) \times 0.9$ is considered high. So in the final stage of the filtering process we reset the values in this range to their representatives V_0 and V_1 respectively. This process inserts a minor offset during the executions as shown in Figures 4.8, 4.9. This offset is propagated as the results are forwarded the outputs are filtered again, thus leading to these errors.

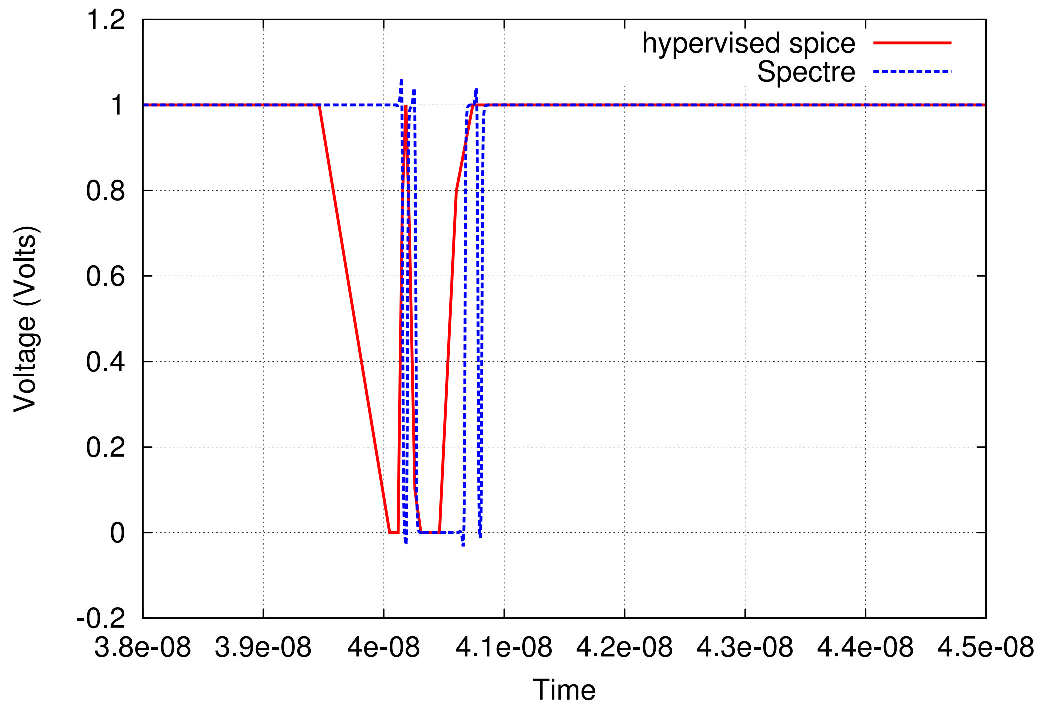


Figure 4.8: Multiple transitions error

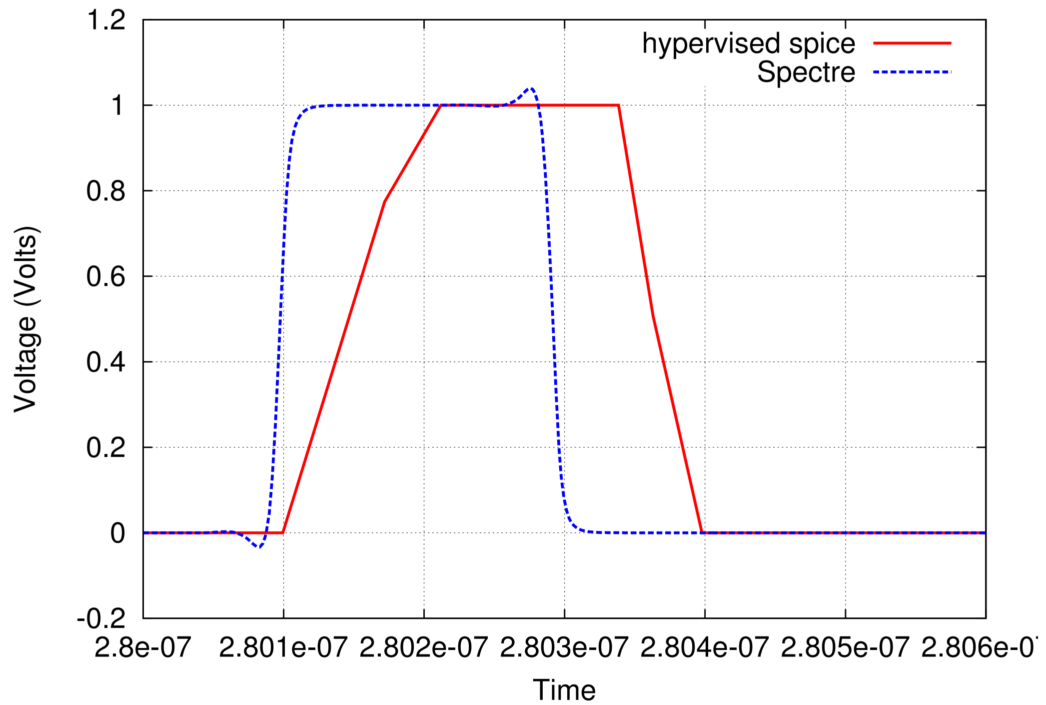


Figure 4.9: Temporal offset error

On the other hand, we have linear execution times with regard to the circuit size on all execution platforms. This is clearly show in Figure 4.10. This proves that the optimisations we performed were well fiting for this application profile.

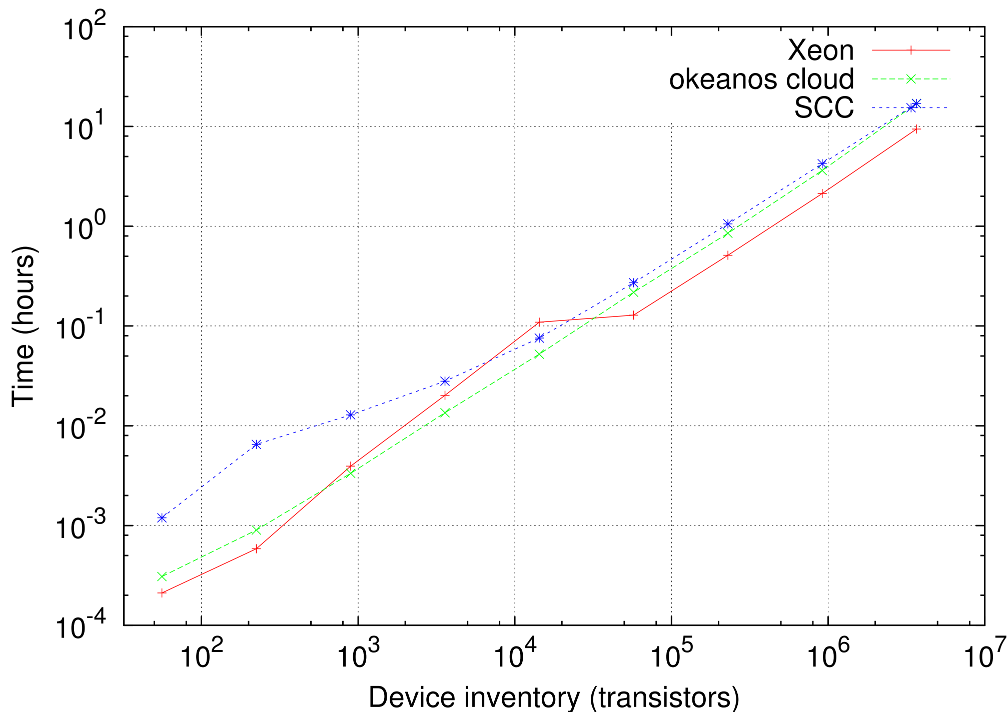


Figure 4.10: Cross platform execution times v2

When comparing the tool performance the execution times are significantly smaller when compared to the commercial **Spectre**. As shown in Figure 4.11 **hypervised spice** is an order of magnitude faster than **Spectre**. Moreover we have to note here that **Spectre**, just as **hspice** in the First Benchmarking Session, fails to produce results for data block sizes greater than a 32 by 32 bit multiplier.

In Figure 4.12 we see a comparison of execution times of all four tools. We can observe that the new implementation is faster by an order of magnitude when compared with either the *Synopsys* or *Cadence* tool. This justifies the steps we took to optimise the performance of the execution, and the bottleneck estimations were valid. However this methods imposed an error factor. This indicates that future work is required to optimise signal filtering in order to reduce these errors.

Finally in Figure 4.13 we see a comparison between the execution times among the two versions. We can observe that the improvement is vast with regard to performance.

