Multi-Level Simulation of Nano-Electronic Digital Circuits on GPUs

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Eric Schneider

aus Hildburghausen, Deutschland

Hauptberichter: Prof. Dr. rer. nat. habil. Hans-Joachim Wunderlich
Mitberichter: Prof. Dr. phil. nat. Rolf Drechsler

Tag der mündlichen Prüfung: 21. Juni 2019.

Institut für Technische Informatik der Universität Stuttgart

2019

Contents

Acknowledgments

I would like to thank Prof. Hans-Joachim Wunderlich for giving me the opportunity to work at the *Institut für Technische Informatik* (ITI) in Stuttgart and to let me experience the life in academia under his professional supervision. I am deeply grateful for his invaluable patience and faith in me, as well as the countless and very often insightful discussions we had. Furthermore, I want to thank Prof. Rolf Drechsler from University of Bremen for taking over the role as my second adviser and also for the nice chats.

In addition, I am very grateful to Stefan Holst, who was my mentor during my undergraduate studies and who helped me getting started at the ITI as a colleague and friend.

I very much enjoyed the work with the other colleagues and friends as well, which altogether made the life at the institute interesting and enjoyable in many ways. Therefore I am happy to thank Michael A. Kochte and Claus Braun as well as Laura Rodrígez Gómez, Dominik Ull, Chang Liu, Rafał Baranowski, Alexander Schöll, Michael Imhof, Abdullah Mumtaz, Marcus Wagner, Sebastian Brandhofer, Ahmed Atteya, Natalia Lylina, Zahra Paria Najafi Haghi and Chih-Hao Wang. I also want to thank Lars Bauer from Karlsruhe Institute of Technology, Matthias Sauer formerly from University of Freiburg, Prof. Sybille Hellebrand and Matthias Kampmann from University of Paderborn as well as Prof. Xiaoqing Wen and Kohei Miyase from Kyushu Institute of Technology and Yuta Yamato formerly from Nara Institute of Science and Technology for all the insightful discussions and collaborations we could establish together.

Thank you, Mirjam Breitling, Lothar Hellmeier and Helmut Häfner. Without your administrative and technical assistance this work would have been impossible for me.

Last but not least, I am also very grateful to my parents for their constant support and caring.

Stuttgart, June 2019

Eric Schneider

Abstract

Simulation of circuits and faults is an essential part in design and test validation tasks of contemporary nano-electronic digital integrated CMOS circuits. Shrinking technology processes with smaller feature sizes and strict performance and reliability requirements demand not only detailed validation of the functional properties of a design, but also accurate validation of non-functional aspects including the timing behavior. However, due to the rising complexity of the circuit behavior and the steady growth of the designs with respect to the transistor count, timing-accurate simulation of current designs requires a lot of computational effort which can only be handled by proper abstraction and a high degree of parallelization.

This work presents a simulation model for scalable and accurate timing simulation of digital circuits on data-parallel *graphics processing unit* (GPU) accelerators. By providing compact modeling and data-structures as well as through exploiting multiple dimensions of parallelism, the simulation model enables not only fast and timing-accurate simulation at logic level, but also massively-parallel simulation with switch level accuracy.

The model facilitates extensions for fast and efficient fault simulation of small delay faults at logic level, as well as first-order parametric and parasitic faults at switch level. With the parallelization on GPUs, detailed and scalable simulation is enabled that is applicable even to multi-million gate designs. This way, comprehensive analyses of realistic timing-related faults in presence of process- and parameter variations are enabled for the first time.

Additional simulation efficiency is achieved by merging the presented methods in a unified simulation model, that allows to combine the unique advantages of the different levels of abstraction in a mixed-abstraction multi-level simulation flow to reach even higher speedups.

Experimental results show that the implemented parallel approach achieves unprecedented simulation throughput as well as high speedup compared to conventional timing simulators. The underlying model scales for multi-million gate designs and gives detailed insights into the timing behavior of digital CMOS circuits, thereby enabling large-scale applications to aid even highly complex design and test validation tasks.

Zusammenfassung

Simulation ist ein wichtiger Bestandteil der Entwurfs- und Testvalidierung von heutigen nano-elektronischen digitalen Schaltungen. Die Herstellungsprozesse von Technologien mit geringen Strukturgrößen im Bereich weniger Nanometer unterliegen strengen Anforderungen an die Zuverlässigkeit und Performanz der zu produzierenden Schaltungen. Daher ist es schon während des Entwurfs eine Validierung der funktionalen Aspekte notwendig und überdies hinaus auch eine akkurate Validierung der nicht-funktionalen Aspekte einschließlich der Simulation des Zeitverhaltens. Aufgrund der steigenden Komplexität des Schaltverhaltens und der stetig steigenden Zahl von Transistoren in den Schaltungen benötigt akkurate Zeitsimulation jedoch enormen Rechenaufwand, welcher nur mit geeigneter Modellabstraktion und einem hohen Grad an Parallelisierung bewältigt werden kann.

In dieser Arbeit wird ein Simulationsmodell zur akkuraten Zeitsimulation digitaler Schaltungen auf massiv daten-parallelen Grafikbeschleunigern (sogenannten ”GPUs”) vorgestellt. Die Verwendung einer kompakten Modellierung und geeignete Datenstrukturen sowie die simultane Ausnutzung mehrerer Dimensionen von Parallelismus ermöglichen nicht nur schnelle und zeitgenaue Simulation auf Logik-Ebene, sondern erstmals auch skalierbare massiv-parallele Simulation mit erhöhter Genauigkeit auf Schalter-Ebene.

Erweiterungen der Modellierung bieten schnelle und effiziente Fehlersimulation von kleinsten Verzögerungsfehlern auf Logik-Ebene, als auch parametrische und parasitische Fehler in elektrischen Parametern erster Ordnung auf Schalter-Ebene. Die Parallelisierung auf den GPU-Architekturen erlaubt darüber hinaus erstmals detaillierte Simulationen und Analysen von realistischen Verzögerungsfehlern unter Prozess- und Parametervariationen für Schaltkreise mit Millionen von Gattern.

Durch Vereinigung der implementierten Methoden in einem mehrere Ebenen umfassenden Simulationskonzept mit gemischten Abstraktionen werden die Vorteile der verschiedenen Abstraktionsebenen kombiniert, wodurch die Effizienz der Zeitsimulation gesteigert und zusätzliche Beschleunigung erzielt werden kann.
Ergebnisse durchgeführter Experimente zeigen, dass der implementierte parallele Simulationsansatz einen hohen Rechendurchsatz mit hohem Grad an Beschleunigung gegenüber konventioneller Zeitsimulation erzielt. Das zugrundeliegende Simulationsmodell skaliert für Schaltkreise mit Millionen von Gattern und gewährt detaillierte Einsicht in das Zeitschaltverhalten digitaler CMOS Schaltungen, wodurch umfassende Anwendungen zur Unterstützung auch für hochkomplexe Aufgaben beim Entwurf und Test nano-elektronischer Schaltungen ermöglicht werden.

List of Figures

List of Tables

List of Abbreviations

ATPG Automatic test pattern generation

CAT Cell-aware test

CCC Channel-connected component

CLF conditional line flip

CMOS Complementary metal-oxide semiconductor

CUDA Computer unified device architecture

DSPF Detailed parasitics file format

EDA Electronic design automation

ELF Early-life failures

EM Electro-migration

FAST Faster-than-at-speed test

FinFET Fin-type field-effect-transistor

FLOP Floating-point operation

GND Ground voltage potential

GPU Graphics processing unit

HCI Hot-carrier injection

IFA Inductive fault analysis

MEPS Million (node) evaluations per second

MIS Multiple-input switching

MOSFET Metal-oxide-semiconductor field-effect-transistor

NBTI Negative-bias temperature instability

PPSFP Parallel-pattern single fault propagation (fault simulation)

ROI Region of interest

RTL Register-transfer level

SDF Standard delay format

SM Streaming multi-processor

SPICE Simulation program with integrated circuit emphasis

STA Static timing analysis

STF slow-to-fall

STR slow-to-rise

TDDB Time-dependent dielectric breakdown

VDD Supply voltage potential

Chapter 1

Introduction

In modern semi-conductor development cycles of nanometer CMOS integrated circuits, the simulation of circuits is one of the most important and complex tasks for design and test validation. Circuit simulation is used for design validation to analyze developed designs with respect to validation targets to indicate the compliance with their specification or customer requirements. In test validation on the other hand, fault simulation is utilized to evaluate the defect coverage of test sets and new test strategies, as well as to assess the quality of the tested products. For this, *electronic design automation* (EDA) tools for design and test validation have to rely on repeated, compute-intensive circuit simulations with extensive runtimes, which can pose a bottleneck, especially for designs with billions of transistors [ITR15].

Simulation itself is best described as "*performing experiments on a model*" of the real (physical) world [Cel91, CK06]. The models for simulation typically provide a set of properties and rules, that represents and explains parts of the complex real world behavior in an abstract manner. Once an algorithm has been derived from the model, a *simulation program* (*simulator*) can be efficiently utilized to perform and observe experiments repeatedly on the model for any given input parameters. The output of the simulation then provides data that can help to make predictions regarding the real world behavior. This way, knowledge can be obtained to validate real world hypotheses without the need of costly (and often destructive) experiments in the real world.

For example, in *circuit simulation* abstract models of the real physical silicon chips are provided, that allow to run experiments to predict and observe the behavior of prototype

designs and compare it to its specification. The designer can then draw conclusions from the results to assess the provided functionality of the prototype design, the correctness as well as other non-functional aspects (i.e., timing reliability, power) [WH11, Wun91].
Fault simulation is an extension of circuit simulation in which the behavior of arbitrary defects in a real physical chip are abstracted to *faults*. Each *fault* describes the altered chip behavior with respect to a given defect mechanism, which is reflected as abstract deterministic misbehavior in the underlying circuit simulation model.
The abstraction of a simulation model itself is crucial to avoid dealing with the overall complexity of the real physical world and to make an evaluation feasible at all. While a high accuracy of a simulation model is generally desirable, it can make simulations infeasible due to a high time- and space complexity of the evaluations. By constraining the model to certain aspects and by using simplified assumptions, the costs of the simulations can be lowered and evaluation can be focused on certain problems of interest. On the other hand, too much abstraction reduces the accuracy of the models and also the viability of the simulation and its results.

Challenges

With the continuing advancements in semiconductor manufacturing processes and shrinking technologies of digital integrated CMOS circuits, traditional circuit and fault simulation models have become insufficient. Nowadays, thorough design and test validation as well as diagnosis require more and more *accurate* simulations and have thus become complex and runtime-intensive tasks, since several aspects have lead to an increase in the overall modeling and simulation effort:

Complexity of Circuit Size

Over the past decades, semiconductor manufacturing technologies and processes made continuous advancements that still allow for an ongoing growth in complexity and higher integration of modern integrated circuits [ITR15]. Although Moore's Law [Moo65, Moo75] was already predicted to end [RS11], newer process technologies, such as the *fin-type field effect transistor* (FinFET), enable a scaling of structure sizes down to a few nanometers

only and the production with higher transistor density [Kin05, Hu11]. For example, in 2017 NVIDIA announced a new generation of parallel GPU accelerators: The NVIDIA® Volta™ architecture. The GV100 GPU chip of the family is manufactured in a 12nm FinFET technology and comprises 21.1 billion transistors on a die of 815 mm^2 in size [NVI17d]. To validate such massive designs, scalable parallel simulation algorithms are required that are able to cope with the increasing design complexity and its millions or billions of transistors.

Complexity of Circuit Behavior

While technology scaling causes the circuits to grow in size and functionality, the behavior of standard cells and circuit structures is becoming more and more complex as well [ITR15]. With the increasing demand for high-performance and low-power embedded designs, chips are running today at their operational limit [SBJ+03, DWB+10, KET16, GKET16]. Furthermore, the circuit structures become more and more prone to process- [ZHHO04, QS10] and parameter variations (e.g., voltage and temperature [BPSB00, BKN+03, TKD+07]), which can impact the reliability of the circuit and effectiveness of tests if neglegted [PBH+11, HBH+10, CIJ+12]. Therefore, accurate simulation models are required that deliver sufficiently high precision and detail to capture both functional as well as non-functional aspects of the circuit with as little abstraction as possible. The circuit models must provide support for accurate representation of the temporal behavior to reflect hazards and glitches, signal slopes and parametric and parasitic dependencies. Also, pattern-dependent delays and multiple-input switching effects should be considered as they can significantly alter the circuit delay [MRD92, SDC94, CS96]. However, such *timing-accurate* evaluations involve the vast processing of many real numbers through complex arithmetic operations, which is several times more expensive in term of computing complexity compared to *untimed* logic simulation [SJN94] and also much harder to parallelize [BMC96].

Complexity of Defect Modeling

The complexity of a simulation quickly rises further as soon as the behavior of defects in a circuit needs to be investigated in fault simulation. While classical fault models and

simulation approaches have been extensively studied and optimized in the past [Wun10, Wun91, WLRI87], modern semiconductor manufacturing processes have to deal with several new fine-grained types of defects [LX12, RAH+14, HRG+14, ITR15]. *Small delay defects* (SDD) [TPC11, RMdGV02], *marginal devices* [RAH+14, KCK+10, KKK+10] and *wear-out*-related effects due to circuit *aging* [LGS09, APZM07, BM09, GSR+14] impact the timing behavior of the circuit and require timing- and glitch-accurate evaluation [Kon00, YS04, HS14], especially when considering process variations on top [PBH+11, HBH+10, CIJ+12]. While certain validation tasks can utilize formal verification methods [DG05], the modeling and evaluation of low-level faults and parameters in explicit simulation is much more intuitive and straight-forward [Lam05], especially when a validated formal model does not exist.

Moreover, the faulty behavior of many *realistic* defects often cannot be represented at higher abstraction levels as their behavior strongly depends on the physical layout as well as the low-level topology and technology of the standard cells [SMF85, FS88]. For this, *cell-aware test* (CAT) approaches have been recently developed [HRG+14] that extensively rely on expensive low-level analog fault simulations with *user-defined fault models* (UDFM). Thus, it is important to efficiently simulate faults with little abstraction as accurately and efficiently as possible [CCCW16].

Parallelism as a Remedy

Assume that a circuit of N nodes with a set $\mathcal{F}$ of faults needs to be simulated for a pattern set $\mathcal{T}$ under different operating parameters $\mathcal{P}$. In a naïve flow, a total of $N \times |\mathcal{F}| \times |\mathcal{T}| \times |\mathcal{P}|$ individual *simulation problems* have to be solved, each of which corresponds to complex arithmetic evaluations of the timing behavior at a node. Thus, with growing design size, the number of *simulation problems* quickly rises since the number of nodes, faults and test patterns also increase.

Although circuit- and fault simulation are inherently parallelizable [IT88, BBC94, BMC96], the parallelization has risen to a whole new level with the introduction of general purpose computing on *graphics processing unit* (GPU) accelerators [OHL+08]. In contrast to conventional processors (e.g., IBM Power9 architecture with 24 cores [STKS17]), the GPU-

architectures provide thousands of compute cores on a single device for running millions of threads concurrently [NVI17d]. With this amount of compute cores, the GPU-based systems attain unprecedented arithmetic computing throughput in the order of petaFLOPS[1] on a single compute node, which has established well in high-performance computing for speeding up many applications [NVI18c].

Many approaches that parallelize logic level simulation on GPUs for acceleration were published [GK08, Den10, KSWZ10, CKG13, WLHW14, LTL18] that isolate and partition the simulation problems for independent parallel execution by individual threads, but only very few are able to consider actual timing. Analog simulation with GPU-support [GCKS09, CRWY15, CUS19] and full GPU-accelerated SPICE implementations [HZF13, HF16, vSAH18] have been proposed as well. However, existing parallel approaches for either logic level or analog simulation are mainly optimized for speeding up single simulation instances only. Since the number of simulation problems is continuing to rise, it is of utmost importance that the approaches are not only efficiently parallelizable, but also *scalable* by providing high-simulation throughput to make an evaluation of large designs for more complex algorithms feasible.

Timing Modeling at Logic Level

For the calculation of the full signal switching histories in a circuit, traditional *unit delay* models [Wun91] have become insufficient. Instead, timing-accurate simulation models are required that are able to process real-valued gate delays for individual gate pins and signal switches [IEE01a]. Since the execution of bare arithmetic floating-point instructions has much higher latency compared to bit-wise logic operations of traditional untimed logic simulation, timing-accurate simulation of a million-gate circuit for a test pattern set can take hours or even days to complete [SHK+15].

On GPUs, long-latency processes and arithmetic operations can be *hidden* by the massive computing throughput provided by the parallel computing cores [OHL+08]. For this, massive parallelization needs to be exploited to effectively speed up extensive logic level simulations with timing-accurate and variation-aware delay models for large designs and test sets.

[1] 1 petaFLOP = 10^{15} floating point operations per second

Switch Level Modeling and Algorithms

Timing-accurate and lower-level models are desirable for accurate simulation of digital integrated CMOS circuits. While simulation at logic level lacks accuracy, SPICE simulation lacks scalability even when accelerated by parallelization. As a trade-off, switch level simulation was introduced [Bry87, Hay87, BBH+88, MV91] which drastically reduces the complexity of the evaluation compared to full analog simulation. While not being as accurate, switch level modeling reflects many properties of CMOS circuits and standard cells that cannot be modeled at logic level. This has recently drawn attention in *cell-aware test* (CAT) generation to reduce the analog fault simulation overhead [CCCW16, CWC17]. Yet, the simulation at switch level is still an order of magnitude slower compared to logic simulation [SJN94] and therefore poses a strong bottleneck in validation tasks, which makes it a prime-example for the acceleration on GPUs. However, to the best of the author's knowledge no algorithms to accelerate switch level simulations on GPUs exist.

Defect Modeling and Fault Simulation

Many defects types exhibit a varying faulty behavior based on the *size* or magnitude of the defect parameter. The behavior of such a parametric defect can range from a hard functional failure of the circuit, to a *weak* impact on the *non-functional* timing behavior that only affects the circuit performance [RMdGV02, FGRC17]. With the continuously shrinking circuit structures, the probability of such defects as well as their severity with respect to the circuit reliability rises [ITR15], while at the same time the faults are becoming harder to detect [LX12, TTC+15]. Thus, thorough test validation requires a fine-grained modeling of the defect parameters to capture deviations in both functional and non-functional aspects of the circuit (e.g., small delays [YS04, LKW17, KKL+18]).

To reflect the behavior of more realistic defects found in CMOS cells, faults must be modeled at the lower levels without introducing too much abstraction [FS88, CCCW16, CWC17]. This vastly increases the computational overhead, as the defect mechanisms not only get more complex, but also many simulations of different individual faults are necessary (e.g., for fault coverage estimation). Thus, the need for parallelization of fault simulation is unavoidable [IT88, Wun91]. With their large amount of parallel arithmetic

compute cores, GPUs provide high arithmetic computing throughput that is promising to simulate even large designs for many faults and test patterns with full timing-accuracy.

Multi-Level Simulation

The modeling and simulation at higher abstraction is fast, but the approaches are limited in accuracy such that often timing and defects cannot be reflected properly [RAH+14]. On the other hand, lower level approaches can provide sufficient accuracy, but exhibit significant runtime complexity and resource requirements, such that the approaches cannot be used solely for a system-wide application (i.e., system-level test validation [Che18, JRW14]). However, advantages from both higher- and lower-level abstractions can still be taken into account at the same time by using *multi-level* techniques.

A *multi-level* simulation provides a combined simulation across multiple different levels of abstractions [GMS88, MC95, RK08, KZB+10, HBPW14]. Accurate and compute-intensive evaluations are typically restricted to a particular region of the circuit [HBPW14, KZB+10], or single cells [CCCW16], whereas the remainder is processed using faster higher-level simulations. Intermediate simulation data is usually exchanged at abstraction boundaries by data aggregation and mapping to the respective higher- or lower-level inputs [JAC+13, HBPW14, CCCW16]. Thus, compared to full simulations at the lowest abstraction, multi-level simulations can provide a trade-off in terms of speed and accuracy and make certain simulation scenarios even feasible at all [JAC+13, HKBG+09, CCCW16].

Applications

High-throughput timing-accurate circuit- and fault simulation is helpful to accelerate applications in a variety of areas, which heavily rely on extensive simulations.

Process variability and timing variation: With the increasing sensitivity of circuits towards process- and parameter variation, the need for statistical evaluations for performance and reliability assessment rises. Under variation, the circuit timing is severely altered [BCSS08, GK09] which tampers with the effectiveness of tests [PBH+11]. By including parameter dependencies in the delay processing, such as supply voltage and ambient temperature [DJA+08, SAKF08], or process variations [ABZ03, ZHHO04], statistical

timing analyses [WW13] and statistical fault coverage analysis [SHM+05, CIJ+12] can be realized by large-scale Monte-Carlo evaluations in highly parallel timing simulation.

Non-functional properties: The timing-accurate switching histories from the simulation can be utilized to analyze *non-functional properties* in a design. Both, thermal design power [HGV+06] and IR-drop from switching activity hotspots [SBJ+03, MSB+15, DED17a] impact the reliability of designs and tests [WYM+05, ZWH+18] and thus need to be evaluated. Moreover, vulnerabilities with respect to single-event upsets [BKK+93] and aging effects, such as *bias temperature instability* (BTI) [GSR+14] or *hot-carrier injection* (HCI) [LGS09, ZKS+15], can be investigated using timing-accurate signal histories.

Power estimation: The static and dynamic power-consumption of a design can be estimated from the circuit signals and switching activity [GNW10]. Power consumption tampers with the functionality of the design not only during functional operation, but also during the application of tests [WYM+05, AWH+15]. While previous approaches relied on untimed logic simulation due to complexity reasons, the power of GPUs enable timing-accurate power estimation for many patterns in parallel. This way more-accurate power-aware test schemes and test pattern selection can be developed [HSW+16, HSK+17, ZWH+18].

Scan test: Timing-accurate simulation is necessary to validate delay tests for small delay defects and marginalities, i.e., *faster than at-speed test* (FAST) [HIK+14, KKL+18]. However, in scan test each test pattern undergoes a long session of consecutive shift cycles before the test can actually be applied and evaluation [GNW10]. During the shift-phase, IR-drop-induced shift-errors from delayed signals [YWK+11] as well as skewed clocks [Sho86] can cause errors during shift-in and shift-out phases of the test pattern application. Again, with the high-throughput computing capabilities of GPUs, parallelization can be utilized to accurately evaluate full test sets with millions of shift cycles concurrently.

Goal of this Thesis and Organization

The document at hand presents highly parallel simulation models and simulation algorithms for fast and scalable timing simulation of nano-electronic digital integrated CMOS circuits on *graphics processing units* (GPUs).

The main contributions of this thesis are summarized in Fig. 1.1. Compact functional as well as timing-accurate temporal modeling are combined with highly parallelized simulation algorithms designed to run on GPU-architectures to exploit their massive computing throughput. These building blocks provide the foundation for developing high-throughput simulation algorithms for timing-accurate logic level and switch level simulation with parallel execution on the GPUs.

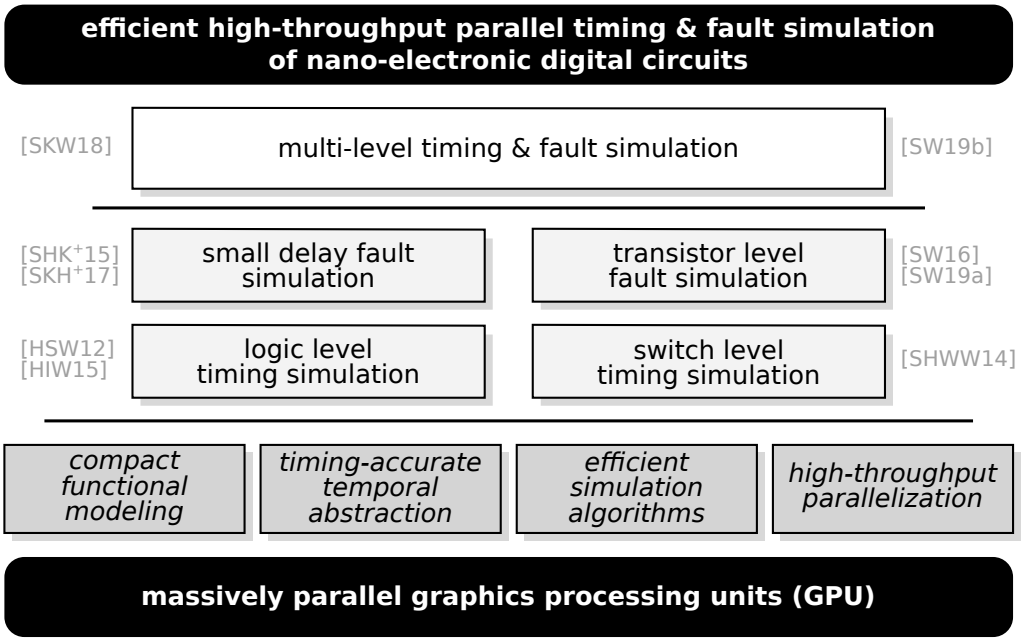


Figure 1.1: Contributions of this work.

The presented simulation models and algorithms are further extended for fault simulation of small delay faults at logic- as well as low-level parametric and parasitic faults at switch level. These allow to fully exploit the massive parallelism found in circuit and fault simulation on the GPU-architectures.

Finally, on top of these the extended simulation models and algorithms are ultimately combined to form an efficient multi-level fault simulation flow that enables additional simulation throughput and simulation efficiency for the lower level fault simulation through the use of mixed abstractions during simulation.

By simultaneously exploiting different dimensions of simulation parallelism, the presented parallel simulation techniques achieve unprecedented simulation throughput on GPUs

thereby enabling scalable timing and fault simulation of multi-million gate designs, which effectively reduces simulation time from several hours or even days to minutes only.
The structure of this document is organized into three parts:
The first part (Chapter 2–4) introduces background on circuit- and fault modeling as well as simulation. Furthermore, the part briefly summarizes the key aspects of data-parallel many-core computing architecture of *graphics processing units* (GPU) and their programming paradigm, along with an overview of state-of-the-art GPU-accelerated simulation techniques.
The second part of this document handles the high-throughput time- and fault simulation, which is the main contribution of the thesis and is composed of the following chapters:

Chapter 5 ("*Parallel Logic Level Time Simulation on GPUs*") – presents the first timing-accurate and variation-aware logic level time simulation on GPUs and outlines the underlying algorithms, kernels and general data structures for multi-dimensional parallelization with high simulation throughput.

Chapter 6 ("*Parallel Switch Level Time Simulation on GPUs*") – introduces the first parallel switch level time simulation for GPUs which utilizes first-order electrical parameters found in CMOS circuits for a more accurate modeling of the functional and timing behavior of CMOS standard cells.

Chapter 7 ("*Waveform-Accurate Fault Simulation on GPUs*") – describes the extension of the logic- and switch level simulation algorithm for fault modeling, syndrome computation and parallelization for first-time detailed and comprehensive parallel simulation of timing-related faults.

Chapter 8 ("*Multi-Level Simulation on GPUs*") – addresses a combination of the presented parallel logic- and switch level algorithms in the first GPU-accelerated multi-level simulator that effectively enhances the performance and efficiency of the lower-level timing and fault simulation.

The third and last part (Chapter 9–13) discusses the evaluation of the aforementioned presented simulation methodologies as well as example applications in five chapters.
Finally, the last chapter provides a summary of all the contributions of this thesis and outlines directions for future work and possible extensions.

Part I

Background

Chapter 2

Fundamental Background

This chapter summarizes the necessary background and fundamentals of circuit- and fault modeling as well as simulation of digital integrated CMOS circuits as found in basic literature [Wun91, BA04, WWX06, WH11]. First, background on standard CMOS technology is provided, followed by circuit- and delay fault modeling on different levels of abstraction. Finally, simulation algorithms and multi-level simulation methods are briefly discussed.

2.1 CMOS Digital Circuits

Today's semiconductor manufacturing processes of digital integrated circuits mainly rely on *complementary metal-oxide semiconductor* (CMOS) technology. CMOS uses two types of *metal-oxide-semiconductor field-effect transistor* (MOSFET) devices, namely NMOS and PMOS transistors, that can basically be considered as voltage-controlled switches. NMOS and PMOS transistors are utilized to form *standard cells* in *standard cell libraries* [Nan10, Syn11] which are the foundation of contemporary semiconductor design. For more details on the structure and mechanics of transistors as well as the manufacturing processes and design of standard cells, the reader is referred to [WH11].

In modern semiconductor process technology nodes, conventional planar MOSFETs are being replaced by the recent *fin-type field-effect transistor* (FinFET) technology [HLK+00, Kin05, BC13]. The FinFET devices allow for scalable manufacturing of feature sizes beyond 20nm, since they have a more compact layout with a faster switching behavior compared to traditional transistors [YMO15]. This enables the production of more complex

designs with higher transistor density that can be further run at higher clock frequencies to achieve even higher device performance. Although the physical structure of planar MOSFET and FinFET is different, their basic working principles and design flows are similar.

2.1.1 CMOS Switching and Time Behavior

In today's nanometer regime, it is often of utmost importance that the designs meet a certain performance requirement that is given by a minimum system clock frequency. The highest frequency a design can reliably run with is usually determined by the length of the *critical path*. The critical path accumulates all the propagation delays of standard cells and interconnects on the longest sensitizable path from a primary or pseudo-primary input to a primary or pseudo-primary output in the design.

In general, the delays of a standard cell are distinguished by *propagation delays* and *rise/-fall times* as shown in Fig. 2.1, which depicts the typical input and output waveform of a inverter cell in a small circuit (15nm FinFET technology [Nan14, BD15] powered with a supply voltage of 0.8V). The maximum signal amplitude on the left axis has been normalized. As shown, the signal switching processes do not occur instantly, but occur merely as a function of the voltage over time as a result of parametric and parasitic electrical elements in the circuit. The change of the voltage of a signal is typically modeled by differential equations that describe electrical charging and discharging processes of capacitances in the circuit [WH11].

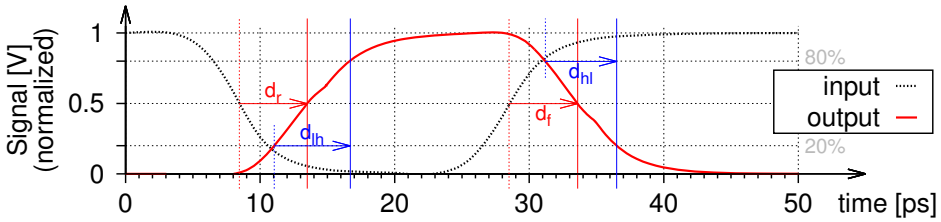


Figure 2.1: *Rising/falling propagation delay* (d_r, d_f) and *rise/fall times* or *slopes* (d_{lh},d_{hl}) of an INV_X1 CMOS inverter cell [Nan14, BD15]. (Adapted from [WH11])

The *propagation delay* is defined for *rising* (low-to-high) and *falling* (high-to-low) output transition polarities (d_r and d_f) and describes the time it takes for a signal transition at a cell input to propagate to the cell output. It is measured from the time point where the input signal reaches the 50% mark of the maximum signal amplitude (i.e., $0.5 \cdot$ VDD

as halfway between VDD and GND potential) to the point where the output crosses 50% of the maximum signal amplitude in response [WH11]. Moreover, as shown in Fig. 2.1, signal transitions are not instantaneous, but follow a certain continuous *transition ramp* or *slope* that are given by the *rise-time* d_{lh} for *rising* and the *fall-time* d_{hl} for *falling* transitions as well. The rise-time (and fall time) of a signal transition is defined as the time it takes to transition from 20% to 80% of the maximum signal amplitude (and vice versa), though these thresholds can vary throughout the literature (e.g., 10%–90%) [WH11].

With the shrinking manufacturing process technologies and increasing complexity, it is important to accurately estimate the possible timing of contemporary and future designs. Thus, suitable models are required to reflect and consider CMOS-related timing effects as early and as accurately as possible during the design phase.

One approach to estimate the delay of a circuit is the RC delay model [WH11], which approximates the non-linear characteristics of transistors considering average resistances and capacitances of the circuit nodes. In the *RC delay model*, the circuit netlist is transformed into an electrical equivalent RC-circuit where all transistors and interconnects are replaced by simple resistors and the interconnect- and gate capacitances in the fanouts are replaced by (in the simplest case) lumped capacitances. A switching transistor causes a change in the state of the RC-model elements by changing the corresponding resistance. This triggers a transient at the output that is typically computed using the first-order or second-order *step response* [Elm48, WH11]. Hence, in case of the rising transient of Fig. 2.1 the step response v_{rise} beginning at some time t_0 can be modeled for the time $t > t_0$ after as

$$v_{rise}(t) := (\text{GND} - \text{VDD}) \cdot e^{\frac{-(t-t_0)}{\tau}} + \text{VDD}, \tag{2.1}$$

where τ is the *time constant* computed as $\tau := R \cdot C$, with R being the effective resistance of the driving PMOS transistor and C being the load capacitance. The propagation delay d_r is then derived from the output signal v_{rise} at the time it crosses $0.5 \cdot \text{VDD}$, which is computed as $d_r := \tau \cdot \log 2$ [WH11]. Similarly, the falling transient v_{fall} beginning at some time t_1 is calculated for $t > t_1$ as

$$v_{fall}(t) := (\text{VDD} - \text{GND}) \cdot e^{\frac{-(t-t_1)}{\tau}} + \text{GND}, \tag{2.2}$$

in which case the resistance R for the time constant $\tau := R \cdot C$ corresponds to the effective resistance of the conducting NMOS transistor, resulting in an estimated falling propagation delay of $d_f := \tau \cdot \log 2$. Both of the modeled rising and falling transients expressed by v_{rise} and v_{fall} are illustrated in Fig. 2.2. The axes have been normalized by τ for the time and by VDD for the voltage, respectively. For the sake of simplicity, it was assumed that the internal resistances of NMOS and PMOS were equal.

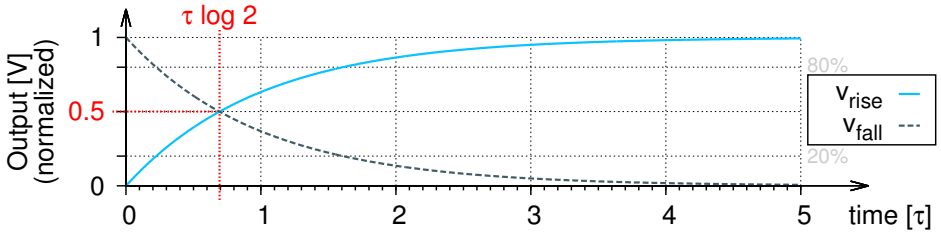


Figure 2.2: First-order step response for rising and falling transient modeling the continuous charging/discharging process of an output load capacitance. (Adapted from [WH11].)

2.1.2 Circuit Performance and Reliability Issues

With the shrinking feature sizes in newer technologies, the semiconductor manufacturing processes get more complex and increasingly prone to process variations and defects [SSB05, PBH+11]. Since the structures, such as fins of FinFET transistors, have a size of only a few nanometers, spot-defects by particles or statistical processes in the different manufacturing steps, such as dopant fluctuations, are getting more and more frequent and influential. These problems can severely alter the physical layout of the circuit structures and lead to improper connections in the design. Moreover, transistor-related parameters, such as the threshold voltage and electron mobility, can shift due to improper manufacturing. The resulting defects, such as open and shorts in either fins or gates [LX12], as well as variation artifacts [HTZ15] primarily impact the timing of the circuit [TTC+15, FGRC17] and ultimately affect the device performance and reliability by introducing delay faults.
Certain parameter shifts in circuit structures exhibit small delays that sometimes indicate *circuit aging* or the presence of a marginal device [KCK+10]. For example, when a design is put under stress (i.e., high switching activity or mechanical stress) after deployment, structures in the circuit can fail due to device degradation and transistor *aging* [BM09]. These wear-out mechanisms appear gradually as a result of continuing static and dynamic stress

in the circuit, i.e., by certain stimuli or circuit activity, as well as environmental conditions, such as temperature. Typical degradation mechanisms are *Negative-Bias Temperature Instability* (NBTI) [LGS09, GBRC09], *Hot-Carrier Injection* (HCI) [EFB81, GSR+14], *Time-Dependent Dielectric Breakdown* (TDDB) [DGB+98] and *electromigration* (EM) [Cle01]. The effects of the degradation range from changes in electrical properties of the transistor devices and interconnects [LQB08] that result in delay faults [BM09] up to hard failures, such as voids or bridges due to transported metal atoms from EM. Imminent wear-out failures can be predicted using specialized hardware (e.g., *aging monitors* [APZM07, LSK+18]), once the circuit has sufficiently *aged* or degraded. Wear-out can also be delayed or mitigated by reducing the stress patterns in the circuit to prolong the system lifetime [ZBK+13].

Failures in the early-life phase of a design are related to infant mortality among circuits, where marginal devices have passed the conventional manufacturing test, but soon break after initial deployment. These failures are called *early-life failures* (ELFs). Marginal devices are usually tested using so-called *Burn-In* tests [VH99], that strongly exercise the circuit under extreme stress conditions to expose the ELF. The indicators of ELFs can be located deep inside the circuitry which can be barely noticeable as small delay deviations that are much smaller than the clock period and not testable under at-speed conditions [KCK+10]. This lead to the recent development of *Faster-than-At-Speed Test* (FAST) [HIK+14, KKS+15] to explicitly test for these *hidden delay faults* (HDF) to identify the marginal devices.

Therefore, to assess and validate the timing of today's designs as well as to thoroughly test for timing-related defects, designers and testers have to rely on timing-accurate models of the circuits and simulation algorithms for test and diagnosis [TPC11].

2.2 Circuit Modeling

Today's multi-billion transistor circuit designs are typically not manufactured starting from scratch using transistors, but rather starting from high-level specifications that describe its behavior in an abstract manner [Wun91]. As shown in Fig. 2.3, the circuit design undergoes different representations at various levels following a V-shaped model. During

the top-down design phase, abstract high-level descriptions are refined step by step by synthesis tools that systematically add more and more implementation details by mapping to lower level structures [ABF90, BA04, WWX06]. For example, at the highest level of abstraction, only the desired function is specified and the actual design itself is treated as a black box since no physical implementation details are present. At this level, modeling and simulation is fast and simple, but the accuracy is the lowest. When moving down towards the lower abstraction levels, the required gates to realize the specified function or even the layout of complete transistor netlists with physical and electrical properties are determined. This drastically increases the modeling complexity and, of course, the simulation effort, but ultimately leads to more accurate models and evaluations.

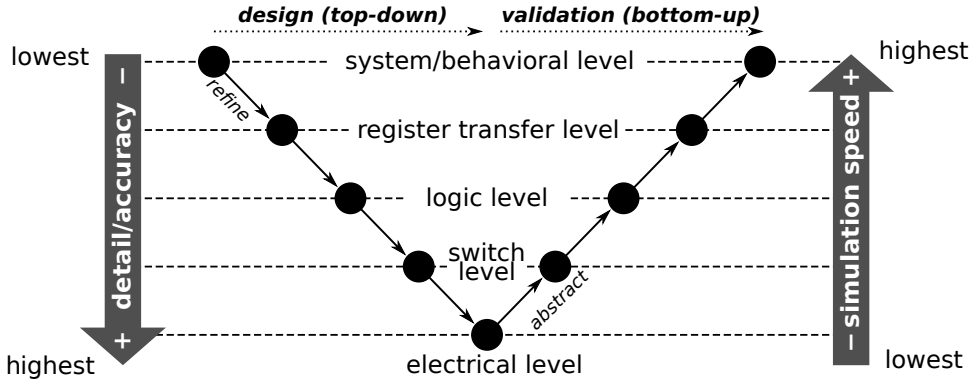


Figure 2.3: Overview of abstraction levels of a circuit in the design and validation.

2.2.1 Electrical Level

The electrical level provides the highest accuracy among the commonly used simulation models as it reflects the physical properties and behavior based on layout. The circuit is modeled using netlists of electrical components and devices, such as transistor models, resistors, capacitors, inductors and voltage sources. Corresponding simulators usually consider geometric information of the physical layout to reflect spatial dependencies of power-grid structures and capacitances between neighboring interconnection lines. Internal node voltages and currents are typically calculated through compute-intensive nodal analyses and differential equations using numerical integration methods. This way, highly waveform-accurate transient analyses by electrical simulations can be provided which also allow to predict the signal integrity as well as the power consumption in a circuit [WH11].

Modeling and simulation at the electrical level is typically performed using SPICE [NP73], which has established as industry golden standard. Common standard cell libraries [Nan10, Syn11] already provide SPICE models of the implemented standard cells, which utilize transistor device model cards that can include hundreds of parameters [DPA+17, Nan17]. However, the high detail and accuracy of the electrical level modeling comes at the cost of vast runtime complexity and memory requirements.

2.2.2 Switch Level

Switch level modeling abstracts the electrical level behavior of the circuit by simplification and linearization of the transistor switching behavior [Bry84, Bry87, Hay87, Kao92]. The circuit is represented as an interconnected network of transistors and first-order electrical components, i.e., resistors and capacitors, as shown in Fig. 2.4. Transistors are treated as simple three-terminal devices that form discrete ideal switches, where the input at the gate terminal controls the connection between drain and current terminal of the device. The resistors in the model reflect the static functional behavior, and the capacitors (sometimes referred to as *wells*) model the temporal behavior [Hay87]. Most switch level models distinguish between signal strengths and consider bi-directional signal flows as well as modeling of charges inside of the networks.

At switch level, signal values are typically represented as pairs $\langle s, v \rangle$ of *signal strength* $s \in \mathbb{N}$ and *signal value* v. The signal values are typically expressed in *ternary* (three-valued) logic over the set $\boldsymbol{E}_3 = \{0, 1, X\}$ [Hay86, BAS97] to distinguish between *low* (0), *high* (1) and *undefined* (X) signals. The *signal strength* reflects the driving strength of the signal source, which is used to resolve multiple drivers at a node or consider bidirectional signal

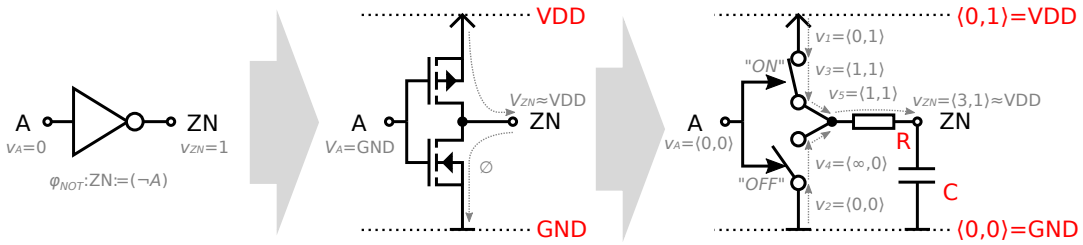


Figure 2.4: Inverter cell with CMOS implementation and switch level representation, where signal values are encoded as pairs $\langle s_i, v_i \rangle$ of strength s_i and logic value v_i [Hay87].

flows as well as stored charges in the network. Signals of strength $s = 0$ are the strongest signals and typically correspond to VDD or GND sources, whereas higher values of s indicate weaker signals. Stronger signal values always override weaker ones. Signals of the same strength, but with different values, result in an *undefined* value.

For the purpose of efficient switch level simulation of digital designs, a common practice is to partition the transistor netlist of circuits and CMOS cells into smaller sub-networks composed of *ideal* PMOS and NMOS transistor meshes, forming a *channel-connected component* (CCC) [Bry87, DOR87, HVC97, BBH+88, BAS97, CCCW16]. The transistors within a CCC are interconnected via their drain and source terminals to each other and connect *power supply* VDD and *ground* GND sources and allow for a bidirectional signal flow within the network. The outputs of a CCC are linked to an output domain that can lead to gate terminals of transistors in succeeding CCCs. However, it is assumed that no current can flow over the gate from one CCC into another.