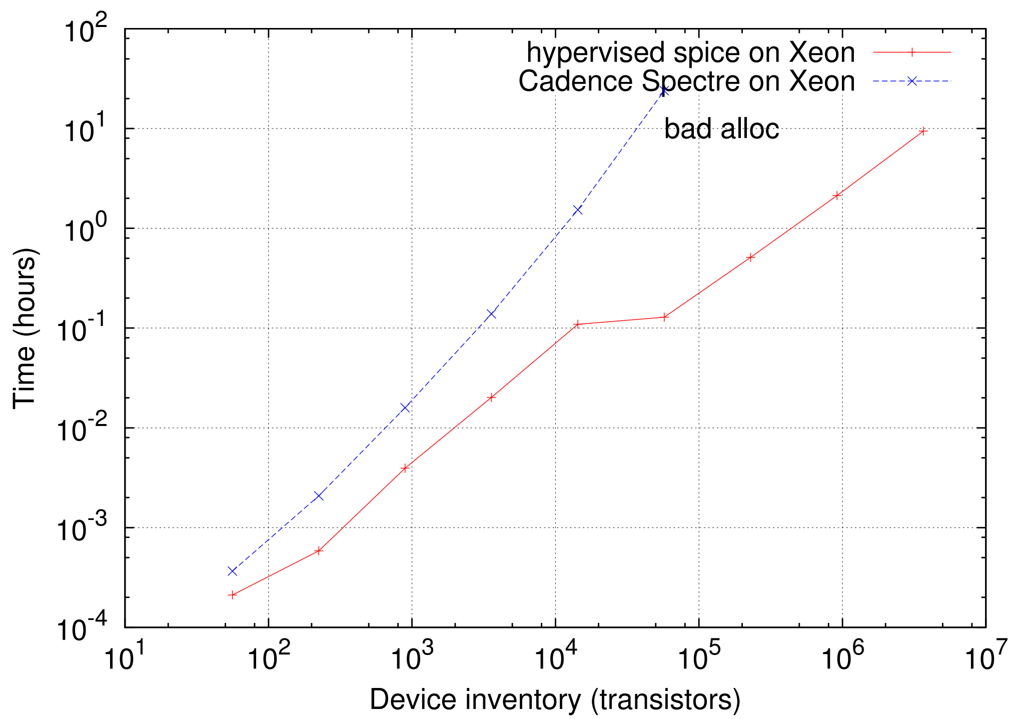


Figure 4.11: Comparison of times of v2 against Spectre

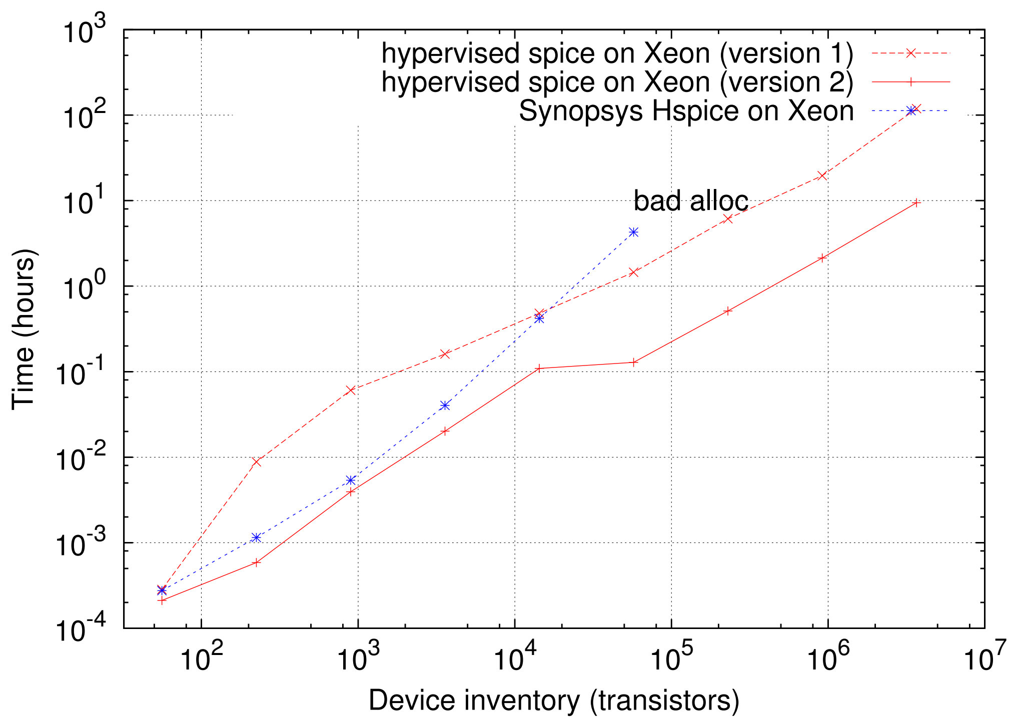


Figure 4.12: Comparison of execution times of v1, v2 against hspice and Spectre

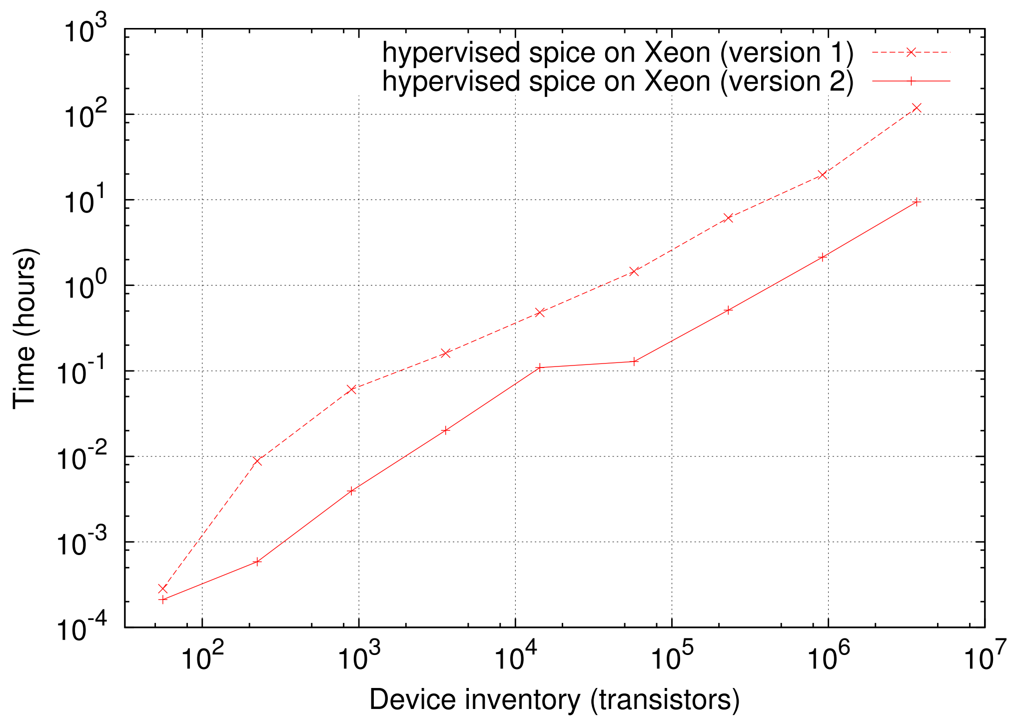


Figure 4.13: Comparison of execution times of v1, v2

Chapter 5

Conclusions & Future Work

5.1 Conclusions

This work has been a multi disciplinary project. It is a merge of ideas from the world of High Performance Computing into the world of Electronic Design Automation.

As part of the technical work, a series of tools and libraries that constitute the framework were designed and implemented.

The backbone of the framework is a configurable **hypervisor** that can handle jobs with dependencies. As a design choice this tool uses a double `fork` and one `execve` system call. This allows the return of the resource through the `waitpid` system call.

The second tool that is used in the framework is the **ngspice wrapper**. This is responsible for all handling input and output signals following a set of strict conventions regarding the naming of the files. Moreover, since it is agnostic regarding the kernel it executes, it also uses the `fork` and `execve` system calls.

The `microsig` library was developed to handle input and output of signals in PWL form and also for performing some simple operations on them. These operations include filtering, addition, interpolation and inversion. Most of the tools are built with the use of this library.

Finally partitioning tools and data mining tools were built with the use of the `microsig` library

When performing the experiments we had three runtime platforms at hand. An Intel®Xeon®CPU X3470 @ 2.93GHz powered server, the Single-Chip

Cloud Computer (SCC) Intel Labs prototype and a virtual machine on GRNET ∼okeanos infrastructure.

For the experiments we used **ngspice** as the main execution kernel, while we used **hspice** by *Synopsys* and **Spectre** by *Cadence* as reference tools for the first and the second round of experiments respectively. The circuit we used as a benchmark was a multiplier.

During the first experiment cycle we observed great accuracy on the results we obtained from our tool. When we compared the execution times with the execution times of **hspice** we found that our tool was strictly slower for small data block sizes but the performance increased and exceeded **hspice** from a certain point forward. A significant point here is that the commercial tool failed to produce results for any block size greater than a 32 by 32 bit multiplier. On the other hand our tool kept producing results for up to 256 by 256 bit multiplier and a device inventory of 3.6 million transistors.

In order to improve the performance we reduced the partitioning factor and inserted a filtering method in the process. We have to note that the step sizing algorithm we used is the one implemented in **ngspice**. Being far from optimal, it raised the need to filter the signals in order to reduce their size. If the execution kernel we used had a better step sizing algorithm, this step wouldn't be needed.

These courses of action greatly boosted the performance. Compared to the industry standard tools our tool was able to produce results faster by an order of magnitude. The commercial tools kept failing at the same point (32 by 32 bit multiplier). However our tool kept producing results up to 256 by 256 proving that we have overcome the memory barrier imposed by the runtime system. On the other hand this performance boost came with a degradation of the accuracy of the tool.

From the above we can conclude that this approach introduces a viable solution to otherwise unfeasible simulations. Furthermore we can identify a trade-off between accuracy and speed. As it can be understood if we increase the speed of the application we can produce results faster with less accuracy.

5.2 Future Work

We approached the field of SPICE simulation for large netlists and extensive workloads and provided a viable solution in order to overcome the memory

limitations of each runtime system. This work can be further extended and improved in various ways.

One step of action would be to improve the **hypervisor**. The hypervisor at the moment uses a simple first come first served scheduling algorithm. If one can prioritise the tasks they can achieve better locality which can improve the performance on certain cases. Furthermore the hypervisor uses a simple `fork` call to spawn the process control threads. An improvement would be to use the `pthread` library or an equivalent `API` which would enable a more fine grained control on the spawns and less strain on the running platform.

Another interesting approach regarding the **hypervisor** would be the concept of local kings. In case of multiple execution nodes one could dispatch, for example, one hypervisor per physical machine. This way you can achieve greater locality and scalability.

Secondly one can improve the **ngspice wrapper**. **Sapp** as it is it performs all the transactions through the file system. This approach is needed in order to overcome the memory wall. Thus one can improve the file system throughput. Solid State Drives can provide the application with a great performance boost if they are available. Moreover a compression layer on the data as they are written and read from the file system can also reduce the strain on the file system thus greatly improving the performance. A third idea would be to implement the signal forwarding through a `Message Passing Interface` if we need even greater scalability.

Due to the discrepancy we observed in the second round of experiments we can argue that the filtering method applied is rather crude for such an application. Thus a more advanced filtering method could greatly improve the accuracy of our approach. Alternatively, improvements of the adaptive step sizing algorithm can enhance the performance of intermediate simulations with minimum accuracy degradation.

Another domain where the hypervisor could alleviate significant CPU times is that of time and workload dependent variability simulations. For instance, a very interesting application would be the atomistic models for Bias Temperature Instability (BTI) and Random Telegraph Noise (RTN) [2].

Last but not least, the trade-off between accuracy and speed always remains. The more invasive the filtering method applied the less points the simulations will parse. This way, depending on the application at hand one can fine tune the performance to be suitable to their needs and deadlines.

Bibliography

[1] Nascentric announces omegasim(tm) gx - the world's first hardware-accelerated spice simulator. http://www10.edacafe.com/nbc/articles/view_article.php?articleid=516299&interstitial_displayed=Yes.

[2] Time and workload dependent circuit simulation. Technical Report EP 2 509 011 A1, 2012.

[3] M. Duranton, S. Yehia, B. de Sutter, K. de Bosschere, A. Cohen, B. Falsafi, G.N. Gaydadjiev, M.G.H. Katevenis, A. Ramirez, O. Temam, and M. Valero. *The HiPEAC Vision, High Performance and Embedded Architecture and Compilation*. HiPEAC Project, January 2010.

[4] S.H. Fuller and L.I. Millett. Computing performance: Game over or next level? *Computer*, 44(1):31 --38, jan. 2011.

[5] Kanupriya Gulati, John F. Croix, Sunil P. Khatr, and Rahm Shastry. Fast circuit simulation on graphics processing units. In *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, ASP-DAC '09, pages 403--408, Piscataway, NJ, USA, 2009. IEEE Press.

[6] Nachiket Kapre. *SPICE2 -- A Spatial Parallel Architecture for Accelerating the SPICE Circuit Simulator*. PhD thesis, California Institute of Technology -- Pasadena, California, 2010.

[7] Vangelis Koukis. ∼okeanos: Delivering iaas to the greek academic and research community, August 2012. Invited talk at VHPC 2012, Rhodes Island, Greece.

[8] Grigorios Lyras, Dimitrios Rodopoulos, Antonis Papanikolaou, and Dimitrios Soudris. Hypervised transient spice simulations of large netlists & workloads on multi-processor systems. In *Design Automa-*

tion and Testing in Europe (DATE), 2013 International, march 2013.

[9] A. Maxim and M. Gheorghe. A novel physical based model of deep-submicron cmos transistors mismatch for monte carlo spice simulation. In *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on*, volume 5, pages 511 --514 vol. 5, 2001.

[10] T. McConaghy, K. Breen, and J. Dyck. *Variation-Aware Design of Custom Integrated Circuits: A Hands-On Field Guide*. Springer, 2013.

[11] Laurence W. Nagel. *SPICE2: A Computer Program to Simulate Semiconductor Circuits*. PhD thesis, EECS Department, University of California, Berkeley, 1975.

[12] Laurence W. Nagel and D.O. Pederson. Spice (simulation program with integrated circuit emphasis). Technical Report UCB/ERL M382, EECS Department, University of California, Berkeley, Apr 1973.

[13] Daniel Payne. New parallel spice start-up company eda thoughts: From an eda marketing insider. `http://www.chipdesignmag.com/payne/2009/12/24/new-parallel-spice-start-up-company/`.