Switch level evaluation algorithms are mainly based on nodal analyses and dependencies that are formulated using linear equations. These equations are solved using relaxation techniques with *sparse matrix vector products* (SMVP) to first find a steady state solution of the node voltages and the node charges inside of the network after each input switch [Bry87]. In combination with timing analysis, node voltages are more accurately represented as a function of time, usually modeled by piece-wise approximations [CGK75, Ter83, Kao92]. In general, switch level models are less accurate than electrical level models, but they are able to reflect many important properties of CMOS technology which are not reflected in classical models based on two-valued Boolean or ternary pseudo-Boolean logic. Due to the higher speed of switch level simulation, it has replaced analog SPICE simulation in recent cell-aware test generation [CCCW16].

2.2.3 Logic Level

The logic level is the most dominant abstraction used in design and test validation of nanoelectronic digital circuits [WWX06, BA04]. At logic level the circuit behavior is described as *netlist* of interconnected Boolean gates. The netlist of a circuit is typically modeled as a directed *graph* $G := (V, E)$. Each vertex $v \in V$ corresponds to a *node* in the netlist, which corresponds to a Boolean gate, flip-flop, or circuit input or output, and basically

represents a functional abstraction of a physical standard cell that determines the logical behavior. The edges $(v, v') \in E \subseteq V \times V$ represent ideal interconnections from a source node $v \in V$ to a sink node $v' \in V$ in the netlist and allow for signal flows. Incoming edges at a node refer to input pins at the corresponding gate, while outgoing edges indicate the information transport from the node output pin to its successors.

Each edge $e \in E$ is associated with a signal value that reflects the logical interpretation of the *voltage level* of the corresponding signal, which are typically defined over two-valued Boolean logic $\boldsymbol{B}_2 = \{0, 1\}$ [Hay86]. The logic values in $\boldsymbol{B}_2$ correspond to discrete signal states, where the logic symbol '0' corresponds to *low* voltage (i.e., ground voltage potential GND) and the symbol '1' corresponds to a *high* voltage (i.e., power supply voltage potential VDD). Multi-valued logic, such as *ternary*, four-valued, or even higher-order logic can be utilized to reflect additional states (e.g., *unknown* due to uncertainties) and dynamic behavior (e.g., *transitions*) of signals [Hay86]. The functional behavior of a node with k incoming edges is modeled by a Boolean function $\phi : \boldsymbol{B}_2^k \rightarrow \boldsymbol{B}_2$ that maps the values of the edges to an output value according to the function.

At logic level, various delay models can be considered to describe the temporal behavior of a gate output signal with respect to input changes. Apart from *zero-delay* modeling, which reflects the untimed simulation case, where input changes at time t can cause a simultaneous change at the output, timing-aware delay models evolve around *unit-* and *multiple-delay* representations to reflect propagation delays for gates (sometimes referred to as transport delays) [ABF90]. Fig. 2.5 provides an overview of the timing-aware delay models on the example of a small inverter gate [Wun91].

In *unit delay* it is assumed that all gates have a constant propagation delay $d := 1$ that corresponds to one time unit. For input transitions at time $t \in \mathbb{N}$, this can cause output transitions to occur at time $t' := (t+d) = (t+1)$ in response. However, different gate types typically have different delays. Moreover, gates usually have different delays for rising and falling output transitions as well. *Multiple-delay* models reflect different transport delays d_r for *rising* and d_f for *falling* output transitions, which are applied accordingly.

Some delay models are able to consider propagation delays that are bounded by a *min-max* interval $[d_{min}, d_{max}] \subset \mathbb{R}$ [Wun91, BGA07]. In interval-based delay modeling, the actual delay of a gate is not known exactly, but it is estimated to be in the interval $d \in$

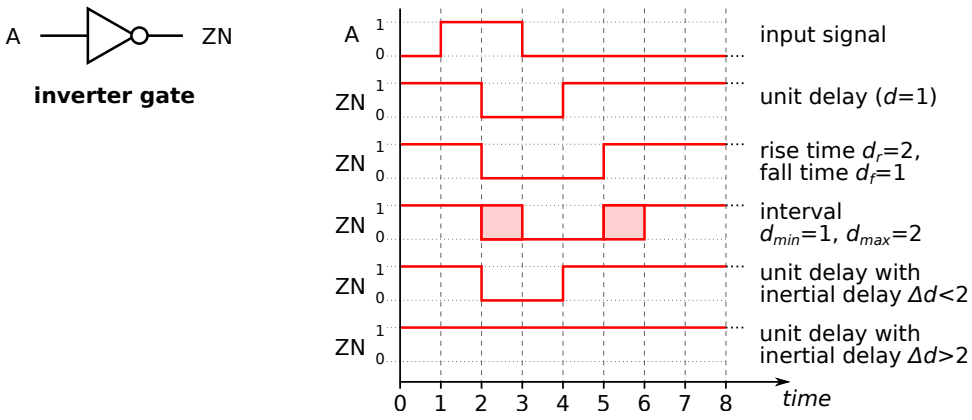


Figure 2.5: Overview of common (timing-aware) delay models in logic level simulation. (Adapted from [Wun91])

$[d_{min}, d_{max}]$. This way, process variations and uncertainties that impact gate delays can be reflected. However, during simulation with such a model the uncertainty of different gates can quickly accumulate and lead to pessimistic simulation results [ABF90].

When the value at a gate output switches, the load capacitance associated with the standard cell implementation at the electrical level is (dis-) charged, which requires some time until the transition is finished. In case a pulse at a gate input is too short, the output does not have sufficient time to charge to the targeted potential, which can prevent the propagation of the pulse, such that the output value is sustained. A so-called *inertial delay* can describe the *minimum pulse width* $\Delta d \in \mathbb{R}$ required to perform a full (dis-) charge of the output. For example, in Fig. 2.5, signal A has two consecutive pulses at times $t_1 := 1$ and $t_2 := 3$, which would lead to transitions at times $t'_1 := 2$ and $t'_2 := 3$ under unit delay. In the last case, the required minimum pulse width is $\Delta d > 2$, therefore the pulse is filtered out since the pulse width given by $(t'_2 - t'_1) = 2$ does not meet the requirement.

Individual delays for each input pin of a gate can be considered in combination with rising and falling distinction. So-called *pin-to-pin* delay models keep a delay value d^i for each input pin i of a gate. An input transition at pin i at time t then propagates to the output at time $t' := (t + d^i)$ with the corresponding delay associated to the pin.

The logic level timing information is usually provided as *standard delay format* (SDF) files [IEE01a] which are obtained during the synthesis of the design. Each SDF file typically contain absolute and incremental delay specifications describing, specific delays for individual ports (PORT), the propagation delay from an input to an output pin (IOPATH) as

well as wire delays (INTERCONNECT) between gate instances. In addition to the propagation delays, *inertial delays* can be specified (via PULSEPATH) that describe the minimum allowed pulse-widths at outputs for the application of pulse filtering [IEE01a, IEE01b].

2.2.4 Register-Transfer Level

At the *register-transfer-level* (RTL) the circuit module descriptions are mapped to a coarser structural description composed of small interconnected components [Wun91]. The components can be registers or memory to store data and operands, and combinational networks to perform operations. Interconnections and buses define the data-flow between the individual components. The data exchange between memory elements is controlled in a synchronized fashion by clock signals. RTL-descriptions are typically expressed as data-flow in hardware-description languages, such as Verilog [IEE01b] or VHDL [IEE09]. While timing can be expressed in cycle granularity, fine-grained propagation delays are not considered at this level, since no implementation details are available.

2.2.5 System-/Behavioral Level

At *system level* the circuits are modeled with the highest abstraction. Abstract functional descriptions of circuit modules are typically expressed as algorithms and functions in higher level programming languages, such as SystemC [IEE12], which are compiled and run as separate processes. Since no details or requirements with respect to the actual implementation of the design is implied, a reasoning about accurate timing is not possible.

2.3 Delay Faults

In test validation and diagnosis, *fault models* are used which abstract the effect of classes of physical defects in a chip for representation and processing on a higher abstraction level [Wun91]. As opposed to classical static fault models, such as stuck-at and bridging faults, a circuit cannot be tested for *delay faults* by using a single test pattern only [WLRI87]. Instead, delay tests with a minimum of two patterns composed of an *initialization* and *propagation test vector* are required. These patterns are applied in consecutive clock cycles to launch signal transitions at (pseudo-)primary inputs. The output responses are then

captured after the clock period has passed, which might reveal output signals that are not yet stabilized.

A generalized way to model arbitrary faults in logic simulation is the so-called *conditional line flip* (CLF) calculus [Wun10]. Each CLF has an activation condition *cond*, which is an arbitrary user-defined Boolean formula to determine the activation of the fault at a signal v. Once a CLF is active (condition *cond* is *true*), it flips (inverts) the value of v to $v' := v \oplus [cond]$. As a simple example: A stuck-at-0 fault is represented in the CLF calculus as $v \oplus [v]$ and a stuck-at-1 fault is represented as $v \oplus [\overline{v}]$.

Transition Faults

The *transition fault* model [WLRI87] assumes a lumped defect at a gate input or output pin that causes a delay at the associated signal line by an amount of time larger than the test clock period (usually infinite), thereby omitting a transition. Transition faults can be distinguished as *slow-to-rise* (STR) or *slow-to-fall* (STF) faults that affect *rising* and *falling* transition polarities, respectively. They are usually considered as *conditional stuck-at* faults, which can be efficiently implemented and simulated using parallel-pattern stuck-at fault simulation by activating the fault upon signal changes of subsequent patterns [Wun10, WLRI87]. Therefore, regarding the modeling and simulation, transition faults are independent of the actual circuit timing and gate delays.

For example, let $v_i \in \boldsymbol{B}_2$ be the value at a signal line after application of the initialization vector, and let $v_{i+1} \in \boldsymbol{B}_2$ be the value after the subsequent propagation vector. The fault activation at a signal line transitioning from v_i to v_{i+1} upon the clock cycle is determined by $active \Leftrightarrow (v_i \oplus v_{i+1}) \cdot \rho$, where the term $(v_i \oplus v_{i+1})$ indicates a change in the signal value and ρ determines the activation of a STR ($\rho := v_{i+1}$) or STF ($\rho := \overline{v}_{i+1}$) fault, respectively. Note that transition faults always affect all sensitized paths in the output cone of the fault site regardless of the actual timing. Therefore, the fault model fails to accurately represent delay faults of smaller quantities, e.g., small resistive open defects.

Path-Delay Faults

The *path-delay fault* model [Smi85] was proposed to model distributed delay effects in the circuit and assumes the accumulation of smaller lumped delays during signal propagation.

In this model, a *path* of nodes $\{i, n_1, n_2, ..., n_j, o\} \subseteq V$ in the netlist graph $G := (V, E)$ from a (pseudo-)primary input $i \in I$ to a (pseudo-)primary output $o \in O$ of the netlist is assumed to suffer from a path-delay fault if the accumulated delay along the associated path exceeds the clock period. The fault is activated, iff the input signal transition is propagated along all nodes of the path to the output. As a result, the signal at the single affected output of the fault is expected to be erroneous. Like transition faults, path-delay faults can be distinguished as STR and STF.

Similarly, the *segment delay fault* model was proposed [HPA96, MAJP00] to model distributed delays along sub-paths and hence combines transition fault and path-delay fault behavior. A segment delay fault is modeled on a path segment of l connected gates $\{n_1, n_2, ..., n_l\} \in V$. If a signal transition enters the segment at node n_1 and propagates through all of its gates, the fault is activated at the segment output node n_l from where it behaves as a lumped transition fault.

The simulation of path-delay faults does not require timing information and can thus be performed in untimed logic simulation or by sensitization analysis [PR08, Wun10]. ATPG-tools for path-delay faults typically constrain the set of relevant paths by topological analyses with timing information to identify longest sensitizable paths through each node [ED12]. By doing so, even small lumped delays at gates can be tested for, whose fault size is larger than the slack of the tested path. However, not all faults can be tested robustly [Lin87, Kon00, ED11] and longest path delays can shift due to process variations [CIJ+12]. Thus, certain faults with smaller delays might be missed, requiring more accurate modeling and simulation of the faults.

Small (Gate-)Delay Faults

A *small (gate) delay* fault is considered as a manifestation at an input or output pin of a node, that slows down the signal propagation through the respective pin by a small finite amount of time [IT88, TPC11]. These faults are mainly caused by weak resistive opens in the design [RMdGV02] and, in contrast to *transition faults* or *path-delay faults*, the delay impact of a *small gate delay fault* is much smaller than the clock period [NG00].

Fig. 2.6 illustrates the behavior of a small delay fault at a gate output in a circuit. After application of a delay test vector, transitions at the circuit inputs are launched which

propagate through the input cone of the fault. Eventually, the fault site is reached and the output transition at the fault site is delayed by the fault size. In the left example (a), the delayed output response is propagated through the output cone of the circuit over sensitized paths to three outputs. However, in the output response vector only two of these outputs (2 and 3) show an error as the signals are captured before the last transition happens. Despite the additional delay due to the fault, the latest signal transition at the first output still arrives in time and a good response value is captured. If the small delay fault was simulated using a simplified transition fault model, all of the outputs of sensitized propagation paths would show an erroneous response. Hence, transition faults typically overestimate the fault coverage of small delay faults with smaller sizes and they are also too optimistic regarding the fault propagation and detection at outputs [IRW90].

In the right example (b) of Fig. 2.6, a non-robust propagation of a small delay fault in presence of reconvergence is illustrated. The reconvergent fault propagation causes a delayed hazard at the output and the fault can only be detected within a small detection window spanned by the hazard. This behavior is not captured by simple transition fault simulation as the corresponding STF-transition fault (corresponding to a stuck-at-1 [WLRI87]) would lead to a constant-1 output signal with no detection window during simulation.

Also, the CLF calculus [Wun10] is not suitable for expressing small delay faults in a practical way, due to the complexity of the temporal modeling. In general, for the fault activation a CLF assumes a single point in the circuit where it is conditionally activated and from which the faulty value is distributed throughout the sensitized output paths. Hence, similar to the transition delay faults, all sets of sensitized paths are affected, similar to the transition delay faults, which does not hold for general small delay faults. One way to fully cover a small delay fault behavior in the CLF calculus would be by introducing multiple-CLF faults in the fan-out of the actual small delay fault site (e.g., CLFs injected at reachable circuit outputs). Then, for each injected CLF all possible combinations of related activation paths and propagation paths need to be encoded in the conditions (not including false paths) to fully capture the timing violations that can occur for the small delay fault. However, this introduces a tremendeous modeling complexity.

In [PB07], so-called segment-network faults were introduced which consider multiple segment delay faults [HPA96, MAJP00] as a (sub-) tree in the netlist starting from a common

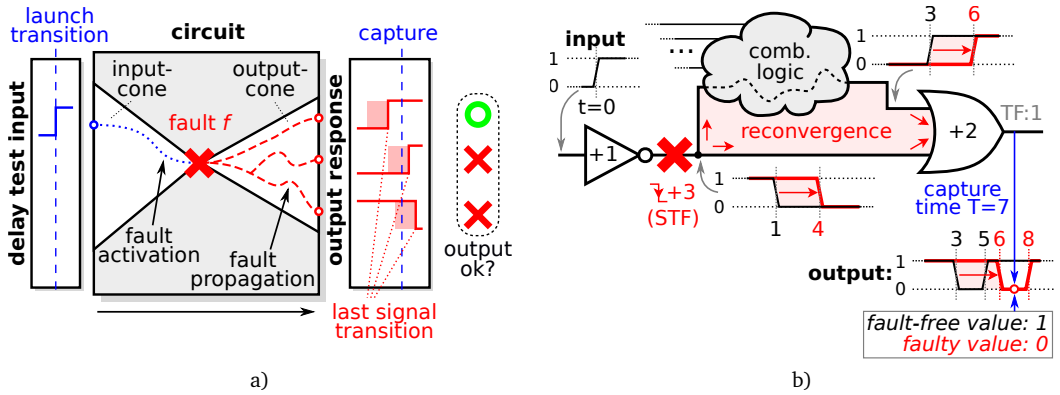


Figure 2.6: Examples of a small gate delay fault in a circuit: a) robust fault-propagation to outputs over sensitized paths and b) non-robust propagation with output hazard.

root node. If a signal is propagated from the root node along a segment, a fault is activated at the corresponding leaf node where the segment ends. While the activation results only from a single root node, the fault modeling provides more control in the propagation of the faults. No hazards are considered in the modeling which can invalidate the detection. Therefore, in order to simulate smallest delay faults for test validation explicit timing-accurate and glitch-aware simulation is required [CHE+08, BGA07].

Impact of Process Variations on the Fault Detection

Process variations cause inaccuracies during manufacturing, which can severely affect the functional and timing behavior of a circuit [Nas01, PBH+11]. The source of variation is typically distinguished as either of *random* or *systematic* nature. The background of *random variation* usually has a quantum mechanical origin. Both, layout and material properties of the circuits are influenced by statistical processes and numerous uncertainties during chip manufacturing and affect electrical properties of the underlying structures. Random variation can influence the switching behavior of gates and interconnects in a circuit (*intra-die*), for example, by *line-edge roughness* (LER) artifacts, and can also increase the threshold voltage of transistors due to *random dopant fluctuation* (RDF). At logic level, this type of variation is typically modeled by assuming *independent random variables* as delays for each gate in a circuit [SSB05, BCSS08]. On the other hand, *systematic variation* takes into account the spatial and parametric dependencies of different process corners

between dies (*inter-die*), wafers (*wafer-to-wafer*) or lots (*lot-to-lot*). This type of variation can affect gates with high spatial correlation or gates of the same type in a similar manner simultaneously [ABZ03, ZHHO04]. The sources of systematic variation relate to irregular material properties as well as limitations of the fabrication processes themselves (e.g., *lithography*, *etching*, *polishing*) that introduce a correlation between neighboring structures or their corresponding nodes in the design [SSB05, ABZ03].

These process variations can have severe impact on the gate delays as well as the circuit timing and ultimately affect the detection of delay faults [CIJ+12, SPI+14]. Therefore, simulation algorithms must be timing-accurate and variation-aware to allow for modeling of the impact of process variations on the delay for statistical timing analyses, variation-aware fault grading and pattern generation [CIJ+12, ZKAH13, SPI+14].

2.4 Circuit and Fault Simulation

A simple way to simulate a circuit is by *compiled-code simulation* [WWX06, BA04]. In compiled-code simulation the combinational netlist is translated into executable instructions for each node with all signal states being kept as variables. The circuit nodes are then evaluated by executing the assigned operations over their input variables. For a successful evaluation of each node, all of its input variables need to be determined first. Hence, the processing of the nodes usually follows a topological order to allow for a hassle-free evaluation, which is obtained by a so-called *levelization* pre-process that partitions the nodes into *levels* ordered by increasing topological distance (depth of the nodes) with respect to the circuit inputs [Wun91].

A basic simulation flow for the simulation of a test pattern $tp \in \boldsymbol{B}_2^{|I|}$ is shown in Algorithm 2.1. During the process, first the input nodes $I \subset V$ are assigned their corresponding values of the input pattern, followed by the ordered computation of the internal node signals until the outputs are reached. Due to the levelization, all input-dependencies of the nodes have been resolved. This type of simulation is typically referred to as *oblivious simulation*, since always every node of the circuit is evaluated upon the application of a new test pattern [BBC94].

Algorithm 2.1: Simulation flow of a plain oblivious logic simulation.

Input: netlist $G = (V, E)$, test pattern tp with $tp[i]$ value of input $i \in I$
Output: values v_n for all $n \in V$

```
1 foreach node n ∈ V in topological order do
2     if n ∈ I then
3         Assign input value v_n := tp[n].
4     else
5         Fetch values v_1, v_2, ..., v_k of fanin(n) (direct predecessors of n).
6         Compute v_n := φ_n(v_1, v_2, ..., v_k).
7     end
8 end
9 return Stored values v_n of all nodes n ∈ V.
```

2.4.1 Event-driven Simulation

Usually, when applying consecutive test patterns to a circuit, not all of the primary or pseudo primary inputs of the circuits change and hence certain signals sustain their state. With the oblivious simulation scheme, this causes a lot of unnecessary node evaluations since these nodes do not require recomputation, yet all nodes are always evaluated regardless of switching activity from signal changes [BA04, WWX06]. To provide a more efficient evaluation in cases of little switching activity, *event-driven simulation* approaches have been proposed [Wun91]. In event-driven simulation the evaluations are constrained to nodes with active switching *events* at their inputs. Thus, the evaluation only follows the path of events during simulation and thereby avoids (unnecessary) evaluation of nodes with constant signals.

Traditional time simulators typically follow a synchronous event-driven *time-wheel* approach [Ulr69], which as proven well for simulation at logic level. A different simulation approach for *asynchronous* event-driven simulation can be realized using the *Chandy-Misra-Bryant* (CMB) algorithm [CM79, Bry77]. As opposed to a global synchronous time schedule, the CMB algorithm assigns a time stamp to each event and utilizes message passing to distribute events from node outputs to input FIFOs of successor nodes. The evaluation of events at different nodes can be realized by individual processes concurrently, which can benefit from parallelization to provide speedup for simulating single circuit instances [BMC96].

In event-driven approaches, the handling of events can get quite complex which quickly increases the runtime overhead when considering more detailed delay models [Wun91].

For example, in inertial delay modeling, many events scheduled during processing might need to be reverted when processing later events in time. Also, the algorithms usually only speed up simulation of single circuit instances by exploiting parallelism from independent gates. They can not benefit from pattern parallelism [BBC94] through simultaneous evaluating multiple patterns in a data-parallel fashion, as they rely on sparse occurrences of events. Since, gate level parallelism can diminish at deeper levels, this strongly limits the simulation throughput and effectiveness of these approaches. Moreover, for the implementation on GPUs these algorithms demand for highly complex control- and dynamic memory management to process all the event lists in the circuit. However, frequent memory operations of the scheduling are expensive and will limit the effectiveness of an acceleration on GPUs [OHL+08].

2.4.2 Fault Simulation

In fault simulation, a circuit is simulated under the behavior of a given defect to determine whether the circuit behavior is altered by the fault [Wun91]. A naïve approach to simulate faults is through serial processing of the provided fault lists. In serial fault simulation, first a simulation of the circuit is performed to obtain the *fault-free* good-value simulation results for a test pattern. The resulting output responses v_o of all circuit (pseudo-)primary outputs $o \in O$ are considered as golden reference, or expected values of the fault-free circuit, which are stored for comparison. The good-value simulation is usually followed by repeated simulations of the pattern for various copies of faulty circuits in which different faults have been *injected*. The *injection* of a fault is performed by modifying the circuit description according to the abstracted fault behavior.
After simulation of a faulty circuit copy, the output responses of the faulty circuit v'_o are compared against the golden reference v_o to compute an output *syndrome* syn_o by

$$syn_o := v_o \oplus v'_o, \tag{2.3}$$

where $\oplus$ corresponds to the bitwise XOR-operation of the response bits of the outputs $o \in O$ in good and faulty response. The output syndrome contains all the differences in the faulty and the fault-free output response and thus indicates the fault detection of the

applied test pattern. A fault f is considered as *detected* at an output $o \in O$, iff $syn_o = 1$ and therefore detected by the pattern. If $\forall o \in O : syn_o = 0$, then f is *undetected*.

Fault simulation is a challenging and compute-intensive task (i.e., especially when done exhaustively for all possible faults of a circuit) due to the additional dimension of complexity from the set of different faults. Often a process called *fault dropping* is applied, in which the simulation of a test pattern set for a fault is stopped as soon the fault has been detected by a certain number n of test patterns (n-detect). The fault is then immediately removed from the fault list and the simulation is continued for the next. While this reduces the overhead of fault simulation, fault dropping in simulation cannot be applied for debug an diagnosis. Especially in diagnosis, often so-called *fault-dictionaries* must be computed which rely on exhaustive simulation of all faults for all patterns to obtain as much syndrome information as possible.

Since the number of faults usually grows with the design size and technology, the increase of the fault simulation performance has been subject of research in test ever since [WEF+85, WLRI87, AKM+88]. Different ways of improving the performance and speeding up fault simulations exist, that exploit parallelism from pattern-parallel simulation and fault-parallel simulation and other optimizations, such as concurrent or deductive fault simulation. For more information the reader is referred to general literature of semiconductor design and test [Wun91, WWX06].

2.5 Multi-Level Simulation

In design and test validation tasks it is often necessary to evaluate parts of the design with higher accuracy. Although high accuracy is generally desirable for any task, the continuing simulation of a circuit at the lowest level at all times can be troublesome, due to the increasing order of magnitude of runtime complexity on lower levels [SN90].

In *multi-level simulation* a design is simulated on more than one level of abstraction at the same time. This is achieved by utilizing mixed abstractions throughout the design during simulation and focusing the accurate and compute-intensive simulations to smaller parts of the design which allows to reduce the overall runtime. Usually, the design is partitioned, such that common parts are evaluated with fast high-level simula-

tion, while *critical* components are processed at the lower abstraction levels with higher accuracy [RK08, KZB+10, SKR10, JAC+13, HBPW14]. Simulators with mixed abstractions usually utilize hierarchical circuit descriptions with the design being represented individually at each desired level of abstraction [SSB89, RGA87]. Those representations are then selected interchangeably for the components during simulation based on the desired accuracy, i.e., for the injection of low-level faults in mixed-level fault simulation [GMS88, MC93, MC95]. By doing so, designers working on individual modules of a circuit can focus the accuracy of the validation on their respective designs without the need of processing the whole circuit at the lowest abstraction, which eventually results in a much faster and more efficient simulation.

2.6 Summary

With increasing technology scaling and shrinking feature sizes, modern circuits become increasingly complex in terms of design size and also prone to newer defect types that often manifest in the timing behavior of the circuit (e.g., small delays). Hence, timing-accurate simulation has become a necessity for design and test validation. However, timing-accurate simulation itself is tremendously complex in terms of runtime, and classical approaches to tackle the afore-mentioned problems either lack accuracy (due to abstraction) or scalability (due to simulation effort).

In order to make accurate and large-scale design and test validation feasible, the simulation algorithms have to be parallelized in a way to increase the throughput performance for coping with an increasing number of gates, test patterns and faults in the designs. This thesis investigates the parallelization of timing simulation on massively parallel *graphics processing unit* (GPU) architectures and provides timing-accurate algorithms for high-throughput simulation as well as multi-level approaches to further enhance the simulation efficiency.

Chapter 3

Graphics Processing Units (GPUs) – Architecture and Programming Paradigm

This chapter introduces the basic concepts of contemporary data-parallel *graphics processing unit* (GPU) accelerators. Both the general GPU-architecture as well as the underlying *parallel programming paradigm* will be outlined. For better comprehensibility, the NVIDIA's *Compute Unified Device Architecture* (CUDA) [NVI18b] programming model and the NVIDIA Pascal GP100 GPU architecture [NVI17c] are used as an example. The GP100 GPU is manufactured in a 16 nm FinFET technology from TSMC featuring 15.3 Billion transistors on a die of 610 mm^2 size.

3.1 GPU Architecture

In recent years, general purpose programming on *graphics processing units* (GPUs) has established well in the context of high-performance computing (HPC). Being used as co-processors, GPUs allow to accelerate computations of many compute-intensive scientific applications by exploiting massive parallelism [NVI18c, ND10, OHL+08]. While conventional CPUs typically provide fewer (usually 8 to 24 [STKS17]), but more general and latency-optimized processing cores, contemporary GPU accelerators consist of thousands of simpler cores that provide high computational throughput due to massively parallel execution of thousands to millions of threads. The execution of threads on the cores is performed in a *single-instruction-multiple-data* (SIMD) fashion, in which the threads each perform the same instruction at a time, but on different data in parallel [HP12].

Fig. 3.1 depicts a block diagram of the NVIDIA GP100 GPU of the recent *Pascal* architecture [NVI17c]. The GPU provides 60 *streaming multiprocessors* (SM) which are partitioned into six *Graphics Processing Clusters* (GPCs) that are further sub-divided into five *Texture Processing Clusters* (TPCs) each. Each SM consists of 64 *single-precision* (SP) and 32 *double-precision* (DP) compute cores summing up to a total of 3,840 single-precision or 1,920 double precision cores. The GPC and TPC [NVI10] provide all key units for graphics processing (e.g., for vertex, geometry and texture processing), however, the actual rendering capabilities will not be used within the context of this work.

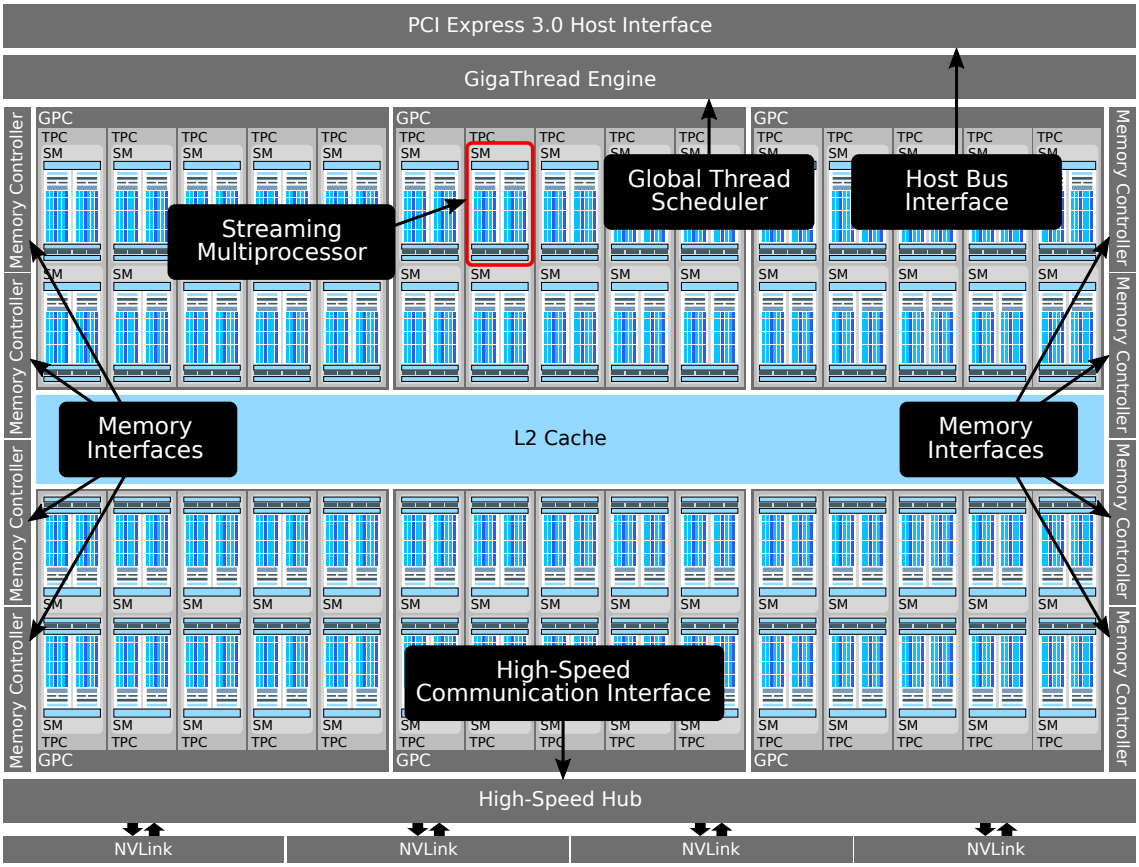


Figure 3.1: Block diagram of the full NVIDIA GP100 Pascal GPU Architecture. (Adapted from [NVI17c])

The SMs have access to a 4,096 KB Level-2 (L2) cache which is connected via eight 512-bit memory controllers to a *High Bandwidth Memory 2* (HBM2) DRAM global device memory providing a memory bandwidth of up to 720 GB/s. A global thread scheduler is responsible for the distribution of workload threads to the available SMs for processing. The

communication with the host system is established via a PCI Express 3.0 interface. In case multiple GPU devices are present in the host, a high-speed hub allows for high-bandwidth communication between the devices via a proprietary interface (NVLink).

3.1.1 Streaming Multiprocessors

The *Streaming Multiprocessors* (SMs) are responsible for performing all the computations and workload processing on the GPU. Each SM is composed of an array of simple compute cores for executing arithmetic operations (*CUDA cores*). In the GP100, each SM provides 64 single-precision (SP) along with 32 double-precision (DP) processing cores as shown in Fig. 3.2. In addition, the SM also contains several *load/store units* (LD/ST) for calculation of memory addresses and *special function units* (SFU) for executing transcendental arithmetic functions, such as *sin* or *square root* [NVI17c].

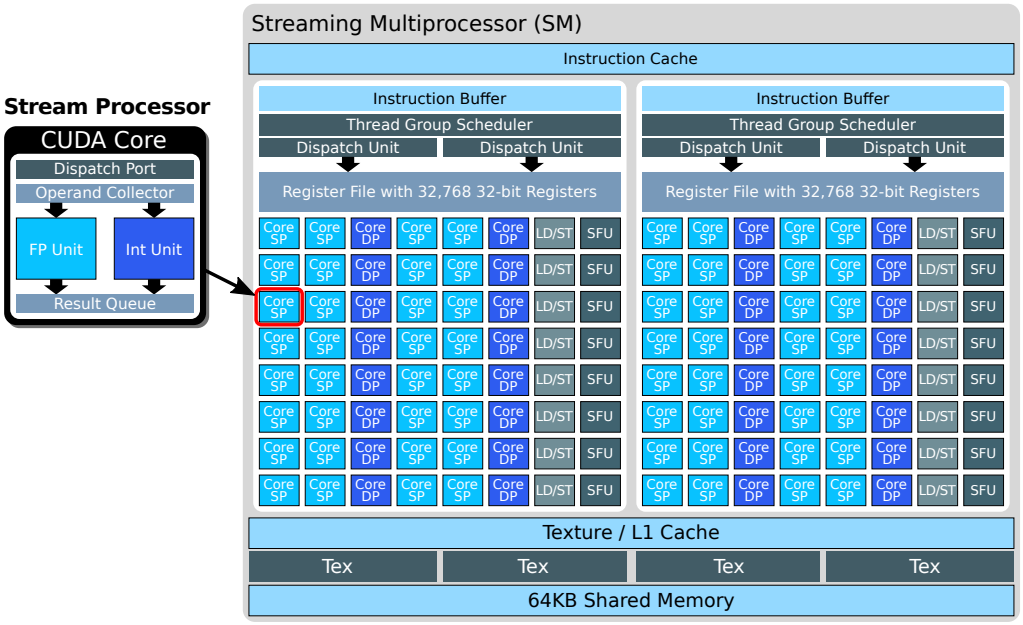


Figure 3.2: Block diagram of a NVIDIA Pascal GP100 Streaming Multiprocessor (SM) with CUDA core. (Adapted from [NVI17c, NVI10])

Each SM is further divided into two processing blocks each of which contains a register file as fast local memory, an instruction buffer of current instructions, a scheduler for scheduling threads as SIMD-compliant *thread groups*, and two dispatch units that can select individual instructions for the selected thread group to be executed in the CUDA

cores. A CUDA core in an SM has access to a pipe-lined floating-point (FP) processing unit compliant with the IEEE-754 floating-point standard [IEE08] as well as an integer (Int) processing unit with their respective precision (32-bit for single-precision cores and 64-bit for double-precision cores). Upon receiving instructions via the built-in dispatch port, the CUDA cores fetch the required operands from the register file for execution. Besides various basic and special arithmetic instructions, including a fast *fused-multiply-add* (FMA) [IEE08, NVI10], the CUDA cores can also execute various bit-wise Boolean operators. The set of available functions and features are dictated by the architectural version or *compute capability* of the SMs, which can vary for different GPU architectures.

3.1.2 Thread Scheduling

With the large amount of computing resources, GPUs can achieve a high computational performance by massively data-parallel execution of *threads* on the available cores. The control and the scheduling of all threads is completely handled by the GPU architecture itself.

The GPU reads host-CPU commands via the PCI Express host interface and copies data and commands to the global memory. The global thread scheduler organizes the threads as *thread blocks* and dispatches the blocks throughout the GPU to the different SMs. The number of threads that can be scheduled on a SM per block depends on the amount of available memory resources, such as local registers and shared memory, as well as the amount of resources required by each thread. For the execution, the SMs partition the thread blocks further into *thread groups* (also called *warps*) each of which is composed of a set of up to 32 threads for parallel execution.

The SMs are responsible for scheduling the thread groups in the assigned thread blocks (up to 32 thread blocks per SM [NVI18e]), as well as the distribution of the threads to the CUDA cores and execution units. While the instructions of the threads of a thread group are executed in a SIMD fashion in parallel, the different thread groups are selected and executed alternately on the multi-processor in different clock cycles. In the GP100 architecture, up to 64 thread groups can be active at a time and each SM-block can issue a maximum of two independent instructions per clock to the cores via the dispatch units. Hence, up to four instructions can be issued in total per SM at a time.

The integrated scheduler utilizes scheduling logic for optimizing the thread execution, such as score-boarding and inter-group (inter-warp) scheduling to identify instructions that are ready to use, as well as scheduling at thread block level by the global thread scheduler. Active thread groups can be suspended when waiting for long-latency memory operations, such that idle thread groups can be selected from the remaining threads for further processing. In the underlying SIMD execution scheme, the threads of an active thread group always execute identical instructions. However, conditional branches in the execution flow can cause some threads of a group to diverge. The resulting branches of diverging threads are executed serially on the SM. During execution of a branch, only the associated threads are active while the other threads in the group are disabled. Eventually, the execution flows converge after the different branches have been processed.

3.1.3 Memory Hierarchy

GPU devices typically provide a full memory hierarchy from fast, scarce local storage to large, slower global DRAM memory, having several caches in between as shown in Fig. 3.3. On the highest level, the GPU architectures provide a register file which is directly addressable by a thread running on a core for the use as private thread-local memory. In the GP100 architecture the register file contains 65,536 32-bit registers per SM, which are equally divided among all active threads running in a thread block.

On the next level of the hierarchy, each SM contains a shared memory and also a combined texture and Level-1 (L1) cache [NVI18b]. The shared memory is a low-latency programmable data storage comprising 64 KB in total. It is accessible by all threads of

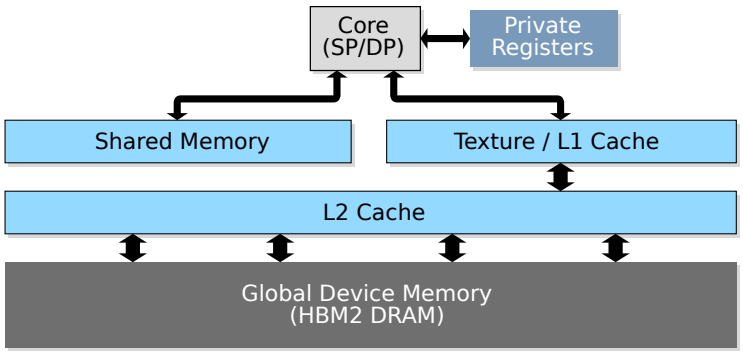


Figure 3.3: GP100 memory hierarchy and thread access. (Adapted from [NVI17c, NVI10])

the thread blocks scheduled in the SM. Within a thread block, the threads can utilize the shared memory for communication and synchronization, as well as for sharing or exchanging data. In the GP100 architecture, a thread block can access a maximum of 48 KB of shared memory [NVI18e]. The combined texture/L1-cache in the SM is utilized for global loads caching up to 24 KB of data and instructions. While the texture memory is for fast cached read-only use, the L1-cache manages and gathers data requests from the threads in the thread groups and also holds thread-local data.

A larger L2-cache on the GPU provides 4,068 KB of storage which is shared among all SMs. It contains the cached accesses of the threads to local and global memory. Each cache line maps to an aligned segment in the global device memory [NVI18b].

The global device memory resides on the lowest level in the memory hierarchy and typically comprises several gigabytes. Data and instructions can be copied between the host system over the host interface into the global memory on the device. Once threads begin to access the memory, the requested data is handed through the memory hierarchy. Data exchange between host and devices is rather expensive compared to the execution of bare arithmetic instructions, hence, memory transfers should be minimized and used scarcely in order to sustain the computing performance [OHL+08, ND10]. The GP100 architecture provides four HBM2 stacked DRAM memory dies with a total capacity of 16 GB [NVI17c]. The register files, shared memories, L1- and L2-caches, and the global device memory are protected by a Single-Error Correct Double-Error Detect (SECDED) ECC-code [NVI18e].

3.2 Programming Paradigm

Contemporary GPU-based systems follow a heterogeneous programming model, where the host CPU assign computing tasks to individual GPUs for computation and acceleration. For efficient parallelization of a program, it has to be partitioned such that independent problem workloads can be distributed over all the available computing resources on the GPUs. The communication among the processors, the management and synchronization of processes and threads, as well as the management of the memory accesses have to be organized in a way, such that full utilization of the computing resources can be achieved and the computational throughput is maximized. With the introduction of GPUs for gen-

eral purpose programming, the era of the many-core processing has begun, which called for a novel programming paradigm to allow the efficient management and handling of many thousands to millions of threads on the GPU devices. This section briefly explains the general GPU programming paradigm on the example of the *Compute Unified Device Architecture* (CUDA) [NVI17a, NVI18b] which was originally launched by NVIDIA in the year 2006. CUDA provides a parallel computing platform and corresponding programming paradigm with a variety of programming constructs that allow for the organization of threads, the synchronization of executions and the management of memory accesses to the GPU device. It comprises extensions of the C/C++ programming language for the implementation of parallel programs (*kernels*) which can be executed on the GPUs.

Fig. 3.4 illustrates a typical execution flow of a C/C++ CUDA program for execution on a heterogeneous compute system with a GPU. As shown, the program structure forms an alternating sequence of sequential code blocks executed on the host CPU and also code blocks that called by the CPU but that are executed in parallel on the GPU. In the following, the organization of threads in kernels as well as the management of memory accesses within threads are explained. Although the CUDA platform is proprietary, the basic principles are similar to other standards, such as OpenCL [Khr17]. More information on the CUDA programming can be found in [NVI18b] and [NVI18a].

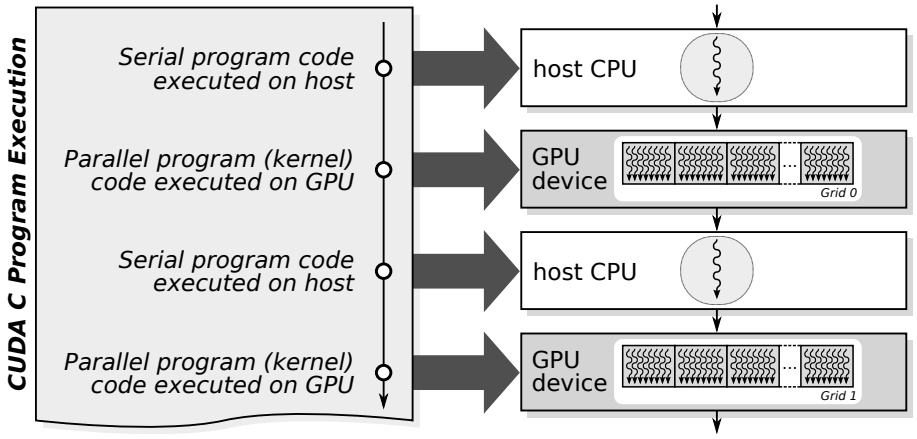


Figure 3.4: Serial execution flow of a heterogeneous program with parallel GPU kernels. (Adapted from [NVI18b])

3.2.1 Thread Organization

In the GPU execution model, a *thread* represents the basic unit of execution in parallel programs running on the GPU processing cores. Each thread of a parallel kernel works on a specific problem instance and has a unique thread index, which is accessible via an in-built *threadIdx* variable. The index variable itself is a three-dimensional vector, which is used to arrange the threads for a structured execution of the parallel programs. For example, two-dimensional arrangements of threads can be implemented to perform matrix operations, where each thread handles the computation of a specific element.

All threads of a program are organized as *thread blocks*, which are one-, two- or three-dimensional arrays of threads of a predetermined size. On the GPU, the threads of a thread block are all processed on the same SM, while each SM can process a different block. The threads within a thread block all have consecutive indices and can communicate via shared memory and synchronize their execution, as well as accesses to the global memory. According to the underlying programming paradigm, all threads are required to work independently of each other, since independent threads allow for an arbitrary order of execution providing better opportunities for scheduling of the threads on the SMs.

On the highest level of the execution of a parallel kernel in CUDA, thread blocks are arranged in a *thread grid*. Thread grids represent one- to three-dimensional arrays of thread blocks of a given size (and corresponding threads) which are distributed to the SMs on the GPU by the global thread scheduler. Similar to the threads, each block has a unique block index (*blockIdx*) to identify its coordinate in the grid. When a kernel function is called for execution on the GPU, the dimension of the thread grid (in number of thread blocks) as well as the dimensions of the thread blocks (in number of threads) must be explicitly stated (in the form of <<<$\#blocks, \#threadsPerBlock$>>> as part of the CUDA C/C++ syntax). At any time during the computation, a thread has access to its own thread index (x, y, z) within the block, the coordinate of its block (x_B, y_B, z_B) in the thread grid, as well as the size of the blocks (B_x, B_y, B_z). This information can be utilized by a thread during computation to determine its respective global coordinate $(x + B_x \cdot x_B, y + B_y \cdot y_B, z + B_z \cdot z_B)$ in the thread grid. This way, the actual problem instance of each thread can be identified which is required to generate problem-specific input data or to compute global memory addresses. Since the available computing and memory

resources are shared by all threads of the active running thread blocks, the dimensions of thread blocks and the grid need to be chosen appropriately and input/output data has to be carefully aligned in memory to fully utilize the processing cores.

3.2.2 Memory Access

The different memories in the GPU memory hierarchy have a different scope and serve different purposes. Fig. 3.5 illustrates the thread organization and the memory hierarchy of the CUDA architecture. On top of the hierarchy, there is the thread-local private memory sector. This comprises all the *local registers* in the SM required by the thread for execution. Registers are allocated by default and the amount of registers required per thread is determined at compile time. The amount of assignable registers per SM and per thread is usually limited (up to maximum of 255 per thread in GP100). However, additional local memory can be obtained by *register spilling* in the device DRAM, which is served through high latency L1- or L2-cache accesses. The lifetime of the private memory sector is on a per-thread basis and is freed once the execution of a thread is finished.

The *shared memory* is a low-latency memory residing in each SM that is be explicitly instantiated by applying a "__*shared*__" qualifier to the targeted variables. Variables declared as *shared* can only be accessed by threads of a thread block and only for as long as the respective block is executed on the SM.

The *global device memory* is allocated by the host and is accessible by all threads of a kernel (e.g., by a "__*device*__" qualifier) through the L1- and L2-cache hierarchy. Its data persists on the device during the whole application context and can be accessed by multiple kernels in succession for as long as the CUDA application is running.

The caches also implement coalescing buffers that are able to merge and combine memory accesses of threads which allows for better utilization of memory transactions. By appropriately aligning data in the global device memory, parallel memory accesses to subsequent addresses by threads of a thread group can be coalesced. The *coalescing* bundles the subsequent addresses to address ranges of the size of memory transactions to the global memory. This way, global device memory requests of the threads can be served efficiently with less memory transactions due to better utilization. The alignment of memory access patterns and the coalescing of the thread accesses is crucial, as misaligned or

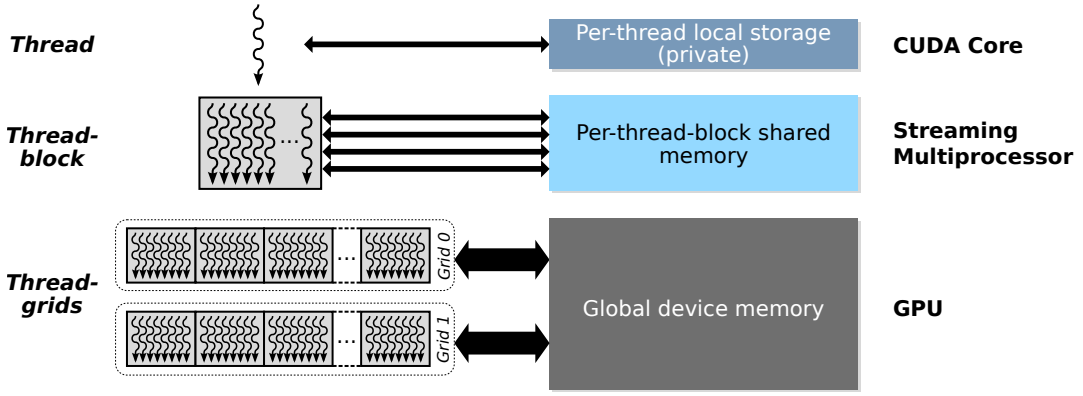


Figure 3.5: NVIDIA CUDA thread organization and accesses through the memory hierarchy on the hardware. (Adapted from [NVI18b])

arbitrary accesses by threads can cause a severe drop in the memory bandwidth and hence the compute performance [NVI18a]. In general, accesses to the global device memory are much more expensive compared to the execution of bare arithmetic instructions. Frequent accesses by the host system (either reads or writes) to larger portions strongly limit the computing performance and should be used as little as possible. More information on the memory management and thread organization can be found in [NVI18b] and [NVI18a].

3.3 Summary

This section introduced the concepts of modern GPUs and their parallel programming paradigm on the example of the NVIDIA CUDA platform [NVI17a]. GPUs provide powerful computing capabilities for parallelized programs by vast computational throughput through concurrent execution of thousands to millions of lightweight threads. This enables high performance computing with massive speedups for many scientific applications [NVI18c]. Due to the SIMD execution scheme, uniform control flows of the kernels, data-independent execution of the threads and regular and aligned memory access patterns are important to achieve optimal performance. Care has to be taken when designing algorithms due to the limited available memory resources and the harsh restrictions of the underlying parallel programming paradigm. Modern compute servers utilize special interfaces for better integration and support of GPUs [FD17, STKS17], allowing to reach peak performances in the petaFLOP range on a single compute node [NVI18d].

Chapter 4

Parallel Circuit and Fault Simulation on GPUs

This chapter provides an overview of state-of-the-art parallelized simulation approaches for the execution on contemporary *graphic processing unit* (GPU) architectures.

4.1 Overview

The simulation of circuits and faults is a time-consuming task and generally poses a serious bottleneck in design and test validation flows, as well as diagnosis tasks. In the past researchers have sought for ways to accelerate simulation and found that circuit simulation is inherently parallelizable on multi-core multi-node architectures when partitioned into independent problems. Common independent problems are related to *structure* (from gates and faults, etc.), and *data* (from patterns, parameters, etc.). However, classical parallel approaches involved high communication and synchronization overhead, that caused the effectiveness of the parallelization to diminish which limited the scope of applications. With the computing power of GPUs [OHL+08, ND10] parallelization has reached a new level and has drawn attention of researchers and developers [DM08, DWM09, Den10, GK10b, CKG13]. The GPUs and their programming paradigm allowed to fully exploit the inherent parallelism for massive acceleration while being cost-effective. For simulations on these architectures, the circuit models must be sufficiently accurate and compact to allow all data structures to fit on the devices. Furthermore, the algorithms need to exploit the available computing resources on the GPU as well as their capabilities to enable an efficient and scalable parallelization applicable to increasing simulation problem sizes.

Hence, maximizing the computational throughput on the GPU architectures is the key for scalable massively parallel simulation and their applications [LTL18].
In the following, approaches and key techniques for GPU-accelerated circuit- and fault simulation are presented.

4.2 Parallel Circuit Simulation

SPICE [NP73] is often considered as golden standard for circuit modeling and simulation in industry and academia due to its accuracy. However, the accuracy comes at the expense of an extremely high runtime complexity which is usually *several* orders of magnitude higher compared to simulations at logic level [SJN94]. Thus, several researchers sought for parallelization methods to accelerate SPICE simulations and to reduce the runtime overhead.
A first approach was presented in [GCKS09] that accelerated simulations at electrical level in SPICE on GPUs. In this approach, the computationally intensive parts of the evaluations of SPICE, i.e., solving the linear equations systems, were off-loaded to the GPU. The size and functionality of the kernels were chosen to fit the hardware resources on the device, while the GPU memories were utilized as much as possible avoiding excessive host-to-device communication and thereby effectively accelerating the simulation. The technique was integrated for the use in a commercial SPICE simulation tool showing average speedups of 2.36×. However, despite the accuracy, a large-scale application for time simulation of many test patterns is not feasible due to the high runtimes.
A fully implemented accelerated SPICE simulation approach on GPUs was presented in [HZF13] and [HF16] that performs many SPICE simulations concurrently allowing for the evaluation of different parameters and input stimuli in parallel. The approach utilizes three-dimensional parametric look-up-tables to parameterize individual transistor devices, and implements an iterative solver for the SPICE simulation. Convergence checks of the solver are performed after a fixed number of iterations to prevent excessive thread-divergence. Although speedups of up to 264× have been reported, the applicability and scalability is limited to small netlists composed of a few transistors only, i.e., for the use in a Monte-Carlo simulation of single standard-cells.

The solving of linear equation systems from sparse matrices is an essential part of SPICE simulations and many optimizations have been proposed for GPU-accelerated solvers that utilize efficient LU-factorization. In [CRWY15] and [HTWS16] the authors show, for example, that right-looking approaches can exploit parallelism from vector columns, submatrices and vector operations simultaneously. The reported speedups reached up to two orders of magnitude compared to conventional solvers. A recent work [vSAH18] that accelerates SPICE can consider parameter variations in the transistor models for modeling aging effects. The simulation is based on CUSPICE, which is a GPU-accelerated version of *ngspice* [CUS19]. It achieved speedups of up to 218× on a recent GPU-architectures and was able to simulate designs with more than 200,000 transistors (128-bit multiplier) in 18.5 hours.

Note that even with speedups achieved, GPU-accelerated SPICE implementations still exhibit relatively high runtimes for small to medium-sized problem sizes. Current works either focus on speeding up single simulation instances [GCKS09, vSAH18] or parallelize over circuit instances [HF16] with no scalability in terms of design size. This is usually related to limitations in the GPU memory or increased communication and synchronization overhead, which limits the problem size and computational throughput, and hence the applicability to larger simulation problems.

4.3 Parallel Gate Level Simulation

In order to cope with larger designs with millions of cells, with even more faults and test patterns, simulators *must* exploit higher level modeling and simulation approaches that are able to exploit massive parallelism from many dimensions simultaneously and that provide high throughput in terms of solving many simulation problems. While GPU-accelerated simulators for RT-level [QD11, BFG12] and system level [NPJS10] exist, they are not suitable for timing-accurate simulation since they neither provide the capabilities to model circuit structure nor the accuracy to reflect timing accurately. Circuit functionality and timing is usually evaluated at (logic) gate level and various research has been conducted on parallelizing gate level simulation on GPUs.

4.3.1 Logic Simulation

[GK08] presented a first accelerated logic simulation in two-valued Boolean logic on GPUs which is also utilized for stuck-at fault simulation. It implements a forward simulation approach by using compact *look-up tables* (LUTs) to compute gate functions, which are stored in the cached read-only constant texture memory on the GPU. The simulation is performed in a levelized manner by calling an evaluation kernel for each level. Only the circuit data of the respective current level is required, thereby avoiding the need to store the entire netlist in the (limited) GPU memory. A compact encoding allows each thread to compute two gates simultaneously. The underlying algorithm exploits *structural parallelism* by concurrent threads for the gates on a level and *data-parallelism* from parallel simulation of patterns (*pattern-parallelism*) for each gate. This way, a speedup of over 238× in average was achieved over a commercial solution.

The evaluation of gates on levels in a levelized circuit by parallel threads is a common method to exploit structural parallelism in circuits. This has been adopted in many other publications [CDB09b, GK09, SRG+11] and this thesis as well.

In [CDB09b] a parallel cycle-based logic level simulator for GPUs was presented where the netlist is partitioned into *clusters* each of which computes the gates in the *cone of influence* of a circuit output. Each cluster is processed by an individual thread block, where the threads of a block concurrently process the gates of the cluster in levelized order. After each level, the threads of the block are synchronized. However, thread blocks can work independently of each other. Truth tables of gates as well as intermediate signal values are kept in the local shared memory, while inputs and outputs of the clusters are stored in the device memory. Independent evaluation of clusters by different thread blocks is achieved through duplication of the netlist gates that reside in the cone of influence of multiple outputs. Experimental results demonstrated a speedup of 14.4× over the sequential simulation of the algorithm. A similar approach using clustering-based method was proposed in [SABM10].

The first GPU-accelerated event-driven logic level simulator was presented in [CDB09a]. It uses a more fine-grained partitioning of the netlist into so-called *macro-gates*, each of which corresponds to a set of connected gates in the original netlist. A macro-gate is designed to be evaluated on a single streaming multiprocessor on the GPU and a sensitivity

list keeps track of any value changes at its inputs. In case the sensitivity list of a macro-gate contains an event, it is scheduled for execution on a multiprocessor with all threads processing the gates level by level. Speedups of 13× over a commercial approach were reported, although, duplication of gates is required for independent evaluation of macro-cells.

[SRG+11] proposes GPU-accelerated logic level simulation approach with a pipelined evaluation of the circuit where alternate memory locations are used for accessing different patterns. When the gates of a circuit level are simulated, the corresponding threads access the patterns in alternate order for consecutive processing thereby allowing to hide write-latencies when storing the output information to the GPU memory, before proceeding with the next level. To avoid loss of intermediate signal values during simulation, additional gates have to be introduced as placeholders that maintain the signal values, which causes a high overhead in gates during evaluation. The authors reported speedups of roughly 10× compared to a serial execution.

The structural independence of gates is an important factor for efficient parallelization of the simulation of a netlist. In order to achieve this, many algorithms rely on duplication of structures, which often introduces a large overhead. While data-structures for the gates are compact and optimized for fast access and execution, the afore-mentioned algorithms only consider information of the *functional behavior*.

4.3.2 Timing-aware Simulation

The consideration of time in circuit simulation requires the modeling of the temporal behavior for gates and signals. Depending on the timing model and abstraction, the timing information requires a large amount of data to be stored and processed on the GPU.

[WLHW14] presented a parallelized *static timing analysis* (STA) to compute the worst-case delay of a circuit on GPUs. The algorithm considers slopes of signals and computes propagation delays using a two-dimensional interpolation over look-up-tables (LUTs) with respect to input slope and output capacitance. Gates are processed in parallel for each level and each streaming multiprocessor on the GPU processes only gates of the same type in a data-parallel fashion. This way, the corresponding LUTs need to be fetched only once

and are cached for more efficient access. The reported speedup of the STA is 12.85× over a CPU-based implementation and three orders of magnitude over a commercial solution.

Other GPU-accelerated STA simulators have been proposed in [DM08] and [Den10] as well. While the first one is simply based on a *maximum* operation, the latter approach is based on *sparse matrix-vector products* (SMVP) showing speedups of 50×. Yet, STA simulators usually only reveal a worst-case timing information without taking switching activity from hazards and glitches into account or identification of *false paths* [MMGA19].

The authors of [GK09] presented the first GPU-accelerated Monte-Carlo-based *statistical static timing analysis* (SSTA) to estimate the delay deviations and yield of a design. It exploits structural-parallelism from data-independent gates on each level in the circuit, as well as *data-parallelism* from simulation of Monte-Carlo instances in parallel. For this, *parallel pseudo-random number generators* (PPRNG) are implemented, such that each thread can generate individual samples to compute the propagation delays of a gate. While Monte-Carlo-based SSTA usually is a compute-intensive task, the presented approach showed a significant boost in speed with an average speedup of 260× on a single GPU. However, similar to STA, SSTA only provides probabilistic worst-case information.

In [ZWD11, WZD10], an event-driven parallel logic level time simulation on GPUs was presented based on the parallel implementation of a message passing algorithm [CM79, Bry77]. In general, the simulator performs three steps to propagate events through the circuit each of which are handled by implemented kernels. First, event queues of signals are handled to input pins of gates, where the events are then processed in temporal order and stored in the respective output event queues of the gates. Individual threads are responsible for fetching the input event queues and assigning them to FIFOs at each gate input pin. Once assigned, the evaluation kernel processes the input events of each gate independently in temporal order by individual threads and writes the output signal in the gate output event queues. A memory paging mechanism is applied to manage the event queues in pages on the GPU that are dynamically swapped during simulation. Although complex dynamic memory management usually reduces the effectiveness of parallelization [BBC94, SG91], the achieved speedups were reported to be 47.4× in average. Yet, similar to other event-driven approaches the algorithm only accelerates the simulation of single circuit instances and it does not benefit from simulation of patterns in parallel.

In previous work [HSW12, HIW15] a novel and innovative logic level timing simulation on GPUs was presented, which aims for higher simulation throughput by simulation of patterns in parallel, rather than aiming for lower latency for single simulation instances. The algorithm computes timing-accurate switching histories (*waveforms*) for each gate output by processing input waveforms in a merge-sort fashion. To exploit both structural parallelism and pattern parallelism, the threads of the kernels are organized as two-dimensional grids of threads. Each thread of the grid computes the waveform of a distinct gate and a test pattern concurrently, which offers high simulation throughput for the timing-accurate evaluation of many patterns in parallel. Experimental results have shown speedups of two to three orders of magnitude thereby effectively reducing the runtime of logic level time simulation from several days to few minutes. Although the simulation is timing-accurate, the delay modeling only considers static pin-to-pin propagation delays, which is not sufficient anymore for today's nano-scale electronic devices.

4.4 Parallel Fault Simulation

Fault simulation is a straight-forward application of the underlying logic simulator itself by repeated simulation of the netlist with *modified* gates that have been injected faults. However, many approaches exploit optimizations, such as structural reasoning, to avoid exhaustive simulation of faults in repeated simulation runs [BA04].

[GK10a] presented a parallelized approach for the generation of fault dictionaries on GPUs. The approach is based on a PPSFP-based algorithm [LH91] and implements a parallelized *critical path tracing* (CPT) [AMM84]. Pattern-parallelism is exploited by encoding multiple patterns into words and performing bit-wise operations in threads, which is further enhanced by the execution of multiple threads in parallel. During CPT, the implemented algorithm traces back all gates of the netlist regardless of the input sensitivity in order to unify the control flow of the parallel threads. The final detection information of all patterns is then merged to a compact representation that indicates the detection of the faults using parallel reduction kernels [NVI18a]. This way, fault detection information of stuck-at faults is obtained for all simulated patterns without the need of fault dropping, which important for logic diagnosis. In the GPU device memory only the data of the cur-

rent gate to be processed is allocated which allows for simulation of larger designs, but in turn causes higher communication overhead with the host. Results on a single-GPU as well as on an eight-GPU setup have shown average speedups of 14.49× and 82.80×, respectively.

The authors of [KSWZ10] presented another parallel fault simulation for the generic (yet untimed) *conditional line flip* (CLF) fault model [Wun10] on GPUs, which implements an efficient PPSFP-based method [WEF+85, WWX06]. The algorithm first performs a pattern-parallel forward simulation to generate all good values followed by a backward-traversal where the fanout-free regions are evaluated with respect to their sensitivity and parallel fault propagation at the fanout stems. This way, a speedup of up to 16× over a serial event-driven implementation was achieved.

The authors of [LXH+10] proposed a stuck-at fault simulator that also determines the n-detectability. It implements a hybrid flow on the GPU with forward simulation to compute the detectability of faults at reconvergent fanout stems and applies backward CPT to reason about the detection of the remaining faults in the fanout-free regions [AMM84]. The netlist as well as the good and faulty responses of the simulation are stored in the global device memory. The netlist data is declared as constant to enable caching in the multi processors with the help of the texture memory for faster and more efficient access. During execution, thread indices map to specific gates in the netlist such that each thread can act independently without the need of additional host communication. The gates residing on a level are partitioned into types with each type being evaluated concurrently by the threads in a thread group. Furthermore, multiple patterns are processed in parallel by exploiting word- and additional thread-parallelism. All data is aligned in the global memory to excite coalescing of the accesses. Results showed average speedups of 25× compared to a commercial fault simulator.

In [LH10] and [LH11] a GPU-accelerated logic level simulator was developed that exploits up to three dimensions of parallelism from patterns, faults and thread blocks. Stuck-at faults are grouped into so-called *compact fault sets*, which are sets of faults with identical fanout region. For the simulation, the gates in each fanout region are indexed in topological order and processed sequentially by a thread block with each thread in the block processing a different pattern. While each thread block handles a compact fault set,

all faults contained in the set are processed in parallel. By processing multiple compact fault sets by different thread blocks, the simulation parallelism is increased on a thread-block level. A parallel *reduction kernel* [NVI18a] is used to efficiently merge detection information of the different test patterns to quickly determine the fault detection. The parallel reduction allows to perform operations over all values in a field, such as summation or bit-wise operations of all values, in logarithmic time as opposed to linear time with conventional approaches. Fault dropping is applied by terminating the simulation of the current compact fault set after all of its contained faults were detected. Experimental results showed average speedups of 150× compared to a sequential execution of the algorithm and 780× over a commercial solution [LH10].

In [AYY+14] a parallel path delay fault simulation approach for GPUs and conventional multi-core architectures with SIMD processing units is presented, which allows to efficiently determine all robustly and non-robustly detectable path delay faults [Smi85]. The general methodology identifies the robustness and non-robustness of each path segment in the circuit leading to a gate input or circuit output for several patterns. This information is stored in tables which are then combined to reason about the detection type of possible path delay faults. During the process, the simulator exploits pattern parallelism at bit-level by processing multiple values per word operation, as well as pattern parallelism at thread-level, where multiple threads handle the same gate, but for different sets of patterns concurrently, thereby obeying the SIMD paradigm. Experimental results showed a speedup of 47–68× over a commercial logic simulator. Yet, the approach considers neither actual timing of the circuit nor explicit simulation of path delay faults.

The authors of [BB17] proposed GPU-accelerated fault simulator for the TRAX (Transition-to-X) [BB12] fault model, which is a type of transition fault [WLRI87] that assumes signal transitions to result in a misbehaved or *undefined* value (X). Similar to transition faults, the TRAX faults require two-vector delay tests, each of which is evaluated by a thread in parallel. Signal values of gates are encoded as four-valued logic value pairs for initial and stable state. For the simulation, a good value simulation kernel first evaluates the circuit for all tests in parallel. During the process, each concurrent thread serially processes for a test pattern all the circuit gates in topological order, and the (intermediate) signals are stored aligned and in order of the gates in the GPU device memory. A fault activation

kernel then marks all gate locations with transitions in their corresponding value-pairs as *active* faults for the respective test patterns. Finally, a fault simulation kernel then dynamically performs pattern-parallel and fault-parallel simulation by mapping and scheduling only active faults and their corresponding detecting patterns to individual threads for simulation. Compared to a commercial simulator for transition faults, the speedups of the presented TRAX fault simulation approach achieved over 10× in average. Although the authors claim, that TRAX reflects *all possible transport-delay changes*, it provides no indication of the actual detection of a fault for a given sample time, since no timing data is involved.

For the fault detection, the previously introduced algorithms reason about the observation using sensitization conditions in the fanout-free regions but do not simulate all faults explicitly. Also, none of the afore-mentioned algorithms considers circuit timing and therefore cannot reflect timing-accurate signal propagation and fault modeling. While this is sufficient for simple fault models, such as stuck-at and transition faults, a fine-grained and detailed simulation of delay faults (i.e., small delay faults) requires *exact* simulation to reveal the actual switching times and possible hazards. Otherwise, important hazards and glitches are missed, which might invalidate the actual fault detection [Kon00, HS14]. Furthermore, many of the logic level simulators exploit pattern parallelism by encoding multiple patterns into machine words. Timing-accurate simulation cannot benefit from this parallelism, since for each pattern usually different signal transitions occur for which also different timing applies.

4.5 Summary

In general, the presented simulation techniques utilize the inherent parallelism found in general circuit- and fault simulation by careful partitioning of structure and data (e.g., gates, patterns, faults, geometry and instances) to individual threads for concurrent execution. For the efficient simulation, these approaches had to overcome common issues of GPU-parallelization:

Limited global device memory and local registers: All threads of a kernel share the same global device memory and within streaming multiprocessors all resources are dis-

tributed evenly to the scheduled threads. Compact data structures must be created to keep the working set and memory footprint of each thread small enough to fit thousands to millions of threads on the GPU.

Long latency global device memory access: Threads should work in cached local memory as much as possible, since accesses to the global device memory are time consuming. Data should be aligned in memory and accesses by the threads should follow regular patterns, such that coalescing of accesses can be exploited.

High thread synchronization overhead and thread divergence: Threads should work independently of each other at all times, since frequent synchronization causes parallelism to diminish. Kernels should be further implemented in a way such that branches in the execution flow are minimized to reduce thread divergence.

High memory transfer overhead between host CPU and GPU device: Copying memory between CPU and GPU is time-consuming, thus data transfer should be minimized or avoided whenever possible. Since global device memory persists for as long as an application is running, initialized memory and intermediate results should be reused in consecutive kernel calls. Furthermore, compacted data structures can help to reduce the size of transactions.

While many untimed higher-level simulation approaches exist, only few timing-aware approaches have been developed. The consideration of accurate timing causes a dramatical growth in simulation complexity [SJN94] when storing and processing timing annotations and signal histories, which can quickly exceed the memory and computing capabilities of GPU devices.

The core of this thesis closely follows the innovative parallelization concept of [HSW12, HIW15], since it meets the high throughput requirements for a fast and scalable timing-accurate simulation of nanoelectronic digital CMOS circuit designs. The concepts are generalized and extended for the application to switch level, fault simulation and eventually for multi-level fault simulation.

Part II

High-Throughput Time and Fault Simulation

Chapter 5

Parallel Logic Level Time Simulation on GPUs

This chapter presents the basic time simulation model for high-throughput logic level time simulation on *graphics processing units* (GPUs) based on [HSW12, HIW15]. It provides the basic circuit modeling, along with the modeling of signals and time and consideration of variations. The general simulation flow is outlined and the underlying algorithms and parallel simulation kernels are presented. Furthermore, the high-throughput parallelization schemes used throughout this thesis are explained which simultaneously exploit parallelism from circuit nodes, test stimuli and parameter variations.

5.1 Overview

The complete flow of the GPU-accelerated logic level time simulation is shown in Fig. 5.1. In general, the flow is composed of mainly two phases: An initialization (1–2) and a simulation phase (3–5). During initialization, the node descriptions are set up by extracting and levelizing the combinational network of the netlist (1). The descriptions are back-annotated with timing information (2) as obtained from *standard delay format* [IEE01a, IEE01b]. In the simulation phase, the input stimuli from pattern files that include (binary) delay test vectors are assigned uploaded to the GPU pattern memory, which are then converted to full stimuli waveforms for the assignment to the circuit inputs (3). Once, the circuit inputs have been assigned stimuli waveforms, the evaluation kernel is called which processes the input waveforms at each node to compute a corresponding output waveform (4). The evaluation procedure is implemented using parallel

adapted kernels for simulating the switching behavior at logic level. During simulation, the kernel further considers combinations of different circuit parameters to reflect process- and parameter variations with impact on the delay behavior of the circuit. After the simulation, the responses of the test patterns are obtained by sampling the output waveforms and extracting the switching activity (5). The simulation terminates, when all input stimuli have been processed.

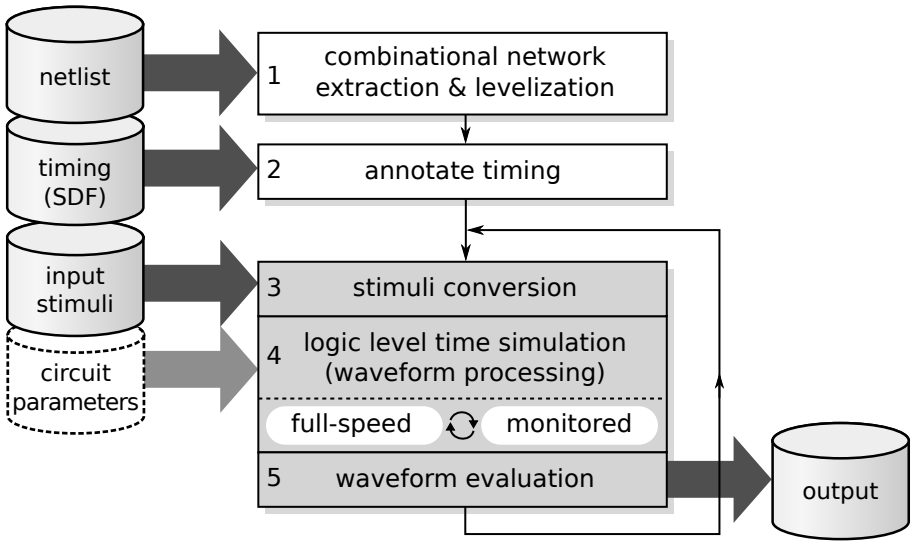


Figure 5.1: Overall flow-chart of the logic level time simulation. Parallel steps and kernels are highlighted.

In the following, the modeling of the circuits and discrete signal values are presented. Then the concept of time is integrated into the signal modeling to form time-continuous and discrete-value signal histories. Afterwards, the delay processing is presented with the basic simulation flow and simulation algorithm. Finally, the applied parallelization concepts and kernel schemes are introduced to run the simulation algorithm under the given modeling aspects with high throughput on the GPU architectures.

5.2 Circuit Modeling

In this thesis *combinational circuits* are considered for simulation. Each combinational circuit implements a defined circuit function which provides a specific output for a given input. A combinational circuit *netlist* is modeled as a directed acyclic graph $G := (V, E)$,

where V is a set of vertices which will be referred to as circuit *nodes* and E is a set of directed edges. The nodes $n \in V$ corresponds to either an *input*, an *output* or a *gate* of the circuit, while each edge $e \in E$ connects a pair of nodes and typically reflect *signal lines* or *interconnections* in the circuit. Vertices $I \subseteq V$ without incoming edges represent the *input nodes* of the circuit G. Vertices $O \subseteq V$ without outgoing edges represent the circuit *outputs nodes* and for all output nodes $o \in O$ at most one incoming edge may exist. Circuit nodes $n \in V$ with both ingoing and outgoing edges represent the *gates* of a circuit. An example of the directed acyclic graph representation of a simple circuit netlist with five input and two output nodes is shown in Fig. 5.2.

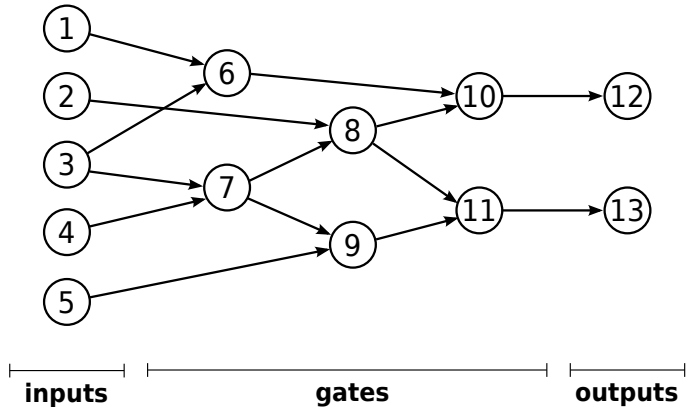


Figure 5.2: Graph visualization of the combinational benchmark circuit *c17*.

The direction of an edge indicates the flow of a signal that transports information from a sending node (called *driver*) to a receiving node (called *receiver*) of the circuit. The set of receivers of a driving node n is referred to as direct *successors* of n. The set of driving nodes of a receiver n is referred to as direct *predecessors* of n. For the sake of simplicity, this work assumes that each driving node passes the same information to all of its receivers, if not mentioned otherwise. If a CMOS cell has multiple outputs which produce different information (e.g., full-adder), the corresponding node can be substituted by a set of vertices where each vertex represents the functional logic of one of the original cell outputs.

The values of input nodes can be assigned Boolean logic values from input stimuli vectors which define value assignments $v_i \in \boldsymbol{B}_2 = \{0, 1\}$ for each circuit input $i \in I$. The values $v_o \in (v_1, v_2, ..., v_m)$ of the m output nodes $o \in O$ of the circuit are considered as test (output) response with respect to the currently applied input stimuli.

For processing the netlist graph during simulation, the nodes are further partitioned into *levels* in a *levelization* pre-process. During *levelization*, the nodes of the combinational netlist are topologically ordered and partitioned into sets $\{L_1, L_2, ..., L_k\}$ of *levels* $L_i \subseteq V$ with $\forall i \neq j : (L_i \cap L_j) = \varnothing$. The order of the L_i is based on the topological distance of the nodes contained to either circuit inputs or outputs. This work utilizes an *as-soon-as-possible* (ASAP) schedule for the ordering, that orders the circuit nodes according to their *depth*. The *depth* of a node $n \in V$ is defined as the maximum topological distance from all input nodes $i \in I$ to n, which is determined recursively as:

$$depth(n) := \begin{cases} 0 & \text{if } (n \text{ is input node}), \\ \max\left(\{depth(p) : p \in fanin(n)\}\right) + 1 & \text{else.} \end{cases} \tag{5.1}$$

Nodes of a certain depth k are then sorted into corresponding levels L_k, such that

$$\forall n \in L_k : depth(n) = k. \tag{5.2}$$

5.2.1 Gates

At logic level a *gate* is the basic functional entity that typically represents the logical equivalent of a physical standard cell at the electrical level (or implementation in silicon). The functional behavior of each gate n of a circuit is modeled by a Boolean function $\phi_n : \boldsymbol{B}_2^k \rightarrow \boldsymbol{B}_2$, which takes the signal values $v_1, v_2, ..., v_k \in \boldsymbol{B}_2 = \{0, 1\}$ sent by each of its corresponding direct predecessors $i_1, i_2, ..., i_k$ as input via the input pins of n. The result of the function $v_n := \phi_n(v_1, v_2, ..., v_k)$ is considered as the *output signal* value of n which is then sent to the gate output pin from where it is distributed to all of its direct successors. For brevity, a vector notation is used to group signal values, such as inputs values or output responses, by $\boldsymbol{v} := (v_1, v_2, ..., v_k)$.

Gates can be *primitive* if they implement a simple primitive Boolean operator such as AND, NAND, OR, NOR, INV or BUF over its set of inputs. On the other hand, *complex* gates implement functions comprised of different operations at the same time. Any Boolean function can be broken down to a set of primitive Boolean operations, each of which has a corresponding primitive gate. Furthermore, gates acting as constant drivers that *tie* signal

lines to a constant value (e.g., TIEL for *low* '0' and TIEH for *high* '1') usually have no input pin and are modeled as circuit input nodes with constant input assignments.
Let $\boldsymbol{v} := (v_1, ..., v_k) \in \boldsymbol{B}_2^k$ and $\boldsymbol{v'} := (v'_1, ..., v'_k) \in \boldsymbol{B}_2^k$ and let v_i and v'_i refer to the i-th component in each vector of $\boldsymbol{B}_2^k$. Then the operator $\leq$: $\boldsymbol{B}_2^k \times \boldsymbol{B}_2^k \rightarrow \boldsymbol{B}_2$ is defined as

$$(\boldsymbol{v} \leq \boldsymbol{v'}) \Leftrightarrow (\forall i \in \{1, ..., k\} : v_i \leq v'_i). \tag{5.3}$$

A gate n is *inverting* if its implemented Boolean function ϕ_n is monotonously decreasing when for any two input vectors $\forall \boldsymbol{v}, \boldsymbol{v'} \in \boldsymbol{B}_2^k : (\boldsymbol{v} \leq \boldsymbol{v'}) \Leftrightarrow (\phi_n(\boldsymbol{v'}) \leq \phi_n(\boldsymbol{v}))$ holds. In case $\forall \boldsymbol{v}, \boldsymbol{v'} \in \boldsymbol{B}_2^k : (\boldsymbol{v} \leq \boldsymbol{v'}) \Leftrightarrow (\phi_n(\boldsymbol{v}) \leq \phi_n(\boldsymbol{v'}))$, the function ϕ_n is monotonously increasing and n is considered as *non-inverting*. For example, types of *inverting* gates are NAND, NOR and INV, while examples of *non-inverting* types are AND, OR and BUF.
Gate types are further distinguished according to their *number of inputs* (e.g., 'NAND2' for a two-input and 'NAND3' for a three-input NAND gate), as well as their *driving strength*, which are annotated in the cell type descriptor by appending 'X1' for single, 'X2' double driving strength, and so on. Gates with higher driving strength, provide more current throughput by implementing parallel transistors structures that allow drive signals with higher fanout much faster and more reliably [Nan10]. As *fanout* of a gate the number of its direct successors is considered during simulation at logic level [GNW10].

5.2.2 Delay

The simulation of the circuit timing requires the representation and modeling of the temporal behavior of gates and cells in the circuit. At logic level the temporal behavior of gates is expressed as transition *propagation delay*, which describes the time it takes to transport the implication of a signal change at a sensitized input pin to the output pin of a node. Propagation delays typically vary for each gate of a circuit and each gate pin (*pin-to-pin* model) as well as the polarity of signal switches (*rising/falling* delay), due to the physical layout structure and the underlying electrical behavior of the corresponding standard cells.
Furthermore, often a so-called *inertial delay* is used where the behavior of changes at node outputs is considered to follow an abstraction of the physical charging process of the

output load of the corresponding gate. Under inertial delay, an output change at a gate is performed only if the output signal will be stable for a pre-defined minimum time after the switch. Therefore, inertial delays act like a filter to remove unreasonable pulses that would be too short to appear at the gate output.
The simulation algorithm proposed in this thesis uses a *pin-to-pin* delay model [HIW15] with individual delay parameters for expressing rising and falling propagation delay as well as inertial delay for all input pins of every node in the circuit. In accordance with common timing models used in industry practice [IEE01a, IEE01b], this work assumes the following delay parameters:

- nominal *propagation delays* $d_f^i \in \mathbb{R}$ and $d_r^i \in \mathbb{R}$ for transitions at an input pin i that cause *falling* and *rising* transition at the gate output, and
- minimum *pulse-widths* $\Delta d_f^i \in \mathbb{R}$ and $\Delta d_r^i \in \mathbb{R}$ for inertial delay pulse filtering of *falling* and *rising* input glitches at a pin i, and
- *variance* parameter $\sigma_i \in \mathbb{R}$ to describe the delay deviation from the nominal propagation delay in presence of delay variation (cf. Sec. 5.2.3).

The timing behavior is represented by *time spec* data structures, each of which contains the delay parameters of a particular node. A *time spec* TS_n of a node n is organized as a set of tuples $ts_i := (d_1^i, d_2^i, ...)$ that contain the delay parameters for one input pin i of the node:

$$TS_n := \{(d_1^1, d_2^1, ...), (d_1^2, d_2^2, ...), ..., (d_1^k, d_2^k, ...)\}. \tag{5.4}$$

Thus, for transitions at an input pin i the associated delay tuple ts_i describes the timing behavior of a signal propagation to the output. While the amount of tuples specified for each node depends on the number of inputs of n, the numbers of parameters in each tuple is related to the underlying timing model and can be adapted depending on the required accuracy of the timing model.

5.2.3 Variations

The delay modeling in this thesis further considers process variations and parameter variations during simulation. While here process variations are considered as passive random

or systematic processes during manufacturing, the same modeling is also applicable to environmental parameters (e.g., temperature) and system parameter variations that are performed online, such as supply voltage scaling and adaptive body biasing [TKD+07]. Based on the active parameter corners of a produced die, the electrical properties might differ and hence its electrical behavior. At logic level, these random and systematic process- and parameter variations typically manifest as delay deviations and thus must be considered during timing simulation [SSB05, BCSS08].

Definition 5.1. A *circuit instance* is a copy of a circuit design under the influence of specific process parameters and operational conditions $P := (p_1, p_2, ..., p_u) \in \mathbb{R}^u$.

In this thesis, a circuit instance is identified by the vector $P := (p_1, p_2, ..., p_u) \in \mathbb{R}^u$ itself. Each of the parameters $p \in P$ reflects a global process or system parameter. Further, it is assumed that an instance $P_{nom} \in \mathbb{R}^u$ exists which corresponds to the nominal circuit instance. Under the nominal parameters P_{nom} the circuit behavior does not exhibit influence of any kind of variation therefore showing nominal timing and functional behavior.
Now, let $P \in \mathbb{R}^u$ be a circuit instance that is described by the parameter setting in $\mathbb{R}^u$. Further, let $d_{nom} \in \mathbb{R}$ be the targeted nominal propagation delay at a gate pin of a node n as specified in the corresponding time spec TS_n. Then the parameterized delay under variation shall be calculated according to the following formula:

$$d' := d_{nom} \cdot (1 + \theta_n(p_1, p_2, ..., p_u)) \tag{5.5}$$

where $\theta_n : \mathbb{R}^u \to \mathbb{R}$ is the corresponding variation function of the node n which expresses the relative delay deviation under the parameters P with respect to the nominal propagation delay d_{nom} in the nominal instance P_{nom}. The delay d' then represents the resulting propagation delay in instance P after the variation has been applied. For P_{nom} the function θ_n should evaluate to $\theta_n(P) = 0$ such that $d' = d_{nom}$.
For *random variation* with normal distributed delays, the *variation function* θ_n is implemented by Gaussian normal distribution kernel $\mathcal{N}(\mu, \sigma^2)$ which generates uniformly distributed random delays with mean $\mu := d_{nom}$ and standard deviation $\sigma \in \mathbb{R}$ [SKH+17]. For *systematic variation*, the delay deviation of θ_n is deterministically expressed by user-defined kernel functions $\theta_n : \mathbb{R}^u \to \mathbb{R}$, such as polynomials. This can cover both, system-

atic process variations [SPI+14] as well as parameter variations with impact on the delay [DAJ+11, DJA+08], such as supply voltage and temperature.
Multiple circuit instances can form a *circuit population* $\mathcal{P} \subseteq \mathbb{R}^u$ that spans a particular parameter sub-space.

Definition 5.2. A *circuit population* $\mathcal{P} := \{P_1, P_2, ...\} \subseteq \mathbb{R}^u$ is a set of circuit instances $P_i \in \mathbb{R}^u$ of a parameter space.

For example, a population can cover *process corners* (e.g., *min-typ-max* delays) of a design or environmental conditions (e.g., temperature). Any $P \in \mathcal{P}$ then corresponds to a sample instance in the corresponding parameter sub-space. The parameters of each instance in a circuit population are usually unique such that for any two instances $P_i, P_j \in \mathcal{P}$ their corresponding parameters differ $P_i \neq P_j$.

5.3 Modeling Signals in Time

In the underlying logic level circuit model, all output and inputs pins of a node as well as interconnection lines are assumed to carry a signal value $v \in \boldsymbol{B}_2$ for an applied test vector at any time point $t \in \mathbb{R}$. Let n be a k-input node corresponding to a gate with an associated Boolean function $\phi_n : \boldsymbol{B}_2^k \to \boldsymbol{B}_2$, and let $I = \{i_1, i_2, ..., i_k\}$ be the set of node inputs. Moreover, let each input $i \in I$ have an assigned stable signal value v_i. The momentaneous output signal state v_n of the node n is given through its node function $v_n := \phi_n(v_1, v_2, ..., v_k)$.