[14] D. Rodopoulos, S.B. Mahato, V.V. de Almeida Camargo, B. Kaczer, F. Catthoor, S. Cosemans, G. Groeseneken, A. Papanikolaou, and D. Soudris. Time and workload dependent device variability in circuit simulations. In *IC Design Technology (ICICDT), 2011 IEEE International Conference on*, pages 1 --4, may 2011.

[15] M. Toledano-Luque, B. Kaczer, P.J. Roussel, T. Grasser, G.I. Wirth, J. Franco, C. Vrancken, N. Horiguchi, and G. Groeseneken. Response of a single trap to ac negative bias temperature stress. In *Reliability Physics Symposium (IRPS), 2011 IEEE International*, pages 4A.2.1 --4A.2.8, april 2011.

[16] Mateo Valero and Nacho Navarro. Multicore: The view from europe. *IEEE Micro*, 30(5):2--4, 2010.

[17] N.H.E. Weste and D.M. Harris. *CMOS VLSI Design: A Circuits and Systems Perspective [With Access Code]*. ADDISON WESLEY Publishing Company Incorporated, 2011.

Chapter 6

Appendix

A: DATE 2013 Preprint

Hypervised Transient SPICE Simulations of Large Netlists & Workloads on Multi-Processor Systems

Grigorios Lyras, Dimitrios Rodopoulos, Antonis Papanikolaou and Dimitrios Soudris
National Technical University of Athens – School of ECE
MICROprocessors and Digital Systems LABoratory
9 Heroon Polytechneiou, Zographou Campus, 157 80 Athens, Greece
Conctact Email: drodo@microlab.ntua.gr

Abstract—**The need for detailed simulation of digital circuits has received the attention of both academia and industry since the early stages of design automation. As the number of integrated devices per silicon die increases, the need for faster, device level, circuit simulations becomes more apparent. These simulations are heavy on the system memory, hence limiting the size of the circuit that can be handled by multi-core systems with a unified memory hierarchy. In this work, we present a parallel implementation of a traditional circuit simulation program, based on structural partitioning of the netlist and temporal partitioning of the input signals. This enables scalability of the overall simulation across units of execution. It allows simulation of very large circuits, which cannot be handled even by commercial tools. The proposed simulation framework is validated through simulations of a benchmark circuit with more than a million MOSFET devices and workloads of extended duration. We observe minimal error in comparison to commercial tools and even a $\times 2.35$ speedup for moderate netlist sizes. Different execution platforms are inspected, thus substantiating the versatility of our implementation.**

I. INTRODUCTION

Computer-aided simulation of circuit activity is a key issue in the manufacturing of digital systems, since it allows the verification of electric and electronic circuits. As a result, it covers a very wide range of applications [1]. Especially in the case of integrated circuits, a definitive piece of work is the "Simulation Program with Integrated Circuit Emphasis" (SPICE) [2]. This software has long been considered as the standard for detailed simulation of integrated circuits. Apart from being a valuable industry and academic tool, SPICE has also triggered a wide variety of research. A significant portion of that research aims at the acceleration of SPICE simulations. One of the major reasons for this research trend is *aggressive integration*. As the device inventory of modern electronics increases, the simulated netlists contain a larger number of devices and require more memory during simulation.

Another valid factor that creates demand for computationally viable SPICE simulations of large netlists and workloads is *device variability*. In sub-micron technologies, the devices feature increased variability at time zero [3]. Also, as the lifetime of the digital system progresses, devices are reported to behave in a variable manner, which includes both a stochastic and a workload-dependent component [4]. Monte Carlo simulations are usually employed to cover time zero device variability, as in the case of [5]. Furthermore, transient simulations of representatively long workloads are required to account for workload-dependent circuit variability, as in [6]. Hence, we can safely claim that *the memory requirements of transient SPICE simulations are increasing both due to larger device inventories and extended netlist workloads*.

By inspecting state-of-the-art approaches on SPICE optimizations, we observe that the majority of the techniques aim to parallelize a SPICE simulation among a set of execution nodes. This is not surprising, if we consider the increasing number of cores which are available in modern large-scale processing systems [7]. The concepts of *node and branch tearing* are very appealing when parallelizing SPICE. These concepts involve the physical partitioning of the initial netlist to smaller subcircuits. Each subcircuit is submitted to a SPICE instance, thus many instances run in parallel to create the solution of the initial netlist. Many papers look into the optimization of this partitioning or parallelize intensive execution stages. Others, deal with the parallel mapping of SPICE on specific hardware. In all the above cases, the demand for main memory is made concurrently to the executing platform, which may prove incapable of fulfilling this requirement.

In the current paper, we differentiate from the state of the art by enabling massive SPICE simulations and overcoming memory capacity issues through the distribution of threads that make minimal memory demands to the available units of execution. We propose the temporal partitioning of the netlist's primary input signals, in what will be referred as *workload tearing*. When combined with node tearing, *workload tearing* enables the execution of small and independent SPICE instances. These instances are sized according to the available computational and memory resources so as to take maximal advantage of the infrastructure capabilities with minimal communication overhead. A hypervisor has been designed to dispatch these SPICE instances across the execution nodes. The proposed framework imposes minimum hardware constraints on the executing platform and is highly reusable. It has been tested on an experimental cloud computer chip, a set of virtual machines and a regular multicore server to illustrate its reusability. It is also compared to the commercial tool HSPICE, which supports multi-threaded execution.

In the next Section, we summarize the state-of the-art dealing with SPICE parallelization. In Section III we elaborate on the proposed structural and temporal partitioning, which creates a set of small SPICE instances. We also present

the hypervisor, which dispatches these instances in order to complete the otherwise infeasible transient simulation. In Section IV, we present the inspected multi-core platforms, the netlist benchmark that was used and simulation results. Finally, conclusions are summarized in Section VI.

II. Related Work

Optimizations of SPICE performance can be split into two major categories, those that target a *single thread* of SPICE execution and solutions that deal with the distribution of a simulation across *different threads* of execution (i.e. parallelization). Naturally, the former category attracted significant amount of research, even from the initial development stages of the SPICE software [8]. As the multi-core trend materialized, research focused on distributed solutions for SPICE acceleration. The Electronic Design Automation (EDA) industry is following the trend of mulit-core SPICE simulations providing a range of related implementations [9].

Definitive steps towards the parallelization of SPICE simulations are the works on node tearing [10] and branch tearing [11]. According to these concepts, the target netlist is partitioned from its nodes or branches and independent voltage or current sources are put in their place. The authors of [12] argue that the addition of extra energy through these independent sources is connected to non-convergence issues. An alternative to node tearing is also proposed in [13]. In this work, a partitioning methodology is proposed aiming at reduced number of connections between partitions, which should also be roughly of the same size. Finally, [14] presents the concepts of direct current connected blocks and strong connected components as two clustering criteria. However, verification of this partitioning scheme with netlists containing more than 10^3 devices is left as future work.

Many SPICE acceleration attempts are using specific hardware to exploit parallel patterns that are observed in the simulation execution. Field Programmable Gate Arrays [15] (FPGAs) can be used to parallelize the tasks performed by the SPICE simulator. In the case of [16], we read about the utilization of Graphics Processing Units (GPUs) for the acceleration of transistor model evaluation.

In view of the related work on SPICE acceleration, it is evident that netlist partitioning has received significant attention. Even though optimized, the partitioning of the circuit is not enough to avoid the violation of the memory constraints imposed by the executing hardware [13]. Other approaches that propose the use of customized hardware to perform the simulations (e.g. GPUs or FPGAs) reduce the versatility of the parallel SPICE simulations. Our parallel framework reduces the memory footprint of the simulation on each execution node, while remaining highly reusable across processing systems usually found in academic or industry environments.

III. Hypervised SPICE Simulations

A. Simulation Framework Concept

Assume a target netlist with a large number of devices and a vector of primitive input signals $\mathbf{V_{in}}(t)$. The purpose of our analysis is to calculate the primitive output signals, namely $\mathbf{V_{out}}(t)$. Application of *node tearing* on the target netlist can create N subcircuits, each one with a number of devices equal to d_i, where $i = 1, 2, ..., N$. *Workload tearing* is the temporal partitioning of the primitive input vector based on Equation 1, where $h(t)$ is the Heaviside pulse and T is a time step.

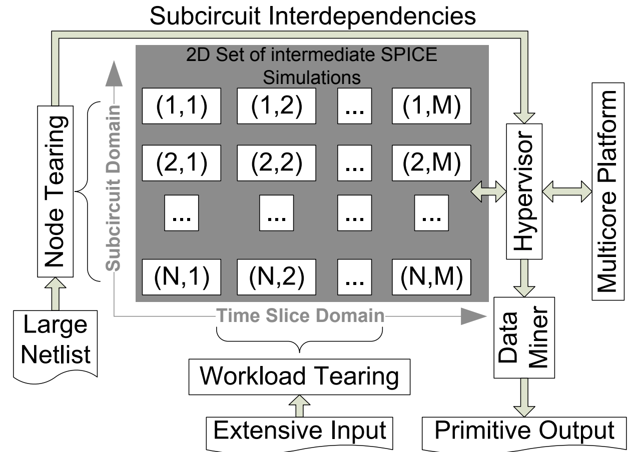


Fig. 1: The designed hypervisor draws from a set of intermediate SPICE simulations to synthesize the primitive outputs of a large netlist, which is simulated over extended workloads.

$$\mathbf{V_{in}}(t) = \sum_{j=1}^{M} \mathbf{V_{in}}(t)\ h(t - jT) \quad (1)$$

This temporal partitioning can be easily implemented at pre-processing, assuming that the primitive input signals are available in a piece-wise linear (PWL) form. Given the time step T, we scan the respective PWL files and create time slices (addends of Equation 1) of the large netlist's primitive input signals. Node and workload tearing create a two dimensional set of simulations (see Figure 1). In the general case, each of these simulations requires a combination of primitive signal slices and intermediate simulation results to produce its output. A simple graph of dependencies between subcircuits indicates the flow of intermediate results across subcircuits of the large netlist. As a result, in order to calculate $\mathbf{V_{out}}(t)$, we require $N \times M$ intermediate SPICE simulations. Each one is uniquely identified by the pair (i, j) as the simulation of subcircuit i, for the time slice j.

A hypervisor dispatches intermediate SPICE simulations from the respective set. Each member is assigned to an execution node of the available multicore platform. Based on the framework of Figure 1, a number of SPICE instances will be running in parallel on the mutlicore platform. As long as all available units of execution are occupied by SPICE instances, the hypervisor remains dormant; it becomes activated only when a SPICE simulation is completed and another has to initiate. Finally, a data miner is synthesizing $\mathbf{V_{out}}(t)$ from the outputs delivered by the intermediate simulations. These simulations demand small amounts of memory due to re-

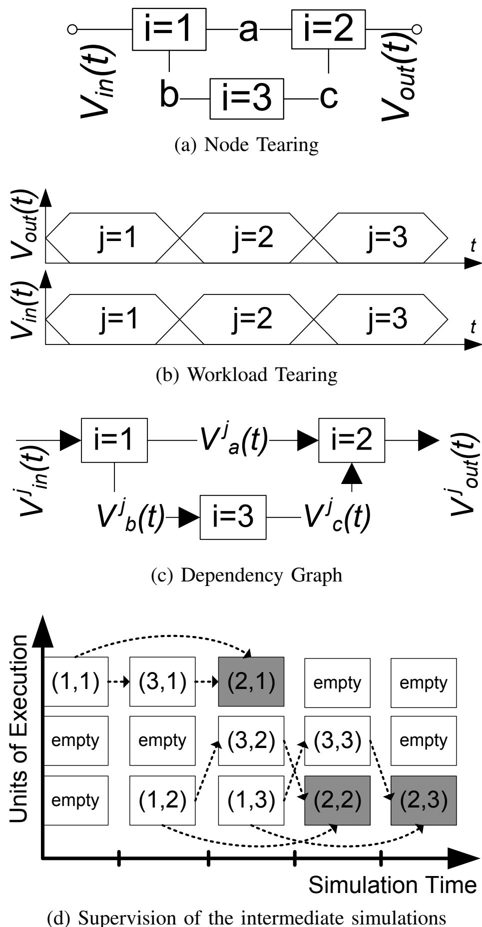


Fig. 2: A qualitative example illustrating our proposed concept.

duced device inventory and input duration. The small memory portions are returned to the system after the intermediate simulation is finished. *This is a major differentiator of our approach from the state-of-the-art and even commercial tools, which require large amounts of memory from the beginning of a transient simulation.*

B. Illustrative Example

In this Subsection, we illustrate the proposed simulation framework with a qualitative example. Assume a netlist, which can be partitioned into three subcircuits, using node tearing. Each subcircuit is uniquely identified by its i value (Figure 2a). The netlist needs to be simulated over a workload of extended duration ($V_{in}(t)$). With workload tearing, we partition the input vector into three time slices (see Figure 2b). In this simulation, we seek to calculate the primitive output $V_{out}(t)$, which will be assembled per time as well. For each time slice, we can notate $V^j_{in}(t)$ and $V^j_{out}(t)$, where $j = 1, 2, 3$.

It is important that, when performing node tearing, we choose nodes, the voltage of which is imposed by the same connected subcircuit during the entire simulated lifetime. In our example, we assume that the nodes a, b and c of Figure 2a fulfill this requirement. The voltage signals of each node will also be created per time slice, hence we can use the notations $V^j_a(t)$, $V^j_b(t)$ and $V^j_c(t)$, where $j = 1, 2, 3$. We can now draw the directed dependency graph between the three subcircuits, displayed in Figure 2c.

In the set of independent simulations that will be used as input to the proposed hypervisor, each member set is uniquely defined by its (i, j) pair. The hypervisor will complete the entire set of simulations, by handling simulation instances with resolved dependencies on a first-come-first serve basis. An example of the hypervisor's mapping on a platform with three execution nodes can be seen in (see Figure 2d). At each simulation time instance, we can see the occupation of execution nodes by members of the intermediate simulation set. Dashed arrows indicate the forwarding of intermediate simulation results between SPICE instances. Shaded boxes represent the intermediate simulations that produce the signals $V^j_{out}(t)$. These outputs will be combined into the primitive output $V_{out}(t)$ by the data miner in an overlap-save fashion.

Platform	UE* Information	#UEs	Hypervisor Residence**	Invocation††
SCC	P54C @533MHz	48	MCPC†	`ssh`
∼okeanos	QEMU Virt. CPU Version 1.0 @2.1GHz	4	1 UE	`fork`
Xeon	X3470@2.93GHz	4	1 UE	`fork`

* UE: Unit of execution, ** The processing system that runs the hypervisor,
† MCPC: Management Console Personal Computer of the SCC,
†† The command required to dispatch an intermediate simulation on a UE.

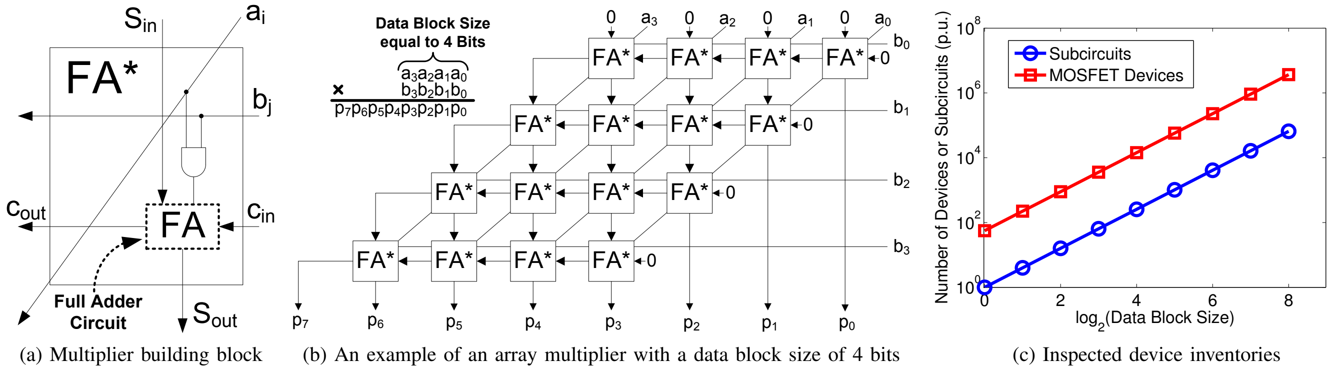
TABLE I: Details of the inspected multicore platforms

IV. Experimental Verification

A. Tested Platforms, Benchmark Netlist & Workloads

To confirm the versatility and correctness of our approach, we have tested the proposed framework on three platforms that are representative of the computation resources usually found among industrial or academic infrastructure. The first platform is the Single-Chip Cloud Computer (SCC) experimental processor, which is a 48-core "concept vehicle" created by Intel Labs as a platform for many-core software research [17]. We have also tested our framework on virtual machines (VMs) of the cloud service ∼okeanos [18]. Finally, an Intel®Xeon® server has been used to complete the set of representative computing resources. All experiments have been performed using an open source SPICE version called `ngspice` [19]. Technical details for each platform are summarized in Table I. As *reference for comparison*, we use the commercial tool HSPICE of Synopsis. We choose an execution with four threads running on a similar 2.33GHz Intel®Xeon® machine (flag `-mt 4`).

The chosen benchmark netlist is an array multiplier of scalable size. An overview of the circuit can be seen in Figures 3a and 3b. We define as *data block size* the length of the multiplier's operands in bits. We increase the device inventory of the netlist by increasing the data block size of the multiplier array, reaching beyond 10^6 devices (see Figure 3c). Arbitrary workloads in the form of a PWL voltage sources



(a) Multiplier building block (b) An example of an array multiplier with a data block size of 4 bits (c) Inspected device inventories

Fig. 3: Circuit diagrams and device inventory of the selected benchmark circuit

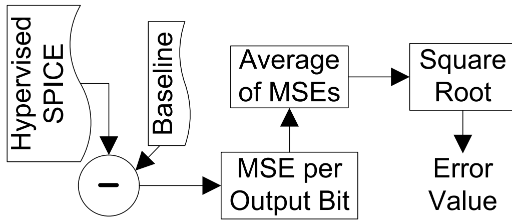


Fig. 4: Error estimation of the hypervised SPICE simulations

have been created for each multiplier size, the duration of which is 800ns and the time step T used for workload tearing is 20ns (see Equation 1). The selected benchmark allows easy node tearing into identical subcircuits of the FA* module of Figure 3a. As a result, the device inventory is the same for all subcircuits. In general, our framework supports non-identical subcircuits, provided that the respective `.subckt` lines exist in the initial large netlist.

B. Simulation Results

We start with the estimation of our framework's performance. For various sizes of the multiplier array, we apply the workloads specified in Subsection IV-A both to our framework and the baseline implementation. First,we measure the processing time that is exclusively dedicated to SPICE simulation (see Figure 5a). For small device sizes, the commercial baseline outperforms our framework. However, beyond the array data block size equal to 32, HSPICE is unable to deliver due to insufficient system memory. At the same time, our proposed framework keeps producing results, even for a netlist with more than 10^6 MOSFET devices. In the execution of the proposed framework, the hypervisor is activated to dispatch a new intermediate simulation, which creates the illustrated temporal overhead (Figure 5b).

We also assess the accuracy of our approach, by calculating the root mean squared error (RMSE) of the simulation outputs with the steps presented in Figure 4. The mean squared error (MSE) is calculated for each product bit of the multiplier. Then, an average of all the MSEs (for all product bits) is calculated. In Figure 5c we can see the root of this average MSE value for all inspected platforms and device inventories. Since the primitive outputs are voltage signals, the respective RMSE values are also measured in Volts. Obviously, we can produce error results only as long as the respective baseline execution is completed successfully (namely up to a data block size equal to 32 bits). In Figure 6 we give transient output excerpts for the output bits with the maximum MSE for different multiplier array sizes. In these graphs we can see that the output signals are reproduced very accurately by our simulation framework. Signal transitions and intermediate jitter are the main causes of output error. In Section V, we propose simple solutions to alleviate these problems.

V. Discussion

The technique proposed in this paper goes further than existing SPICE parallelization approaches by adding a partitioning of the workload on top of existing netlist partitioning. Workload tearing can be adjusted to improve the accuracy of the results presented in Figure 6, by applying workload tearing so that the transitions of the inputs are kept within each inspected time slice. The output signals of Figure 6 show a poor reproduction of the signal overshoot. We believe that the addition of correct gate capacitances at the subcircuit outputs can solve this problem and correctly reproduce the waveforms.

A major differentiator of our proposed concept is also the flexible memory allocation to circuit simulations. Most state-of-the-art tools (even the commercial ones) allocate memory for the entire simulation, thus it is often that the simulation is halted due to insufficient memory. With our framework, we partition the simulation to small instances, thus making sure that the transient analysis is completed without making massive memory requests to the system. We also need to note that the communication between SPICE instances that we propose is performed at the level of system storage. As a result, use of modern storage devices (e.g. solid state disks) or of a more customized file system would only enhance the performance of our proposed framework.

In the results presented above, the resolution of the signals that are propagated through the netlist and across the netlist components was not altered (as delivered from `ngspice`).

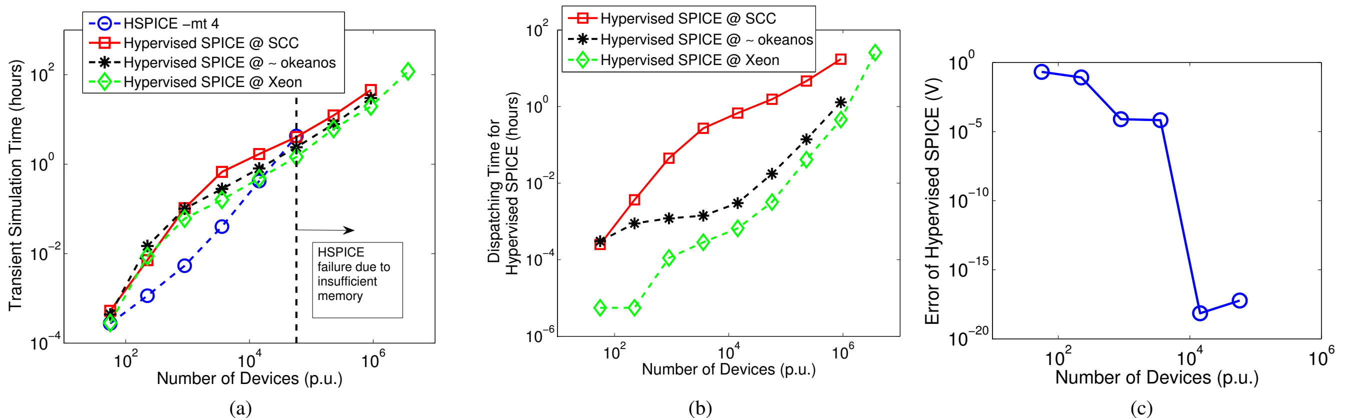
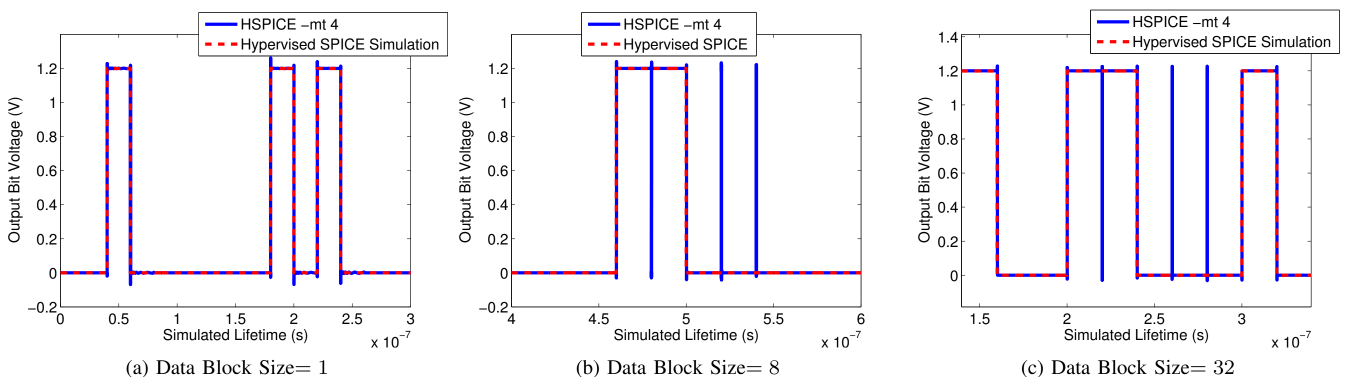