5.3.1 Events and Time

When a new test vector is applied to the circuit inputs, each change of a node input assignment from v to v' can also change the corresponding node output states $\phi_n(..., v, ...) \neq \phi_n(..., v', ...)$. These changes can propagate throughput the circuit to the (pseudo) primary outputs. This work uses so-called *events* in order to model and process signal changes in the circuit over time.

Definition 5.3. An *event* e is a signal change from a value v to a specific value v' at a given point in time $t \in \mathbb{R}$ over a duration $\Delta t \geq 0$.

Each event is modeled as a tuple $e := (a_1, a_2, ..., a_m) \in \mathbb{R}^m$ of m ordered parameters, which correspond to either *time point* of the event, *signal values*, *transition duration*, or other parameters that control the *shape* of the signal change. The number of parameters $m \geq 1$ and semantics of each event depend on the underlying abstraction. For the sake of simplicity, the first parameter in each event shall refer to the time $t \in \mathbb{R}$ of the occurrence.

Example 5.1. *A signal transition at time t (from any state) to a stationary value v can be modeled as time-value pair $e = (t, v)$ to express signal step-functions [CHE+08]. Signal slopes, such as linear transitions to value v, can be simply modeled by an event $e' = (t, v, \Delta t)$, where Δt corresponds to the transition duration (slope parameter).*

5.3.2 Delay Processing

An event at time t at a sensitized node input can cause the output signal of the corresponding node to change (*event propagation*). This *output event* typically occurs at a later time point $t' := (t + d) \geq t$ due to the propagation delay $d \in \mathbb{R}$ of the node. If the node is not sensitized, the output signal is not affected by the input transition. The delay model used in this thesis utilizes an elemental delay processing that comprises multiple stages during the processing of events at node inputs as illustrated in Fig. 5.3.

In the first stage, all events of the k input waveforms at the node n inputs are added the respective propagation delays d_r^i or d_f^i of the time spec TS_n that were assigned to their respective gate pin i for the corresponding transition polarities (e.g., *rising* or *falling*). By adding the propagation delay, the input events are *transported* to the node output and possible event times are predicted. During this process, *intermediate* input waveforms

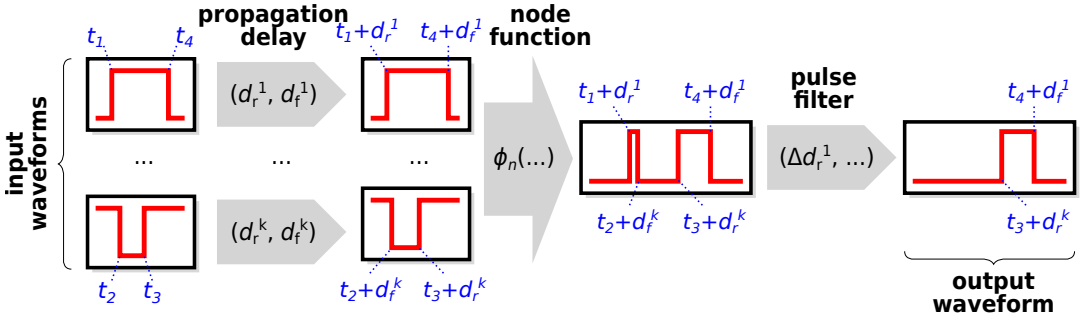


Figure 5.3: Delay processing at a node [HIW15].

are generated. At this stage, the new events are checked for *race conditions* as part of a natural input pulse filter for filtering consecutive transitions that would result in a reversed temporal order at the output while being propagated. For example in case of Fig. 5.3, although $t_1 \leq t_4$ the predicted output event times can be in reversed order with $(t_4 + d_f^1)$ first then $(t_1 + d_r^1)$ in case the propagation delays are of different magnitude (e.g., $d_r^1 \geq (t_4 - t_1) + d_f^1$). If such a race condition is detected, the corresponding events are removed. At the second stage, the signal transitions of the intermediate waveforms are passed in temporal order to the Boolean function ϕ_n of the node n to determine the corresponding output values v_n. For this, the output function is evaluated at all points in time thereby generating a new *temporary* response output waveform.
After evaluation, a *pulse filter* stage is responsible for filtering out hazards and glitches in the temporary response output waveform. The pulse filter removes consecutive events which do not conform with the specified minimum pulse-widths Δd_r^i and Δd_f^i for rising and falling pulses. From an electrical point of view, these pulses are considered too short to allow for a full charge or discharge of the underlying gate output capacitance and are therefore rendered as invalid.

5.3.3 Waveforms

To represent full histories of the switching activity of a signal, a *waveform* data structure is used that stores sequences of events in temporal order [HSW12, HIW15]. The modeling is compliant with the IEEE *standard delay format* (SDF) [IEE01a, IEE01b] for timing accurate simulation and full representation of the circuit switching activity at logic level. In the following, the modeling of signals and time is presented for Boolean logic, which is extended in Chapter 6 for switch level voltage waveforms and ternary logic in Chapter 8.

Definition 5.4. A *waveform* w_n describes the switching history of a signal n over time as a sequence of $K \in \mathbb{N}$ temporally ordered events:

$$w_n := \{e_1, e_2, e_3, ..., e_K\}. \tag{5.6}$$

Each waveform shall be modeled as a list of $K \in \mathbb{N}$ events $e_1, ..., e_K$ with the amount of stored events being bound by a *waveform capacity* K. The waveform capacity typically

varies from circuit node to circuit node as it strongly depends on the node switching activity. All events $e_i \in w$ of a waveform w are temporally ordered in increasing $t_i \in e_i$. Thus, if two events of a waveform $e_i, e_j \in w$ have corresponding event times t_i and t_j with $t_i < t_j$ then their index positions $i, j \in \mathbb{N}$ in the waveform event lists are $i < j$.

For the sake of simplicity, this work assumes that *ordinary* events of actual signal switching processes occur at times $t \in \mathbb{R}$ with $0 \leq t < \infty$. Events at (larger) negative time points, i.e., $t = -\infty$, are used to assign and control initial signal values at a node.

Definition 5.5. The *waveform function* $\boldsymbol{w}(t)$ of a waveform w of a node determines the signal value $v_t \in \boldsymbol{B}_2$ of w present at the node at time t:

$$\boldsymbol{w} : \mathbb{R} \rightarrow \boldsymbol{B}_2, \boldsymbol{w}(t) := v_t. \tag{5.7}$$

For the evaluation of the function $\boldsymbol{w}(t)$, the events $e_i \in w$ are processed in temporal order of their corresponding $t_i \in e_i$. Starting from the earliest event in the waveform, the processing continues until all events up to and including time t have been *handled*. The signal value $v_t \in \boldsymbol{B}_2$ at time t is then determined by the latest event processed which is eventually returned by $\boldsymbol{w}(t) = v_t$ as result. The *handling* process of each event itself depends on the semantics of all events which is specified by the modeling as well as abstraction.

For example, one *naïve* way to model Boolean logic waveforms is by using a two-value representation for events $e_i = (t_i, v_i)$, which explicitly describes signal transitions to discrete values $v_i \in \boldsymbol{B}_2$ at the given time $t_i \in \mathbb{R}$. The complete waveform is stored as a list of ordered tuples $w := \{(t_1, v_1), (t_2, v_2), ..., (t_K, v_K)\}$ with $t_1 < t_2 < ... < t_K$ and $v_i \in \boldsymbol{B}_2$ [CHE+08]. In between two consecutive events e_i and e_{i+1}, signal values are assumed to be constant v_i for all $t \in [t_i, t_{i+1})$ which represents a piece-wise constant function. For the evaluation of the waveform function $\boldsymbol{w}(t)$, the events $e_i \in w$ are processed as regular in temporal order. The latest event $e_j = (t_j, v_j) \in w$ at event time t_j with the highest list index $j = \max\{i : e_i = (t_i, v_i) \in w, t_i \leq t\}$ then determines the corresponding signal value $\boldsymbol{w}(t) := v_j$.

This thesis adopts a sophisticated and compact representation for storing logic level switching information as proposed in [HSW12, HIW15]. When processing events in temporal

order, the modeling implicitly utilizes the knowledge of the current signal state to determine the transition types of events, as the signal always transitions between the two logic states *high* '1' and *low* '0' in $\boldsymbol{B}_2$. Hence, the target value of a transition does not have to be explicitly stored for each event, but only the event time. This avoids redundancy in the data structures and thereby reduces the amount of storage required per waveform.

In the $\boldsymbol{B}_2$-waveform representation of [HSW12, HIW15] events $e \in w$ stored in a waveform w are of the form $e = (t)$ with no transition value information attached. Instead, it is assumed that all waveforms w start with a low '0' value and their signals are considered valid for the time interval $0 \leq t < \infty$ only. The time of the first event stored in the waveform is allowed to be negative $t = -\infty$ which describes the initialization of a signal to '1' $\in \boldsymbol{B}_2$ and no other event is allowed to occur before. Similarly, the last event stored in each waveform is always at time $t = \infty$. This last event takes place in *infinity* with no more events to follow, thus indicating the end of a waveform as a *sentinel*.

All events $e_i = (t_i) \in w$ with time $0 \leq t_i < \infty$ are considered as *ordinary* events which cause the current waveform signal to switch to the opposite of its current value. Hence, $v_{i+1} := (v_i \oplus 1)$ for $v_i, v_{i+1} \in \boldsymbol{B}_2$, where $\oplus$ denotes the two-valued Boolean logic XOR operation. Note that for every pair of two consecutive events in the waveform the previous signal value is retained. As a result, the waveform function $\boldsymbol{w}(t)$ of $\boldsymbol{B}_2$-waveforms can be simply computed as

$$\boldsymbol{w}(t) := |\{e_i : e_i = (t_i) \in w, t_i \leq t\}| \mod 2. \tag{5.8}$$

Examples of Boolean logic level waveforms representations following [HSW12, HIW15] are shown in Fig. 5.4. Note that since each waveform is assumed to start with value '0', all signals that have an initial value of '1' need to provide an initialization event $e = (-\infty)$, that models the initial transition at time $t = -\infty$ (see Fig. 5.4, "A NAND B"). For constant signals the corresponding waveform representations are $w_0 := \{(\infty)\}$ for *low* and $w_1 := \{(-\infty), (\infty)\}$ for *high* values. Again, all waveforms are terminated by a sentinel event $e = (\infty)$. Thus, assuming a *waveform capacity* of K events, waveforms that start with an initial value of '0' can store a maximum of $K - 1$ transitions, while for waveforms with an initial value of '1' only $K - 2$ transitions can be stored.

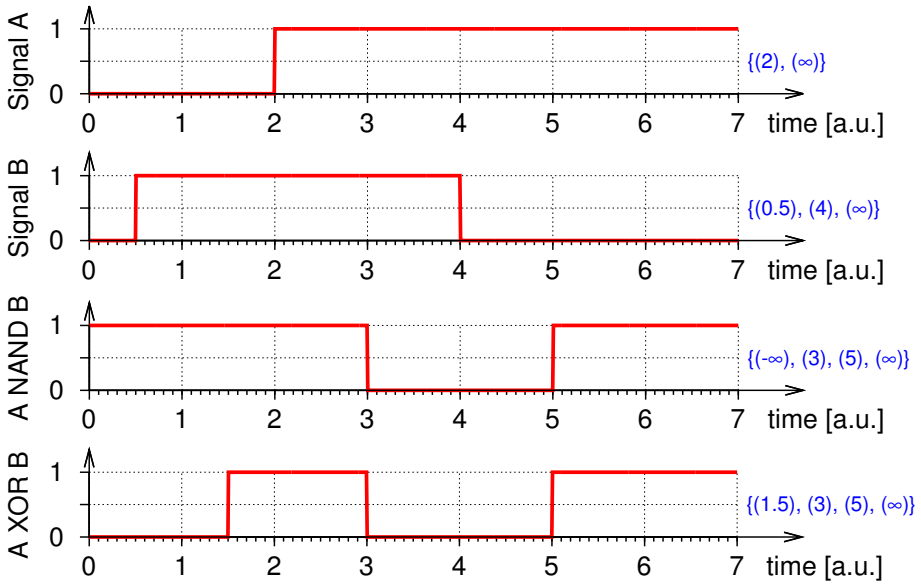


Figure 5.4: Logic level waveforms ($\boldsymbol{B}_2$) of signals A and B before and after passing through NAND and XOR nodes with a delay of 1 time unit [HSW12].

5.4 Algorithm

For the simulation of a circuit, the implemented logic level time simulator utilizes an *oblivious* circuit simulation approach in which the waveform processing is performed for each circuit node regardless of the presence of switching activity at its inputs (as opposed to event-driven approaches). The waveform processing algorithm executed for every node of the circuit computes the output waveforms at each circuit node including the (pseudo) primary circuit outputs in (levelized) topological order. This simulation process has to be repeated for each pattern stimuli applied. Thus, the remainder of this section first describes the basic waveform processing algorithm. Then the parallelization concepts to run the algorithm in parallel on GPUs are introduced.

5.4.1 Serial Algorithm

The overall procedure of the logic level waveform processing is composed of three parts as shown in Algorithm 5.1. For any k-input node $n \in V$, the input to the algorithm is the node description of n, the circuit instance parameters $P \in \mathbb{R}^u$ for variation (if applicable), the input waveforms w_i of all gate pins $i \in I$, as well as the time spec $TS_n := \{ts_1, ts_2, ..., ts_k\}$

of n with the delay tuples ts_i containing the nominal propagation delays for each of the k input pins. Additionally, delays of interconnection wires can be considered which can be treated in two ways. Either, each wire (1) is mapped to an interconnect buffer (BUF gates) that is inserted in the netlist between the driving and receiving gate to which the time spec can be directly assigned to, or (2) the wire is expressed in terms of propagation delays at an input pin in the node *time specs* according to [HIW15]. In the latter case, no additional growth or modification of the circuit netlist is required.

If no pulse widths are provided in the SDF source file, the minimum pulse-widths are set to the gate propagation delays (d_f, d_r). This is a valid assumption, since pulses should not be shorter than the time required for the gate to perform the actual switching [IEE01b].

The general processing at a node is controlled through a sliding *time window* with a lower and an upper bound in order to process the events of all node inputs in correct temporal order. The lower bound refers to the time of the last event processed that caused a change in the output, and the upper bound marks the time of the earliest next output change that may be caused by an event at an input. The algorithm uses a small table for scheduling the processing of local events in the input waveforms. Each line of the table holds for a certain node input pin $i \in I$ the current value of the input waveform v_i, the event e^i_{curr} of the next input switch, the predicted time t^i_{curr} of a possible transition at the node output caused by this event, and the succeeding event e^i_{next} in the input waveform along with the respective predicted output time t^i_{next}. In the algorithm, the time window $[t_{last}, t]$ spans the time of the latest output event computed and $t = \min_{i \in I}(\{t^i_{curr}\})$ refers to the predicted time of the earliest next event at the node output.

When executed for a node n, the algorithm first computes the node- and instance specific propagation delays based on the provided time spec TS_n and the circuit instance parameters P. For random variation, the variation function θ_n executes a *pseudo-random number generator* (PRNG) that is implemented in parallel on GPU similar to [GK09]. Each PRNG requires a unique *seed* to draw individual sequences of uniformly distributed random numbers. For this, *simulation slots* (corresponding to a parallel circuit copies) are assigned *instance IDs* as *base seed* $i \in \mathbb{N}$ which is stored in the *parameter memory* on the GPU device as part of the instance parameter vector P. By applying the Box-Muller method, the generated random number sequences of θ_n are efficiently transformed to a

Algorithm 5.1: Main logic-level node evaluation algorithm (single thread)

Input: Node n with instance parameters P, input waveforms w_i for node input pins $i \in I$, time spec $TS_n = \{ts_1, ..., ts_k\}$.
Output: Event sequence of the output waveform of n.

1 // Initialize local variables data structures.
2 Create copy of time spec TS_n and compute delay variation θ_n according to parameters P.
3 // A.1 load initial waveform events and determine values
4 **foreach** *input i of node n* **do**
5 Load first two ordinary events e^i_{curr} and e^i_{next} from waveform w_i. // skip events at $-\infty$
6 Determine initial input signal waveform value $v_i := \boldsymbol{w}_i(-\infty)$.
7 Compute time of event propagation $t^i_{curr} := t^i_{curr} + d_i[\neg v_i] \in ts_i$ and $t^i_{next} := t^i_{next} + d_i[v_i] \in ts_i$.
8 **while** $(t^i_{curr} \geq t^i_{next}) \wedge (t^i_{next} \neq \infty)$ **do**
9 // race detection in consecutive input events
10 Skip pulse and load next two input events e^i_{curr} and e^i_{next} from waveform.
11 Compute corresponding output event times t^i_{curr} and t^i_{next}. (cf. line 7)
12 **end**
13 **end**
14 // A.2 compute node output state and initialize output waveform
15 Initialize output state $S := \phi_n(v_1, v_2, ..., v_k)$.
16 **if** $(S = 1)$ **then**
17 Append new event $e = (-\infty)$ to output waveform. // sets initial value '1' in $\boldsymbol{B}_2$ logic
18 **end**
19 // A.3 bounds of time window
20 Set $t_{last} := -\infty$.
21 Set upper bound of time window and time t of next event to process: $t := \min_{i \in I}(\{t^i_{curr}\})$.
22 // B. process switching of input events in temporal order
23 **while** $(t \neq \infty)$ **do**
24 // B.1 pick input pin and consume event
25 Select pin i with $t^i_{curr} = t$.
26 Set $t^i_{curr} := t^i_{next}$ and update input value $v_i := \neg v_i$. // negation in $\boldsymbol{B}_2$ logic
27 **if** $(t^i_{next} \neq \infty)$ **then**
28 Fetch next event e^i_{next} from input waveform and compute t^i_{next}.
29 Perform race detection (lines 8–12) and update $e^i_{curr}, e^i_{next}, t^i_{curr}$ and t^i_{next} accordingly.
30 **end**
31 // B.2 compute node state
32 $S_{next} := \phi_n(v_1, v_2, ..., v_k)$.
33 **if** $(S \neq S_{next})$ **then**
34 // apply output filter by checking pulse width
35 **if** $((t - t_{last}) \geq \Delta d^i[v_i] \in ts_i)$ **then**
36 Set $S_{last} := S$, $S := S_{next}$. // update node state and backup old
37 Output new event $e = (t)$.
38 Move lower time window bound $t_{last} := t$.
39 **else**
40 // pulse too short
41 Remove last event in output waveform and revert output state: $S := S_{last}$.
42 Update lower bound of time window $t_{last} := t$ w.r.t. current last event e at output.
43 **end**
44 **end**
45 Move upper time window bound t to next event to process: $t := \min_{i \in I}(\{t^i_{curr}\})$.
46 **end**
47 // C. set termination event
48 Output new sentinel event $e = (\infty)$.

Gaussian normal distribution [Knu97b] which are then applied to the nominal time specs. If no specific parameters P are provided, the algorithm continues with the nominal delays. Then the first two ordinary events e^i_{curr} and e^i_{next} of each input waveform are fetched and the initial waveform values v_i are determined. These are used to predict possible output event times t^i_{curr} and t^i_{next} of both events after propagation (line 7) by selecting the appropriate delay of the *time spec* tuple $ts_i \in TS_n$ of the node n using the numerical interpretation of the current signal logic value v_i as index. The race condition filter is applied (lines 8–12) to ensure the correct temporal order of the output events with respect to the input events, in case the order is reversed ($t_{next} < t_{curr}$). When the filter is triggered, the current events e^i_{curr} and e^i_{next} are invalidated and the algorithm proceeds with the next two events in the waveform.

After the initial events and input waveform values have been loaded, the output state S of the node is computed and both the output waveform and the sliding time window are initialized (line 15–21). The window is bound by the time of the last output change and the time of the earliest next switch caused by the current input event to be processed. This window will be used to apply the *glitch filtering* at the node output.

In the main loop (lines 23–46) the waveform processing is performed in temporal order of the event. First, the input i is selected that corresponds to the predicted next output switch $t := t_{next}$. The current input event is then *consumed* by setting the current event time t_{curr} to the follow-up event t_{next} of the input waveform and toggling the input signal value (line 26). The algorithm then looks-up and pre-fetches the next event in the respective input waveform (lines 27–30).

After the input event has been consumed, the new node output state S_{next} is computed. In case the output state changes ($S_{next} \neq S$), the time window is checked for the minimum pulse-width requirement (line 35) of the glitch-filter. If the time elapsed since the last switch t_{last} is larger than the required minimum pulse-width Δd^i, a new event $e = (t)$ is written to the output waveform and the lower bound of the sliding time window is updated accordingly. The current state variable S is backed up in S_{last} and the new output value is assigned as the current state $S := S_{next}$. If the time elapsed since the last output switch is smaller than the minimum pulse-width, the output pulse filtering is active and the previously added output event is deleted from the waveform with the corresponding

node output state being restored from S_{last} (line 41). Also, since the last output switch was reverted, the lower bound t_{last} of the time window needs to be set to the time of the current latest event in the waveform ($t_{last} := t$) in order to ensure proper glitch-filtering for the remaining events in the sequence. In case the new output state S_{next} is equal to the current state S, the consumed input event does not cause a new output event and hence the input switch can be ignored. Finally after processing of the input switch, the upper bound of the time window $[t_{last}, t]$ is moved to the next earliest output event (line 45). Eventually, all input waveform events have been processed after reaching their sentinel events (∞), which terminates the main loop of the waveform processing. The output waveform is then terminated by adding the sentinel event accordingly (line 48).

Example

Fig. 5.5 illustrates the relation between the input and output events using the time window on the example of a small two-input NOR-gate with a rising delay of $d_r = 2$ and a falling delay of $d_f = 3$ units for each pin. The first input A has waveform $w_A := \{(1), (4), (11), (16), (\infty)\}$ and the second input B has waveform $w_B := \{(8), (\infty)\}$. The corresponding output waveform is $w_{ZN} := \{(-\infty), (3), (7), (10), (\infty)\}$. In this example, the first event in w_A at time $t = 1$ (rising transition) has already been processed, which caused a falling transition (inverting gate) in the output waveform at time $(t+d_r) = 3$ with the current node output state being $S = 0$. Hence, the *lower bound* of the time window is $t_{last} = 3$.

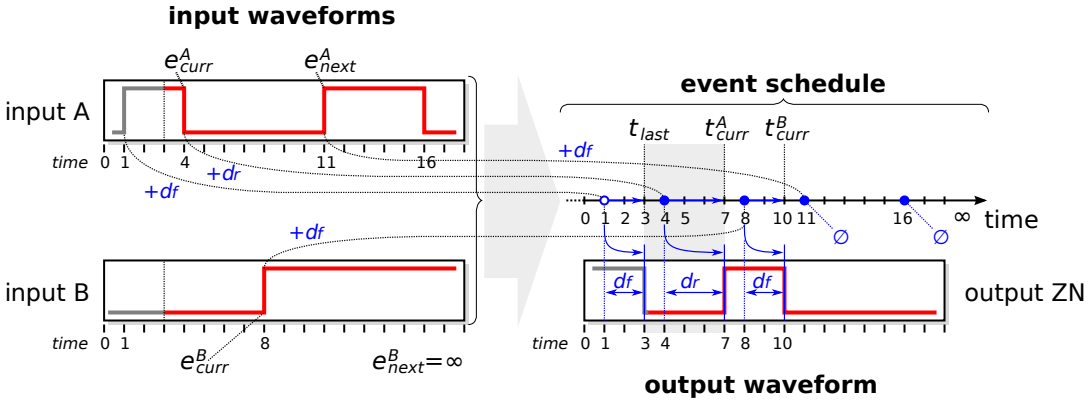


Figure 5.5: Logic level waveform processing at a two-input NOR-gate (*inverting*) with a *rising* transition delay of $d_r = 3$ and *falling* transition delay of $d_f = 2$ time units.

In a next step, the output times of the current input events $e^A_{curr} = (4)$, $e^B_{curr} = (8)$ to process at the two inputs are pre-computed, which are $t^A_{curr} = 4 + d_f = 7$ and $t^B_{curr} = 8 + d_r = 10$. Similarly, the times of the next events are computed as well to check for possible race conditions at the input pins, which are $t^A_{next} = 11 + d_r = 13$ and $t^B_{next} = \infty + d_f = \infty$. As no race detection occurred, the *upper bound* of the time window is updated ($t := t^A_{curr}$) and the input pin A is being selected for the event processing.

Now, the new input value of $v_A := \boldsymbol{w}_A(t) = 0$ is computed and the node function is evaluated to obtain the next state value $S_{next} := \phi_{NOR2}(v_A, v_B) = \phi_{NOR2}(0, 0) = 1$. Since $S \neq S_{next}$, a new event $e = (t^A_{curr}) = (7)$ is added to the output waveform. The lower bound t_{last} of the time window is set accordingly ($t_{last} := t^A_{curr}$). The event e^A_{curr} is finally consumed by stepping forward to $e^A_{curr} := e^A_{next} = (11)$ and fetching the new event $e^A_{next} := (16)$ from the input waveform. After computing the new upper bound, the rising transition event $e^B_{curr} = (8)$ at input B is processed, which causes a falling transition at the node output at time $t^B_{curr} = 10$. The remaining events (11) and (16) at input A do not cause any output state changes and hence no additional events are added to the output waveform. For brevity, the handling of initialization and sentinel events has been omitted in this example.

5.4.2 Parallelization

For the parallelization, the *oblivious* simulation flow is divided into serial tasks some of which are executed on the GPU and some of which are handled by the CPU on the host system. The host CPU is only responsible for minor tasks such as initialization and synchronization of the simulation process which is handled in a serial manner. All compute-intensive tasks on the other hand are executed on the GPU in parallel. Each GPU-task is executed as a kernel that can simultaneously exploit multiple different dimensions of parallelism from both structure and data to maximize the simulation throughput [HSW12, HIW15]. This is illustrated in Fig. 5.6, which shows the parallel evaluation by a simulation kernel that processes different nodes of the circuit each of which is also evaluated for different input stimuli at the same time.

The evaluation of a circuit for a particular stimuli or test pattern is referred to as a *slot*. Within each slot, a subset of different nodes of the circuit is evaluated concurrently form-

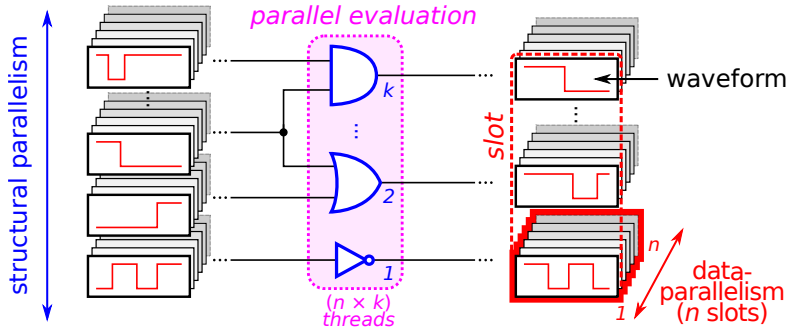


Figure 5.6: Two-dimensional evaluation by utilizing *structural parallelism* from nodes and *data-parallelism* from waveforms [HSW12, HIW15].

ing one dimension of parallelism, called *structural parallelism*. A second dimension is provided by processing *slots* for different input data at the same time, and hence, the processing of the nodes for different input data in parallel (*data-parallelism*).

Node-Parallelism

The first dimension of parallelism exploited is *node-parallelism*, a type of structural parallelism from the evaluation of different nodes in the circuits that have *mutual* (data-) independence. Let $fanin^*(n)$ be the transitive fanin (also called input-cone) and $fanout^*(n)$ the transitive fanout (output cone) of a node $n \in G$.

Definition 5.6. If two circuit nodes $n_1, n_2 \in G$ with $n_1 \neq n_2$ are neither in the input-cone $((\{n_1\} \cap fanin^*(n_2)) \cup (\{n_2\} \cap fanin^*(n_1)) = \varnothing)$ nor in the output-cone of each other $((\{n_1\} \cap fanout^*(n_2)) \cup (\{n_2\} \cap fanout^*(n_1)) = \varnothing)$, then n_1 and n_2 are mutually *data-independent*.

While under data-dependency, the node providing the input to the other one must be evaluated first, otherwise the input remains *unspecified* and unable to be processed. Mutually data-independent nodes on the other hand do not depend on the output data of each other and hence the exact order of evaluation is not subject of matter and can be performed in any order or even at the same time. On parallel architectures, this property is exploited to evaluate data-independent nodes concurrently [SHWW14, HIW15].
Fig. 5.7 depicts the parallelization scheme of the simulation kernels executed for a single simulation slot on the GPU. The evaluation of the nodes is performed in a (partially)

ordered sequence obtained from *levelization* which ensures the provision of specified input waveforms for all nodes on each level. For each level $L_i \subseteq V$ of the circuit ($i = 1, ..., d$), the kernels are invoked by running $k_i := |L_i|$ individual threads for each node in the slot to be evaluated at the current level. The threads concurrently process the previously computed input waveforms of their corresponding nodes independently of the others and compute their respective output waveforms or process and aggregate data.

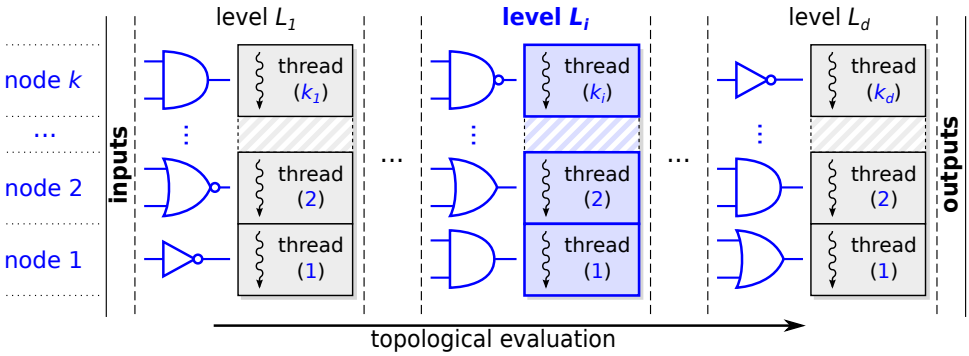


Figure 5.7: Thread-organization for exploiting structural parallelism from levels of mutually data-independent nodes in a simulation slot.

Waveform-Parallelism

The implemented simulator further exploits the parallelism from data-centric aspects to provide higher simulation throughput. For this, a two-dimensional parallelization scheme is adopted [HSW12, HIW15] in which different nodes are processed concurrently that are also evaluated for different stimuli at the same time (*waveform-parallelism*).

In each step of the simulation, the invoked parallel kernels start a *two-dimensional grid* of $n \times k$ execution threads as illustrated in Fig. 5.8, where n is the number of waveform stimuli and k is the number of gates for concurrent evaluation. As shown, within the thread grid the threads in the *vertical* direction form a *slot* in which the different nodes of a level are evaluated in parallel for a particular input stimuli. In the *horizontal* direction, all threads evaluate the same node, but each thread operates on an individual simulation slot each of which can provide a different input stimuli. The horizontal and vertical thread grid dimensions depend on the number of available simulation slots, and the number of nodes on the current level to be processed. While the number of gates usually varies from level to level depending on the amount of available nodes, the number of simulation

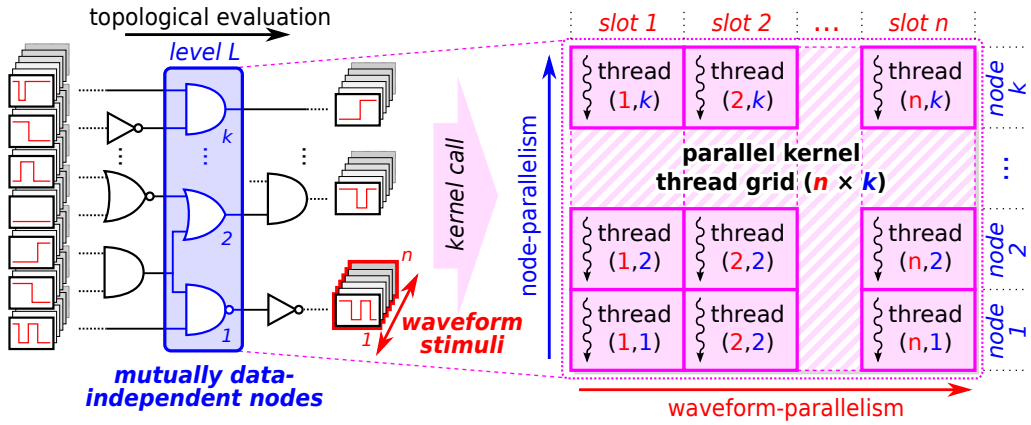


Figure 5.8: Two-dimensional thread-organization for simultaneous exploitation of structural parallelism and data-parallelism for k nodes and n slots.

slots remains constant throughout the simulation. Since each thread of a thread group processes the exact same node function, the threads closely follow the same execution path during the evaluation without excessive control flow divergence due to complex branching in the kernels. Note that larger sets of independent test stimuli can be partitioned and distributed to multiple GPU devices for execution, which allows for a further increase in the simulation throughput.

Instance-Parallelism

The concept of exploiting data-parallelism in multi-dimensional kernels is further applied to consider *instance-parallelism* during simulation by processing multiple circuit instances of a population with unique gate delays in parallel [SKH+17, SW19a]. For this, the two-dimensional thread organization of the parallel simulation is extended to allow simulation of a different *circuit instance* in each simulation slot as illustrated in Fig. 5.9. Within each simulation slot the corresponding threads then process the gates of a specified circuit instance for the assigned waveform stimuli. In the example, the n simulation slots on the GPU have been organized to systematically process m different stimuli for a population of i different circuit instances.

Each simulation slot is assigned circuit instance specific parameters which are accessible by the threads through a dedicated memory allocated on the GPU. The threads access the respective parameters of their slot in order to parameterize the circuit data (i.e, node

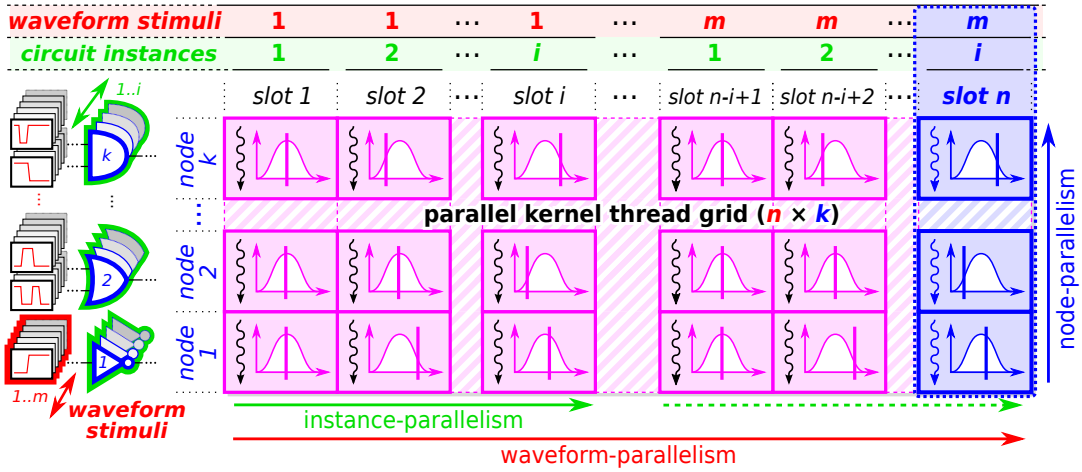


Figure 5.9: Multi-dimensional thread organization for parallel simulation of i parameterized circuit instances for m stimuli.

delays) and the simulation kernels in the beginning of each call. After loading the node descriptions, each thread then calculates thread-local delays based on its parameters accordingly which are then used throughout the node evaluation.

Evidently, since the threads in each row of the thread grid compute the same node function, both the node-parallelism and the control flow uniformity of the underlying kernels in this scheme are sustained and therefore remain untouched. Hence, the combination of the data-parallelism types allows to increase the simulation throughput by maximizing the utilization of the GPU resources and fully occupying the memory during simulation. Again, since the stimuli-instance combinations are data-independent, this parallelization scheme allows workload distribution to multiple parallel GPU devices as well.

5.5 Implementation

In the following, the overall simulation flow as well as the kernel organization of the presented parallel logic level time simulation and their realizations are explained.

Fig. 5.10 depicts the general data flow of the parallel kernels implemented for the simulation. In general, throughout the simulation, the *circuit description* is kept in the global memory of the GPU (i.e., circuit description memory), which contains both the functional and timing descriptions of each node in the circuit. The *simulation state*, containing all

computed signals and simulation results, is kept in the global memory of the device as well (i.e., waveform memory). The *parallelized kernels* take commands and parameters from the host system as input (1), which are used to select and fetch node descriptions from the *circuit description* (2) to process the selected nodes. During processing, the kernels fetch the current *simulation state* as well as the assigned inputs from the memory and write the computed result back into the simulation state again (3). The *output* of a kernel (4) can range from a simple status indicator, over an abstracted mapping of the simulation state, such as output response patterns or other simulation parameters (i.e., weighted switching activity) to a return of the full simulation state. Certain outputs can be used as input of consecutive simulation runs (5), such as state assignments in sequential state-elements from pseudo-primary outputs to pseudo-primary inputs. Finally, both outputs as well as inputs can be redirected to annotate or manipulate the circuit description (6).
In total, the following memories are allocated on the global device memory GPU prior to the actual simulation:

1. *circuit description memory* to store the functional circuit and timing information,
2. *pattern memory* for storing compact input stimuli and output response information,
3. *waveform memory* for storing all signal waveforms (simulation state) which is the largest portion,
4. *parameter memory* to assign instance parameters to the simulation slots, and
5. *scratch memory* to collect miscellaneous information (e.g., overflows) for temporary use only.

Besides a uniformly organized kernel structure, it is necessary to avoid large working sets, diverging control flows and arbitrary memory access patterns in the global memory [OHL+08, GK08]. All threads should work independently of each other in order to avoid synchronization and they should require a minimum amount of resources (e.g., thread-local memory). For the efficient execution on GPUs, the type of data structures and their organization in the memory poses a major hurdle. Therefore, another key-aspect is the focus on compact data structures and the memory organization of both, the functional as well as the timing-description of the circuit and the time-continuous waveform signals.

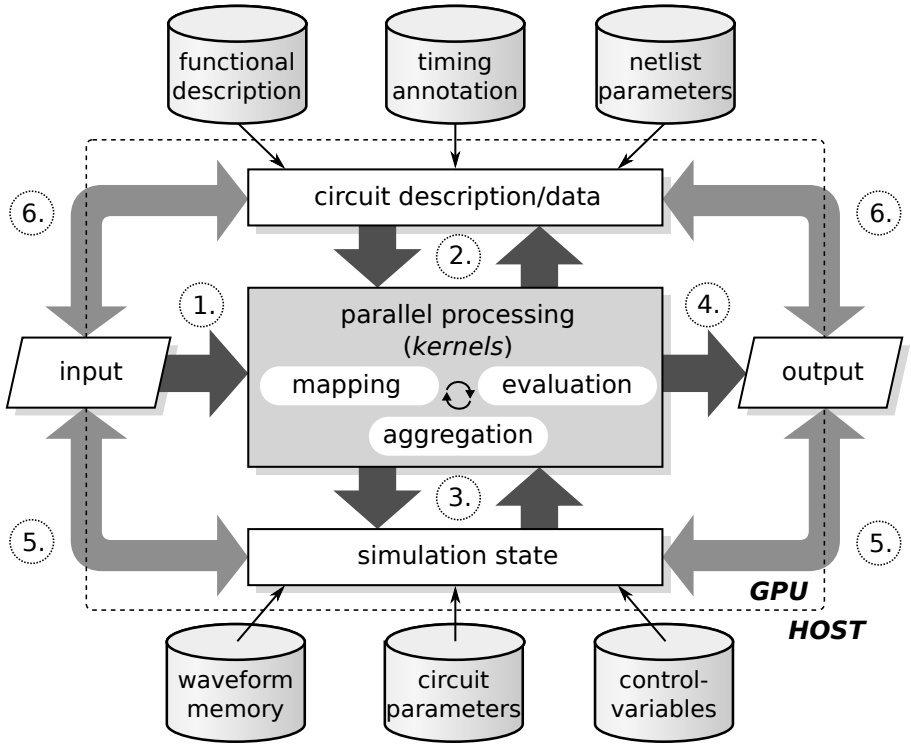


Figure 5.10: General data flow of the kernels used during simulation on the GPU.

5.5.1 Waveform Allocation

To allow an efficient processing and storage of the waveforms on the GPU with simple memory organization and regular memory access patterns, a *buddy system* memory allocation technique is utilized [Kno65, Knu97a], that provides an efficient management for the waveforms [HIW15]. In the buddy system the waveform memory is split into chunks of fixed equal size, that are organized as a binary tree. Each leaf node of the tree corresponds to a chunk of consecutive memory addresses, while parent nodes represent the union of the memory space of all its children. The leaf nodes of a parent node can be merged to link the associated memory chunks in powers of two (2^i at the i-th level) thus forming a larger consecutive memory space. Similarly, all waveforms during simulation are built from memory chunks.

The memory chunk of a leaf node is referred to a *waveform register*. All waveform registers have a fixed capacity $\kappa \in \mathbb{N}^+$ of events (*waveform capacity*). In this thesis, an initial capacity of $\kappa = 10$ is utilized. Subsequent waveform registers are merged to form larger waveforms according to the buddy tree system, in case the memory of a register is insuffi-

cient to store all events. Hence, for allocations at leaf nodes in the tree (level 0), the event capacity of a waveform is $K = \kappa$. For waveform allocations at the i-th level, the memory chunks of 2^i waveforms registers have been merged thus resulting in an event capacity of $K = (2^i \cdot \kappa)$ per waveform. While the allocations are kept for every level of the circuit, each waveform registers can be freed after processing a certain level in the circuit. Once a waveform is not read in the subsequent stages anymore, it is freed to provide memory space for new allocations.

Waveforms are stored on the GPU in the dedicated waveform memory as illustrated in Fig. 5.11. The *waveform memory* is split into *parallel-waveform registers* that form a connected block of memory and contain the waveforms of a specific node of the circuit. The parallel-waveform registers hold one instance of a waveform for each available simulation slot. All waveforms are stored within a register in a way, such that the j-th parameter of the i-th event in waveforms of subsequent slots are stored at consecutive memory addresses. Thus, while processing the waveforms, the 32 threads of each thread group on a stream multi-processor utilize coalescing of the memory accesses addressing the current parameters and merge the addresses into ranges for single memory transactions.

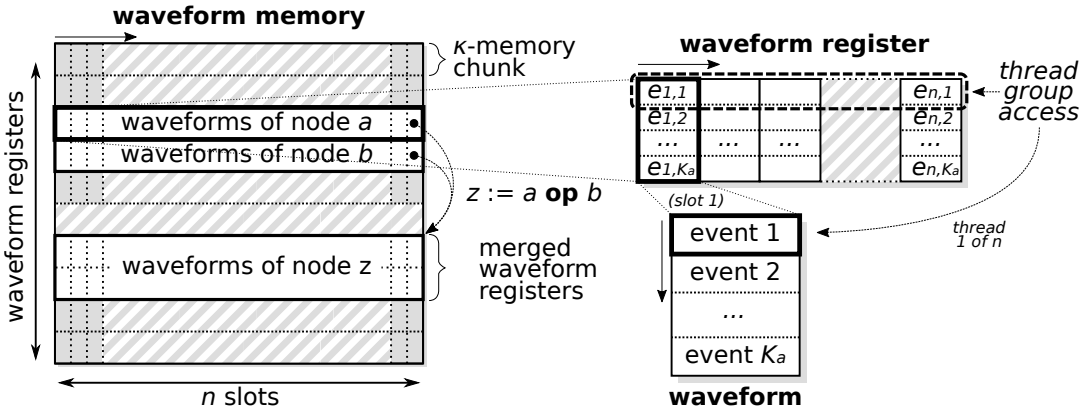


Figure 5.11: Organization of the memory for storing waveform data. For node z two memory chunks are merged into one waveform register ($K_z = 2\kappa$).