Fig. 5: Performance and accuracy evaluation of hypervised SPICE simulations against the baseline execution



(a) Data Block Size= 1 (b) Data Block Size= 8 (c) Data Block Size= 32

Fig. 6: Transient view of the primary output with maximum MSE, for different sizes of the multiplier array

By simplifying these signals into simple ramps, the simulation time will be considerably reduced and the accuracy of timing and power metrics will be degraded. Here lies a fundamental trade-off between simulation time and result quality. This can prove very handy in the context of simulating digital systems in presence of variability and/or reliability degradation. Analog artifacts, such as threshold voltage shifts, in MOSFETs of a huge design can be simulated and the impact of such artifacts on the global design can be evaluated. Up to now this capability has only been available at the IP component level or through library characterization.

Having presented the experimental results, we can revisit the comparison to the state-of-the-art with the three key metrics of Table II. The maximum reported simulation speedup and inspected device inventories are presented for the related work cited in the Section II. Also, the target platform is given for each case, as an indication of the implementation's versatility. We can safely claim that the device inventories that we address are very much increased in comparison to the state-of-the-art, whereas the proposed framework is not limited to specific hardware. Finally, we can identify a minor speedup achieved by our parallelization, based on Figure 5a and specifically for the a data block size of 32 bits. This reduced acceleration is to be expected, since the primary goal of this work is the viability of massive, transient SPICE simulations from a memory footprint point of view.

Cited Work	Max. Speedup	Max. #MOSFETs	Executing Platform
[12]*	N/A	6	2-Processor System
[13]	$^{1}\times 5.96$	19,995	SGI Challenge Server (12 CPUs)
[14]	$^{2}\times 2.09$	96	Intel 2.66GHz PC
[15]	$^{3}\times 11$	27,995**	Xilinx Virtex-6 LX760 FPGA
[16]	$^{4}\times 3.07$	7,682	NVIDIA GeForce GPU
Proposed Hypervised SPICE	$^{5}\times 2.35$	3,670,016	SCC, VMs of a Cloud, Intel Xeon Server

* Few experimental details are provided, apart from a quantitative example with three inverters, ** Maximum size of netlist matrix.
[1] vs. single core execution, [2] vs. Fiduccia-Mattheyses partitioning, [3] vs. SPICE on Intel i7-965, [4] vs. Intel Core 2 Quad Core, [5] vs. HSPICE -mt 4.

TABLE II: Comparison of the proposed transient SPICE simulation framework with works from the state-of-the-art

VI. Conclusions

In this paper, we have presented a simulation framework that allows transient SPICE simulations of large netlists for extended workloads. We employ the concept of node tearing in order to partition the target netlist into subcircuits, which are simulated independently. The primitive input signals of the netlist are also temporally partitioned. Workload and node partitioning create a set of small SPICE instances, that have reduced memory requirements. Being independent, these SPICE instances can be mapped to the execution units of a multi-core platform, thus achieving a data partitioning of the transient simulation problem. With our simulation framework, the demand for main memory is not performed massively and concurrently to the executing platform, as observed in the state-of-the-art and even commercial tools. Instead, by scaling the initially large SPICE simulation, we avoid hitting the memory constraints of all inspected platforms, while simulating netlists with over 10^6 devices. Furthermore, our proposed parallel implementation is flexible and has minimum hardware requirements. We have verified the performance and accuracy of our framework on three platforms, covering the range of cloud service infrastructure, regular multi-core systems and advanced chips with an increased number of integrated processors. Thus, we substantiate the reusability of our approach across computing systems, which are usually found in academia or industry.

Acknowledgments

The SCC and its MCPC were provided by Intel Labs Braunschweig, Germany in the context of the FP7-INFSO-IST-248789 TRAMS project of the European Commission.

References

[1] F. N. Najm, *Circuit Simulation*, 1st ed. Hoboken, New Jersey, US: Wiley-IEEE Press, 2010.

[2] L. W. Nagel and D. Pederson, "Spice (simulation program with integrated circuit emphasis)," EECS Department, University of California, Berkeley, Tech. Rep. UCB/ERL M382, Apr 1973.

[3] Y. Ye, F. Liu, M. Chen, S. Nassif, and Y. Cao, "Statistical modeling and simulation of threshold variation under random dopant fluctuations and line-edge roughness," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 19, no. 6, pp. 987 –996, june 2011.

[4] M. Toledano-Luque, B. Kaczer, P. Roussel, T. Grasser, G. Wirth, J. Franco, C. Vrancken, N. Horiguchi, and G. Groeseneken, "Response of a single trap to ac negative bias temperature stress," in *Reliability Physics Symposium (IRPS), 2011 IEEE International*, april 2011, pp. 4A.2.1 –4A.2.8.

[5] A. Maxim and M. Gheorghe, "A novel physical based model of deep-submicron cmos transistors mismatch for monte carlo spice simulation," in *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on*, vol. 5, 2001, pp. 511 –514 vol. 5.

[6] D. Rodopoulos, S. Mahato, V. de Almeida Camargo, B. Kaczer, F. Catthoor, S. Cosemans, G. Groeseneken, A. Papanikolaou, and D. Soudris, "Time and workload dependent device variability in circuit simulations," in *IC Design Technology (ICICDT), 2011 IEEE International Conference on*, may 2011, pp. 1 –4.

[7] S. Fuller and L. Millett, "Computing performance: Game over or next level?" *Computer*, vol. 44, no. 1, pp. 31 –38, jan. 2011.

[8] L. W. Nagel, "Spice2: A computer program to simulate semiconductor circuits," Ph.D. dissertation, EECS Department, University of California, Berkeley, 1975.

[9] M. Rewieński, "A perspective on fast-spice simulation technology," in *Simulation and Verification of Electronic and Biological Systems*, P. Li, L. M. Silveira, and P. Feldmann, Eds. Springer, 2011, pp. 23–42.

[10] A. Sangiovanni-Vincentelli, et al., "An efficient heuristic cluster algorithm for tearing large-scale networks," *Circuits and Systems, IEEE Trans. on*, vol. 24, no. 12, pp. 709 – 717, dec 1977.

[11] F. Wu, "Solution of large-scale networks by tearing," *Circuits and Systems, IEEE Trans. on*, vol. 23, no. 12, pp. 706 – 713, dec 1976.

[12] F. Wei and H. Yang, "Transmission line inspires a new distributed algorithm to solve the nonlinear dynamical system of physical circuit," in *Computer Sciences and Convergence Information Technology (ICCIT), 2010 5th Int. Conf. on*, 30 2010-dec. 2 2010, pp. 816 –821.

[13] N. Frohlich, V. Glockel, and J. Fleischmann, "A new partitioning method for parallel simulation of vlsi circuits on transistor level," in *Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings*, 2000, pp. 679 –684.

[14] X. Zhou, Y. Wang, and H. Yang, "Dccb and scc based fast circuit partition algorithm for parallel spice simulation," in *ASIC, 2009. ASICON '09. IEEE 8th International Conference on*, oct. 2009, pp. 1247 –1250.

[15] N. Kapre, "Spice2 – a spatial parallel architecture for accelerating the spice circuit simulator," Ph.D. dissertation, California Institute of Technology – Pasadena, California, 2010.

[16] K. Gulati, J. F. Croix, S. P. Khatr, and R. Shastry, "Fast circuit simulation on graphics processing units," in *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 403–408. [Online]. Available: http://dl.acm.org/citation.cfm?id=1509633.1509733

[17] J. Howard, S. Dighe, S. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, and R. Van Der Wijngaart, "A 48-core ia-32 processor in 45 nm cmos using on-die message-passing and dvfs for performance and power scaling," *Solid-State Circuits, IEEE Journal of*, vol. 46, no. 1, pp. 173 –183, jan. 2011.

[18] V. Koukis, "∼okeanos: Delivering iaas to the greek academic and research community," August 2012, invited talk at VHPC 2012, Rhodes Island, Greece.

[19] P. Nenzi and H. Vogt, "Spice (simulation program with integrated circuit emphasis)," Tech. Rep. UCB/ERL M382, July 2012.

B: Source Code

Copyright (C) 2013 Gregory Lyras

The following source code is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

The following source code is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with the following source code. If not, see <http://www.gnu.org/licenses/>.

Hypervisor

```
using namespace std;

void child(int core, const char *cmd)
{
#if RUN_ON_MITSOS
    run(cmd);
#elif RUN_ON_SCC
    run_on_SCC(core, cmd);
#endif /* where do you wanna run */
    exit(core);
}


int main(int argc, char **argv)
{
    pid_t p;
    //children queue (pids)
    /*
     * there are two queues that hold the available resources
     * cores and readyToSpawn, when either one is empty, the hypervisor waits
     * there could be a better policy there but we leave that to future work
     */
    deque<int> available_cores;
    deque<int> readyToSpawn;
    /*
     * allNodes is the main structure of the hypervisor
     * it holds all the data needed by the hypervisor
     * regarding the jobs it has to dispatch
     * it also holds dependencies
     * and children for each
     */
    vector<execNode *> allNodes;
    int status;
    int cores;
    int jobs;
```

```
36        int jobs_remaining;
37        /*
38         * procIDbuf is used to hold each process ID
39         * this is unique and different from the OS pid
40         * it is defined during the parsing of the executionGraph
41         */
42        int procIDbuf;
43        int coreIDbuf;
44        int coreIDstatus;
45
46        /*
47         * it is the number of currently running processes
48         * not needed but usefull for debugging
49         * might be marked as deprecated
50         * XXX: removed as it wasn't used
51         */
52        /* int processesCurrentlyRunning; */
53
54        /*
55         * iterator used for children waking for each node
56         */
57        vector<int>::iterator it;
58        /*
59         * buffer which holds the children of each node every time
60         */
61        vector<int> children;
62        /*
63         * procsOnCores holds a mapping
64         * it knows which currentlyRunning process
65         * is on which core
66         */
67        vector<int> procsOnCores;
68
69        if( argc != 3)
70        {
71           fprintf( stderr, "no arguements given\n" );
72           fprintf( stderr, "usage: %s cores genericGraph\n", argv[0] );
73           exit( EXIT_FAILURE );
74        }
75
76        sscanf( argv[1], "%d", &cores );
77        procsOnCores.resize(cores);
78        fill(procsOnCores.begin(),procsOnCores.end(),0);
79
80
81        /*
82         * and now the fun part
83         */
84
85
86        /*
87         * hocus pocus
88         * read the configuration
89         * and fill allNodes with the data i need
90         */
91        parse(allNodes,argv[2]);
92
93
94        /*
95         * add all jobs that are ready to run
96         * in the readyToSpawn queue
97         */
```

```
98      jobs = allNodes.size();
99      for ( int i = 0 ; i < jobs ; ++i )
100     {
101         if ( allNodes[i]->ready() )
102             readyToSpawn.push_back(i);
103     }
104         jobs_remaining = jobs;
105
106     /*
107      * this builds the corelist of the available resources
108      */
109     for( int i = 0 ; i < cores ; ++i )
110     {
111         available_cores.push_back(i);
112     }
113     int processesCurrentlyRunning = 0;
114
115     for ( int i = 0; i < jobs ; )
116     {
117         //if there is an available core, go for it
118         //else wait for a child to end
119         if( available_cores.empty() || readyToSpawn.empty() )
120         {
121             if (waitpid(-1, &status, 0) < 0) {
122                 perror("unreachable state\n");
123                 exit(EXIT_FAILURE);
124             }
125             processesCurrentlyRunning--;
126             coreIDstatus = WEXITSTATUS(status);
127             /* the exit status is the coreID */
128             /* this way when a child exits we know where it was running
129              * and with the array procsOnCores we know which process was
130              * running on that core
131              */
132             coreIDbuf = coreIDstatus;
133             available_cores.push_back(coreIDstatus);
134             procIDbuf = procsOnCores[coreIDbuf];
135
136             if ( allNodes[procIDbuf]->childrenN() > 0 )
137             {
138                 children = allNodes[procIDbuf]->getChildren();
139
140                 for ( it = children.begin() ; it != children.end() ; it++)
141                 {
142                     if( allNodes[*it] && allNodes[*it]->decDeps() )
143                     {
144                         readyToSpawn.push_back(*it);
145                     }
146                 }
147             }
148         }
149         else
150         {
151             procIDbuf = readyToSpawn.front();
152             coreIDbuf = available_cores.front();
153             procsOnCores[coreIDbuf]=procIDbuf;
154             p = fork();
155             processesCurrentlyRunning++;
156             if ( p == 0 )
157             {
158                 //child wfd,core,pid,cmd
159                 child(coreIDbuf, allNodes[procIDbuf]->getCmd());
```

```
160                }
161                else
162                {
163                    printf(">>>> spawning: %16d on: %3d | %16d to go\n", procIDbuf+1, coreIDbuf, --jobs_remaining);
164                    readyToSpawn.pop_front();
165                    available_cores.pop_front();
166                }
167                ++i;
168            }
169        }
170        printf("\n");
171        for(; processesCurrentlyRunning > 0; processesCurrentlyRunning--)
172        {
173            waitpid(-1, &status, 0);
174        }
175
176        return 0;
177    }
```

```
1    int run_on_SCC(int core,const char *command)
2    {
3        char cmd[COMMAND_SIZE];
4        char dir[COMMAND_SIZE];
5        getcwd(dir,COMMAND_SIZE);
6        snprintf(cmd,COMMAND_SIZE,"ssh root@rck%02d 'cd %s && %s' ",core , dir, command);
7        return system(cmd);
8    }
9
10   int run(const char *command)
11   {
12       return system(command);
13   }
```

```
1    #ifndef EXEC_NODE_H
2    #define EXEC_NODE_H
3
4    #include <vector>
5    #include <string>
6    #include <cstring>
7    #include <iostream>
8    #include <iterator>
9
10   class execNode {
11           std::string command;
12           int dependencies;
13           std::vector<int> children;
14       public:
15           execNode(const execNode &N);
16           execNode(int deps, std::string cmd);
17           ~execNode();
18           bool decDeps();
19           bool ready();
20           void addChild(int i);
21           const char *getCmd();
22           int childrenN();
23           std::vector<int> getChildren();
24           void print();
25
26   };
27   #endif
```

Sapp

```
#include <stdio.h>
#include <stdlib.h>
#include "sapp_common.h"
#include "replace.h"
#include "signal.h"
#include <sys/times.h>
#include <sys/timeb.h>
#include <time.h>



int main( int argc, char** argv )
{
    int circID;
    int timeslice;
    int timesliceDuration;
    int Ninputs;
    int Noutputs;
    int spice_ret;
        double dtime;
        FILE *timelog;
        char timelog_buf[CIR_NAME_SIZE];
        struct timeb start,end;
    signal *tslices;
        ftime(&start);

    char **inputs;
    char **outputs;
    int i;
    char *gen_cir;
    char my_cir[CIR_NAME_SIZE];
    char buf1[COMMAND_SIZE];
    char buf2[COMMAND_SIZE];
    char btifile[BTI_NAME_SIZE];

        if (argc == 2)
        {
                sscanf( argv[1], "%d", &circID );
                if(circID == 0)
                {
                        exit(EXIT_SUCCESS);
                }
        }
    if ( argc < 8 )
    {
        help( argv[0] );
        exit( EXIT_FAILURE );
    }

    /*
     * Here be input parsing
     */
    sscanf( argv[1], "%d", &circID );
    sscanf( argv[2], "%d", &timeslice );

    sscanf( argv[3], "%d", &Ninputs );

    if ( 7+Ninputs > argc )
    {
```

```
60            error( ERR_INPUT_SIZE );
61            help( argv[0] );
62            exit( EXIT_FAILURE );
63        }
64
65        inputs = (char **) malloc( Ninputs * sizeof( char* ) );
66        for( i = 0; i < Ninputs; i++ )
67            inputs[ i ] = argv[i+4];
68
69        sscanf( argv[4+Ninputs], "%d", &Noutputs );
70
71        if ( Noutputs+Ninputs+6 > argc )
72        {
73            error( ERR_OUTPUT_SIZE );
74            help( argv[0] );
75            exit( EXIT_FAILURE );
76        }
77
78        outputs = (char **) malloc( Noutputs * sizeof( char* ) );
79        for( i = 0; i < Noutputs; i++ )
80            outputs[ i ] = argv[i+5+Ninputs];
81
82        gen_cir = argv[5+Ninputs+Noutputs];
83        /*
84         * End of input parsing
85         */
86
87
88        /*
89         * copy the file and rename it
90         * format: timeslice_circuitID.cir
91         */
92
93        snprintf( timelog_buf, CIR_NAME_SIZE, "timelogs/%04d_%05d.log", timeslice, circID );
94            timelog = fopen(timelog_buf,"w");
95
96        snprintf( my_cir, CIR_NAME_SIZE, "%04d_%05d.cir", timeslice, circID );
97        my_cir[CIR_NAME_SIZE-1]='\0';
98        copy( gen_cir, my_cir );
99
100       /*
101        * replace all INPUT in generic circ
102        * so that they correspond to the output
103        * of previous executions
104        */
105       for( i = 0; i < Ninputs ; i++ )
106       {
107           create_input_file( timeslice, circID, i, inputs[ i ] );
108           snprintf( buf1, COMMAND_SIZE, ".include INPUT%03d", i+1 );
109           buf1[COMMAND_SIZE-1] = '\0';
110           snprintf( buf2, COMMAND_SIZE, ".include %04d_%05d_%03d.in", timeslice, circID,  i );
111           buf2[COMMAND_SIZE-1] = '\0';
112           replace_line_in_file( my_cir, buf1, buf2 );
113       }
114
115
116       tslices = read_sig("tslices.out");
117       timesliceDuration = (int)(tslices->value[timeslice-1]*1e9);
118       delete_sig(tslices);
119       snprintf( buf2, COMMAND_SIZE, ".tran %dns %dns", timesliceDuration, timesliceDuration );
120       buf2[COMMAND_SIZE-1] = '\0';
121       replace_line_in_file( my_cir, ".tran", buf2 );
```

```
122
123     snprintf( buf1, COMMAND_SIZE, NGSPICE_CMD_LINE, my_cir, timeslice, circID, timeslice, circID);
124
125     /*
126      * Setup the environment variable for the BTI output filename
127      * used in dimitris stamoulis' version
128      */
129     snprintf( btifile, BTI_NAME_SIZE, BTI_FILENAME, circID);
130     if (setenv(BTI_FILENAME_VAR, btifile, 1) != 0)
131     {
132         perror("Failed to set BTI_FILENAME");
133         exit(1);
134     }
135
136     buf1[COMMAND_SIZE-1] = '\0';
137     spice_ret = system( buf1 );
138     if(spice_ret != 0)
139     {
140         perror("SPICE Failed...");
141         exit(spice_ret);
142     }
143
144
145     parse_spice_output( timeslice, circID, Noutputs, outputs );
146
147     clean_files( timeslice, circID, Ninputs, inputs );
148
149
150     free(inputs);
151     free(outputs);
152
153         ftime(&end);
154         dtime = (end.time-start.time)*1e6+(end.millitm-start.millitm);
155
156
157         fprintf(timelog,"%lf\n", dtime/1e6);
158         fclose(timelog);
159     exit( EXIT_SUCCESS );
160
161 }
```