Obviously, the waveform memory occupies the largest portion of the global device memory. Given a specified portion of the global device memory on the GPU that will be dedicated to the sole purpose of storing waveforms and the amount of memory of the waveform registers required for allocating the waveforms of a (single) simulation instance,

then the maximum number of possible simulation slots $n \in \mathbb{N}$ is determined [HIW15] by finding the largest n such that

$$GlobalWaveformMemory \geq n \cdot MemoryPerInstance. \tag{5.9}$$

5.5.2 Kernel Organization

The high-throughput simulation flow uses a fixed set of *kernels*, each of which serves a unique purpose and also has a different thread grid organization. These kernels are categorized as one of the following four:

1. *pattern-to-waveform* conversion kernels,
2. *waveform processing* evaluation kernels,
3. *waveform-to-pattern* conversion kernels,
4. *data manipulation/aggregation* kernels.

The kernels of the first category (*pattern-to-waveforms*) are used to convert the binary test stimuli sequences provided in a dedicated *pattern memory* to fully expanded waveforms stored in the *waveform memory* on the GPU. These kernels are usually applied to the nodes on the input level and drastically reduce the computing and memory overhead as no raw waveform data has to be transferred between the host system and the device.
As for the second category (*waveform processing*), the kernels perform the actual waveform-based time simulation and evaluation of nodes on a single level as presented in Algorithm 5.1. The evaluation kernels are executed for each level of a circuit until all levels have been processed. After simulation of a particular level $L \subseteq V$, the waveforms of all its nodes $n \in L$ are present in the GPU waveform memory which can be accessed for further processing.
The third category (*waveform-to-pattern*) kernels provide the reverse operation of the category-1 kernels by sampling waveform values of given nodes (e.g., circuit outputs) and reducing them to compact test response vectors on the GPU. After the conversion, the test responses can then be fetched by the host system with little memory overhead for further processing.

For the execution of kernels of the three categories above, the set of threads is partitioned into a two-dimensional grid of two-dimensional blocks of threads. Each thread works independently on a different node and a different slot and no synchronization of threads from different blocks is required. Also, in the kernels of the first and second category, the threads within each thread group are independent as well. Thus, no synchronization among or within the thread groups is required. Kernels of the third category, utilize schemes based on one-dimensional *reduction* [NVI18a] in order to efficiently encode information of multiple threads working on different slots into single data words. The *shared memory* in each streaming multi-processor is used to collect the data of each thread and to provide a consistent view [NVI17b]. The data of the thread group is then encoded in a synchronized iterative *reduction phase*, before eventually written back by a thread. Thus, synchronization is only required for threads within a thread group, but different thread groups can still run independently of each other.

The kernels of the last category (*data manipulation/aggregation*) are used to extract and quickly merge information (such as sampled values, overflows, event counts, etc...) from waveforms in the memory. The parallel aggregation is utilized to reduce the overall latency and to avoid excessive look-ups of individual numbers in the device memory by the host. Kernels of this category are primarily used to perform fast overflow checking in waveforms and collect waveform attributes of either all active waveform registers in a slot, or of all slots in a particular waveform register. Depending on the kernel the extracted data is merged into one- or two-dimensional fields, or even into a single scalar, and is then stored in a separate memory on the GPU device. When the data is reduced to a one-dimensional field or single scalar, the original two-dimensional high-throughput kernel scheme cannot be applied, due to the high data-dependency over either the slots or the waveform registers. Since the synchronization among threads of different thread blocks is expensive, a different thread grid organization is applied.

For the reduction of the data, basically two different reduction kernels are offered: (1) *horizontal reduction* and (2) *vertical reduction* as shown in Fig. 5.12.

During the *horizontal reduction*, waveform data of each node $n \in G$ is aggregated usually by addition (e.g., for event counting or determining overflows) into a single struct over all available slots s that is accessible via the node index in a separate memory. The kernel

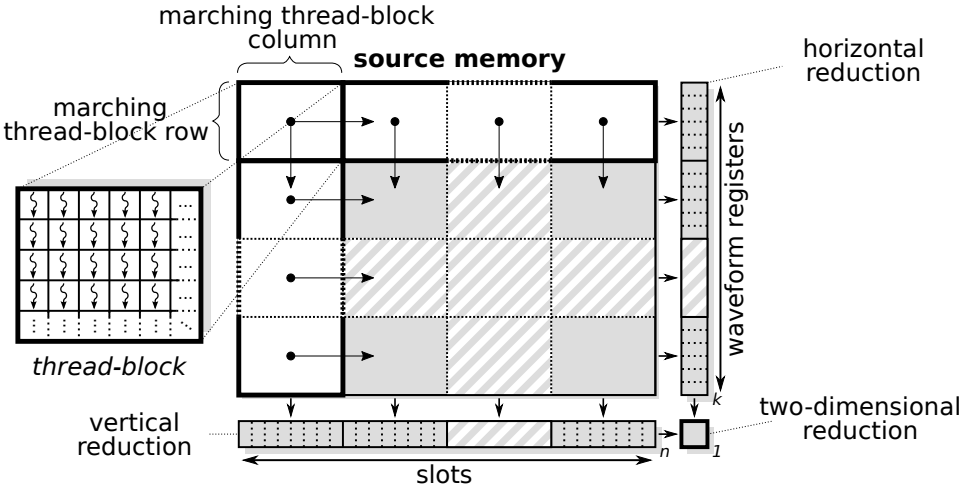


Figure 5.12: One-dimensional (*horizontal*, *vertical*) and two-dimensional memory reduction for quick data aggregation and compaction.

is executed as a marching column of thread blocks, with each thread group of a block summing up the data of a particular node by traversing over all the available slots in bundles of 32 consecutive slots. Throughout the traversal, the acquired data of the slots is held in the shared memory of the streaming multi-processors. Eventually, all slots have been processed and a final one-dimensional reduction over the shared memory reduces the 32 slot values of the thread group into a single struct to be stored for the node. The memory then provides the merged data for each node individually. Similarly, the *vertical reduction* aggregates the data of all nodes in the slots by a marching thread-block row. The resulting data is then stored in the separate memory for each slot individually. By applying another reduction over the results, a two-dimensional reduction is formed, which allows to reduce the information to a single struct. In this work, the two-dimensional reduction is primarily used for the *fast overflow checking*, which allows to determine the presence of overflows by looking up a single scalar value.

For the kernel execution, the thread grid of each kernel is sub-divided into two-dimensional sub-grids or *thread blocks* which are scheduled for execution on the streaming multi-processors. Each instantiated thread block of a simulation kernel has a fixed dimension of $(blockDim.X, blockDim.Y) \in \mathbb{N}^+ \times \mathbb{N}^+$ threads. The horizontal dimension of each thread block was chosen as $blockDim.X := 32$ to match the number of maximum allowed threads in a *thread group*, which maximizes the thread group occupancy on the streaming multi-

processors. As for the vertical dimension $blockDim.Y$, the available resources distributed evenly among all the threads according to the following formula:

$$blockDim.Y := \left\lfloor \frac{\#ResPerMultiProcessor}{blockDim.X \times \#ResPerThread} \right\rfloor \tag{5.10}$$

where $\#ResPerMultiProcessor$ corresponds to the overall available registers on the steaming multi-processors and $\#ResPerThread$ corresponds to the registers required per thread. For example, if a streaming multi-processor provides 65,536 local registers and a simulation kernel requires 100 registers per thread, then the thread block dimensions are chosen as $(blockDim.X, blockDim.Y) := (32, 20)$ according to Eq. (5.10). For these dimensions, the resulting thread blocks consist of $20 \times 32 = 640$ threads each.

Prior to calling a kernel on the GPU, both the dimensions of the thread blocks as well as the dimensions $(dimGrid.X, dimGrid.Y) \in \mathbb{N}^+ \times \mathbb{N}^+$ of the two-dimensional thread grid itself need to be specified. Given the thread organization as shown in Fig. 5.8, the horizontal X-dimension of the grid corresponds to the number of available simulation slots on the GPU and the vertical Y-dimension corresponds to the number of nodes on the current level under evaluation. Since the simulation kernels are executed for a given circuit level, the grid is evenly subdivided into a two-dimensional grid of thread blocks as follows:

$$dimGrid.X := \left\lceil \frac{\#Slots}{blockDim.X} \right\rceil \text{ and } dimGrid.Y := \left\lceil \frac{\#NodesOnLevel}{blockDim.Y} \right\rceil. \tag{5.11}$$

The two-dimensional kernel processing allows to evaluate nodes for different stimuli within a thread group in a SIMD fashion as shown in Fig. 5.8. In combination with the aforementioned organization of the waveform memory, the waveform accesses of the threads can be merged to create fully utilized memory transactions ($32 \times 4 = 128$ Byte) when processing the single-precision floating-point event parameters of waveforms. With efficient coalescing and caching of memory accesses within each thread block, the overall amount of global memory transactions is reduced, allowing to increase the computational throughput [NVI17b].

The two-dimensional parallelization scheme is applied throughout the different kernels used during the simulation, from test pattern conversion, through node evaluation and signal sampling. The maximum amount of stimuli (simulation slots) that can be processed

in a single simulation run is bound by the available global memory on the GPU device, as well as the memory required to store the waveforms in a simulation slot. In case more input stimuli are provided, than able to fit on the GPU memory, the set of stimuli is split into chunks, which are processed serially on a single GPU. The indices of the stimuli pairs simulated in a given simulation pass ($pass = 0, 1, ...$), depends on the pass number as well as the the number of available slots $\#Slots$ on the GPU. The index of the stimuli in the first slot ($s = 0$) is determined by $(pass \times \#Slots)$ and the index of the last stimuli pair processed ($s = \#Slots - 1$) is $((pass + 1) \times \#Slots) - 1$. Again, if the host system contains multiple GPU devices, the simulation parallelism can be further enhanced by distributing the stimuli chunks over the devices to separate simulator instances. Thus, a larger global memory and more GPU devices allow for higher degree of parallelism by processing more stimuli concurrently. In contrast to the structural parallelism from the nodes, the effective parallelism from the data remains constant throughout the simulation.

5.5.3 Simulation Flow

The overall time simulation flow is comprised of two major parts: A full-speed run, and a monitored run, as shown in Fig. 5.13. Parallel tasks on the GPU are indicated as shaded boxes, while tasks of white boxes are run on the serial host CPU.

The *full-speed run* provides a fast evaluation, while the *monitored run* takes measures to ensure the integrity and completeness of all waveform information throughout the simulation. In the beginning, the circuit description and the test patterns are uploaded into their dedicated memories on the GPU (step 1–2). Then a parallel kernel is called on the GPU that transforms the test patterns into stimuli waveforms for the primary and pseudo-primary inputs (3), after which the parallel level-by-level simulation of the circuit is performed (4). In order to prevent loss of overflow information after freeing waveform registers, the information of the presence of overflows is propagated through the circuit via the waveforms. For this, the first event in each waveform is reserved as header information for holding the number of overflow occurrences in the current waveform and those of its inputs. An overflow check kernel collects the information (5) propagated to the output nodes and indicates whether information loss occurred and a calibration run is

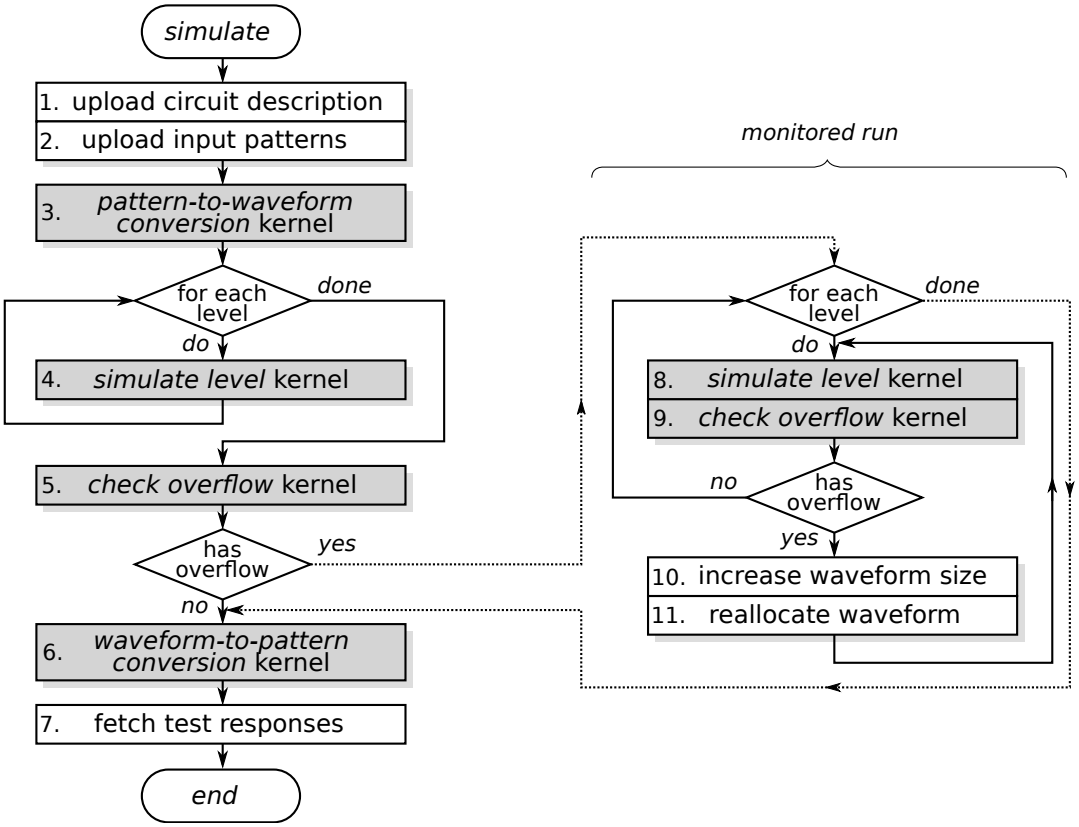


Figure 5.13: Overall simulation flow and call structure of the parallel time simulation with *full-speed* run (left) and *monitored run* (right). Shaded boxes reflect parallel tasks.

necessary. If no overflow did occur, all output waveforms are consistent and are converted back to test pattern responses for further processing on the GPU (6) or the host system (7). In case overflows did show up, the monitored simulation procedure is called, which repeats the level-by-level simulation of the circuit for the previously assigned input stimuli. In contrast to the full-speed run, an overflow check is performed after simulation of each level (8–9) during the monitored simulation. As soon as overflows on a level have been detected, the culprit waveform registers present in the memory are identified and increased in size (10). Due to the increased waveform size, a costly reallocation of all upcoming waveforms is necessary (11). The simulation of the level is then repeated, until all registers on the level eventually satisfy the required waveform sizes. After all levels have been successfully simulated, the output responses can be fetched and processed.

In the beginning, all waveform registers have a small initial capacity $K = \kappa$ and they gradually grow in size due to a massive amount of overflows caused. Yet, throughout the

simulation the total number of overflows saturates quickly, as the registers attain the appropriate sizes to store all events [SHK+15]. The calibration of the waveforms reduces the probability of new overflows and also the need for monitored simulation, thus allowing for full-speed simulation runs. Once calibrated, the new capacities of the individual waveform registers are reused in subsequent simulation runs. After simulation, The capacities of the waveform registers can be stored externally for later reuse. When a new simulation instance is started, the capacities are read and assigned to the waveform registers for initial allocation as part of the initialization, which allows to skip initial calibrations.

5.5.4 Test Stimuli and Response Conversion

Prior to the waveform processing of the circuit nodes, input stimuli waveforms are assigned to the circuit inputs in the simulator. The input stimuli waveforms are obtained from delay test vector pairs as specified from either random or deterministic sources. In each simulation slot, the processing of one delay test vector pair is performed.
In order to timing simulate a particular delay test, the input assignments of the test pair must be translated into an equivalent input stimuli waveforms which are applied to their corresponding inputs $i \in I$ of the circuit. Providing the waveforms by the host system itself is an expensive task and causes high memory overhead. Instead, the compact stimuli bit-sequences of test pattern files are transferred to the dedicated pattern memory on the GPU device, which are then expanded and converted to waveforms on the GPU for all the inputs. Similarly, all output waveforms are evaluated at given sample times and translated back to compact test responses for each test vector.

Input Stimuli Conversion

The test sets for simulation are considered as sequence of subsequent test vectors or test vector pairs with corresponding assignments to the circuit inputs. All test vector assignments of the sequence are stored as 64-bit `Long` integers, with each `Long` representing a sequence of 64 consecutive 1-bit assignments to a specific circuit input. The assignments of each `Long` are further subdivided into pairs for 32 slots, each pair of which holds a delay test pair composed of a value $v \in \boldsymbol{B}_2$ as *initialization vector* value that is followed by another value $v' \in \boldsymbol{B}_2$ of the test vector sequence as *propagation vector* value.

Algorithm 5.2 outlines the conversion process to obtain a valid stimuli waveform from a delay test vector pair assignment at a circuit input pin $i \in I$ composed of initialization $\boldsymbol{v} \in \boldsymbol{B}_2^{|I|}$ and propagation vector $\boldsymbol{v}' \in \boldsymbol{B}_2^{|I|}$. First the waveform is initialized with the value of the initial assignment in $v_i \in \boldsymbol{v}$ (line 2–4). For the propagation vector value $v_i' \in \boldsymbol{v}'$ the algorithm appends an event to the waveform at the specified transition time t (typically $t = 0$), if the assignment changes (line 5). Note that the propagation vector $\boldsymbol{v}' \in \boldsymbol{B}_2^{|I|}$ can also be used as initialization vector of the next test vector pair.

Algorithm 5.2: Input waveform obtained from a delay test pair assignment (v_i, v_i').

Input: delay test pair (v_i, v_i') at a circuit input $i \in I$, transition time t
Output: logic level input assignment w_i at input node i

```
1 Initialize empty waveform w_i := ∅.  // current value is '0'
2 if (v_i ≠ 0) then
3 |   Append new event e = (−∞) to w_i.  // raise initial signal to '1'
4 end
5 if (v_i' ≠ v_i) then
6 |   Append new event e = (t) to w_i.  // toggle signal value at time t
7 end
8 Append new event e = (∞) to w_i.  // termination event
9 return w_i
```

The algorithm is implemented by a two-dimensional kernel with each thread providing a stimuli waveform of a test for an input node. The corresponding thread grid dimensions $(X_{P2W}, Y_{P2W}) \in \mathbb{N}^+ \times \mathbb{N}^+$ of the kernel were chosen according to the number of simulation slots and the number of inputs of the circuit, such that

$$(X_{P2W}, Y_{P2W}) := (\#Slots, \#inputs). \tag{5.12}$$

For conventional two-pattern delay test sets, the patterns are encoded in data structures that use 64-bit `Long` integers to represent sequences of 32 consecutive initialization and propagation vector pair assignments of a specific input which fits the size of a SIMD thread-group. In each simulation slot the circuit is applied a single vector pair. The number of delay tests stored in a `Long` integer corresponds to the number of threads in a thread group. Thus, each thread group can process a `Long` exclusively. The threads of a thread group access the respective vectors upon loading the data structure as shown in Fig. 5.14 and operate by using bit-wise operations involving *shifting*, *masking* and *comparison*. This way, stimuli can be provided in a compact way in order to avoid costly memory transfers

between the host system and the GPU device. Once transmitted, input stimuli can reside in the GPU memory for reuse in subsequent simulation runs.

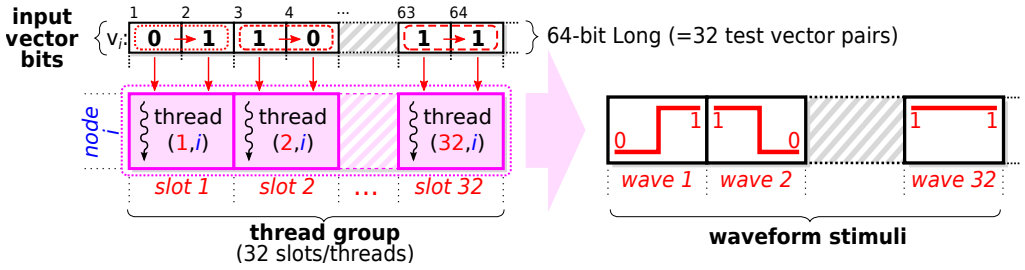


Figure 5.14: Parallel *pattern-to-waveform* conversion by threads transforming pairs of initialization and propagation vectors into stimuli waveforms of the slots.

Output Response Sampling

After processing the circuit, all output waveforms of the output nodes are available in each slot of the waveform memory. At this point the output waveforms can be copied from the GPU device to the host system memory for further processing. However, the waveforms can also be efficiently evaluated in parallel on the GPU in order to avoid large and costly memory operations by parallel sampling and conversion to compact test response vectors. The general output sampling for obtaining the value of a waveform w_o of an circuit output $o \in O$ at a given sample time t implements the corresponding waveform function $\boldsymbol{w}_o(t)$ for the Boolean logic waveforms. The algorithm is shown in Algorithm 5.3. The waveform value at a given time point t, is obtained by iterating over the events $e_i = (t_i) \in w_o$ until the time window $[t_i, t_{i+1}) \ni t$ is reached. In each iteration, the initial waveform value v_o is negated which reflects a signal toggle due to the event processing. The value v_o at time t_i then represents the sampled value $v_o := \boldsymbol{w}_o(t)$ $(= \boldsymbol{w}_o(t_i))$.

For the parallel sampling of the waveforms, a two-dimensional kernel is called with each thread processing an output waveform of a node within a slot. The thread grid dimensions $(X_{W2P}, Y_{W2P}) \in \mathbb{N}^+ \times \mathbb{N}^+$ of the kernel are chosen according to the number of simulation slots and the number of circuit outputs to process in order to exploit maximum parallelism on the device:

$$(X_{W2P}, Y_{W2P}) := (\#slots, \#outputs). \tag{5.13}$$

Algorithm 5.3: Value sampling of a Boolean logic waveform.

Input: output waveform w_o of output $o \in O$, sample time t
Output: test response $\boldsymbol{w}_o(t)$

```
Initial value of output waveform v_o := 0.
Get first waveform event e_i = (t_i) ∈ w_o with i = 0.
while (t_i ≤ t) ∧ (t_i ≠ ∞) do
    v_o := ¬(v_o).  // process event and toggle signal state.
    Set i := i + 1.
    Get next event e_i = (t_i) ∈ w_o.
end
return v_o
```

The parallel output evaluation and the conversion into response vectors is shown in Fig. 5.15, which depicts the waveform evaluation using sampled values within a single thread group of the two-dimensional kernel scheme. The kernel evaluates the waveform at two *sample times* $t_a \in \mathbb{R}$ and $t_b \in \mathbb{R}$ given as input parameters. Each thread that is invoked then accesses and evaluates the response waveform w of its respective output and slot. The obtained responses for both $\boldsymbol{w}(t_a)$ and $\boldsymbol{w}(t_b)$ of each waveform are then encoded and stored in 64-bit `Longs` in the dedicated pattern memory. Note that by choosing sampling times $t_a = -\infty$ and $t_b = \infty$ in a *fault-free* simulation, the resulting response vectors in the memory correspond to those of an untimed zero-delay logic simulation.

Each active thread group evaluates waveforms with respect to two sample times for 32 different slots in parallel. Since all waveform events are accessed in order by each thread and each thread accesses the same event index, the memory addresses by a thread group form consecutive address ranges which allow to coalesce the accesses and maximize their utilization. The signal values sampled by the threads are merged into 64-bit `Long` integers

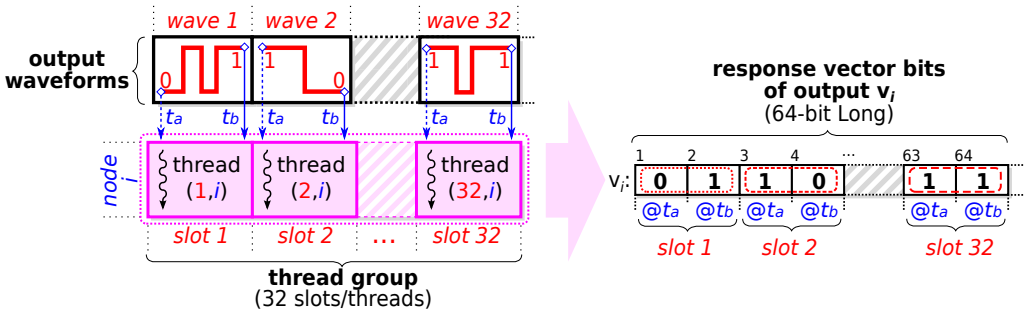


Figure 5.15: Parallel *waveform-to-pattern* conversion in a thread group with each waveform being sampled at two distinct times $t_a = -\infty$ and $t_b = \infty$.

using *parallel reduction* over the shared memory on the SMs [NVI17a]. Eventually, the first thread of the thread group writes the merged result into the pattern memory, which can then be accessed in a compact way by the host system.

5.6 Summary

This chapter presented a simulation model for high-throughput logic level time simulation on graphics processing units (GPUs). In this chapter, the basic modeling of the circuit and signals in time using waveforms were presented including the organization of the data as well as the kernel structures. Compact data structures and algorithms allow to efficiently evaluate the circuit timing with industry-standard timing accuracy [IEE01a, IEE01b], such as individual pin-to-pin delays, rise and fall times as well as pulse-filtering, as used in many commercial simulators. The time simulation algorithm itself is based on a mergesort-based processing of the input waveforms and involves a small working set and memory footprint, which is especially suitable for GPUs. A small table keeps track of signal states and immediate next events in input waveforms, which are evaluated in temporal order until all input events have been processed. The parallel execution is performed on the GPUs by utilizing multi-dimensional thread grids that organize the individual threads as arrays in order to effectively exploit structural and data parallelism. As shown later in this thesis, the modeling and kernel structures provide the foundation for switch level time simulation (cf. Chapter 6) [SHWW14, SW19a], fault simulation (cf. Chapter 7) [SHK+15, SKH+17] as well as for combined multi-level simulation with mixed abstractions (cf. Chapter 8) [SKW18, SW19b].

Chapter 6

Parallel Switch Level Time Simulation on GPUs

This chapter describes the extension of the presented parallel GPU-accelerated time simulation of Chapter 5 for the application to switch level. First, the basic switch level circuit model and the waveform representation are introduced, along with the required data structures for efficiently processing and storing the switch level signal information. Then, the simulation procedure and the signal evaluation are explained on a detailed example.

6.1 Overview

In CMOS technology many electrical effects exhibit impact on the timing behavior, which are not sufficiently captured or not captured at all in logic level simulation using simple static delay models. Although logic level time simulation with rising and falling delays can provide timing-accurate results close to the real propagation delay of signal transitions at single cell input pins [IEE01a], the output switching behavior of the cell can significantly change depending on the applied input.

For example, many cells exhibit *pattern-dependent* (or *data-dependent*) delays, where a signal transition at a cell input pin can result in different propagation delays of the cell depending on the signal values applied at the side inputs [SDC94]. This is due to the fact that the electrical driving capability (for charging or discharging the cell output) depends on the states of the transistors in the cell, which in turn change with the applied inputs. While the topology of the internal transistor netlist of the cells is typically neglected at

logic level, pattern-dependent delays can be reflected in gate timing descriptions by using conditional delays [IEE01a].

Fig. 6.1 illustrates a special case of pattern-dependent delays on the example of a three-input NOR3_X2 cell, where multiple input signals switch at the same time causing a so-called *multiple-input switching* (MIS) effect [MRD92, CGB01]. Here, all input pins of the cell were initialized with zero volts in the beginning leading to a *high* output signal. Then the voltage at the inputs was raised to *high* for one, two and three input pins respectively with the output slope being measured between the interval of 90% to 10% of the voltage range [WH11]. As shown, the more inputs change at a time the steeper the output slope gets (lasting as high as 20.7ps over 10.8ps down to 7.6ps) and therefore the propagation delay decreases as more transistors collectively discharge the output load in parallel. Note that this speed-up in the output switching can vary depending on the timely proximity of the simultaneous input transitions [CS96]. Also, the change in the signal slope also impacts the timing behavior of succeeding cells, as the different PMOS and NMOS transistors begin to switch at different times depending on the steepness [WH11].

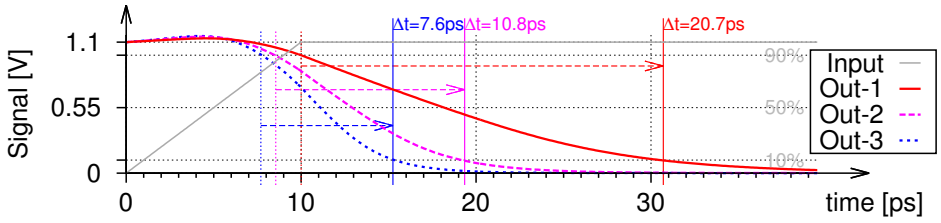


Figure 6.1: MIS effect on the 90/10 output slope of a NOR3_X2 cell [Nan10, ZC06] in SPICE for simultaneous changes at *one* (Out-1), *two* (Out-2) and *three* (Out-3) inputs.

Since lower level information is typically not available in logic level simulation, many effort has been put into representing pattern-dependent delays [SDC94], multiple-input switching effects [MRD92, CGB01, CS96] and also to incorporate the parameter-dependent switching behavior of different signal slopes [BJV06] into gate delay models. However, conventional look-up table based approaches to calculate the non-linear delay behavior [WH11] quickly become impractical due to a large amount of different parameters and conditions that have to be considered (e.g., slope, output load, current charge at cell output, signal values and latest arrival times at side inputs). The complexity rises even further as soon as faults need to be simulated for test validation. Since electrical level

simulation is costly in terms of runtime, recent test approaches [CCCW16] utilize switch level simulation [Bry87] as trade-off in evaluation speed and accuracy [CWC17].
In this chapter, a highly-parallelizable time simulator is presented to evaluate timing of combinational circuits at switch level. The simulator utilizes an intuitive modeling both the functional as well as the timing behavior of CMOS cells, and is able to reflect the aforementioned delay effects allowing to achieve more accurate time simulation on GPUs with unprecedented simulation throughput.

6.2 Switch Level Circuit Model

At switch level, the circuit is typically modeled as a mesh of interconnected MOSFET transistors, each of which is considered as ideal switch [BA04]. Conducting paths in the mesh determine signal values, strengths and charges at signal nodes [Bry84, Hay87]. By abstracting electrical properties of transistors and interconnects (e.g., voltage and current characteristics of transistors, resistances and capacitances), node voltages as well as charging and discharging processes of signal nodes are reflected. This way signal strengths and changes in signal values can be expressed in terms of discrete voltage steps or functions of time [CGK75].
The switch level modeling in this thesis assumes an extended modeling of the combinational circuits as introduced in Chapter 5 for simulation. As a pre-processing, the gates of the circuit netlist are substituted by their respective CMOS standard cell descriptions, which describe the interconnections of the PMOS and NMOS transistors in the cells. The resulting transistor netlist of the circuit is then partitioned into primitives for simulation.

6.2.1 Channel-Connected Components

In switch level simulation, PMOS and NMOS transistors are typically viewed as three-terminal devices in which a *gate* terminal controls the conductance between a *drain* and a *source* terminal that form a bidirectional channel. For the partitioning, the transistor netlist of the design is searched for *channel-connected components* (CCCs) [Bry84, HVC97] as shown in Fig. 6.2. Each CCC connects the VDD and the GND source by meshes of PMOS and NMOS transistors interconnected via their drain and source terminals. Within a CCC

current can flow freely through the drain-source channels. However, no current is allowed to enter or leave a CCC through the insulating dielectric oxide at the gate terminals and currents are only drawn from the associated power supplies. All gate terminals of transistors are associated with CCC inputs and each CCC is assumed to drive a signal output load associated with a node of its transistor mesh. The output is obtained by analyzing the transistor states and hence the conducting paths in the mesh. Depending on the input applied to the transistor gates, the output load of a CCC can be charged (or discharged) through the VDD (or GND) power supply. The accumulated charge at the output load then determines the output voltage and hence the signal value of the CCC.

Fig. 6.2 shows an AOI21 (And-Or-Inverter) cell [Nan10] in a small circuit composed of three CCCs. A lumped output load capacitance C_{load} models the wire to the succeeding CCC and its transistor gate capacitances. The voltage v_o at the output load capacitor C_{load} represents the output signal which is determined by the effective charge in the capacitor. The capacitor can be charged or discharged depending on the state of the transistors in the cell, which in turn is controlled by the input voltages applied at the respective gate terminals.

Given an arbitrary starting node of a combinational circuit netlist, CCC are extracted by traversing along the bidirectional drain-source channels of the transistors and resistive paths from interconnects. Each traversal ends at any power supply VDD, ground GND or gate terminal. Once, the traversal has terminated in all directions from the starting node, the traversed net represents the extracted CCC. The net is removed from the netlist and the extraction process is repeated at the remaining nodes until all nodes have been

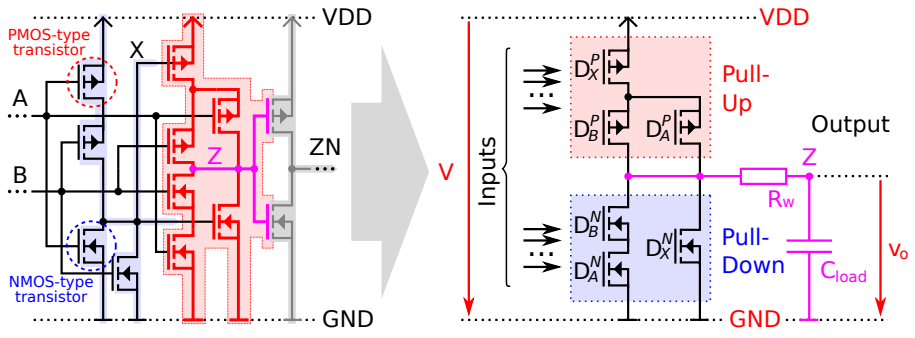


Figure 6.2: *Channel-connected components* in a small transistor-netlist. The red highlighted CCC (left) represents an AOI21_X1 cell [Nan10] (right) with output load C_{load}.

processed. Note that the majority of primitive standard cells found in academic digital standard cell libraries [Nan10, Syn11] already form single CCCs. Yet, certain types of cells (e.g. two-input XOR cell) comprise multiple CCCs [Nan10].

6.2.2 RRC-Cell Model

In this thesis, so-called *Resistor-Resistor-Capacitor* (RRC-) cells are used as basic switch level simulation entities as shown in Fig. 6.3, which provide a simple and intuitive unidirectional model for representing the functional behavior of CMOS technology standard cells [SHWW14]. An RRC-cell covers an individual channel-connected component of the circuit netlist and describes the interconnected PMOS and NMOS transistor mesh.

The description of an RRC-cell consists of a compact device description for each transistor device contained in the respective CCC mesh as well as first-order electrical parameters of the channels and metal interconnects, which are the major contributor to the cell functional behavior and timing [WH11]. The RRC-cell is further connected to power supply and ground potentials. However, for the sake of simplicity, only one power supply VDD $\in \mathbb{R}$ and one ground potential GND $\in \mathbb{R}$ are assumed per cell. All RRC-cells are assigned a *cell type* for which the mesh topology of the transistor interconnections in each cell is uniquely determined. For a given cell type the inputs of the CCC, and hence of the RRC-cell, are assigned to the respective gate terminals of the associated transistors. The output terminal of the cell is associated with a lumped load capacitance C_{load} which is modeled as the sum of the interconnect and gate capacitances in the cell fanout [SHWW14].

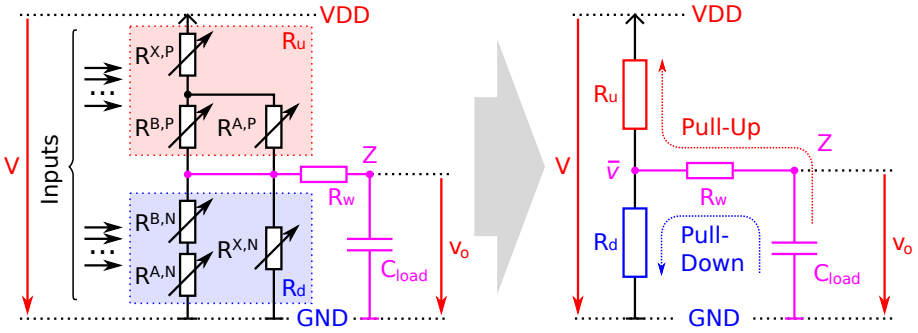


Figure 6.3: First-order equivalent circuit model of the AOI21_X1 cell and RRC-cell showing the input-controlled voltage-divider (R_u, R_d) driving an output load C_{load}.

Each transistor device D of the RRC-cell is viewed as a voltage-controlled resistor $R^D(v) \in \mathbb{R}$ that determines the drain-source conductance of the transistor based on its applied gate voltage $v \in \mathbb{R}$. The representative resistor R^D of device D is modeled by a threshold-based binary switch that is controlled by the voltage level v present at the gate terminal of D. A transistor assumes either a *conducting* state, which is modeled by a *low* drain-source resistance R_{on}, or *blocking* state modeled as high drain-source resistance R_{off}. Both resistances are obtained from voltage- and current analysis of the underlying transistor model cards. Given a threshold voltage V_{th} and a gate voltage v as input, the *state* of the transistor D is described as follows:

$$R^D(v) := \begin{cases} R_{off} & \text{if } (v < V_{th}), \\ R_{on} & \text{else.} \end{cases} \tag{6.1}$$

Each transistor device is modeled using a 3-tuple $D := (V_{th}, R_{off}, R_{on})$ as part of the RRC-cell description. The voltage V_{th} is defined as the input voltage potential at the transistor gate terminal over either VDD (for PMOS types) or GND (for NMOS types) at which the transistors change their *state*.

All PMOS (NMOS) transistors in a standard cell form *pull-up* (*pull-down*) networks that connect the cell output terminal to the VDD (GND) sources with some resistance. From the view of the output terminal, both pull-up network and pull-down network form a voltage divider (R_u, R_d) that drives the output load C_{load} of the RRC-cell with a stationary voltage, where $R_u \in \mathbb{R}$ and $R_d \in \mathbb{R}$ are the net-resistances of the pull-up and pull-down network, respectively. Both parametric resistances of the transistors as well as interconnect resistances can contribute to R_u and R_d. For a given cell input, the *stationary voltage* $\overline{v} \in \mathbb{R}$ of the RRC-cell [SHWW14] is computed as

$$\overline{v} := s \cdot V + \text{GND}, \tag{6.2}$$

with $V := (\text{VDD} - \text{GND})$ as the voltage difference from VDD to GND and $s \in \mathbb{R}$ as the voltage divider ratio (R_u, R_d) defined by

$$s := \frac{R_d}{R_u + R_d}. \tag{6.3}$$

The computation of R_u and R_d plays a major role during the RRC-cell evaluation as the stationary voltage has to be computed after any transistor changes its state. The CMOS standard cells found in current academic standard-cell libraries with design processes down to 15nm [Syn15, Nan14] follow the duality principle of CMOS by the rule of *conduction complements* [WH11], where the transistors in the networks are interconnected either as *series* or in *parallel*, or in an arbitrary combination of both forming more complex meshes. For these nets, *Kirchhoff's laws* for currents (KCL) and voltages (KVL) for parallel and series circuits can be applied to compute the equivalent mesh resistance R_u of the pull-up and R_d of the pull-down network.
For example, in case of the AOI21_X1-cell of Fig. 6.3, the pull-up mesh resistance R_u is actually a series of resistors

$$R_u = R^{X,P} + R' \tag{6.4}$$

where R' substitutes the parallel sub-network composed of device D_A^P and D_B^P, with

$$\frac{1}{R'} = \frac{1}{R^{B,P}} + \frac{1}{R^{A,P}} \Longrightarrow R' = \frac{R^{B,P} \cdot R^{A,P}}{R^{B,P} + R^{A,P}} \tag{6.5}$$

$$\Longrightarrow R_u = R^{X,P} + \frac{R^{B,P} \cdot R^{A,P}}{R^{B,P} + R^{A,P}}. \tag{6.6}$$

Similarly, for the pull-down network R_d the equivalent resistance of the transistor series D_A^N and D_B^N is computed first

$$R'' = R^{B,N} + R^{A,N} \tag{6.7}$$

which is then used to solve the parallel part by

$$R_d = \frac{R'' \cdot R^{X,N}}{R'' + R^{X,N}} = \frac{(R^{B,N} + R^{A,N}) \cdot R^{X,N}}{R^{B,N} + R^{A,N} + R^{X,N}}. \tag{6.8}$$

6.2.3 Switch Level Waveform Representation

The switching process at a cell output from one voltage to another takes time due to various intrinsic and extrinsic capacitances (e.g., diffusion, wire and gate capacitances) which have to be charged or discharged over time [WH11]. The transient response behavior

of cells can be approximated with varying degree of accuracy. The response is typically calculated from differential equations over non-linear models [NP73] with stepwise evaluation of the differential equations [CGK75]. Simplified models exist, with piecewise linear assumptions [Kao92], stepwise constant I/V characteristics [VR91], or piecewise exponential waveforms representing the first-order step response [BMA88] which make evaluations less complex at the sacrifice of accuracy.

Transient Response Modeling

In the RRC-cell model (cf. Fig. 6.3), the voltage at the output load capacitor C_{load} is considered as the output signal of the cell. In order to model a transient response, the first-order electrical parameters of the cell are reduced to a lumped RC low-pass equivalent circuit. The resulting RC circuit is composed of a lumped resistor that includes the effective driving resistance of the voltage divider (R_u, R_d) and the lumped interconnect resistance $R_w \in \mathbb{R}$ of the fanout, as well as the lumped output load capacitance C_{load} which can include diffusion and fanout gate capacitances. When a transistor device in an RRC-cell changes its state, the pull-up (R_u) or pull-down (R_d) resistance of the cell changes and hence the voltage divider ratio s (cf. Eq (6.3)) that determines the stationary voltage $\overline{v}$. This causes the cell output capacitor C_{load} to (dis-)charge via the voltage divider from its current level to voltage $\overline{v}$ with the effective resistance $R_{\mathit{eff}} := (s \cdot R_u)$, which delivers the *time constant* $\tau \in \mathbb{R}$ which characterizes the transient response of the underlying RC circuit:

$$\tau := (R_{\mathit{eff}} + R_w) \cdot C_{load}. \tag{6.9}$$

Note, that only first-order electrical parameters are considered in the RRC-model while second-order capacitances (e.g., gate-source) are neglected. Hence, overshoots in the signal voltages due to trapping of charges are not modeled [WH11]. Therefore, given an arbitrary transistor switch at time $t_i \in \mathbb{R}$, the transient response of the RRC-cell output v_o can be derived from Kirchhoff's Current Law (KCL) following a monotonous [RPH83] exponential function $v_o(t)$ for time $t \geq t_i$:

$$v_o(t) := (v_o(t_i) - \overline{v}) \cdot e^{-\frac{\Delta t}{\tau}} + \overline{v}, \tag{6.10}$$

where $v_o(t_i)$ is the voltage at the capacitor C_{load} at the time t_i and $\Delta t := (t - t_i)$ is the elapsed time after the switch. The voltage $v_o(t)$ then strives asymptotically towards the stationary voltage $\overline{v}$ for $t \to \infty$ and will saturate eventually.
Between any two consecutive transistor switches at times t_i and t_{i+1}, all resistances of transistors and nets in the RRC-cell, and hence the stationary voltage at the voltage divider, remain constant. Therefore, based on Eq. (6.11), the monotonous curve within each interval $[t_i, t_{i+1}]$ of consecutive switches can be described entirely by a tuple composed of three parameters $e_i = (t_i, \overline{v}_i, \tau_i)$ assuming that the value $v_o(t_i)$ at the beginning is known. Such a three-tuple will be referred to as a switch level *event* e_i, whose components are:

- $t_i \in \mathbb{R}$: The time of the *start of a new exponential curve* expressed by the event e_i as a consequence of a transistor switch. This also marks the end of the previous segment (e_{i-1}).

- $\overline{v}_i \in \mathbb{R}$: The *targeted stationary voltage* of the new curve expressed by e_i, which is derived from the resistances of the voltage divider, the supplying voltage VDD and ground voltage GND.

- $\tau_i \in \mathbb{R}$: The *signal slope* of the new curve segment of e_i corresponding to the time constant calculated from the resistances at the voltage divider and the lumped load capacitance.

Fig. 6.4 shows the result of a SPICE transient analysis compared to the approximation of the signal with fitted ideal exponential curve segments. Compared to logic level time simulation, these curves approximate the electrical output behavior quite well and are able to contain sufficient information to express many of the delay effects found in CMOS.

Switch Level Waveform Modeling

The complete switching history of a signal over time is summarized as *switch level waveform* similar to the data structure presented in Chapter 5. Each waveform is modeled by a list of temporally ordered events $\tilde{w} = \{e_1, e_2, ..., e_k\}$ each of which stores the parameters of an exponential curve segment. This way, multiple continuous signal transitions can be modeled over time using a piecewise exponential approximation as depicted in Fig. 6.5.

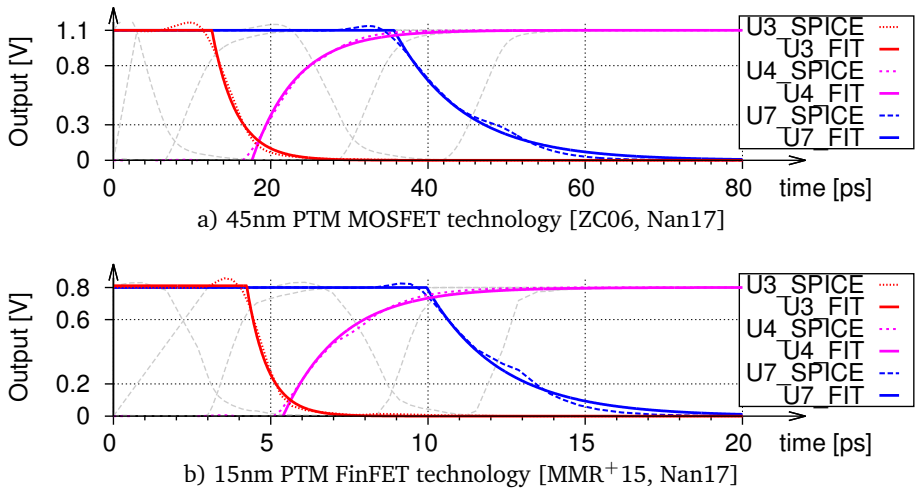


Figure 6.4: Signal transitions of cells from SPICE transient analysis with varying output load (dotted) and fitting with exponential functions (full strokes).

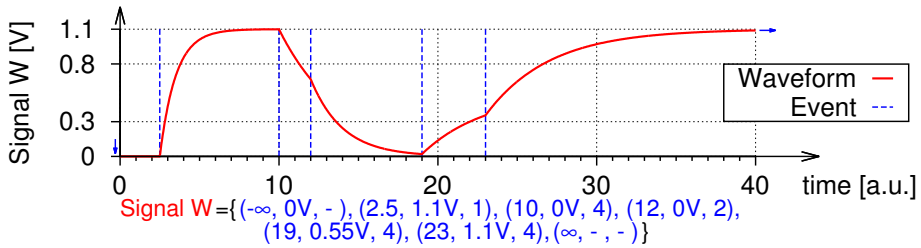


Figure 6.5: Switch level event representation and visualization of an arbitrary signal. "*Don't care*"-values are denoted by '-'.

Besides the ordinary events, each waveform contains an *initialization event* $(-\infty, v_{init}, -)$ at the head of the list that marks a constant initial voltage v_{init} of a cell output (the latter '$-$' denotes a "*don't-care*"-value). In addition, the event list is terminated by a *termination event* $(\infty, -, -)$ with time $t = \infty$ as sentinel to indicate the end of the waveform data structure. The waveform signal is assumed to approach the stationary voltage $\overline{v}$ of the last ordinary event processed. Hence, fully detailed signal information is processed with smallest memory overhead, as no sampling of signal values is necessary, allowing for an efficient time- and value-continuous evaluation even for longer signal histories.

Each switch level waveform $\tilde{w}$ is evaluated by a switch level *waveform function* $\tilde{\boldsymbol{w}} : \mathbb{R} \to \mathbb{R}$ that delivers the voltage value present at the given point in time $t \geq 0$ in the waveform $\tilde{w}$.

Following Eq. (6.10), the function is defined recursively as

$$\tilde{\boldsymbol{w}}(t) := (\tilde{\boldsymbol{w}}(t_i) - \overline{v}_i) \cdot e^{-\frac{\Delta t}{\tau_i}} + \overline{v}_i, \tag{6.11}$$

with

$$(t_i, \overline{v}_i, \tau_i) := \max_{j \in \mathbb{N}} \{(t_j, \overline{v}_j, \tau_j) \in \tilde{w} | t_j < t\}, \tag{6.12}$$

where $\tilde{\boldsymbol{w}}(t_i)$ expresses the signal value at the time of the previous transient event $e_i := (t_i, \overline{v}_i, \tau_i) \in \tilde{w}$ in the waveform before t and $\Delta t := (t - t_i)$ as the elapsed time in the current exponential curve segment.

6.2.4 Modeling Accuracy

In contrast to the logic level simulation model, the presented switch level models all signals continuous in time and value using the presented voltage waveforms. Furthermore, many of the aforementioned CMOS-related delay effects are reflected:

Signal slopes are implicitly covered by the first-order transient response (cf. Eq. (6.10)) of the RC-circuit equivalent that is built from the RRC-cell parameters.

Pattern-dependent delays are directly captured by the transistor mesh structure of each RRC-cell type, since the transient response depends not on a single input, but on the states of *all* transistors in the mesh.

Multiple-input switching is implicitly considered by the pattern-dependent delays, since consecutive input events cause immediate updates in the output transient regardless of the transition proximity.

Hazards and pulse filtering is implicitly performed at an RRC-cell output when an event transient with a shallow signal slope is prematurely interrupted by a succeeding event before the signal can fully reach the stationary voltage.

6.3 Switch Level Simulation Algorithm

The general RRC-cell evaluation algorithm [SHWW14] processes all input waveforms of a single RRC-cell and generates a new output waveform that describes the output voltage

over time. The algorithm determines all discrete transistor switching events from the input waveforms to manipulate the cell internal states changes and transient responses to produce a new output waveform.

6.3.1 RRC-Cell Simulation

Algorithm 6.1 proposes an efficient solution for tackling the evaluation of RRC-cells on GPUs. It outlines the evaluation done by a single thread, which can operate freely and data-independent of other threads. The algorithm assumes that all events in waveforms are already sorted temporally in order to allow processing of the input waveforms within a single pass using a merge-sort approach. A new output waveform is computed with all events already sorted in order, hence sustaining the premise for subsequent RRC-cell evaluations. By processing switches in a merge-sort fashion, only single reads and writes are required per event and transistor switch processing, which only use cell-local data and thus allow to keep the working set of the thread as small as possible.
The evaluation process is controlled by a time window $[t_{min}, t_{max}] \subseteq \mathbb{R}$ defined by two time points. A time window *slides* over all the input events by moving the boundaries further ahead as soon as inputs are processed. When the earliest switch has been processed, the lower boundary t_{min} is increased, while the upper boundary t_{max} is moved forward as soon as a new event is loaded from the memory. The local memory of a thread consists mainly of two tables to track the state of the cell within $[t_{min}, t_{max}]$: the *event table* and the *device table*. While the event table contains the current curve parameters of all the input waveforms and describes the voltage change in the current time window, the device table keeps track of all transistor states at time t_{min} and holds the switching times until t_{max}.
During simulation of an RRC-cell the events of all inputs are processed in temporal order. To determine transistor switches for each curve segment $[t_i, t_{i+1})$ of an input waveform, the algorithm looks for possible intersection points with a threshold value V_{th} of an associated transistor device D. Since each curve segment expressed by an event $e_i \in \tilde{w}$ is *monotonous* in value, either exactly one or no solution of an intersection point $x \in [t_i, t_{i+1})$ exists, which is determined by:

$$x := t_i - \tau_i \cdot \log\left(\frac{V_{th} - \overline{v}_i}{v(t_i) - \overline{v}_i}\right). \tag{6.13}$$

Algorithm 6.1: Main RRC-cell evaluation algorithm (single thread)

Input: RRC-cell description of node n with device parameters D, output load C_{load} voltage potentials VDD and GND, input waveforms w_i for all inputs $i \in I$.
Output: Event sequence of the switch level output waveform of n.

```
// A.1 load initializing events
foreach line in event table do
    Load initial event (t_i, v̄_i, τ_i) and t_{i+1} from memory
    // Note: t_i and τ_i undef. (cf. Sec. 6.2.3)
    v_i := v̄_i and v_{i+1} := v̄_i
end
// A.2 determine initial states of all internal transistors
foreach column k in device table do
    R_k^D := R^D(v_i from event table), using Eq. (6.1)
    t_next^k := ∞
end
// A.3 curve interval bounds until first switch
t_min := −∞ and t_max := min({t_{i+1} in event table})
// A.4 calculate R_u, R_d and store initial output event
Output initialization event e = (−∞, s · V + GND, −)  // with s as the voltage divider ratio
// B. process switching of input events in temporal order
while (t_max ≠ ∞) do
    // B.1 determine time of next event
    t_min := t_max
    // process event(s) at the time t_min and fetch data of next event
    foreach line in event table with t_{i+1} = t_min do
        Set v_i := v_{i+1} and t_i := t_{i+1}
        Load next τ_i, v̄_i and t_{i+1} from memory
        Calculate v_{i+1} using Eq. (6.11)
    end
    // update time of next event
    t_max := min({t_{i+1} in event table})
    // compute time of next transistor switches for all inputs
    foreach column k in device table do
        x := t_i − τ_i · log((V_th^k − v̄_i) / (v_i − v̄_i))  // determine threshold intersection time
        if t_min < x ≤ t_max then
            t_next^k := x
        else
            t_next^k := ∞
        end
    end
    // B.2 update network resistance states R_u, R_d and add output events accordingly
    foreach column k in device table with t_next^k < ∞ in ascending order of t_next do
        R_k^D := R^D(u_{i+1} from event table), using Eq. (6.1)
        Output new event e = (t_next^k, s · V + GND, s · R_u · C)
        t_min := t_next^k
    end
end
// C. set termination event
Output new event e = (∞, −, −)
```

In case the resulting intersection point lies within the time interval of the current curve segment $x \in [t_i, t_{i+1})$, the input signal crosses the V_{th} voltage level which causes a switch of the associated transistor and hence produces a new event in the output waveform. If no such solution exists, the boundaries of the time window are shifted forward and the event processing is continued for the next input segment until the waveform terminates.

Execution Example

The execution of the cell evaluation algorithm is demonstrated on a small step-by-step example involving a two-input NOR cell with the input transitions as shown in Fig. 6.6. The figure shows detailed timing and voltage information as computed by the RRC-cell evaluation algorithm. The falling transition at one input and the rising transition at the other create a hazard at the cell output. The intermediate output results after each step of the evaluation as well as the internal states of both the event table and the device table during evaluation are summarized in Table 6.1 and Table 6.2.

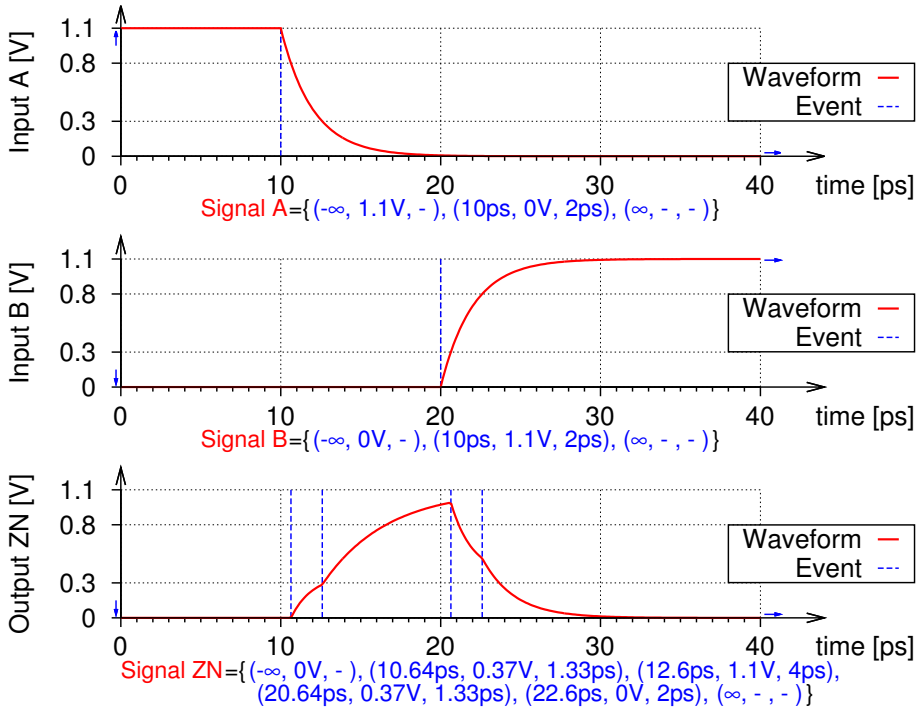


Figure 6.6: Input stimuli and computed output waveform of a two-input NOR-cell. All events are marked as blue dashed lines. [SHWW14]

For the sake of simplicity, all transistors are assumed to have an internal drain-source resistance of $R_{off} := 20\text{M}\Omega$ in blocking and $R_{on} := 2\text{k}\Omega$ in conducting state. The supply voltage is VDD $= 1.1$V over the ground level GND $= 0$V. The threshold voltages of the PMOS and NMOS transistors have been chosen as $V_{th}^{P} := -0.3$V and $V_{th}^{N} := 0.3$V, which corresponds to intersection points for the input signals at voltages 0.8V and 0.3V respectively. Furthermore, the RRC-cell output terminal drives a lumped load capacitance of $C_{load} := 1$fF.

The *event table* is shown in Table 6.1, which also summarizes the performed steps during the evaluation of the example circuit in the first column. Each row in the table (Col. 2–6) contains a line for each input of the cell. A line contains the signal voltage v_i (Col. 2) at the time the current event to be processed (Col. 3–5) starts. The data of the current event e_i contains the start time t_i (Col. 3) of the curve segment, as well as the targeted stationary voltage $\overline{v}_i$ (Col. 4) and the time constant τ_i (Col. 5). Further, each line contains the pre-fetched time t_{i+1} (Col. 6) of the next event. The boundaries of the sliding time window are given in columns 7 and 8.

The *device table* is summarized in Table 6.2 in each row. It contains a column for each transistor in the cell (Col. 2–5, four in total in the NOR2-cell example) with two lines for each entry. While the line R^D holds the resistance of the transistors at time t_{min}, the row t_{next} contains the time of the next switch of the transistor within the interval of the sliding window $[t_{min}, t_{max}]$. If the transistor does not switch within the interval, the value of t_{next} is ∞. Any actions taken after a step has been performed are shown in the last column.

Initialization (Step 1): In Algorithm 6.1 the initialization events of the input waveforms of signals A and B are loaded into the columns 3–5 (t_i, $\overline{v}_i$, τ_i) of the event table (Alg. lines 1–15). The evaluation of the time window starts at $t_{min} := \infty$. These initialization events are not ordinary events and they only describe the initial signal voltages (for $t \to -\infty$), thus, the voltages v_i and v_{i+1} (at time of the next event) are initialized by $\overline{v}_i$. Meanwhile, the times t_{i+1} of the next events in each input waveform are loaded and the upper bound of the time window t_{max} is updated accordingly and set to the minimum of these times. The time window $[t_{min}, t_{max}]$ is now fully specified, and within this initial window all input waveforms sustain their initial signal value. These initial voltage values are used to determine the initial states of all the transistors in the cell (line 8–11) in

Table 6.1: Event table with waveform data during the evaluation of the NOR-cell example (updates are highlighted).

Current Step[1]		Input	Event Table					t_{min}[7]	t_{max}[8]
			v_i[2]	(t_i[3],	$\overline{v}_i$[4],	τ_i[5])	t_{i+1}[6]		
1	load initial events ($e_i = 0$), initialize R^D, $t_{min} := -\infty$.	A: B:	**1.1V** **0.0V**	$-\infty$ $-\infty$	**1.1V** **0.0V**	– –	10ps 20ps	$-\infty$	**10ps**
2	advance input A ($e_A := 1$), $t_{min} :=$ 10ps, update t_{next}.	A: B:	1.1V 0.0V	**10ps** $-\infty$	**0.0V** 0.0V	**2ps** –	∞ 20ps	**10ps**	**20ps**
3	process D_A^P switch, $\rightarrow$ generate output event.							**10.64ps**	20ps
4	process D_A^N switch, $\rightarrow$ generate output event.							**12.60ps**	20ps
5	advance input B ($e_B := 1$), $t_{min} :=$ 20ps, update t_{next}.	A: B:	1.1V 0.0V	10ps **20ps**	0.0V **1.1V**	2ps **2ps**	∞ ∞	**20ps**	∞
6	process D_B^N switch, $\rightarrow$ generate output event.							**20.64ps**	∞
7	process D_B^P switch, $\rightarrow$ generate output event.							**22.60ps**	∞
8	$t_{min} := \infty$ $\rightarrow$ terminate.							∞	∞

Table 6.2: Device table with transistor states and threshold intersection times as well as simulation output.

Current Step[1]		Device Table				Action Taken[6]
		$D_A^{P\,(2)}$	$D_B^{P\,(3)}$	$D_A^{N\,(4)}$	$D_B^{N\,(5)}$	
1	R^D: t_{next}:	**20MΩ** ∞	**2kΩ** ∞	**2kΩ** ∞	**20MΩ** ∞	output ($-\infty$, 0.0V, –)
2	R^D: t_{next}:	20MΩ **10.64ps**	2kΩ ∞	2kΩ **12.60ps**	20MΩ ∞	
3	R^D: t_{next}:	**2kΩ** **∞**	2kΩ ∞	2kΩ 12.60	20MΩ ∞	output (10.64ps, 0.37V, 1.33ps) set $t_{min} :=$ 10.64ps
4	R^D: t_{next}:	2kΩ ∞	2kΩ ∞	**20MΩ** **∞**	20MΩ ∞	output (12.60ps, 1.1V, 4.00ps) set $t_{min} :=$ 12.60ps
5	R^D: t_{next}:	2kΩ ∞	2kΩ **22.60ps**	20MΩ ∞	20MΩ **20.64ps**	
6	R^D: t_{next}:	2kΩ ∞	2kΩ 22.60ps	20MΩ ∞	**2kΩ** **∞**	output (20.64ps, 0.37V, 1.33ps) set $t_{min} :=$ 20.64ps
7	R^D: t_{next}:	2kΩ ∞	**20MΩ** **∞**	20MΩ ∞	2kΩ ∞	output (22.60ps, 0.0V, 2.00ps) set $t_{min} :=$ 22.60ps
8	R^D: t_{next}:	2kΩ ∞	20MΩ ∞	20MΩ ∞	2kΩ ∞	output (∞,–,–)

order to fill the R^D rows of the device table with the corresponding resistance values (cf. Eq. (6.1)). In addition, the t_{next} row is initialized with ∞, since all input values remain stable until the end of the time window t_{max}. The ratio of the voltage divider is then computed from the transistor resistances according to the cell-internal transistor network topology, after which the resulting output voltage of the cell is computed (line 15) and a first output event can be written to the output waveform. Note that, like in the input waveforms, this first output serves for initialization purposes only as it is not considered as an ordinary curve segment.

Advance window (Step 2a): Everything until time t_{max} has been processed and the lower bound of the time window t_{min} is moved forward to t_{max} (line 19). In line 21–25, the earliest next events that defined the previous t_{max} are now loaded into the event table (t_i, $\overline{v}_i$, τ_i) again, as well as the time of the succeeding event t_{i+1} (row of input A in this case). After fetching the next events, the upper bound t_{max} of the time window is updated to $t_{i+1} := 20$ps (line 27). Since the sliding time window has progressed, a new output voltage v_i is calculated using Eq. (6.11) for time t_{min}.

Compute transistor switch times (Step 2b): As a next step, for each entry t_{next} in the device table the time points of transistor threshold crossings has to be determined for the current curve segments (lines 29–36). Since the represented exponential curve segments are either (strong) monotonously increasing or decreasing, each curve can cross a certain transistor threshold and cause it to switch states at maximum once per event. Whenever there is no switch at a transistor in the current time window $[t_{min}, t_{max}]$, then the respective switching time is $t_{next} = \infty$.

Generate output events in order (Step 3 and 4): The earliest transistor switch identified at time t_{max} will be executed (multiple transistors can be processed if they occur at the same time) and its associated resistance R^D will be updated (lines 38–42). Each transistor switch causes the internal resistances of the RRC-cell to change, such that the voltage at the internal voltage divider shifts and the cell output will transition to a new level. Thus, after updating the device table with the new transistor states, the resistances of the transistor networks are updated and ratio of the voltage divider is calculated again to obtain the new stationary voltage $\overline{v}$ as well as the time constant τ. As a result, a new event $e := (t_{next}, \overline{v}, \tau)$ is added to the output waveform (line 40) after which the lower

bound t_{min} advances to t_{next}. In the example, the switch of device D_A^P at time 10.64ps is consumed first, since the input signal A crosses the threshold at 0.8V, and then the switch of D_A^N at 12.60ps is processed after reaching 0.3V.

Process remaining events (Step 5 to 7): All events in the time window $[t_{min}, t_{max}]$ have been processed and the event table is now updated similar to step 2a for row B. With the rising input signal, the loaded event crosses the threshold voltages of the transistors in reverse order within the time window $[20\text{ps}, \infty]$. First, the NMOS transistor D_B^N switches after the input crosses 0.3V with the PMOS transistor D_B^P following at 0.8V. In steps 6 and 7, the corresponding events are added to the output as shown in steps 3 and 4.

Step 8 (Termination condition): After t_{max} has reached ∞, all events in the input waveforms have been processed which initializes the termination routine. The output waveform is terminated by adding a *termination event* $(\infty, -, -)$. As a result, a voltage waveform with a pulse is computed. Short pulses in the circuit might get filtered by subsequent RRC-cell stages, due to the implicit pulse filtering introduced by the slopes of the signal transitions.

The changes in the RRC-cell internal pull-down and pull-up network resistances as well as the stationary voltage during the evaluation are shown in Fig. 6.7. Again, these parameters only have to be computed with the switching of transistor states. As shown, the pull-up and pull-down resistances in the RRC-cell are piece-wise constant and they mutually follow the duality principles of the rules of *conducting complements* in CMOS [WH11]. Due to the different switching times of the PMOS and NMOS transistors, short time frames exist (around 10.64–12.60ps and 21.64–22.60ps), where both meshes exhibit a high conductance shorting VDD and GND. During this time, the RRC-cell output targets an intermediate voltage level.

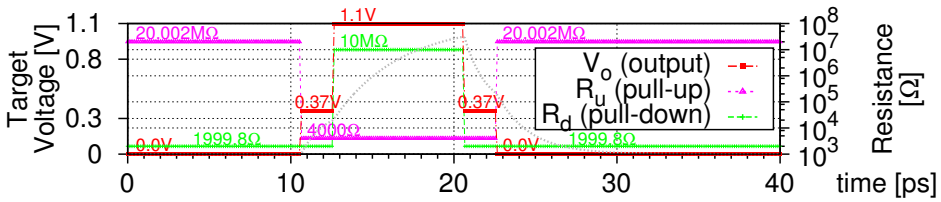


Figure 6.7: Time-continuous simulation state with stationary voltage $\overline{v}_o$ (left axis), pull-up and pull-down resistances R_u and R_d (right, log-scale) during evaluation of the NOR cell.

6.4 Implementation

In order to enable an efficient simulation on GPUs the circuit netlist is partitioned into RRC-cells which are levelized to resolve any data-dependencies during simulation of the levels from circuit inputs to outputs. Similar to Chapter 5, structural parallelism is exploited by processing mutually data-independent RRC-cells after levelization in parallel. In addition data-parallelism is provided from the delay test stimuli, as they are also considered independent. Thus, each thread is able to operate on an independent cell for an input stimuli, such that the multi-dimensional GPU kernel organization can applied to switch level as well. Note that the parallel threads, that compute an RRC-cell for different stimuli, always compute the same functions but for different data, hence execution flow uniformity is widely sustained. Since all threads act independently of each other, costly synchronizations are avoided [OHL+08, GK08].
The RRC-cell parameters are extracted from *detailed standard parasitics format* (DSPF) files, which are obtained from layout synthesis [IEE10], the structure of the respective cell transistor netlists given in the standard-cell libraries [Nan10] as well as the SPICE transistor model cards [ZC06, MMR+15, Nan17]. SPICE transient analyses are used to extract the threshold voltages and the first-order parameters. All RRC-cell parameters descriptions and waveform descriptions are expressed in 32-bit single-precision floating point numbers and allow for a continuous parametrization with machine-precision.

Input Assignment

While circuit inputs can be manually assigned voltage waveforms, the presented simulator generates input waveforms from delay test vectors provided by *Automatic Test Pattern Generation* (ATPG) tools. During the process, all specified input assignments $v \in \boldsymbol{B}_2$ of the test vectors are mapped to corresponding voltages by the following function:

$$val_{\boldsymbol{B}} : \boldsymbol{B}_2 \to \mathbb{R}, val_{\boldsymbol{B}}(v) := \begin{cases} \text{VDD} & \text{if } (v = 1), \\ \text{GND} & \text{else.} \end{cases} \tag{6.14}$$

The generation of a switch level waveform $\tilde{w}_i$ from a delay test assigned to a circuit input $i \in I$ is shown in Algorithm 6.2. For common transition delay test patterns, this

results in the provision of the initialization vector value $v_i \in \boldsymbol{B}_2$ and the propagation vector $v_i' \in \boldsymbol{B}_2$ of the tests. Each input can be assigned a time constant $\tau_i \in \mathbb{R}$ that can be derived from the driving RC-circuit [Elm48, WH11] to express an initial signal slope. For $\tau_i \to 0$, the resulting signal transition closely resembles the logic transition with negligible error [SKW18].

Algorithm 6.2: Mapping of a delay test vector to a switch level waveform.

Input: input assignments (v_i, v_i') for circuit input $i \in I$, time constant τ_i
Output: switch level input assignment $\tilde{w}_i$ at input i

```
1  Initialize empty waveform w̃ := ∅.
2  Append new event e = (−∞, val_B(v_i), −) to w̃_i.  // initialization event (the last parameter is omitted)
3  if (v_i' ≠ v_i) then
4  |  Append new event e = (0, val_B(v_i'), τ_i) to w̃_i.  // propagation vector event
5  end
6  Append new event e = (∞, −, −) to w̃_i.  // termination event
7  return w̃_i
```

Stationary Voltage Computation

The computation of the pull-up and pull-down network resistances R_u and R_d is performed thread-locally and immediately after a new transistor switch has occurred. The general structure of these kernels is outlined on the example of the pull-up net resistance R_u in Algorithm 6.3, which uses the current voltage levels at the cell inputs and Eq. (6.1) to compute the resistance R^D of each transistor device.

The formulas of the RRC-cell types are derived from the respective topology of the transistor netlists of the standard cells using from Kirchhoff's Current Law (KCL). This is ap-

Algorithm 6.3: Kernel structure for computing the pull-up resistance R_u.

Input: RRC-Cell n with device descriptions D, voltage levels $V := \{v_A, v_B, v_X, ...\}$ at each input $A, B, X, ...$
Output: equivalent resistance R_u

```
1  switch cellType(n) do
2  |  case NOR2 do
3  |  |  R_u := R^{A,P}(v_A) + R^{B,P}(v_A)
4  |  end
5  |  case AOI21 do
6  |  |  R' := (R^{B,P}(v_B) · R^{A,P}(v_A)) / (R^{B,P}(v_B) + R^{A,P}(v_A))
7  |  |  R_u := R^{X,P}(v_X) + R'
8  |  end
9  |  ...
10 end
11 return R_u
```

plicable to most CMOS standard-cells (e.g., AND, NAND, XOR, AOI22,...) found in digital standard cell libraries [Nan10, Syn11] including their driving strengths variants. In a similar fashion, the kernel for computing the pull-down net resistance R_d is computed. All of the operations in the kernel require extensive use of floating-point operations as well as transcendental functions. Yet, they can be efficiently handled by the high arithmetic computing throughput of the GPUs.

6.5 Summary

In this chapter, the first high-throughput GPU-accelerated switch-level timing simulation of CMOS circuits was presented [SHWW14, SW19a]. The simulation utilizes RRC-cells to model the functional and timing behavior of CMOS cells with transistor granularity based on first-order electrical effects found in CMOS technology. A sophisticated waveform representation models voltage waveforms to represent continuous charging and discharging processes of the cells. This way, transition ramps, pattern-dependent delays, multiple input switching effects and implicit glitch-filtering are reflected during simulation, which achieve a more accurate representation of the timing behavior than compared to conventional timing simulation at logic level.

Chapter 7

Waveform-Accurate Fault Simulation on GPUs

This chapter presents an extension of the GPU-accelerated parallel timing simulation for simulation of *small delay faults* [SHK+15, SKH+17] at logic level and *parametric faults* at switch level [SW16, SW19a]. A highly parallel waveform-accurate evaluation of the faults is enabled by exploiting additional parallelism from faults (*fault-parallelism*) with efficient fault injection and sophisticated syndrome calculation.

7.1 Overview

The overall flow of the implemented parallel fault simulation approach is outlined in Fig. 7.1. The flow itself is mainly composed of two parts involving a *pre-processing* (Step 1–3) and the *simulation* (Step 4–6), which are applicable to all of the simulation methods presented in this thesis.

During the pre-processing, the netlist is read in, topologically ordered and being assigned its respective circuit parameters (i.e., timing or first order parameters) (1). The provided fault set is *collapsed* (2) by removing equivalent faults in order to reduce the overall simulation overhead. The remaining fault set is then partitioned into suitable *fault groups* for parallel simulation (3) by a grouping heuristic as presented in [SKH+17]. During the actual simulation (Step 4–7), the obtained fault groups are processed one after another. First, the faults are *injected* into the circuit (4). Then, the main waveform-processing kernel is called (6) that performs a timing-accurate simulation of the *faulty* circuit copies. The waveform-processing is followed by an output evaluation kernel (7) that computes

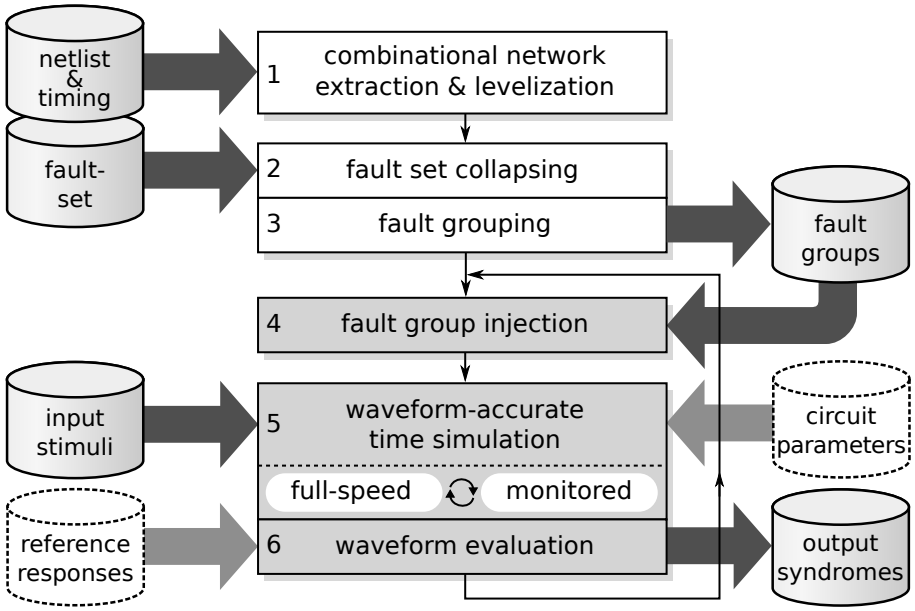


Figure 7.1: Overall flow-chart of the fault simulation. Shaded steps are parallelized.

the *syndromes* for determining the fault detection. Reference responses of a *good-value* simulation for comparison can be provided externally (e.g., from fast untimed logic-level simulation). Yet, for some fault models, both fault- and good-value simulation can be performed simultaneously without introducing any additional overhead [SHK+15].

In the following, the general modeling of timing-related faults at logic-level and parametric faults at switch level, as well as their general injection methods are introduced, followed by the syndrome calculation for determining and assessing the fault detection. Finally, the fault grouping algorithm for parallel fault simulation on GPUs is presented.

7.2 Fault Modeling

Here, a *fault* describes the deviation from the nominal functional or timing behavior of a structure in the circuit due to a delay- or parametric defect. The modeling of faults as such is based on manipulation of the nominal circuit parameters. Independent of the abstraction level and regardless of the implemented fault model, all faults are represented in this work as a tuple:

$$f = (loc, \{\delta^0, \delta^1, ...\}). \tag{7.1}$$

composed of a *fault location* (loc) and a set of fault parameters $\{\delta^0, \delta^1, ...\}$, that describe the *fault size*. The *fault location* loc describes the position of a fault and can refer either to a particular pin of a node at logic level or address a specific parameter in an RRC-cell at switch level. The *fault size* refers to the actual fault effect at the given fault location. The number of parameters in the fault size description and their semantics varies for the different abstraction levels and fault models as presented in the following.

7.2.1 Small Gate Delay Fault Model

A small (gate) delay fault is typically considered as a lumped manifestation at a gate pin that introduces an additional delay [IRW90, RMdGV02, CHE+08] and delays signal propagation by a specified time. Although the additional delay is considered much smaller than the global clock period (as opposed to transition faults [WLRI87]), the fault can lead to errors when propagating along long paths under at-speed operating conditions.
In this work, a *small (gate) delay fault* is represented according to Eq. (7.1) by a tuple:

$$f := (loc, \{\delta_r, \delta_f\}) \tag{7.2}$$

where the location loc can refer to either input or output pin of any node, $\delta_r \in \mathbb{R}$ is the *rising delay* and $\delta_f \in \mathbb{R}$ is the *falling delay* of the fault. Small delay faults can be specified to affect *both* transition polarities at the same time ($\delta_r = \delta_f > 0$), or a single polarity only (*rising* or *falling* for which the delay of other is set to zero).
Since the *time spec* delay annotations describe the delay of a node for its input pins only (cf. Chapter 5), small delay faults at output pins of a node $n \in V$ cannot be modeled directly. Instead, small delay faults at node outputs therefore need to be mapped to the input pin *time spec* tuples $ts_i := (d_r^i, d_f^i) \in TS_n$ resulting in a *multiple* fault distributed over the node inputs. A small delay fault at a node output affects all events caused by incoming signal transitions at its inputs, thus, it behaves as if the fault is present at all inputs. Let t_1 be the time of an event at an input with propagation delay $d \in \mathbb{R}$ that causes an output event at time $t_2 = (t_1 + d)$. Assume a small delay fault of size $\delta \in \mathbb{R}$ at the output, that delays the output event to $t_2' = (t_2 + \delta)$. Then $t_2' = (t_2 + \delta) = (t_1 + d) + \delta = (t_1 + \delta) + d = (t_1' + d)$, where $t_1' = (t_1 + \delta)$ corresponds to the input event that is delayed by the fault size δ. As

shown, the behavior of the fault at the output and the mapping to the input faults are identical since they lead to the same transition times.

In case the small delay fault has different *rising* and *falling* delays, the mapping to the input pins has to take into account the *(non-)inverting* property of a node type as shown in Fig. 7.2. If the target node is of an *inverting* type (e.g., NOT, NAND, NOR), the rising and falling delays of the output fault description have to be swapped for the mapping to the input pin time spec. As opposed to *non-inverting* types (e.g., BUF, AND, OR), *rising* (*falling*) transitions in the input of inverting nodes always cause a *falling* (resp. *rising*) transition at the output, due to the inversion property.

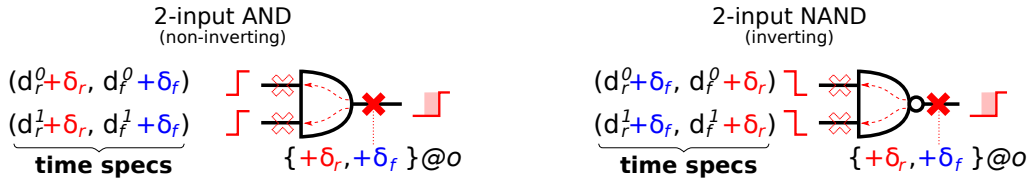


Figure 7.2: Mapping of a small delay fault $f := (o, \{\delta_r, \delta_f\})$ at a node output o the input pin *time specs* of a non-inverting AND- (left) and inverting NAND-gate (right).

7.2.2 Switch-Level Fault Models

The implemented switch level simulation approach allows to model low-level parametric and parasitic low-level faults based on the first-order electrical parameters of RRC-cells [SW16]. In this work, cell-internal *resistive opens* and *bridges* in either the *pull-up* or *pull-down* network topology are considered, as well as *capacitive* and *voltage-related* faults. For many of these faults models, the low-level defect information and their mechanisms can be directly employed into the simulation model with little or no abstraction at all. Therefore, the common modeling restrictions that are typically faced on logic level and which require severe abstraction of functional and timing behavior can be avoided [Wad78, CHE$^+$08, Wun10]. Methods for layout-aware extraction of relevant defects and the logical and topological abstractions to fault types at either logic or electrical level have been proposed in *inductive fault analysis* (IFA) [SMF85, FS88, STO95]. In the following, different fault models of the RRC-cell-based switch level simulation and their realization will be presented.

Resistive Opens and Shorts

The impact of faults in the resistive parameters of a cell depends on the location and parametric size of the fault. Their behavior range from a simple resistive open (wire) impacting the timing [RMdGV02] up to a (transistor) open fault, that can also change the functional behavior of the affected cell [Wad78, HM91]. In certain CMOS cell types, transistor and cross-wire opens can disconnect meshes within the pull-up or pull-down network topology which can cause *floating* outputs [HM91, Kon00, HS15]. The activation of such faults can be invalidated by hazards in the circuit [Sin16], which therefore requires timing-accurate and glitch-aware simulation methods for validation.

In this work, a *resistive fault* in a RRC-cell is described by a tuple $f := (loc, \{\Delta R_f\})$ composed of a fault location loc and a *fault size* $\Delta R_f \in \mathbb{R}$. The fault location refers to a *resistive parameter* $R \in \mathbb{R}$ of the RRC-cell description, which can be a *blocking resistance* (R_{off}) or *conducting resistance* (R_{on}) of one of the transistor devices $D := (V_{th}, R_{off}, R_{on})$, or point to a *static resistance* (from vias or wires) within an RRC-cell. The deviation $\Delta R_f \in \mathbb{R}$ of the selected parameter is given in Ohms, which changes the targeted parameter to $R' := (R + \Delta R_f)$. This allows for modeling of the following common fault types:

- *transistor open fault*, if loc refers to the *conducting resistance* R_{on} of a transistor device D and $\Delta R_f > 0$,
- *shorted transistor fault*, if loc refers to the *blocking resistance* R_{off} of a transistor device D and $\Delta R_f < 0$,
- *cross-wire open fault*, if loc refers to a low-ohmic *static resistance* and $\Delta R_f > 0$, and
- *cross-wire bridge*, if loc refers to a high-ohmic *static resistance* and $\Delta R_f < 0$.

As for the bridges, the RRC-cell transistor meshes have to be extended by additional wires that connect mesh-internal nodes via static resistors. In the *fault-free* case, these *interconnections* have no significant influence on the timing or functional output ($R \approx \infty$ Ohms). However, if the resistance is lowered by a bridge fault ($R' \approx 0$ Ohms), both the functional and timing behavior of the cell can be impacted.

Capacitive Faults

Defects with additional metal in interconnect wires or polysilicon at receiving gate terminals can cause an increase in the cell fanout capacitance. Compared to resistive faults in cells, capacitive faults do not affect the functional behavior but rather contribute to the temporal behavior of cells [Elm48, RPH83].

In this work, a *capacitive fault* is modeled by a tuple $f := (loc, \{\Delta C_f\})$, that introduces a lumped parasitic capacitance $\Delta C_f \in \mathbb{R}^+$ to the RRC-cell output load capacitance C_{load}, such that $C'_{load} := (C_{load} + \Delta C_f)$. The changes of the load capacitance C'_{load} cause an increase in the *time constant* τ (cf. Eq. (6.9)) which impacts the timing of the output transient (cf. Eq. (6.11)) by flattening the slope.

In [CHE+08], resistive and capacitive first-order electrical parameters were used to calculate *small delay faults* of fixed size. However, the delay introduced by the capacitive faults at switch level is also pattern-dependent, since the time constant (and hence the resulting delay) can vary for different side inputs. Hence, by explicit modeling of capacitive faults a more realistic timing behavior of lumped capacitive faults in CMOS circuits is reflected.

Voltage-related Faults

Transistor aging effects, such as NBTI or HCI [LGS09, GSR+14] impact the switching delay of a transistor by shifting its threshold voltage. Furthermore, fluctuations in the power supply grid of a circuit (e.g., caused by IR-drop and ground bounce) [SGYW05] cause significant delays. In this work, the RRC-cell voltage parameters allow for modeling of voltage-related faults.

A voltage-related fault is represented as a tuple $f := (loc, \{\Delta V_f\})$, where the fault location loc refers to a voltage parameter of an RRC-cell (e.g., VDD, GND or the threshold V_{th} of transistor) and $\Delta V \in \mathbb{R}$ refers to the deviation of the respective voltage level in Volts.

In case the fault location loc refers to a transistor threshold voltage V_{th}, f is considered an *aging fault* that models a shift in the threshold voltage thereby delaying the transistor switching. For NMOS transistors, V_{th} is usually positive and therefore must be increased by ΔV_f to inject the fault ($V'_{th} := V_{th} + \Delta V_f$). The threshold of PMOS transistors is usually negative and injection is performed by decreasing V_{th} ($V'_{th} := V_{th} - \Delta V_f$).

Similarly, power supply faults in VDD and GND parameters of an RRC-cell are modeled. A drop in VDD (or increase in GND) can delay the transistor switching, as all transistor thresholds are specified with respect to their associated ground potential. In addition, power supply faults can also lead to a functional fault, as the output strength of the signal is lowered.

7.2.3 Fault Injection

The *injection* of any of the presented faults is conducted by modifying the node- or timing-descriptions according to the behavior specified by the underlying fault models. For the injection of a fault, the corresponding values in the node-description memory at the fault location are adjusted prior to the actual simulation run. During the injection process, all nodes with an active fault injected are marked as *faulty*. The presence of injected faults is completely *transparent* to the evaluation kernels as they do not interfere with the control flow of any thread during execution.

After the simulation of a fault has been completed, the node descriptions of the fault sites currently marked as *faulty* are reverted back to their original specification and their marks are removed. Once the original circuit description has been restored, the simulator is ready for the next fault simulation run. This fault injection scheme requires only a small amount of compact memory operations and allows to keep the communication and synchronization between host and device at a minimum [SKH+17, SW16].