Microsig library

```
1  #ifndef SIGNAL_H
2  #define SIGNAL_H
3
4  #include <math.h>
5  #include "filter_common.h"
6
7  #define SIG_INIT_SIZE 100
8  #define SIG_REALLOC_STEP 2
9  #define FILE_DATA_WIDTH 2
10
11 #define SIG_AVG_STEPPING 1
12 #define SIG_RMSE_STEPPING 1
13
14 #define SIG_SLOPE_LIMIT 1e-3
15
16 struct _signal {
```

```
17      uint32_t len;
18      uint32_t alloc;
19      double *time;
20      double *value;
21  };
22
23  typedef struct _signal signal;
24
25  /*
26   * SIG utils
27   */
28  signal *create_sig(uint32_t init);
29  signal *delete_sig(signal *sig);
30  void extend_sig(signal *sig, double t, double v);
31  void append_sig(signal *sig1, signal *sig2);
32  void trim_sig(signal *sig);
33  signal *add_sig(signal *sig1, signal *sig2);
34  double slope(signal *sig, uint32_t i);
35  double interpolate(signal *sig, uint32_t p1, uint32_t p2, double t);
36  double average(signal *sig);
37  double rmse(signal *errors);
38  signal *resample_sig(signal *sig, uint32_t points);
39  signal *zeros(double step, uint32_t points);
40  void invert_sig(signal *sig);
41  void shift_sig(signal *sig, double offset);
42
43
44  /*
45   * SIG IO
46   */
47  signal *read_sig(const char *fname);
48  signal *read_n_filter_sig(const char *fname);
49  signal *read_n_filter_pwl_sig(const char *fname);
50  signal *read_pwl_sig(const char *fname);
51  void print_sig(signal *sig);
52  void fprint_sig(FILE *f, signal *sig);
53  void write_sig(const char *fname, signal *sig);
54
55  #endif
```

```
1   #include "signal.h"
2   #include "filter.h"
3
4   signal *create_sig(uint32_t init)
5   {
6       signal *sig = NULL;
7
8       sig = (signal *) malloc(sizeof(signal));
9       ASSERT(sig != NULL);
10
11      sig->len = 0;
12      sig->alloc = init;
13      sig->time = (double *) malloc(init*sizeof(double));
14      sig->value = (double *) malloc(init*sizeof(double));
15      return sig;
16  }
17
18  signal *delete_sig(signal *sig)
19  {
20      free(sig->time);
21      free(sig->value);
22      free(sig);
```

```
23	    return NULL;
24	}
25	
26	void *my_realloc(void *p, size_t size)
27	{
28	    void *buf = NULL;
29	    ASSERT(p != NULL);
30	    buf = realloc(p, size);
31	    ASSERT(buf != NULL);
32	    return buf;
33	}
34	
35	/*
36	 * There may be points we dont want in the signal
37	 * YOU SHALL NOT PASS!
38	 */
39	static bool gandalf_point(signal *sig, double t, double v)
40	{
41	    return  ((t == sig->time[sig->len-1]) && (v != sig->value[sig->len-1])) || \
42	                 (t < sig->time[sig->len-1]);
43	}
44	
45	void extend_sig(signal *sig, double t, double v)
46	{
47	    ASSERT(sig != NULL);
48	    if(sig->len > 0)
49	    {
50	        if (gandalf_point(sig, t, v))
51	        {
52	            debug("error in signal input ignoring entries...:\n");
53	            debug("times: %le (ignored) %le\n", t, sig->time[sig->len-1]);
54	            debug("values: %le (ignored) %le\n", v, sig->value[sig->len-1]);
55	            return;
56	        }
57	    }
58	    if(sig->len == sig->alloc)
59	    {
60	        sig->alloc *= SIG_REALLOC_STEP;
61	        debug("need to extend to %d\n", sig->alloc);
62	        ASSERT(sig->alloc > sig->len);
63	        sig->time = (double *) my_realloc(sig->time, (sig->alloc) * sizeof(double));
64	        sig->value = (double *) my_realloc(sig->value, (sig->alloc) * sizeof(double));
65	    }
66	    sig->time[sig->len] = t;
67	    sig->value[sig->len] = v;
68	    sig->len++;
69	}
70	
71	void trim_sig(signal *sig)
72	{
73	    return;
74	    ASSERT(sig != NULL);
75	    ASSERT(sig->time != NULL);
76	    ASSERT(sig->value != NULL);
77	    sig->time = my_realloc(sig->time, (sig->len)*sizeof(double));
78	    sig->value = my_realloc(sig->value, (sig->len)*sizeof(double));
79	    sig->alloc = sig->len;
80	}
81	
82	double interpolate(signal *sig, uint32_t p1, uint32_t p2, double t)
83	{
84	    double v1 = 0;
```

```
85      double v2 = 0;
86      double t1 = 0;
87      double t2 = 0;
88      double rate = 0;
89
90      if(p1 < sig->len)
91      {
92          t1 = sig->time[p1];
93          v1 = sig->value[p1];
94      }
95      else
96      {
97          t1 = 0;
98          v1 = 0;
99      }
100
101     if(p2 < sig->len)
102     {
103         t2 = sig->time[p2];
104         v2 = sig->value[p2];
105         rate = (v2 - v1)/(t2-t1);
106     }
107     else
108     {
109         t2 = 0;
110         v2 = 0;
111         rate = 0;
112     }
113
114
115     return v1 + (t-t1)*rate;
116 }
117
118 signal *add_sig(signal *sig1, signal *sig2)
119 {
120     uint32_t i1, i2, i3;
121     double t1, t2;
122     double v1, v2;
123     double ts, vs;
124     signal *sum_sig = NULL;create_sig(SIG_INIT_SIZE);
125     i1 = 0;
126     i2 = 0;
127     i3 = 0;
128     t1 = t2 = ts = 0;
129     v1 = v2 = vs = 0;
130
131     sum_sig = create_sig(SIG_INIT_SIZE);
132     /*
133      * Ok here we create a signal with all the timestamps of each of sig1 sig2
134      * this will be used as a buffer to add the signals and it will be returned
135      */
136     while(i1 < sig1->len && i2 < sig2->len)
137     {
138         t1 = sig1->time[i1];
139         t2 = sig2->time[i2];
140         if(t1 < t2)
141         {
142             extend_sig(sum_sig, t1, 0);
143             i1++;
144         }
145         else if (t1 > t2)
146         {
```

```
147                 extend_sig(sum_sig, t2, 0);
148                 i2++;
149             }
150             else
151             {
152                 extend_sig(sum_sig, t1, 0);
153                 i1++;
154                 i2++;
155             }
156         }
157         while(i1 < sig1->len)
158         {
159             t1 = sig1->time[i1];
160             extend_sig(sum_sig, t1, 0);
161             i1++;
162         }
163         while(i2 < sig2->len)
164         {
165             t2 = sig2->time[i2];
166             extend_sig(sum_sig, t2, 0);
167             i2++;
168         }
169 
170         i1 = 0;
171         i2 = 0;
172         i3 = 0;
173 
174         t1 = sig1->time[i1];
175         t2 = sig2->time[i2];
176         v1 = sum_sig->value[i1];
177         v2 = sum_sig->value[i2];
178 
179         vs = v1+v2;
180         sum_sig->value[i3] = vs;
181         ts = sum_sig->time[i3];
182         if(t1 <= ts)
183         {
184             i1++;
185         }
186         if(t2 <= ts)
187         {
188             i2++;
189         }
190 
191         for(i3 = 1; i3 < sum_sig->len && i1 < sig1->len && i2 < sig2->len; ++i3)
192         {
193             t1 = sig1->time[i1];
194             t2 = sig2->time[i2];
195             ts = sum_sig->time[i3];
196 
197             v1 = sig1->value[i1];
198             v2 = sig2->value[i2];
199 
200             if(t1 > ts)
201             {
202                 ASSERT(ts == t2);
203                 vs = v2 + interpolate(sig1, i1-1, i1, ts);
204                 sum_sig->value[i3] = vs;
205                 i2++;
206             }
207             else if (t2 > ts)
208             {
```

```
209                 ASSERT(ts == t1);
210                 vs = v1 + interpolate(sig2, i2-1, i2, ts);
211                 sum_sig->value[i3] = vs;
212                 i1++;
213             }
214             else
215             {
216                 ASSERT(ts == t1 && ts == t2);
217                 vs = v1 + v2;
218                 sum_sig->value[i3] = vs;
219                 i1++;
220                 i2++;
221             }
222         }
223 
224         v2 = sig2->value[sig2->len-1];
225         while(i1 < sig1->len)
226         {
227             vs = v2 + sig1->value[i1];
228             sum_sig->value[sig2->len - 1 + i1] = vs;
229             i1++;
230         }
231 
232         v1 = sig1->value[sig1->len-1];
233         while(i2 < sig2->len)
234         {
235             vs = v1 + sig2->value[i2];
236             sum_sig->value[sig1->len - 1 + i2] = vs;
237             i2++;
238         }
239 
240         trim_sig(sum_sig);
241         return sum_sig;
242     }
243 
244     void shift_sig(signal *sig, double offset)
245     {
246         uint32_t i;
247 
248         for(i = 0; i < sig->len; ++i)
249         {
250             sig->time[i] += offset;
251         }
252     }
253 
254 
255     signal *read_n_filter_sig(const char *fname)
256     {
257         signal *sig = NULL;
258         signal *rsig = NULL;
259         sig = read_sig(fname);
260         rsig = filter(sig);
261         delete_sig(sig);
262         return rsig;
263     }
264     signal *read_n_filter_pwl_sig(const char *fname)
265     {
266         signal *sig = NULL;
267         signal *rsig = NULL;
268         sig = read_pwl_sig(fname);
269         rsig = filter(sig);
270         delete_sig(sig);
```

```
271     return rsig;
272 }
273
274 signal *read_pwl_sig(const char *fname)
275 {
276     FILE *f = NULL;
277     signal *sig = NULL;
278     double stamp = 0;
279     double value = 0;
280     char dump = '\0';
281
282     f = fopen(fname, "r");
283     if(f == NULL)
284     {
285         perror("read_pwl_signal failed");
286         exit(EXIT_FAILURE);
287     }
288
289     sig = create_sig(SIG_INIT_SIZE);
290     if(sig == NULL)
291     {
292         perror("failed to initialize sig");
293         exit(EXIT_FAILURE);
294     }
295
296     while((dump = fgetc(f)) != '(' && dump != EOF)
297         ;
298
299     if ((fscanf(f, "%le", &stamp) == 1) && (fscanf(f, "%le\n", &value) == 1))
300     {
301         extend_sig(sig, stamp, value);
302     }
303     else
304     {
305         perror("read_pwl_signal failed");
306         exit(EXIT_FAILURE);
307     }
308
309     while((fscanf(f, "+%c", &dump) == 1) && (fscanf(f, "%le", &stamp) == 1) && (fscanf(f, "%le\n", &value) == 1))
310     {
311         extend_sig(sig, stamp, value);
312     }
313     trim_sig(sig);
314     fclose(f);
315     return sig;
316 }
317
318 signal *read_sig(const char *fname)
319 {
320     FILE *f;
321     signal *sig;
322     double stamp;
323     double value;
324
325     f = fopen(fname, "r");
326     if(f == NULL)
327     {
328         fprintf(stderr, "read_signal failed %s\n", fname);
329         exit(EXIT_FAILURE);
330     }
331
332     sig = create_sig(SIG_INIT_SIZE);
```

```
333
334     while((fscanf(f, "%le", &stamp) == 1) && (fscanf(f, "%le", &value) == 1))
335     {
336         extend_sig(sig, stamp, value);
337     }
338     trim_sig(sig);
339     fclose(f);
340     return sig;
341 }
342
343 void write_sig(const char *fname, signal *sig)
344 {
345     FILE *f = NULL;
346
347     f = fopen(fname, "w");
348     if(f == NULL)
349     {
350         perror("write_sig failed");
351         exit(EXIT_FAILURE);
352     }
353     fprint_sig(f, sig);
354     fclose(f);
355 }
356
357 void print_sig(signal *sig)
358 {
359     fprint_sig(stdout, sig);
360 }
361
362 void fprint_sig(FILE *f, signal *sig)
363 {
364     uint32_t i = 0;
365     fprintf(f, "%le %le\n", sig->time[0], sig->value[0]);
366     for(i = 1; i < sig->len; ++i)
367     {
368             fprintf(f, "%le %le\n", sig->time[i], sig->value[i]);
369     }
370 }
371
372 double slope(signal *sig, uint32_t i)
373 {
374     ASSERT(i < sig->len);
375     double v1, v2;
376     double t1, t2;
377
378     v1 = sig->value[i-1];
379     v2 = sig->value[i];
380     t1 = sig->time[i-1];
381     t2 = sig->time[i];
382     if(t1 == t2)
383     {
384         debug("%le %le %le %le\n", t1, v1, t2, v2);
385         ASSERT(v1 == v2);
386         return 0;
387     }
388
389     return (v2-v1)/(t2-t1);
390 }
391
392 void append_sig(signal *sig1, signal *sig2)
393 {
394     uint32_t i;
```

```
395     for(i = 0; i < sig2->len; ++i)
396     {
397         extend_sig(sig1, sig2->time[i], sig2->value[i]);
398     }
399 }
400
401 void invert_sig(signal *sig)
402 {
403     uint32_t i;
404     for(i = 0; i < sig->len; ++i)
405     {
406         sig->value[i] *= -1;
407     }
408 }
409
410 double average(signal *sig)
411 {
412     uint32_t i;
413     double avg = 0;
414     double avg_buff = 0;
415     for(i = 0; i < sig->len; ++i)
416     {
417         avg_buff = sig->value[i];
418         avg += avg_buff/sig->len;
419     }
420     return avg;
421 }
422
423 double rmse(signal *errors)
424 {
425     uint32_t i;
426     double mse = 0;
427     double mse_buff = 0;
428     double diff = 0;
429     for(i = 0; i < errors->len; ++i)
430     {
431         diff = errors->value[i];
432         mse_buff = diff*diff;
433         mse += mse_buff;
434     }
435     return sqrt(mse/errors->len);
436 }
437
438 signal *zeros(double step, uint32_t points)
439 {
440     double curr;
441     uint32_t i;
442     signal *sig;
443
444     sig = create_sig(SIG_INIT_SIZE);
445     curr = 0;
446     for(i = 0; i < points; ++i)
447     {
448         extend_sig(sig, curr, 0);
449         curr += step;
450     }
451     trim_sig(sig);
452     return sig;
453 }
454
455 signal *resample_sig(signal *sig, uint32_t points)
456 {
```

```
457     double duration = 0;
458     double step = 0;
459     signal *resample = NULL;
460     signal *ans = NULL;
461
462
463     duration = sig->time[sig->len - 1];
464     step = duration/points;
465     resample = zeros(step, points);
466
467     ans = add_sig(sig, resample);
468     return ans;
469 }
```

```
1  #include "filter.h"
2
3  static state in_thresolds(signal *sig, uint32_t p)
4  {
5      double v;
6      ASSERT(sig != NULL);
7      ASSERT(p <= sig->len);
8      v = sig->value[p];
9      /*
10     if(v > VALUE_HIGH_OVER_THRESHOLD || v < VALUE_LOW_UNDER_THRESHOLD)
11     {
12         return STATE_ERROR;
13     }
14     else
15     */
16     if(v <= VALUE_LOW_OVER_THRESHOLD)
17     {
18         return STATE_LOW;
19     }
20     else if(v >= VALUE_HIGH_UNDER_THRESHOLD)
21     {
22         return STATE_HIGH;
23     }
24     else
25     {
26         return STATE_MID;
27     }
28
29 }
30 static void quantinise(signal *sig)
31 {
32     state s;
33     uint32_t i;
34     for(i = 0; i < sig->len; ++i)
35     {
36         s = in_thresolds(sig, i);
37         switch(s)
38         {
39             case STATE_ERROR:
40                 fprintf(stderr, "value out of bounds %lf\n", sig->value[i]);
41             case STATE_LOW:
42                 sig->value[i] = VALUE_LOW;
43                 break;
44             case STATE_HIGH:
45                 sig->value[i] = VALUE_HIGH;
46                 break;
47             default:
48                 break;
```

```
49            }
50        }
51    }
52
53    static bool pass(signal *sig, uint32_t p1, uint32_t p2, uint32_t p3)
54    {
55        state s1, s2, s3;
56        s1 = in_thresolds(sig, p1);
57        s2 = in_thresolds(sig, p2);
58        s3 = in_thresolds(sig, p3);
59
60        ASSERT(s1 != STATE_ERROR);
61        ASSERT(s2 != STATE_ERROR);
62        ASSERT(s3 != STATE_ERROR);
63
64        /*
65         * XXX: for now we will not filter out any STATE_MID points
66         */
67        return s1 == s2 && s2 == s3 && s1 != STATE_MID;
68
69    }
70
71    static uint32_t same(signal *sig, uint32_t start)
72    {
73        uint32_t stop;
74
75        stop = start+1;
76        while(stop < sig->len-1 && pass(sig, start, stop, stop + 1))
77        {
78            stop++;
79        }
80        return stop;
81    }
82
83    signal *filter(signal *sig)
84    {
85        signal *fsig;
86        uint32_t start;
87        uint32_t stop;
88
89        ASSERT(sig != NULL);
90
91        fsig = create_sig(SIG_INIT_SIZE);
92
93        ASSERT(fsig != NULL);
94
95
96        start = 0;
97        while(start < sig->len)
98        {
99            extend_sig(fsig, sig->time[start], sig->value[start]);
100           stop = same(sig, start);
101           extend_sig(fsig, sig->time[stop], sig->value[stop]);
102           start = stop + 1;
103       }
104
105       trim_sig(fsig);
106       quantinise(fsig);
107       return fsig;
108   }
109
110   signal *crude_filter(signal *sig)
```

```
{
    signal *fsig;
    signal *rsig;
    state s;
    uint32_t i;
    ASSERT(sig != NULL);

    fsig = filter(sig);
    rsig = create_sig(SIG_INIT_SIZE);

    ASSERT(fsig != NULL);
    ASSERT(rsig != NULL);


    for(i = 0; i < fsig->len; ++i)
    {
        s = in_thresolds(fsig, i);
        if(s == STATE_LOW || s == STATE_HIGH)
        {
            extend_sig(rsig, fsig->time[i], fsig->value[i]);
        }
    }
    trim_sig(rsig);
    quantinise(rsig);
    delete_sig(fsig);
    return rsig;
}
```