In conventional commercial event-driven and compiled-code simulation, the timing descriptions are typically provided as external files, whose annotations are globally assigned to the netlist during the instantiation of the simulation instance. Any changes in the annotations require the set up procedure of a new instance, which is a costly procedure since it can introduce whole re-runs of the netlist optimization pre-processing and re-compilations of the simulator code. However, with the *oblivious* plain simulation scheme followed by the presented timing simulation, the timing descriptions residing in the node description memory on the GPU can be modified without the need of rebuilding the simulation model. Any modifications of node descriptions are allowed at both global scale (for all nodes) and local scale (for single nodes) with negligible overhead. This is a key-feature of

the implemented fault simulator [SKH+17, SW16] as the injection of any of the faults of the presented faults models requires only node-local changes in the netlist.

7.3 Syndrome Computation and Fault Detection

Once the simulation of the faulty circuit is finished, the time-continuous waveforms of all circuit output nodes are resident in the *waveform memory*. In order to reason about the detection of faults, a mapping of the waveforms to logic symbols is required to analyze the output and to distinguish right (*faulty*) from wrong responses (*fault-free*). For the sake of simplicity, the symbol w denotes either logic level or switch level waveforms.

7.3.1 Signal Interpretation

The value range of all modeled waveform types (either logic or switch level) can be considered as bounded by a lower bound $V_L \in \mathbb{R}$, that represents the *low* state in digital circuits, as well as an upper bound $V_H \in \mathbb{R}$, which corresponds to the *high* state of the signal (i.e., '0' and '1' at logic- or GND and VDD at switch level). At logic level, these bounds together already reflect the value domain $\boldsymbol{B}_2$ themselves. While at switch level voltages close to GND or VDD can be considered as *low* and *high* respectively, other voltages do not show a definite *high* or *low* behavior as succeeding cells can interpret the signal differently. For those values, the output behavior is usually considered as *undefined* (X) [Hay86].

This work uses a *threshold-based* characterization to logically interpret (continuous) voltage waveforms to distinguish between *high*, *low* and *undefined*. A *threshold interval* $[V_{thL}, V_{thH}] \subset [V_L, V_H]$ is defined, that separates the value domain into value domains of a defined logic symbol. Signal values that fall within $[V_L, V_{thL})$ (or $(V_{thH}, V_H]$) are considered as *low* (resp. *high*) due to amplification of voltage levels at cell outputs in CMOS technology [WH11]. Values in the range of $[V_{thL}, V_{thH}]$ are *intermediate* and have no clear defined state in digital circuits and are therefore considered as *undefined* (symbol 'X') and *possibly erroneous*. These thresholds can be obtained by characterization of the *transfer functions* of the CMOS cells which describe the input/output relation [WH11].

The mapping of a (waveform) value $v \in \mathbb{R}$ to a discrete logic symbol of the ternary logic domain $\boldsymbol{E_3} = \{0, 1, X\}$ [Hay86] is described by a function:

$$val_{\boldsymbol{E}} : \mathbb{R}^3 \rightarrow \boldsymbol{E_3}, val_{\boldsymbol{E}}(v, V_{thL}, V_{thH}) := \begin{cases} 1 & \text{if } (v > V_{thH}), \\ 0 & \text{else if } (v < V_{thL}), \\ X & \text{else.} \end{cases} \tag{7.3}$$

7.3.2 Discrete Syndrome Computation

To determine the presence of deviations or *errors* at the outputs of a faulty circuit the corresponding waveforms are first captured at a given sample time $t_{samp} \in \mathbb{R}$. The *captured* faulty values are compared against the *fault-free* circuit to determine the output difference (or *syndrome*) caused by the fault. In this work, *syndrome waveforms* [SKH+17, SW19a] are utilized that allow to continuously determine the detection of a fault over time.

Definition 7.1. A *syndrome waveform* syn expresses the difference of a time-continuous output waveform w with respect to the *fault-free* potential at any given point t in time.

At logic level, this reference *potential* represents the output of the *good*-value simulation, i.e., the *high* or the *low* state. At switch level, the corresponding voltage potentials (VDD and GND) are used as a reference. The difference in the output responses at a time point $t \in \mathbb{R}$ then determines the syndrome waveform value $syn(t) \in \boldsymbol{E}_3 = \{0, 1, X\}$ similar to Eq. (7.3), which is considered

- *faulty*, if the syndrome value is $syn(t) = 1$,
- *fault-free*, if $syn(t) = 0$, and
- *unknown* for $syn(t) = X$.

Syndrome waveforms are obtained directly from the output waveforms by a transformation process within a single pass over the waveform events. The transformation procedure compares the values of a waveform w with the *fault-free* reference responses, denoted as $\boldsymbol{w}(\infty)$. It is assumed that the *fault-free* responses always have a clear *high* or *low* signal value $\boldsymbol{w}(\infty) \in \{V_H, V_L\}$, which are defined by a *high* potential ($V_H \in \mathbb{R}$) and *low* potential value ($V_L \in \mathbb{R}$) as reference. The corresponding syndrome $syn(t)$ is then acquired by

mirroring the values of $\boldsymbol{w}(t)$ at the center potential level $\frac{1}{2} \cdot (V_H + V_L)$ according to the reference value $\boldsymbol{w}(\infty)$, such that

$$syn(t) := \begin{cases} val_{\boldsymbol{E}}(\boldsymbol{w}(t), V_L, V_H) & \text{if } \big(\boldsymbol{w}(\infty) \leq \frac{1}{2} \cdot (V_H + V_L)\big), \\ val_{\boldsymbol{E}}(V_H - \boldsymbol{w}(t) + V_L, V_L, V_H) & \text{else.} \end{cases} \tag{7.4}$$

When applied to logic simulation over $\boldsymbol{B}_2 = \{0, 1\}$, $syn(t)$ corresponds to the Boolean difference of the waveform value $\boldsymbol{w}(t)$ and the reference value for any time t, therefore $syn(t) \equiv (\boldsymbol{w}(t) \oplus \boldsymbol{w}(\infty))$. In case of a *small gate delay fault* of finite size $\delta_f < \infty$ the fault free response can be directly acquired from the faulty waveform itself by sampling $\boldsymbol{w}(t_{samp})$ at time $t_{samp} \to \infty$. In the switch level simulation, the $syn(t)$ function performs a logical interpretation of the absolute voltage difference of the signal with respect to the level of the fault-free potential. Over $\boldsymbol{E}_3 = \{0, 1, X\}$, the presence of *undefined* values (X) at outputs will lead to a pessimistic *unknown* syndrome $(\boldsymbol{w}(t) \hat{=} X) \Rightarrow (syn(t) = X)$.

7.3.3 Fault Detection

With the use of the computed output syndromes, each fault can be classified as *detected*, *undetected* and *possibly detected*. For a given sample time t_{samp}, the classification of a fault f is performed by looking-up each syndrome $syn_o(t_{samp})$ of all reachable circuit outputs $o \in O(f)$ in its output cone $O(f) \subseteq O$. In general, a fault is considered as *detected* at an output $o \in O(f)$, if the corresponding syndrome is $syn_o(t) = 1$ at time t and *undetected*, if $syn_o(t) = 0$. If neither of the above cases applies the syndrome is considered as *unknown*, which cannot provide reliable information regarding the fault detection. Thus, the fault classification based on the captured syndrome responses is realized as follows:

- *detected* (DT) iff *at least* one output $o \in O(f)$ in the output cone of f shows a *faulty* syndrome $(\exists o \in O(f) : syn_o(t_{samp}) = 1)$,

- *undetected* (UD) iff *all* output $o \in O(f)$ in the output cone of f show only *fault-free* syndromes $(\forall o \in O(f) : syn_o(t_{samp}) = 0)$,

- *possibly detected* (PD) iff a *non-empty* subset of outputs $o \in O(f)$ in the output cone of f show an *unknown* ('X') syndrome while no other output is considered *faulty* $(\exists o \in O(f) : syn_o(t_{samp}) = X) \wedge (\forall o \in O(f) : syn_o(t_{samp}) \neq 1)$.

Setup-/Hold- Times

The waveform information obtained can be used to identify possible wrong signal captures of signal values in the associated storage elements due to violations in the *setup-* and *hold-times*. In the standard delay format [IEE01a] these times are provided for any storage-element cells through the `SETUP` and `HOLD` timing properties. Each of the properties defines a time margin in an interval before (for `SETUP`) or after (`HOLD`) the sample time t_{samp} in which the data-input signal must sustain a stable value in order to be properly captured. A *violation* in a setup- or hold-time of a storage element occurs, if the input signal is not *stable*, i.e., due to a signal value transition, which is assumed to cause uncertainty in the capturing process. Thus, the stability of a signal within a time interval $[t_S, t_H] \subseteq \mathbb{R}$ with lower bound $t_S := (t_{samp} - t_{setup})$ and upper bound $t_H := (t_{samp} + t_{hold})$ must be checked in order to identify these violations. The times $t_{setup} \in \mathbb{R}$ and $t_{hold} \in \mathbb{R}$ are added to the node description memory as part of the description of each output node.
A *setup-time violation* $S_o \in \boldsymbol{B}_2$ at an output $o \in O(f)$ is raised ($S_o = 1$) iff

$$S_o \Leftrightarrow (\exists t \in [t_S, t_{samp}] : syn_o(t) \neq syn_o(t_{samp})). \tag{7.5}$$

For determining hold-violations, the signal value at the lower bound t_{samp} is used as a reference and the traversal of the waveform events is continued until time t_H is reached. Similarly, a *hold-time violation* $H_o \in \boldsymbol{B}_2$ is then indicated once an event e_i with $t_{samp} \leq t_i \leq t_H$ is found, such that the syndrome value changes:

$$H_o \Leftrightarrow (\exists t \in [t_{samp}, t_H] : syn_o(t) \neq syn_o(t_{samp})). \tag{7.6}$$

Regarding the impact of the setup- and hold-time violations on the output capturing, all captured outputs $o \in O(f)$ with either $S_o = 1$ and $H_o = 1$ will be pessimistically considered as *unknown* 'X' and *possibly erroneous* [SW19a].

7.4 Fault-Parallel Simulation

The high-throughput parallelization scheme is extended for the simulation of structurally independent faults which is applicable to both logic and switch level simulation [SHK+15, SKH+17, SW16]. The simulation effort is further reduced by applying *fault collapsing* to reduce the initial fault set by removing *timing equivalent* faults. A *fault grouping* heuristic is then applied to partition the remaining faults into *fault groups* for parallel injection and simulation. An efficient *syndrome* calculation kernel allow for comprehensive and exhaustive fault analysis.

7.4.1 Fault Collapsing

Prior to the waveform-accurate simulation, the number of fault locations is reduced by using structural *fault collapsing* of the fault list. For this, *equivalence classes* need to be identified, which are sets of faults that lead to identical output behavior in the simulation model [SKH+17]. Hence, the simulation of only one *representative* fault is necessary in order to evaluate all faults of an equivalence class. In this work, two faults f_1 and f_2 are *timing equivalent* ($f_1 \equiv f_2$), iff the waveforms at the circuit outputs on the sensitized propagation paths *match* for all possible input stimuli.
At logic level, the equivalence rules based on the *transition fault* model [WLRI87] can be applied and adapted for small delay faults as follows. Let $\mathcal{F}$ be the exhaustive set of small delay faults in a circuit (i.e., all input pins and output pins of each node) with a specified rising delay δ_r and falling delay δ_f for each fault $f_i := (loc_i, \{\delta_r, \delta_f\}) \in \mathcal{F}$. In case the faults affect both transition polarities by the same delay amount ($\delta_r = \delta_f$), then the timing equivalence of each fault $f_1, f_2 \in \mathcal{F}$ in the fault set can be determined as follows:

(1) If a node has a single predecessor as input, then the fault f_1 at the node input and the fault f_2 at the node output are *timing equivalent* ($f_1 \equiv f_2$).

(2) If a node has a single successor as output, then the fault f_1 at the output and the fault f_2 at the input of the fanout-node are *timing equivalent* ($f_1 \equiv f_2$).

Regarding the first rule, assume a node compliant with the premise of rule (1) with propagation delay of $d \in \mathbb{R}$ and let $e_1 = (t_1)$ be an input event at time t_1. The cor-

responding event at the node output $e_2 = (t_2)$ then takes place at time $t_2 = (t_1 + d)$. Further assume a small delay fault f_1 of size $\delta := \delta_r = \delta_f$ (both polarities) at the node input, then the input transition is delayed taking place at time $t_1' := (t_1 + \delta)$. Similarly, a fault f_2 at the node output could delay the output event at time t_2 by the same amount δ, such that $t_2' := (t_2 + \delta)$. For f_1 the resulting time of the output event is $t_1' + d = (t_1 + \delta) + d = (t_1 + d) + \delta = (t_2 + \delta) = t_2'$. Therefore, both faults produce the same output events for all input combinations, hence being timing equivalent $(f_1 \equiv f_2)$. In case the second rule (2) applies, the faulty output event from fault f_1 arrives at time $t_1' = (t_1 + \delta)$, at the input of the successor. The succeeding node processes the input event at t_1' by adding the propagation delay specified in the time spec to generate the *intermediate* waveform (cf. Chapter 5) that predicts a output event at time $t_2 = (t_1' + d)$. Note that the predicted time is equal to the output caused by fault f_2 at the successor which is $t_2' = t_1 + (d + \delta) = (t_1 + \delta) + d = t_1' + \delta = t_2$.

The above rules are further extended to collapse faults with different rising and falling delays based on the *inverting* and *non-inverting* properties of the nodes. If the node type is *inverting* and complies with the premise of rule (1), a small delay fault fault $f_1 = (i, \{\delta_r, \delta_f\})$ with a rising delay δ_r and falling delay δ_f at the input pin i of a node behaves identical to a fault $f_2 = (o, \{\delta_f, \delta_r\})$ at its output pin o, as the output signal has the inverted polarity of the input signal when propagated. If a node is *non-inverting* and complies with the premise of rule (1), then any fault $f_1 = (i, \{\delta_r, \delta_f\})$ at the input pin i is equivalent to a fault f_2 at the output pin o with fault size $f_2 = (o, \{\delta_r, \delta_f\})$ having the same delay with respect to each transition polarity. This is also applicable to the second rule (2) above, as a single fanout interconnection can be viewed as an interconnect buffer node, which also has a *non-inverting* property.

All of the above rules obey a transitive relationship, which allows chaining of multiple equivalence rules to maximize the size of the equivalence classes of a fault set $\mathcal{F}$:

$$\forall f_1, f_2, f_3 \in \mathcal{F} : (f_1 \equiv f_2) \wedge (f_2 \equiv f_3) \Rightarrow (f_1 \equiv f_3). \tag{7.7}$$

Without loss of generality, the fault closest to the circuit outputs is used as the *representative fault* of the equivalence. Hence, since the equivalence rules consider only the

relation between directly connected fault locations, all representative fault locations of a circuit $G = (V, E)$ can be determined with linear time complexity $\mathcal{O}(|V|)$.

In contrast to logic level, fault collapsing of parametric faults at switch level is not as trivial [SW16, SW19a]. While the modeled small delay faults affect signals only in terms of the time by shifting the transitions according to the fault size, the presented parametric faults at switch level can also affect the *slope* of the signals. These changes in the slopes can cause further differences in the waveform shapes of succeeding nodes when propagating through the circuit, due to different rising and falling delays. This makes it difficult to identify equivalent faults from different RRC-cells that lead to the exact same output waveforms as required by the *timing equivalence criteria*. However, some timing equivalences of resistive parametric faults can be identified on a RRC-cell-internal level [SW19a] allowing to perform a fault collapsing based on the structure of the resistor networks [BKL+82, LNB83].

Resistive parametric faults f_1 and f_2 of the same size $\Delta R_f \in \mathbb{R}$, can be marked as equivalent iff the resistances $R_1 \in \mathbb{R}$ and $R_2 \in \mathbb{R}$ of the associated fault locations are connected such that all direct paths from GND or VDD to the node output pin always pass both locations. Hence, in the RRC-cell model any current flowing from or to the node output always passes through both or none of the locations. Therefore, the faults lead to the same electrical output behavior as an injection in R_1 or in R_2 delivers an identical total resistance:

$$(R_1 + \Delta R_f) + R_2 = R_1 + (\Delta R_f + R_2). \tag{7.8}$$

Hence, any two faults f_i and f_j located at a resistive parameters $R_i, R_j \in \mathbb{R}$ of an RRC-cell are *timing equivalent* ($f_i \equiv f_j$) iff for any path in the equivalence circuit mesh from either VDD or GND to the cell output, R_i, R_j both lie on the path or *none* of them at all [SW19a].

7.4.2 Fault Groups

For fault-parallel simulation the simulator employs a parallelization scheme based on the simultaneous injection of *output-independent* faults [IT88]. Let $O(f_1), O(f_2) \subseteq O$ be the reachable outputs in the transitive fanout of two faults $f_1, f_2 \in \mathcal{F}$ of a fault set $\mathcal{F}$, then the two faults are considered *output-independent*, if they do not share any output logic

($O(f_1) \cap O(f_2) = \varnothing$). Output-independent faults have no mutual influence by adding or masking themselves, since the fault effects propagate to distinct parts of the circuit. For determining the detection of each fault, only the reachable outputs in the respective output cones need to be investigated. Therefore, the faults can be injected and evaluated in the same simulation instance allowing for a parallel simulation. In the following, a set of mutually output-independent faults will be referred to as a *fault group* denoted as FG.

Definition 7.2. A *fault group* $FG \subseteq \mathcal{F}$ is a set of mutually *output-independent* faults of a fault set $\mathcal{F}$, such that:

$$\forall f_i, f_j \in FG : (f_i \neq f_j) \Rightarrow O(f_i) \cap O(f_j) = \varnothing. \tag{7.9}$$

The optimal selection of fault groups can be viewed as a *minimum graph coloring* problem, or *chromatic number problem*, that is applied to an *output-dependence graph*. In this graph each fault is represented by a node and two nodes are connected iff the associated faults share common output logic. Hence, each edge directly indicates that the associated faults are ineligible for a parallel injection. During the coloring, each node is assigned a color, such that no edge connects two nodes of identical color. Each color then represents a specific fault group and all nodes of the same color belong to the same fault group.
In order to achieve maximum fault parallelism and simulation throughput, it is favorable to keep the fault groups large and the total group count low to process as many faults in parallel as possible with the least amount of simulation runs. The minimum graph coloring problem has the additional requirement that the number of colors used is also minimal. The graph coloring problem as such has been proven to be NP-*complete*, with its optimization being NP-*hard* [Kar72] which is not applicable to fault sets of designs with millions of faults.

7.4.3 Grouping Algorithm

To enable a fast and efficient computation of fault groups for both sparse and exhaustive fault sets even for larger problem sizes, a *greedy* fault grouping heuristic [SHK+15, SKH+17] is applied. Unlike [IT88], the heuristic follows a breadth-first grouping approach. This way an efficient grouping for exhaustive as well as sparse fault sets is en-

abled. The general flow of the algorithm is outlined in Fig. 7.3, which starts with an empty set of fault groups. Given a particular set of faults $\mathcal{F}$ as input, first the algorithm extracts the spatial information of each fault to determine all fault locations in the circuit. First, all fault locations of $\mathcal{F}$ are scheduled in an *as-late-as-possible* (ALAP) manner by sorting them in reversed topological order from outputs to inputs (1).

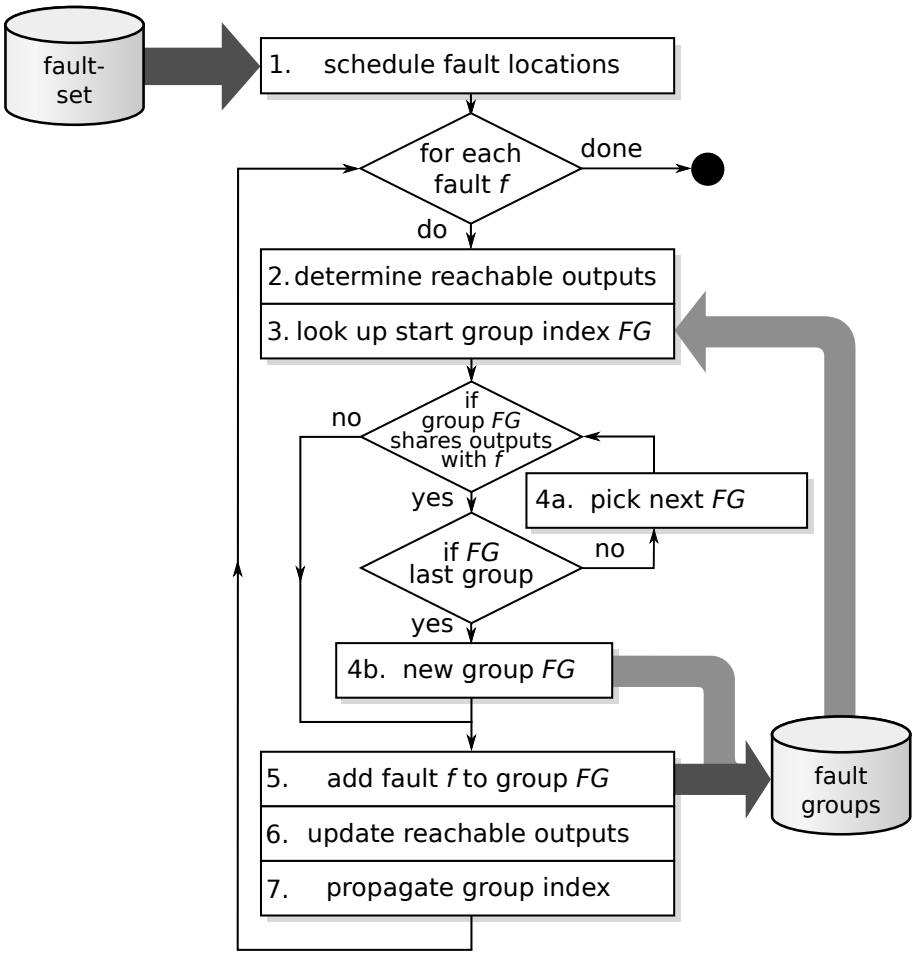


Figure 7.3: Flow-chart of the greedy fault grouping heuristic.

Each fault $f \in \mathcal{F}$ is processed one after another within a loop by assigning it to a suitable fault group. For this, the reachable outputs in the output cone of a fault are compared to those of the combined output cones of the faults in each obtained fault group. The reachable outputs $O(f) \subseteq O$ of a fault f are determined by traversing through the netlist towards the circuit outputs O and stored in a hash-set (2), that provides fast look-ups of outputs for quick comparisons with other faults and groups. In addition, for each fault

group FG a hash-set is stored that holds the union of reachable outputs of each fault contained $O(FG) := \bigcup_{f \in FG}\{O(f)\}$. Initially, the hash-set of a fault group is empty, and reachable outputs are added and updated only when a fault is being assigned to the group.

During the grouping of a fault f, a *start group index* $start(f) \in \mathbb{N}$ is utilized (3), which points to the first fault group FG_i ($i = 1, 2, 3, ...$) to be compared with. The computation of the index of a particular fault f is performed by looking up the highest group index $max\_group : V \rightarrow \mathbb{N}$ of any reachable output $o \in O(f)$ in the output cone of the fault. Initially, the group index of every output node $o \in O$ in the circuit is initialized with $max\_group(o) := 0$. For a particular fault f at any location, the start group index is then computed by

$$start(f) := \max\{max\_group(o) \ : o \in O(f)\} + 1. \tag{7.10}$$

The outputs in $o \in O(f)$ are then compared with the reachable outputs of the fault group starting from $idx := start(f)$ to identify any common output logic. As soon as a shared output is detected, the index is increased ($idx := idx + 1$) and the next group in the list is taken (4a), and again checked for any shared outputs. In case none of the existing fault groups is disjoint with f, a new group is created for the fault (4b). If the reachable outputs $O(f)$ of a fault f and the reachable outputs $O(FG)$ of a fault group FG are disjoint ($\forall o \in O(f) : o \notin O(FG)$), then f can be assigned to FG (5). Whenever a fault f is added to a group, the reachable outputs of the fault are added to the reachable outputs of the fault group (6), and the group index idx is assigned to all output nodes $o \in O(f)$ in its output cone by $max\_group(o) := idx$ (7). The reversed topological order of the processing from outputs to inputs allows to use the start group index $start(f)$ to skip unnecessary comparisons with groups as the transitive structural dependencies of all previously scheduled fault locations in the support cone are sustained.

The application of the fault grouping algorithm is illustrated in Fig. 7.4. Given a fault set, the faults are sorted in reverse topological order from outputs to inputs (a) with the output-dependencies shown in the output-dependence graph (b). As all circuit outputs are initialized with a group index 0, fault f_1 is assigned to the first group FG_1 (Eq. (7.10)) and the group index is assigned to the reachable outputs in $O(f)$. In the next step, the fault f_2 is processed. Due to the output dependence of f_2 and f_1, the start group index

of f_2 is $start(f_2) = 1$, thus the comparison with the first group is omitted and the fault is assigned to a new group FG_2. For fault f_3, no output dependence so far has been discovered $start(f_3) = 1$ and after comparing the reachable outputs the fault is assigned to group FG_1 along with fault f_1. Regarding f_4 and f_5, both of the faults share outputs common with f_2 (Fig. 7.4a) allowing to skip FG_1 and FG_2. Furthermore, the ranges of the reachable outputs of the two faults are denoted as $O(f_4)$ and $O(f_5)$, respectively. Since both faults do not share common output logic ($O(f_4) \cap O(f_5) = \varnothing$) they can be scheduled in the same fault group FG_3.

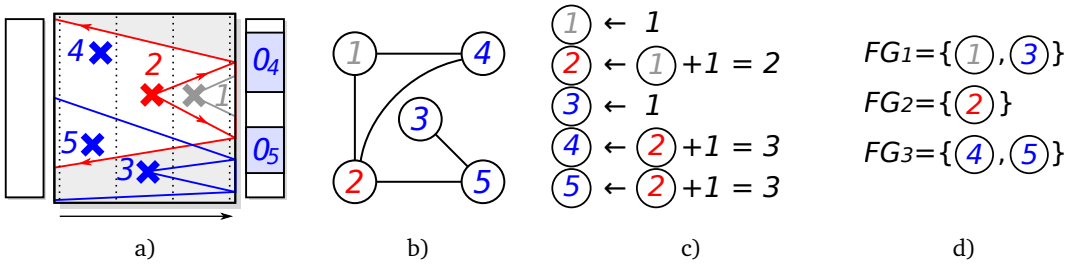


Figure 7.4: Fault grouping example: a) fault set $\mathcal{F} := \{f_1, f_2, f_3, f_4, f_5\}$ in reverse topological order, b) output-dependencies, c) group indices and d) resulting fault groups.

Once all fault groups have been extracted, the simulator can process the fault groups in consecutive simulation runs. During each run, all faults $f \in FG$ of a fault group are simultaneously injected into the circuit with all associated fault locations being marked *faulty*. Once the timing simulation is complete, the detection of each fault $f \in FG$ is determined by checking only the set of reachable outputs $O(f)$ of the respective faults. After finishing the evaluation, the node descriptions of the marked locations are reverted back to their original specification thereby removing all injected faults from the circuit.

7.4.4 Parallel Fault Evaluation

The calculations of the syndrome is performed by a two-dimensional kernel as shown in Fig. 7.5, which illustrates the syndrome computation for the switch level simulation.
Each thread in the grid computes the syndrome symbol ('0', '1' or 'X') for a particular output and stimuli at a given sample time t_{samp}. The threads first access the output waveforms of their respective node in the waveform memory and sample the voltage $w(t_{samp})$

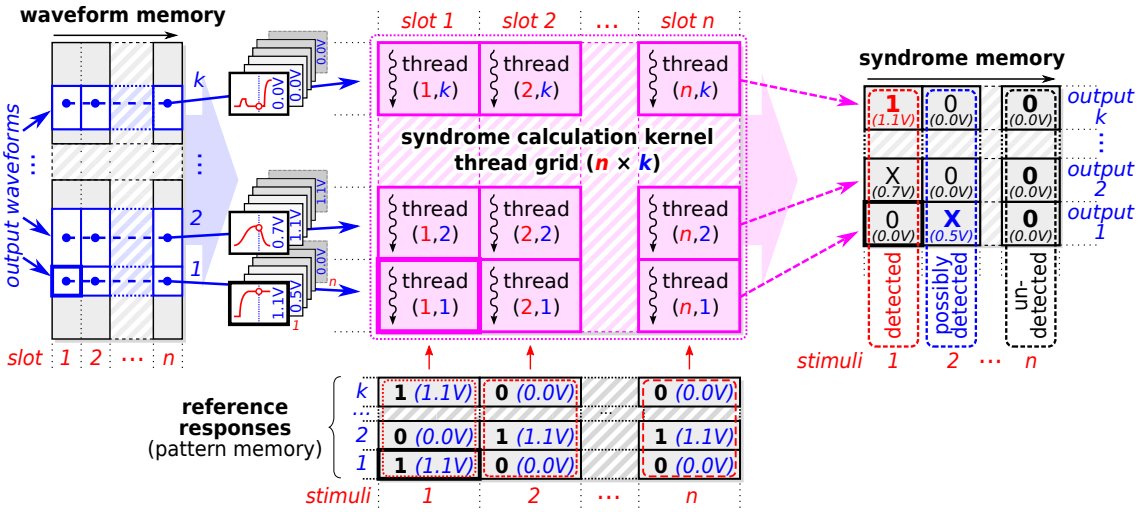


Figure 7.5: Parallel syndrome calculation for switch level waveforms in a two-dimensional kernel with additional syndrome memory.

at time t_{samp}. In the next step, each thread accesses the reference response $w(\infty) \in \mathbb{R}$ in the pattern memory on the GPU. In case of the switch level simulation, the logic symbol is mapped to the corresponding voltage value using Eq. (6.14). The *high* (V_H) and *low* (V_L) voltage potentials have been chosen as 1.1V and 0.0V respectively. Similarly, the thresholds for distinguishing clear *high* and *low* signal values from *unknown* 'X'es were chosen as $V_{thH} = 0.8\text{V}$ and $V_{thL} = 0.3\text{V}$. The syndrome $syn(t_{samp})$ is then computed according to Eq. (7.4) and stored as binary encoded symbol in the dedicated *syndrome memory*. The syndrome memory then allows to look-up and determine the fault detection for each stimuli individually.

Since during the evaluation all waveforms remain untouched in the waveform memory on the GPU, the signal capturing of waveform signals can be performed multiple times in succession at varying points in time without the need for additional simulation runs. Also, each output node can be assigned an individual offset with respect to the capture time t_{samp}, which allows for modeling of *clock skew* due to unbalanced signal propagation in the clock-distribution tree.

7.5 Summary

In this chapter, an extension of the high-throughput timing simulation model was presented to enable timing-accurate parallel fault simulation on GPUs. Fault models have been presented for *small gate delay fault* simulation at logic level [SHK+15, SKH+17], as well as *parametric* and *parasitic* faults at switch level [SW16, SW19a]. Fault-parallelism is exploited to cope with the additional problem complexity from the faults. For this, the timing-equivalence was defined for the aforementioned fault models and a fault grouping heuristic was presented that identifies groups of output-independent faults of a fault-set for parallel injection and efficient simulation. A highly-parallel and comprehensive syndrome evaluation then allows to reveal timing-accurate detection information for each fault directly from the waveforms.

Chapter 8

Multi-Level Simulation on GPUs

This chapter presents the first multi-level fault simulation approach on GPUs [SKW18, SW19b] which selectively trades off *speed* against *accuracy* by mixing abstractions in order to maximize the simulation efficiency. The approach utilizes similarities in data structures and kernel organization of the presented simulation models and their implementations and provides a transparent transition between the abstraction levels.

8.1 Overview

In timing simulation, high abstraction modeling provides higher simulation speedup, but the modeling capabilities with respect to both functional and timing behavior are limited, which ultimately constrains the simulation accuracy. However, often it is not necessary to simulate the full circuit with the lowest abstraction [MC93]. Therefore, mixing abstractions is a way to speed-up and increase the *efficiency* of the simulation.

Assume a multi-level simulator that utilizes both lower and higher abstraction levels (e.g., switch and logic level nodes) simultaneously. Further assume a given mixed-abstraction scenario $x \in [0, 1]$ that describes the ratio of circuit nodes at switch level, i.e., from full logic ($x = 0$) to full switch level simulation ($x = 1$). Let $T_{ML}(x) \in \mathbb{R}$ be the runtime of the multi-level simulation for the scenario x, and let $T_{ref} \in \mathbb{R}$ be the reference runtime required to run a full simulation at the lower abstraction level, i.e., switch level with $T_{ref} := T_{ML}(1)$.

Definition 8.1. (*multi-level simulation efficiency*) The *multi-level simulation efficiency* of a mixed-abstraction scenario $x \in [0,1]$ processed in a multi-level simulator ML is defined as

$$\mathit{eff}_{ML} : [0,1] \to \mathbb{R},\ \mathit{eff}_{ML}(x) := \frac{T_{\mathit{ref}}}{T_{ML}(x)} \left(= \frac{T_{ML}(1)}{T_{ML}(x)} \right). \tag{8.1}$$

Example 8.1. *For example, given a simulation problem x that requires time $T_{ref} = 10s$ for evaluation at a targeted lower abstraction level and time $T_{ML}(x) = 1s$ for the evaluation in a multi-level simulator ML. The simulation efficiency of the multi-level approach is computed by $\mathit{eff}_{ML}(x) = 10s/1s = 10$, i.e., the multi-level simulator can solve in the same time ten times the amount of simulation problems compared to the full low level simulation.*

In the context of GPUs, the mixing of abstractions during multi-level simulation generally conflicts with the underlying many-core programming paradigm on the SIMD architectures [OHL+08]. High- and low-level simulations typically utilize different data-structures and algorithms, each of which involves its own working set with varying memory footprint. This complicates the parallelization and portability of the individual algorithms to the GPUs eventually causing inefficiencies for the use in a multi-level simulation environment due to diverging execution flows.
The presented time simulator is extended to form a mixed-abstraction multi-level fault simulation, that allows to switch accuracy wherever and whenever required. While this work focuses on combining the presented logic level and switch level time simulation (cf. Chapter 5 and 6), the general concepts can be extended for incorporating additional abstraction levels. Throughout the simulation, the different abstractions are processed interchangeably on the GPU by utilizing *unified* data structures and kernels. The key features of the multi-level simulator are:

- scalable, flexible and extensible waveform-accurate parallel multi-level time simulation with full accuracy and time trade-off,
- mixed-abstraction simulation with efficient memory reuse and low overhead switching between abstractions for efficient consecutive simulations runs,
- and full transparency of abstraction levels in the kernels causing no additional control flow divergence during evaluation.

8.2 Multi-Level Circuit Modeling

In the multi-level simulation, the circuit netlist is modeled as an *abstraction-independent* graph, which stores the general structure of the circuit including interconnections of the circuit nodes. For each node, *abstraction-dependent* functional and timing descriptions are assumed to be available for all abstraction levels. However, during simulation the description of only one abstraction level is used at a time per node. All output waveforms inherit the same abstraction level as their respective nodes.
In the following, the general approach for switching between different abstractions during simulation and the simulation flow is explained.

8.2.1 Region of Interest

The implemented multi-level simulation uses the concept of *regions of interest* (ROI) in order to switch abstraction levels in critical regions throughout the simulation [SKW18]. While the simulator is expected to run with the highest abstraction to maximize speed (logic level), nodes in the circuit netlist G can be *activated* to perform as ROI by assuming a lower abstraction level through swapping of their abstraction-dependent descriptions.

Definition 8.2. A *region of interest* (ROI) is a connected sub-graph $H \subseteq G$ of a netlist G with nodes of lower abstraction (switch level) compared to adjacent nodes.

The *activation* of ROIs at single or multiple nodes then allows to locally lower the abstraction and gradually increase the modeling accuracy over the circuit. Different ROI activation schemes allow to change the accuracy during simulation:

- *spots* representing a single circuit node (e.g., standard cell),
- *paths* as sequence of nodes from an input to an output of the circuit,
- *cones* that cover a complete input or output cone, or cone of influence of a node, or
- *areas* to model a connected sub-graph of the circuit netlist (e.g., module).

For example, ROI *spots* at single nodes can be utilized in order to introduce more accurate descriptions in case of *complex-cells* or injection-points for low-level fault modeling, which

otherwise would not be represented with sufficient accuracy at higher abstraction levels, or not represented at all. *Paths* can be activated, such as longest or critical paths required for investigation (i.e., paths that are likely to cause errors under a resistive open delay fault). ROI-*cones* enable provision of more accurate stimuli to a node and also enable low-level signal propagation to outputs. Similarly, the activation of ROI-*areas* lowers the abstraction in connected regions of the circuit (e.g., sub-module).

The multi-level circuit modeling can be utilized to simultaneously inject and simulate faults of mixed abstractions. When simulating a fault, it is injected into the node description corresponding to the abstraction level of the fault. The node description with faults are then used actively during the simulation until the faults are removed. This way, all of the logic- and switch level fault models presented in Chapter 7 are supported.

8.2.2 Changing the Abstraction

When activating or removing a ROI in the netlist, the original netlist graph $G = (V, E)$ is modified [MC95] in the process to generate a substitute graph $G' := (V', E')$ that includes the ROI. Let $V_n \subseteq V$ be a set of nodes of one abstraction connected to direct predecessors $i \in I \subseteq V$ and direct successors $o \in O \subseteq V$ in the graph such that $(I \cup O) \cap V_n = \emptyset$. All nodes $n \in V_n$ are connected to the predecessors via edges $E_i := \bigcup_{i \in I} \{(i, n) \in E\}$ and to the successors via $E_o := \bigcup_{o \in O} \{(n, o) \in E\}$. Further, let $E_n := \bigcup_{i,o \in V_n} \{(i, o) \in E\}$ be the set of edges that connect nodes internally in V_n. Then $G_n := (V_n, E_n \cup E_i \cup E_o)$ forms the connecting sub-graph of V_n embedded in G.

Now, let $G'_n := (V'_n, E'_n \cup E'_i \cup E'_o)$ be sub-graph of n in the other abstraction composed of a set of nodes V'_n, with internal $E'_n \subseteq V'_n \times V'_n$, in-going $E'_i \subseteq I \times V'_n$ and outgoing $E'_o \subseteq V'_n \times O$ edges. The swapping from one abstraction to another is performed by substitution of the corresponding sub-graphs G_n and G'_n in the original netlist G, to generate the substitute graph $G' = (V', E')$ where $V' := \{V \setminus V_n\} \cup V'_n$ and $E' := \{E \setminus E_n\} \cup E'_n$.

8.2.3 Mixed-abstraction Temporal Behavior

In mixed-level simulation of the circuit, all nodes are evaluated by the waveform processing algorithm of their respective abstraction which computes output waveforms in the corresponding abstractions. Therefore, waveforms of different abstractions can coexist

during simulation. Each time a waveform encounters a node of a different abstraction, it crosses an ROI boundary. To allow the receiving node to process the waveform, the waveform needs to be *transformed* to the abstraction of the node during the input processing. This work utilizes two transformations to map between high-level waveforms with discrete logic values and low-level waveforms with continuous voltages as presented in [SKW18]. These transformations map the events from *logic- to switch* and back from *switch- to logic* level accordingly.

Ternary Logic Waveforms

As mentioned earlier, switch level waveforms are continuous in value and for certain values an *undefined* logic behavior is interpreted which can propagate through the circuit. The logic level waveforms are therefore extended to support ternary logic $\boldsymbol{E}_3 = \{0, 1, X\}$ over a pseudo-Boolean algebra in order to model *undefined* values [Hay86].
For this, all events $e \in w$ of a logic level waveform are distinguished between *ordinary events* and *special events*. With the exception of the initialization and sentinel event at $t = -\infty$ and $t = \infty$, all *ordinary* events e are considered to occur only at times $0 \leq t < \infty$. For the extension the special events $e := (t) \in w$ are allowed *negative* event times $t < 0$. By consideration of the sign bit [IEE08] additional signal transition rules can be formulated for the waveforms. The *sign* of an event time t is accessed via the sign-function $sgn : \mathbb{R} \to \{0, 1\}$, where $sgn(t)$ delivers 1, if the sign bit of the corresponding event time is set (t negative), or 0 otherwise (number is positive). The absolute value $|t|$ of $t \in e$ is considered as actual event time for temporal evaluation. Within each waveform all events are stored in temporal order according to their (absolute) event time $|t|$. Hence, for any two events $e_i, e_j \in w$ with $|t_i| > |t_j|$, the respective indices in for the ordering are $i > j$.
In case $sgn(t) = 1$ holds for an event $e = (t)$ with time t, e is declared a special event which triggers an irregular transition, that switches between *undefined* ('X') and *high* ('1') or *low* ('0') values. All transitions between the different values are illustrated in the following state diagram of Fig. 8.1 from which the waveform function $\boldsymbol{w}$ is derived. Initially, each waveform has a *low* ('0') initial value (*init*). For each following event e_i with $sgn(t_i) = 0$ the signal state then transitions in temporal order between *high* ('1') and *low* ('0'). In case $sgn(t_i) = 1$, the signal transitions to the *unknown* state ('X') at time $|t_i|$, which can be left

to either high or low depending on the next event in the order. Since the IEEE floating point standard [IEE08] distinguishes between -0 (sign-bit set) and 0 (sign-bit not set), transitions to *unknown* at time $t = 0$ are supported as well. For constant signals the corresponding waveform representations are $w_0 := \{(\infty)\}$ for *low*, $w_1 := \{(-\infty), (-\infty), (\infty)\}$ for *high* and $w_X := \{(-\infty), (\infty)\}$ for *unknown* values.

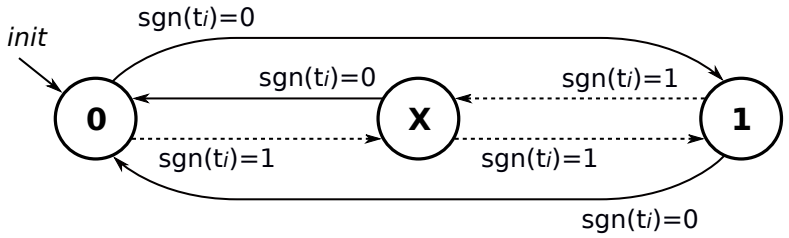


Figure 8.1: State transitions of events $e_i = (t_i)$ in *ternary* $\boldsymbol{E}_3$ logic waveforms.

A more complex waveform using the ternary logic representation is illustrated in Fig. 8.2. As shown, the new waveform modeling can express all possible transitions in the three-valued $\boldsymbol{E}_3$ logic domain in a compact way. If no undefined values are present during simulation, the processing of the waveforms is similar to the previous approach in [HSW12, HIW15] except for signal waveforms with an initial value of '1' for which an additional event at $t = -\infty$ has to be stored.

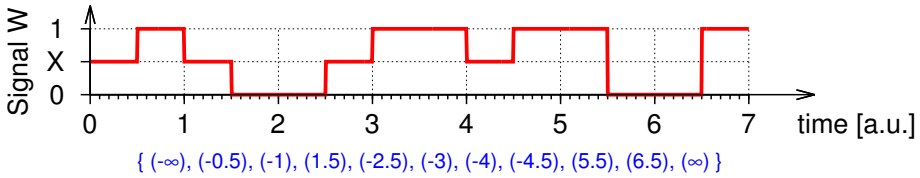


Figure 8.2: Example of an arbitrary waveform over the ternary logic value domain $\boldsymbol{E_3}$.