Partitioner

```
#include "signal.h"
#include "filter.h"
#include "party.h"

#define CWD_SIZE 1024

#include <math.h>
#include <dirent.h>
#include <unistd.h>
#include <fnmatch.h>

char *pattern = NULL;

static int files_filter(const struct dirent *d)
{
    int r = d->d_type == DT_REG && !fnmatch(pattern, d->d_name, 0);
    return r;
}

int main(int argc, char **argv)
{

    signal *aggregated_sig = NULL;
    signal *profile = NULL;
    uint32_t available_timeslices = 0;
    uint32_t slices = 0;
    int file_cnt = 0;
    char cwd[CWD_SIZE] = "";
    struct dirent **file_names = NULL;
```

```
30      int i = 0;
31
32      if(argc != 3)
33      {
34          fprintf(stderr, "Usage: party #slices <fname patern>\n");
35          exit(EXIT_FAILURE);
36      }
37      sscanf(argv[1], "%u", &slices);
38
39      pattern = argv[2];
40
41      getcwd(cwd, CWD_SIZE);
42      printf("%s\n", cwd);
43
44      file_cnt = scandir(cwd, &file_names, files_filter, alphasort);
45
46
47      debug("So I have to process %d files... hmm...\n", file_cnt);
48      for(i = 0; i < file_cnt; ++i)
49          debug("Filename %s\n", file_names[i]->d_name);
50
51      aggregated_sig = party_prof(file_cnt, file_names);
52      if(aggregated_sig == NULL)
53      {
54          exit(EXIT_FAILURE);
55      }
56      write_sig("aggregated_sig.out", aggregated_sig);
57      profile = party_times(aggregated_sig);
58
59      /*
60       * At this point sig1 holds the indices that the addition of signals can be
61       * partitioned. More specifically the right end of those signal slices
62       */
63      available_timeslices = profile->len;
64      slices = slices < available_timeslices ? slices : available_timeslices;
65      printf("Available timeslices: %u Slices to be applied: %u\n", \
66              available_timeslices, slices);
67      write_sig("times.out", profile);
68
69
70      for(i = 0; i < file_cnt; ++i)
71      {
72          debug("processing signal %s...\n", file_names[i]->d_name);
73          party(file_names[i]->d_name, profile, slices);
74      }
75
76      delete_sig(aggregated_sig);
77      delete_sig(profile);
78      for(i = 0; i < file_cnt; ++i)
79      {
80          free(file_names[i]);
81      }
82      free(file_names);
83
84      exit(EXIT_SUCCESS);
85  }
```

```
1   #include <math.h>
2   #include <string.h>
3
4   #include "signal.h"
5   #include "filter.h"
```

```
6   #include "party.h"
7
8   /*
9    * Generates the profile signal used later on for slicing
10   */
11  signal *party_prof(int file_cnt, struct dirent **file_names)
12  {
13      signal *sig1 = NULL;
14      signal *sig2 = NULL;
15      signal *asig = NULL;
16      signal *fsig = NULL;
17      int i;
18
19      if(file_cnt == 0)
20      {
21          return NULL;
22      }
23
24
25      debug("profiling signal %s...\n", file_names[0]->d_name);
26      sig1 = read_pwl_sig(file_names[0]->d_name);
27      fsig = crude_filter(sig1);
28      sig1 = delete_sig(sig1);
29      sig1 = fsig;
30      asig = sig1;
31
32      for(i = 1; i < file_cnt; ++i)
33      {
34          debug("profiling signal %s...\n", file_names[i]->d_name);
35          sig2 = read_pwl_sig(file_names[i]->d_name);
36          fsig = crude_filter(sig2);
37          sig2 = delete_sig(sig2);
38          sig2 = fsig;
39
40          asig = add_sig(sig1, sig2);
41          sig1 = delete_sig(sig1);
42          sig2 = delete_sig(sig2);
43          sig1 = asig;
44      }
45      return asig;
46  }
47
48  /*
49   * Used to generate the times that the signal can be cut
50   */
51  signal *party_times(signal *profile)
52  {
53      signal *sig1;
54      uint32_t i;
55      double dt;
56      double s;
57
58      sig1 = create_sig(SIG_INIT_SIZE);
59      extend_sig(sig1, 0, 0);
60      for(i = 1; i < profile->len; ++i)
61      {
62          s = fabs(slope(profile, i));
63          dt = profile->time[i] - profile->time[i-1];
64          if( s < SIG_SLOPE_LIMIT && (dt > PARTY_MIN_DURATION))
65          {
66              //extend_sig(sig1, i, asig->time[i] - asig->time[i-1]);
67              extend_sig(sig1, 0.5 * (profile->time[i] + profile->time[i-1]), 0);
```

```
68            }
69        }
70        trim_sig(sig1);
71
72        return sig1;
73    }
74
75    /*
76     * Does the partitioning based on the signal profile generated previously
77     */
78    void party(const char *fname, signal *profile, uint32_t timeslices)
79    {
80        uint32_t available_timeslices = 0;
81        int step = 0;
82        uint32_t i = 0;
83        uint32_t j = 0;
84        double t1 = 0;
85        double t2 = 0;
86        double ts = 0;
87        double tprev = 0;
88        double v = 0;
89        double vprev = 0;
90        char newf[64];
91        signal *input = 0;
92        signal *output = 0;
93        FILE *slicelog = fopen("tslices.out", "w");
94
95
96        input = read_n_filter_pwl_sig(fname);
97        available_timeslices = profile->len;
98        step = (int) ceil((double) available_timeslices / (double) timeslices);
99
100
101       for(i = step, j = 0; i < available_timeslices; i += step)
102       {
103           output = create_sig(SIG_INIT_SIZE);
104           t1 = profile->time[i-step];
105           t2 = profile->time[i];
106           ts = input->time[j];
107           if(ts >= t1)
108           {
109               if (t1 == 0)
110               {
111                   v = input->value[0];
112               }
113               else
114               {
115                   v = interpolate(input, j - 1, j, t1);
116               }
117               extend_sig(output, t1, v);
118           }
119
120           for(; j < input->len && ts >= t1 && ts < t2; ts = input->time[++j])
121           {
122               v = input->value[j];
123               extend_sig(output, ts, v);
124           }
125
126           if(ts >= t2)
127           {
128               if (t2 == 0)
129               {
```

```
130                 v = input->value[0];
131             }
132             else
133             {
134                 v = interpolate(input, j - 1, j, t2);
135             }
136             extend_sig(output, t2, v);
137         }
138 
139         trim_sig(output);
140 #if SET_DEBUG
141         snprintf(newf, 48, "%04d_dbg_%s.prty", i/step, fname);
142         write_sig(newf, output);
143 #endif /* SET_DEBUG */
144         snprintf(newf, 48, "%04d_%s.prty", i/step, fname);
145         shift_sig(output, -t1);
146         fprintf(slicelog, "%le\t%le\n", t1, output->time[output->len-1]);
147         write_sig(newf, output);
148         delete_sig(output);
149     }
150 
151     output = create_sig(SIG_INIT_SIZE);
152     t1 = profile->time[i-step];
153     ts = input->time[j];
154     if(ts >= t1)
155     {
156         v = interpolate(input, j - 1, j, t1);
157         extend_sig(output, t1, v);
158     }
159     for(; j < input->len; ts = input->time[++j])
160     {
161         v = input->value[j];
162         extend_sig(output, ts, v);
163     }
164     trim_sig(output);
165 #if SET_DEBUG
166     snprintf(newf, 48, "%04d_dbg_%s.prty", i/step, fname);
167     write_sig(newf, output);
168 #endif /* SET_DEBUG */
169     snprintf(newf, 48, "%04d_%s.prty", i/step, fname);
170     shift_sig(output, -t1);
171     fprintf(slicelog, "%le\t%le\n", t1, output->time[output->len-1]);
172     write_sig(newf, output);
173     delete_sig(output);
174     fclose(slicelog);
175 }
```