8.2.4 Simulation Flow

The overall flow of the multi-level simulation approach combining both logic and switch level is shown in Fig. 8.3. After reading in the design (i.e., as netlist), the abstraction-independent combinational network of the netlist is extracted, topologically ordered and annotated with abstraction-dependent timing data (1). Once the circuit data is prepared, the user can define the *regions of interest* (ROI). The ROIs can be defined by specifying

individual nodes for simulation with switch level accuracy (2), or by providing a fault set, where the individual fault locations provided are used as ROIs. Similar to faults, the defined ROIs can be grouped (3) for parallel simulation using the presented *grouping heuristic* (cf. Chapter 7).

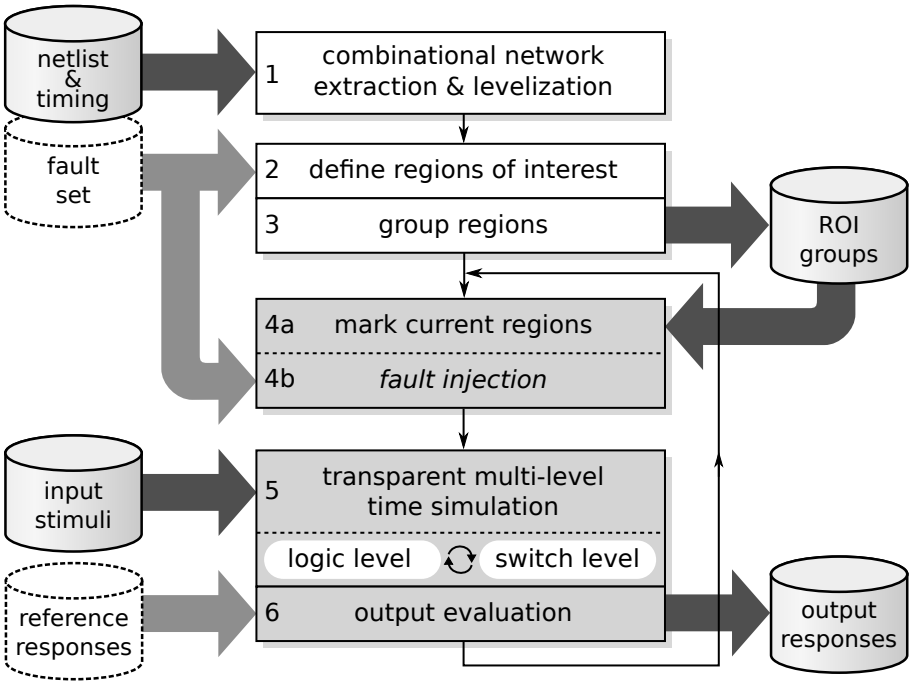


Figure 8.3: Overall flow-chart of the implemented multi-level fault simulation for combined logic- and switch level simulation. Shaded tasks are parallelized.

The ROI groups are then processed individually in subsequent simulation runs (step 4–6). While processing a ROI, the simulator first marks all nodes of the current ROIs as *active* (4a). During this process, the timing descriptions of all activated ROI nodes are updated accordingly. Once the timing descriptions are set, the simulator is ready to perform the *fault injection* (4b), which allows for injection of logic level faults as well as transistor level parametric faults. The provided input stimuli set is then assigned to the circuit inputs and all nodes of the design are processed in topological order from inputs to outputs (5). Eventually, all output waveforms of the circuit have been computed and are ready for evaluation (6). In case of a fault simulation, *reference responses* provided via the response pattern memory are used to compute the output syndromes. After the evaluation, the ROI marks of all *active* nodes are cleared and the descriptions are restored.

8.3 Transparent Multi-Level Time Simulation

The input to the algorithm is a levelized netlist G with each node being described at either logic or switch level, along with a set of input stimuli that are assigned to all primary and pseudo-primary inputs of the circuit stored in the *waveform memory*. The execution of the algorithm itself is performed again in topological order level by level from inputs to outputs and for each node the multi-level evaluation algorithm is called. The multi-level evaluation at a circuit node is outlined in Algorithm 8.1 [SKW18, SW19b].

Algorithm 8.1: Transparent multi-level simulation algorithm (single thread).

Input: Node n, input waveforms w_i for each input pin $i \in I$
Output: Event sequence of the output waveform w_n

```
1  // Initialize local variables and data structures.
2  Create copy of node description and determine abstraction of node n.
3  // A. initialization
4  foreach node input i ∈ I do
5      Look-up abstraction level of input waveform w_i.
6      Set-up corresponding data structures and determine initial waveform value v_i.
7      Get first event e_i of w_i and put into event schedule E.
8  end
9  Compute initial state S of n and Initialize output waveform w_n.
10 // B. event processing
11 while Events to process in schedule E do
12     // B.1 consume next event
13     Remove earliest event e_i from E.
14     if node n is ROI then
15         Transform e_i to switch level event and compute input value v_i.
16         Compute new switch level state S of n. // switch level kernel
17     else
18         Transform e_i to logic level event and compute input value v_i.
19         Compute new logic level state S of n. // logic level kernel
20     end
21     // B.2 output update
22     if S causes output change on n then
23         Compute output event and add to w_n.
24     end
25     Get next event e_i of input waveform w_i and put into E.
26 end
27 // C. finalization
28 Append sentinel event at time ∞ to w_n.
29 return Output waveform w_n.
```

When a node n is processed, first the abstraction of the node is determined and the functional and timing description are loaded accordingly (line 2). Then the input waveform w_i of each input $i \in I$ of the current node n (line 4) are fetched and the abstraction level that is encoded in the header of each waveform w_i is determined. Depending on the waveform

type, the respective data structures for holding the current events of a waveform are set up as well as the initial waveform value of each waveform is calculated (line 4–8). The first event e_i in each input waveform w_i is fetched and put into the local event table E for scheduling, which keeps the immediate next event to be processed for each node input. After determining the initial signal value $v_i := \boldsymbol{w}_i(-\infty)$ of each input waveform w_i, the initial state S of the node output of n is determined and the output waveform w_n is initialized (line 9). The initialization of the output waveform comprises the annotation of the abstraction level as well as writing the initial node output value of n corresponding to the computed initial state S.

The event processing loop (line 11–26) then processes all the events scheduled in the event table E in temporal order from earliest to latest. The earliest next event e_i is *consumed* (line 13) by removing it from the schedule. Since each event e_i indicates a change in the associated input signal value, the implications of the value change need to be transformed to the targeted abstraction level of the current node under evaluation. In case n is marked as a *region of interest* (ROI), the input signal value needs to be mapped to switch level (line 15) and the new switch level state of n is computed by calling the switch level evaluation kernel (line 16). Otherwise, the input signal is mapped to logic level (line 18) which is then processed using the logic level evaluation kernel (line 19).

If an input event e_i causes a change in the output state of the cell, a new switching event e is appended to the output waveform w_n (line 23), which is a switch level event in case n is marked as ROI and a logic level event, if otherwise. After the event has been processed, the next event of the current input waveform w_i is fetched and scheduled into the event table E. The loop terminates when all events in the event table have been processed and the output waveform is terminated.

8.3.1 Waveform Transformation

The waveform transformation is performed by mapping the respective events from one abstraction level to representative events of the other type during simulation before they are put into the *event schedule*. This way at most one transformation per input waveform event is required at each node during the evaluation. If the abstraction level of an input

waveform matches the one of the node under evaluation, the event processing algorithm can proceed regularly without the need of explicit waveform transformation overhead. In the following, the two waveform transformations are explained in more detail.

Logic-to-Switch Level Transformation

Logic level waveforms are transformed to switch level by mapping the discrete logic value domain $\boldsymbol{E_3}$ to stationary voltages at switch level. Given the voltage levels of the power supply VDD and ground GND potential to represent *high* and *low* logic values, *undefined* values are mapped to the intermediate voltage level $X \rightarrow \frac{1}{2} \cdot (\text{VDD} + \text{GND})$ [Hay86].
All events $e_i \in w$ of the original logic level waveform description are then translated by substituting each event $e_i = (t_i)$ one after another by switch level events e_i'. Based on the targeted logic value of a logic level event e_i and a chosen *time constant* τ_ε, the stationary voltage is selected as follows to obtain the switch level event e_i':

$$(t_i) \mapsto e_i' := \begin{cases} (t_i', \text{VDD}, \tau_\varepsilon) & \text{if } (t_i \textit{ rising}), \\ (t_i', \text{GND}, \tau_\varepsilon) & \text{else if } (t_i \textit{ falling}), \\ (t_i', \frac{1}{2} \cdot (\text{VDD} + \text{GND}), \tau_\varepsilon) & \text{else,} \end{cases} \tag{8.2}$$

where t_i' is the time of the new event e_i' in the corresponding switch level waveform $\tilde{w}$. The time t_i' is computed according to Eq. (6.13), where the original t_i is replaced by the new t_i' and the equation is solved for $x = t_i$, such that the resulting curve segment of e' crosses $\frac{1}{2} \cdot (\text{VDD} + \text{GND})$ Volts at the time of the source logic level transition t_i. For $\tau_\varepsilon \rightarrow 0$, the shape of e_i' closely resembles an instantaneous event with a steepness similar to transitions in the visualized logic level waveforms. The absolute error $\epsilon = |t_i' - t_i|$ of the resulting t_i' can be estimated by using Eq. (6.13), where $t_i' := t_i - \tau_\varepsilon \cdot \log(0.5)$. Since for the last term $|\log(0.5)| < 1$, it follows that the absolute error $\epsilon < \tau_\varepsilon$. Thus, for very small $\tau_\varepsilon \rightarrow 0$ the resulting curve is getting steeper and the shape is well expressing the logic level transition.
The RC-parameters of the input ports can be utilized to fit the curve segment of each resulting event $e_i' := (t_i', v_i', \tau_i')$ by adjusting the time point t_i' and the slope τ_i' of the transition. Again, the curve parameters can be fitted using the *log function* in Eq. (6.13)

to determine the time of an intersection with the threshold. According to the general definition of the *output transition time* [IEE01a], switches at logic level occur at a time point t_i when the underlying signal at the electrical level passes the threshold at $V_{th} = \frac{1}{2} \cdot (\text{VDD} + \text{GND})$ Volts (50% of VDD). Assuming a *rising* (*falling*) transition from a waveform value $\tilde{\boldsymbol{w}}(t_i) = \text{GND}$ (VDD) to the stationary voltage $\overline{v}'_i = \text{VDD}$ (GND), the inner term of the *log*-function delivers

$$\frac{V_{th} - \overline{v}'_i}{\tilde{\boldsymbol{w}}(t_i) - \overline{v}'_i} = \frac{\frac{1}{2} \cdot (\text{VDD} + \text{GND}) - \overline{v}'_i}{\tilde{\boldsymbol{w}}(t_i) - \overline{v}'_i} = \frac{(\text{VDD} + \text{GND}) - 2 \cdot \overline{v}'_i}{2 \cdot (\tilde{\boldsymbol{w}}(t_i) - \overline{v}'_i)}$$

$$(\textit{rising}) \Longrightarrow \frac{(\text{VDD} + \text{GND}) - 2 \cdot \text{VDD}}{2 \cdot (\text{GND} - \text{VDD})} = \frac{-\text{VDD} + \text{GND}}{2 \cdot (-\text{VDD} + \text{GND})} = \frac{1}{2}, \tag{8.3}$$

$$(\textit{falling}) \Longrightarrow \frac{(\text{VDD} + \text{GND}) - 2 \cdot \text{GND}}{2 \cdot (\text{VDD} - \text{GND})} = \frac{\text{VDD} - \text{GND}}{2 \cdot (\text{VDD} - \text{GND})} = \frac{1}{2}. \tag{8.4}$$

Hence, from Eq. (8.3) for the rising transition and Eq. (8.4) for the falling transition parameters the inner *log*-term shortens to

$$\frac{V_{th} - \overline{v}'_i}{\tilde{\boldsymbol{w}}(t_i) - \overline{v}'_i} = \frac{1}{2} = 0.5, \tag{8.5}$$

which can be applied to Eq. (6.13) to finally determine the time t'_i of a corresponding switch level curve with time constant τ'_i that crosses the 50% voltage threshold at the given intersection point t_i:

$$t_i \stackrel{!}{=} t'_i - \tau_i \cdot \log(0.5) \quad \Longrightarrow \quad t'_i := t_i + \tau_i \cdot \log(0.5)\,.$$

This way, the transformed events can reflect the transient response over RC-elements in the circuit, which allows for a more realistic representations of the input waveforms at the current node under evaluation [SKW18].

Fig. 8.4 illustrates the transformation of a binary logic level waveform ("*source*"), for a small time constant $\tau_\varepsilon := 10^{-5}$ ("τ_ε *trans.*") as well as for time-adjusted events based on RC-characteristics ("*RC* 50%"). As shown, the τ_ε-curve overlaps the logic level representation without any noticeable error. In the latter case, the events exhibit the shape of exponential curves that cross the 50% voltage threshold at the exact times of the logic

level transitions. In addition, different RC-characteristics were chosen for the rising and falling transition, which resulted in different time offsets and slopes.

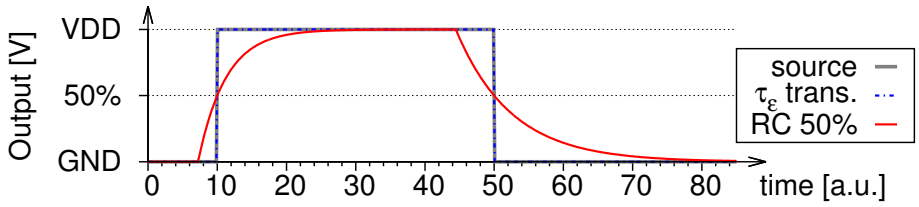


Figure 8.4: Multi-level transformation of a logic level source waveform to switch level. The RC-characteristics vary for the *rising* and *falling* transition [SKW18].

Switch-to-Logic Level Transformation

For the switch level waveform transformation a threshold-based signal characterization from continuous voltage to discrete ternary logic values is applied based on Eq. (7.3). Two thresholds for low ($V_{thL} \in \mathbb{R}$) and high ($V_{thH} \in \mathbb{R}$) values are chosen to partition the voltage range $[\text{GND}, \text{VDD}] \subseteq \mathbb{R}$ into distinct intervals $[\text{GND}, V_{thL})$, $[V_{thL}, V_{thH}]$ and $(V_{thH}, \text{VDD}]$ each corresponding to the value range of a logic symbol in $\boldsymbol{E}_3$. The voltage intervals are assumed to be disjoint and in order to map each voltage value to a unique logic symbol such that

$$\text{GND} < V_{thL} < \frac{1}{2} \cdot (\text{VDD} + \text{GND}) < V_{thH} < \text{VDD}. \tag{8.6}$$

Algorithm 8.2 outlines the general switch to logic level transformation of a waveform. Starting from the initial waveform value $v \in \boldsymbol{E}_3$ at $t = -\infty$ (line 1), the algorithm loops over the events of the switch level signal (line 5) as obtained from the *event table* in temporal order (cf. Chapter 6). Eq. (6.13) is used to determine possible intersection points within each of the curve segment. Each crossing of a threshold level $V_{thL} \in \mathbb{R}$ and $V_{thH} \in \mathbb{R}$ at time t_i then causes a change in the interpreted logic value v of the signal. Therefore the intersection points at times t_i can be directly mapped to corresponding logic level events. Again, for any transition to *undefined*, or from *undefined* to *high* state, the negative event time annotation is applied (line 8) in order to be compliant with the state transition diagram over the ternary logic $\boldsymbol{E}_3 = \{0, 1, X\}$ (cf. Fig. 8.1).

Algorithm 8.2: Threshold-based switch to logic level waveform transformation.

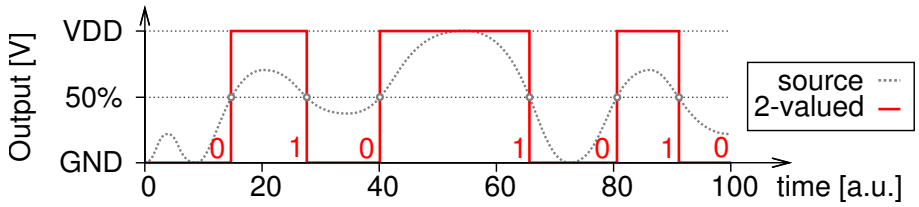
Input: switch level waveform $\tilde{w}$, thresholds V_{thL} and V_{thH}
Output: logic level waveform w

```
1  Compute initial logic state v := val_E(w̃(−∞), V_thL, V_thH).
2  Initialize output waveform w with v.  // v : 0 ↦ ∅, X ↦ {(−∞)}, 1 ↦ {(−∞), (−∞)} (cf. Fig. 8.1)
3  Remember last state v_last := v.
4  // loop over threshold intersection points (using Eq. (6.13)):
5  foreach threshold intersection point t_i of events e_i ∈ w̃ in temporal order do
6      Compute new logic state v := val_E(w̃(t_i), V_thL, V_thH).
7      if ((v = X) ∨ ((v_last = X) ∧ (v = 1))) then
8          Append new event e = (−t_i) to w.  // transition to 'X' or from 'X' to '1' (cf. Fig. 8.1)
9      else
10         Append new event e = (t_i) to w.
11     end
12     v_last := v.
13 end
14 Append sentinel e = (∞) to w.
15 return w.
```

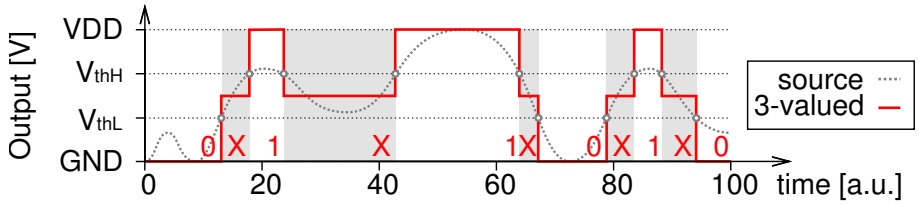
Fig. 8.5 illustrates the low-level signal transformation from an arbitrary continuous voltage waveform to a *Boolean logic* (a) and a *ternary logic* (b) waveform using a threshold interval. For the Boolean logic transformation a single threshold has been chosen at $V_{th} := \frac{1}{2} \cdot (\text{VDD} + \text{GND})$ corresponding to 50% of the VDD to GND voltage level. The thresholds of the ternary logic transformation were chosen as $V_{thL} := 33\%$ and $V_{thH} = 66\%$ in the example. As shown, within these regions signal *uncertainties* are assumed (X).

8.4 Multi-Level Data Structures and Implementation

The implemented multi-level timing simulation approach processes *logic level* (cf. Chapter 5) and *switch level* (cf. Chapter 6) abstractions simultaneously and interchangeably throughout the simulation. Therefore the kernels on the GPU need to process node descriptions and waveform representations of both abstraction types present in the circuit The presented time simulation models at logic and switch level allows for reuse of the implemented data structures and their memory organization to enable an efficient and transparent mixed-abstraction simulation.



a) Transformation to Boolean logic values $\boldsymbol{B}_2 = \{0, 1\}$.



b) Transformation to ternary logic values $\boldsymbol{E}_3 = \{0, 1, X\}$.

Figure 8.5: Threshold-based waveform transformation of an arbitrary continuous signal to a logic waveform [SKW18].

8.4.1 Multi-Level Node Description

In order to distinguish the abstraction of a node, a ROI-flag $ROI \in \{0, 1\}$ is added to each node description, which indicates the level of abstraction as shown in Fig. 8.6. If the flag is *cleared* ($ROI = 0$) the node is simulated at *logic level*. If the flag of a particular node is *set* ($ROI = 1$), it is simulated with *switch level* accuracy. Upon execution of a node kernel by a thread, first the ROI flag and the type information is loaded from the node description memory. This information is utilized to select the evaluation kernel corresponding to the ROI flag, and to set up the data structures for loading the remaining portion of the node description that include the pointers to the input successors and the functional- and timing information.

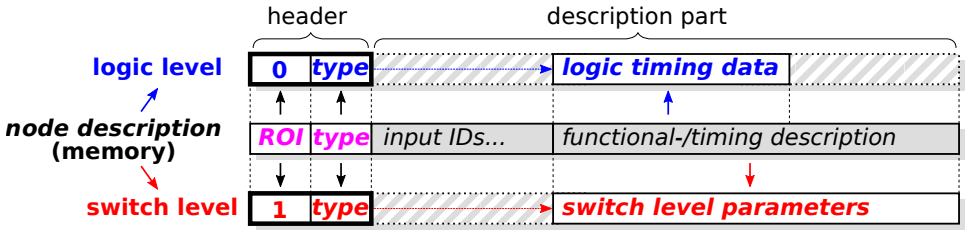


Figure 8.6: Generic multi-level node description struct in memory with logic- and switch level implementations. The ROI flag and type indicator is located in a common header.

Regarding the allocation of the node description in the memory, let $size\_of_{swl}(n) \in \mathbb{N}$ be the amount of memory of the switch level node description of a node n (e.g., in bytes), and let $size\_of_{log}(n) \in \mathbb{N}$ be the size of the corresponding description at logic level. The resulting portion of memory $size\_of_{ML}(n) \in \mathbb{N}$ that needs to be allocated in total for storing either switch or logic level description of a node in the multi level simulation is the maximum of both, hence

$$size\_of_{ML}(n) := \max\left(size\_of_{swl}(n), size\_of_{log}(n)\right). \tag{8.7}$$

While the memory allocated for each node description increases, the resulting node description memory is smaller than storing both descriptions at the same time. In order to perform the *ROI activation* process of any *ROI* or their composites, the associated node descriptions in the node description memory are swapped prior to the actual simulation of the circuit. Each swap transaction comprises the modification of the ROI *flag* in the node description memory, as well as the update of the node description payload describing the actual functional and timing behavior at the respective abstraction. The graph information about the waveform registers of the inputs and the output signal does not have to be changed as the positions of the waveforms in the waveform memory remain untouched. The swap of a description consists of memory transactions of a few hundred bytes only, which allows for a fast activation and deactivation of ROIs.

8.4.2 Multi-Level Netlist Graph

In certain cases, a node description on the higher level has to be substituted by multiple nodes of the lower level. For example, the transistor netlist of a single logic level AND-gate [Nan10] contains two *channel-connected components* that correspond to a NAND and an INV RRC-cell. Hence, for those cases, a one-by-one substitution scheme cannot be applied and a substitution of sub-graphs with different numbers of nodes must be performed.

This work employs *expansion* and *collapsing* for graph structures that require different amounts of nodes in the abstractions. For this, a *fully expanded netlist graph* G^* is constructed that assumes the netlist graph being maximally expanded and with all nodes n substituted by their corresponding larger sub-graphs G_n. This sub-graph is levelized and

embedded into the netlist prior to simulation to determine the *waveform registers* to store the waveforms in the GPU memory. Assuming the sub-graph G_n has depth d, all of its nodes are scheduled on levels L_i through L_{i+d} in their respective topological order. The corresponding node n is always scheduled on the highest level L_{i+d} in the graph G^*.
Both *node expansion* and *node collapsing* are illustrated in Fig. 8.7, where nodes are transformed between switch and logic level. In this example, a logic level node n (AND gate) is represented by a set of two nodes n' and n'' at switch level corresponding to a NAND and INV RRC-cell. Note, that the switch level sub-graph has an additional edge that points to the predecessor waveform of node n'. While the waveform register of n' is the same as of n, node n'' requires allocation of a new waveform register in the memory for storing the additional signal history. Yet, the required memory is freed at latest after processing the head level $(j + 1)$, since there are no further references to the waveforms by other nodes in the graph, but only by nodes in the corresponding sub-graph. While the node n is active on logic level, the memory space for node n'' is marked as *empty* in the netlist. Empty nodes do not execute any function but remain in the netlist, which allows to avoid reorganization of the graph structure which would otherwise cause costly reassignment of the associated *waveform registers*.

8.4.3 Multi-Level Waveforms

Fig. 8.8 depicts the generic waveform data structure in memory for a waveform with a *capacity* of $K \in \mathbb{N}$ *entries*. The memory for the waveform is divided into a header information field of $H \in \mathbb{N}$ entries and the payload for storing the actual waveform events ($K - H$ entries in total). Similar to the node descriptions, a ROI-flag *ROI* $\in \{0, 1\}$ in the waveform header indicates the abstraction level of the waveform in multi-level simulation, which is directly inherited from the *head node* that computes it.
In the example, a value of *ROI* $= 0$ indicates logic level abstraction, where all events events are stored in consecutive entries within the remaining payload area. For *ROI* $= 1$, a switch level waveform is assumed, where the following payload entries are grouped in packs of three for storing the event 3-tuples of the presented switch level simulator. Since the payload data is grouped, the switch level waveforms store less events compared to logic level waveforms within the same amount of memory, which can cause additional wave-

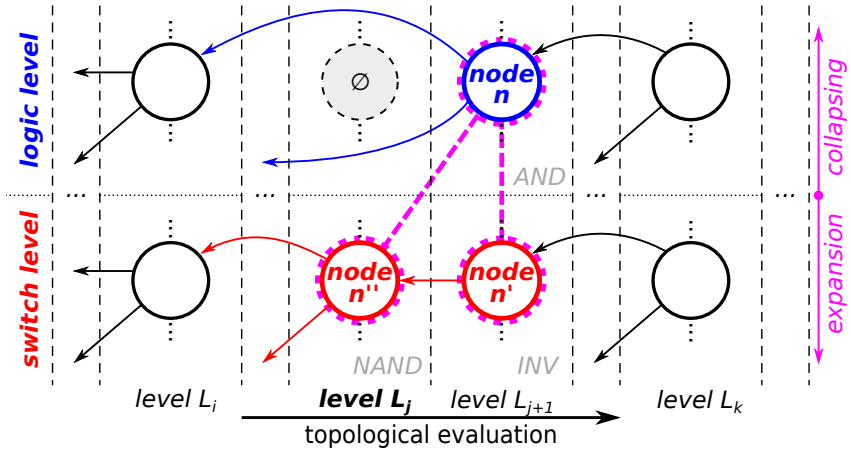


Figure 8.7: Example transformation of a node between two types of abstractions using *node expansion* and *node collapsing*.

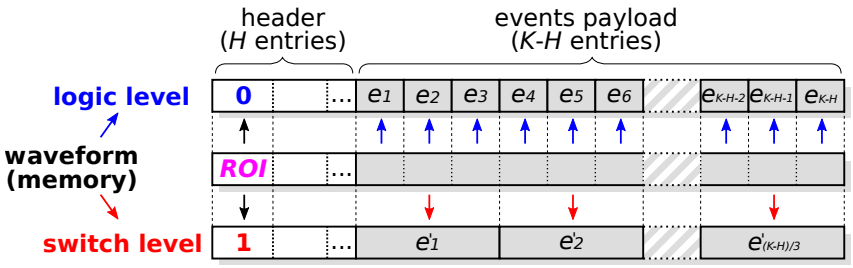


Figure 8.8: Generic waveform data structure with logic and switch level implementation. The ROI flag is located in the header.

form overflows. However, the *waveform calibration* (cf. Chapter 5) will simply increase the *waveform register* size in case *overflows* occur during simulation.

8.4.4 Parallelization

The applied parallelization scheme of the multi-level simulation kernel is depicted in Fig. 8.9, which reflects the general two-dimensional parallelization of the presented high-throughput time simulation (cf. Chapter 5). As shown, the evaluation threads of a node process input waveform sets of either abstraction, which are transformed during the process, and the resulting output waveforms in the output waveform set have the same abstraction as the node itself. Each thread in the horizontal direction of the grid evaluates the same node (of either switch or logic level abstraction) and all threads in the horizontal dimension are processed in the same *thread groups* in parallel but for different stimuli.

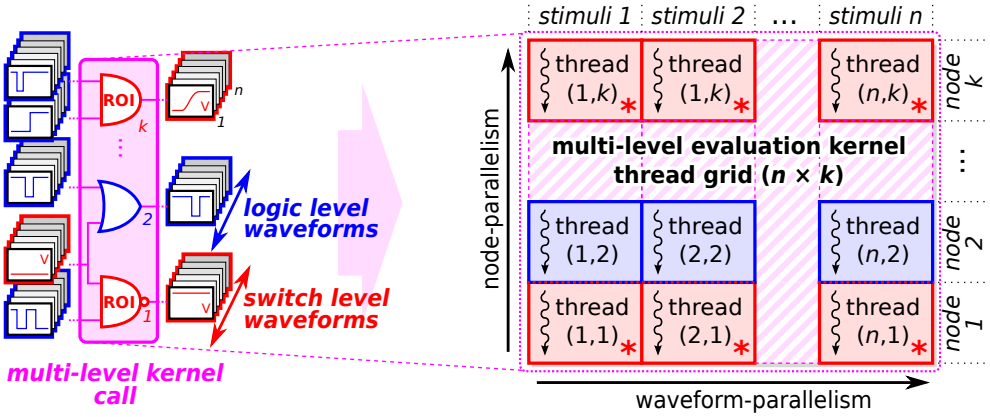


Figure 8.9: Transparent parallelization of the multi-level simulation kernel. Threads processing nodes at switch level are marked with '∗'.

During the waveform processing, the different types of the waveforms are completely transparent to the evaluation kernel, as all events are transformed to the abstraction level of each node before the events are processed. Therefore, although the abstraction levels can vary from node to node throughout the circuit, no additional control flow divergence is caused during execution, since the threads of a thread group always execute the same simulation algorithm and transformations.

8.5 Summary

This chapter presented the first timing-accurate multi-level fault simulation approach for GPUs [SKW18, SW19b] combining fast simulation at logic level with low-level simulation at switch level. It provides a trade-off in speed versus accuracy that is finely controlled by the activation of user-defined *regions of interest* (ROIs) that allow to *locally* lower the abstraction level. By carefully exploiting similarities in data structures and modeling of both abstraction levels, the mixed-abstraction multi-level simulation enables higher simulation efficiency in terms of simulation throughput and modeling accuracy than compared to a full switch level simulation, while being completely transparent to the kernels.

Part III

Applications and Evaluation

Chapter 9

Experimental Setup

This chapter briefly describes the general setup of all the experiments performed in this work. It summarizes the relevant information of the benchmark circuits and software used, and provides an overview of the used hardware as well as the specifications of the used GPU accelerators. The general experimental evaluation part is split into four main sections over the following next chapters:

Chapter 10 provides results of the implemented logic and switch level time simulation,

Chapter 11 presents logic and switch level fault simulation results, and

Chapter 12 investigates the application for timing-accurate power estimation,

Chapter 13 discusses the presented multi-level simulation with mixed abstractions.

9.1 Benchmarks

This work utilizes commonly used circuits from *ISCAS'89* [BBK89] and *ITC'99* [ITC99] benchmarks for evaluation purposes and comparison, in addition to industrial benchmark designs provided by *NXP Semiconductors* (see Appendix A). All designs have been synthesized down to physical layout using the academic NanGate 45nm open cell library [Nan10] in a commercial synthesis tool flow. For all circuits, *full-scan design* is assumed, such that each input and output (either primary or pseudo-primary) is accessible. During the synthesis process, all sequential elements have been removed from the netlist, which were

replaced by pseudo-primary inputs and pseudo-primary outputs thus leaving only the combinational structure of the designs. The timing information of the resulting designs was extracted through *standard delay format* (SDF) [IEE01a] and *detailed standard parasitics format* (DSPF) files [IEE10], which are used as timing annotations at logic and switch level simulation respectively.

9.2 Host System

All simulation experiments were conducted on a host system equipped with different GPUs of the Pascal™ architecture [NVI17c] and its predecessor Kepler™ [NVI14] from NVIDIA® as used in [SW19a, SKW18, SKH+17, SW16, SHK+15, SHWW14].

Table 9.1 shows the specifications of each of the used devices. All devices vary in their chip family (Col. 2), available global device memory (Col. 3) and peak memory bandwidth (Col. 4), number of processing cores (Col. 5) and maximum clock frequency (Col. 6). The last column shows the compute capability of the respective architectures which identifies the available features of the SMs [NVI18b].

The GPU-accelerators were mounted in a host system composed of two Intel® Xeon® E5-2687W v2 processors clocked at 3.4GHz with access to 256GB main memory. The Tesla™ series devices are designed for high-performance computing tasks and provide a larger amount of global memory which also supports ECC-protection. The GeForce™ family devices are targeted for gaming applications providing much higher clock frequencies. The more recent Pascal™ architecture outperforms its Kepler™ predecessor with its additional cores and high memory throughput, thus allowing for a much higher degree

Table 9.1: Specifications of the GPU devices on the targeted host system.

Device Name(1)	Family(2)	Global Memory(3)	Peak Memory Bandwidth(4)	Cores(5)	Maximum Clock(6)	CUDA Arch.(7)
Tesla K40c	Kepler (GK110B)	12GB	288GB/s	2880	745MHz	sm_35
GeForce GTX 1080	Pascal (GP104)	8GB	320GB/s	2560	1.73GHz	sm_61
GeForce GTX 1080 Ti	Pascal (GP102)	11GB	484GB/s	3584	1.67GHz	sm_61
Tesla P100	Pascal (GP100)	16GB	732GB/s	3584	1.33GHz	sm_60

of parallelization and higher simulation throughput. Note that, although originally the GP100 GPU architecture specifies 60 SMs the devices only use up to 56 SMs [NVI17c].

For all devices of the `sm_35`, `sm_60` and `sm_61` architectures the number of maximum processor registers per thread block is 65,536 registers with a limitation of 256 assignable registers per thread. The thread group sizes are 32 threads per group (also called *warp*). On each SM a maximum of 64 active thread groups can be scheduled totalling up to 2048 active theads. The maximum size of a thread block for scheduling is 1024 threads. This information is used to adjust the thread block and thread grid dimensions for each simulation kernel during compilation. Both host systems used in the experiments were running a *CentOS 6* linux distribution with CUDA version 9.1.

9.3 Simulation Environment

All experiments were performed in a diagnostic simulation environment which was implemented in Java 1.8. The general tasks of the simulation environment provide reading in and processing netlists as well as summarizing and evaluating the simulation data obtained. The actual simulation kernels running on the GPU-accelerators were written in CUDA C [NVI18b].

Within the main Java process, the CUDA C binaries are executed as a child process. The Java simulation environment then issues commands via communication over *standard input* and *standard output* streams to the process, such as calls for individual simulation kernels or requests for memory operations (e.g., `cudaMemcpy` or `cudaMemcpu` commands). For exchanging larger amounts of data (i.e., such as netlist information) between host system and GPU devices, mapped-memory files are used that are accessible by both the Java and the CUDA process. The data consistency between both the copies of the memory-mapped files and the memories allocated on the GPU-accelerators is guaranteed by comparison of generation values prior to each transaction of either host or device code.

All GPU-executable CUDA binaries were compiled using the NVIDIA® CUDA compiler driver `nvcc` from CUDA™ version 9.1 for each of the respective architectures and host systems used.

9.4 Data Representation

In order to ease the readability of the experimental results, the order of magnitude of numbers that express certain counts or amounts are expressed using letter symbols. The following Table 9.2 summarizes the notations used in the number representations (Col. 1) and the corresponding order of magnitudes or multiplier (Col. 2). As an example, the notation of "42k" equals to $42 \times 10^3 = 42,000$.

9.5 Performance Metrics

In order to provide a consistent metric for the assessment and comparison of the measured simulation performance across different platforms and simulation tools, the simulation throughput is given in *million node evaluations per second* (MEPS) [SW19a].
Given the number of nodes of a design $\#nodes$, the number of pattern pairs in the test set under evaluation $\#pattern\_pairs$ and the runtime of the simulation T_{sim} for processing the provided test set, the throughput performance of a simulation sim in MEPS is defined as the fraction of million node evaluations and the simulation time:

$$MEPS_{sim} := \frac{\#nodes \times \#pattern\_pairs}{T_{sim} \times 10^6}. \tag{9.1}$$

The throughput metric allows both, the assessment of the performance and scalability of the simulation with respect to the size of a given design in number of nodes, as well as with respect to different numbers of pattern pairs under evaluation.
For relative runtime comparisons of different simulations running on the same host system, the simulation speedup (denoted by 'X') can be derived from the throughput metric. Given simulation throughput from two simulators $MEPS_{sim}$ and $MEPS_{ref}$ running on the

Table 9.2: Number symbols and represented order of magnitude (multiplier).

Symbol[(1)]	Magnitude[(2)]
k	$\times 10^3$
M	$\times 10^6$
B	$\times 10^9$

same host system, the speedup of the simulation sim over ref is obtained by

$$SpeedUp(sim, ref) := \frac{MEPS_{sim}}{MEPS_{ref}} \overset{(9.1)}{=} \frac{T_{ref}}{T_{sim}}. \tag{9.2}$$

Similar to [HIW15, SKH+17, SW19a], a commercial event-driven simulator for timing-accurate evaluation of circuits at logic level was used to provide the baseline reference runtimes. The commercial simulation has been executed on the same host to obtain the respective reference runtimes for comparison. Each reference simulation was running serially, due to the lack of parallelization features. All runtimes stated in the following only reflect the bare simulation time for processing the provided test-set and do not include reading-in, optimizing and compilation of the netlist and timing annotations. Furthermore, the commercial reference simulation does not evaluate the computed waveforms, due to costly file operations, which would otherwise further bias the comparison and put the reference simulation in a disadvantageous position [HSW12, HIW15].

Chapter 10

High-Throughput Time Simulation Results

This chapter evaluates the presented high-throughput simulation approach for the application to logic level and switch level as proposed in Chapter 5 and Chapter 6.

The presented GPU-accelerated logic and switch level simulation were investigated in terms of throughput performance and simulation accuracy. For both logic- and switch level simulation, the test stimuli generated from the commercial ATPG tool were applied. Comparisons were made across device generations with varying amount of compute cores, global device memory and system clock.

In the following, first the logic level simulation results are presented, followed by the switch level simulation results.

10.1 Logic Level Timing Simulation Performance

In the logic level simulation the netlists of the designs under evaluation are annotated timing information from SDF-files [IEE01a] as obtained from synthesis. Throughout the evaluation, both results from uninitialized simulation runs with *uncalibrated* waveform register sizes (cf. Chapter 5), as well as simulation runs with *fitting* waveform register sizes are compared. For each circuit node an initial *waveform register capacity* of $\kappa := 10$ events was chosen. While the uninitialized runs represent the *worst case* simulation scenario involving lots of monitored calibration runs, the simulation runs with fitting waveform register sizes represent the *best case* simulation without the need of register calibration [HIW15].

10.1.1 Runtime Performance

Detailed information about the runtimes and the throughput performance of a fault-free simulation at logic level are provided in Table 10.1. For each circuit listed (Col. 1), column 2 and 3 show the number of pattern pairs of the generated n-detect pattern set ($n = 10$) to process as well as the number of pattern pairs (*slots*) that can be processed in parallel on the GPU (cf. Section 5.4.2). Despite the availability of 16GB of global memory, the thread grid dimensions have been limited to the respective test pattern sets in case the number of provided pattern pairs (cf. Col. 2) is smaller than the number of available *slots* on the GPU (cf. Col. 3). This avoids spawning and scheduling of thread blocks with unnecessary threads on the GPU multi-processors and hence allows to save runtime. In case the number of test patterns to process is larger than the available simulation slots on the GPU, the stimuli are processed as bunches in consecutive simulation runs. Column 4 contains the reference runtime of the simulation using the unparallelized event driven commercial solution for processing the provided pattern set with the base throughput performance in MEPS given in column 5. Next, the initial *worst case* performance of the proposed GPU-accelerated simulation is shown with the absolute runtime given in column 6, the throughput performance in MEPS given in column 7 and the speedup compared to the unparallelized commercial reference simulation in column 8. Finally, columns 9 through 11 present the *best case* simulation runtime performance of the GPU-accelerated simulation.

As shown, the simulation time of the unparallelized commercial simulation (Col. 4) quickly rises from seconds for very small circuits up to minutes for the medium-sized circuit and even up to hours or even days for circuits with more than a million nodes. The measured simulation throughput ranged from a maximum of 2.60 MEPS (circuit b17) to as little as 0.39 MEPS (circuit p3726k) with an average of 1.15 MEPS per circuit. However, the experiments showed that for larger circuit sizes with more than a million nodes (i.e., p951k to p3881k), the general throughput reduces down to 0.57 MEPS in average due to costly memory operations.

As for the implemented GPU-accelerated simulation, the *worst case* runtime (Col. 6) ranges from milliseconds to a few hours with a simulation throughput (Col. 7) ranging from 14.9 (p3847k) up to 207.9 MEPS (p35k). Compared to the reference simulation times,

Table 10.1: Logic level simulation performance (NVIDIA Tesla P100 accelerator).

Circuit[1]	Pattern-Pairs		Logic Level Simulation (*Fault-Free*)							
			Event-Driven		Worst (GPU)			Best (GPU)		
	source[2]	*max.*[3]	Time[4]	MEPS[5]	Time[6]	MEPS[7]	X[8]	Time[9]	MEPS[10]	X[11]
s38417	348	53.1k	3.11s	2.13	408ms	16.2	8	12ms	551.0	260
s38584	563	44.8k	5.29s	2.45	428ms	30.3	13	18ms	721.0	294
b14	904	147.0k	5.30s	1.62	426ms	20.2	13	12ms	716.1	442
b17	2135	59.6k	35.13s	2.60	876ms	104.3	41	60ms	1522.2	586
b18	3174	18.5k	6:42m	0.99	2.86s	139.2	141	261ms	1523.8	1542
b19	4651	9728	0:17h	1.14	7.92s	146.9	130	696ms	1672.2	1469
b20	1097	90.4k	14.41s	1.40	611ms	33.0	24	23ms	876.8	627
b21	1154	88.6k	16.60s	1.34	559ms	39.7	30	22ms	1009.9	755
b22	1190	62.9k	25.39s	1.31	618ms	53.6	42	31ms	1069.0	819
p35k	4096	21.2k	2:22m	1.38	1.04s	188.9	137	114ms	1724.5	1249
p45k	2417	25.3k	49.96s	2.13	738ms	144.4	68	68ms	1567.4	735
p77k	1979	24.4k	3:02m	0.76	4.81s	29.0	38	158ms	883.4	1156
p81k	795	13.6k	2:12m	0.83	1.13s	96.8	118	63ms	1742.5	2110
p89k	2460	15.3k	2:26m	1.64	1.36s	175.9	108	133ms	1804.2	1103
p100k	2809	14.9k	3:17m	1.37	1.62s	166.6	122	159ms	1699.0	1242
p141k	2043	8128	5:34m	1.09	2.54s	143.4	132	227ms	1602.6	1473
p267k	3181	6048	0:14h	0.81	3.34s	207.9	257	573ms	1212.6	1498
p330k	5928	5248	0:38h	0.74	9.79s	173.6	235	2.28s	746.9	1009
p418k	3676	3424	0:29h	0.92	20.66s	78.3	86	3.00s	538.9	586
p500k	5012	3200	0:49h	0.89	25.09s	105.3	118	3.07s	859.5	962
p533k	3417	2336	1:07h	0.57	30.02s	77.0	135	3.04s	761.8	1331
p951k	7063	1024	3:00h	0.71	2:45m	46.6	66	11.95s	644.3	905
p1522k	17980	1440	8:21h	0.65	5:04m	64.2	99	23.66s	827.2	1273
p2927k	22107	768	18:17h	0.56	0:18h	33.7	61	58.25s	634.7	1131
p3188k	26502	576	42:02h	0.50	1:02h	20.3	41	2:07m	591.3	1185
p3726k	15512	480	39:24h	0.39	0:40h	22.8	59	1:38m	558.4	1434
p3847k	31653	480	40:12h	0.65	1:44h	14.9	24	3:08m	496.7	768
p3881k	12092	352	23:53h	0.52	0:44h	16.8	33	1:47m	413.4	797

the GPU-accelerated approach was able to increase the simulation speed in the range from 8× to 257× (Col. 8). For smaller circuits, the speedups are lower compared to the medium-sized circuits due to under-utilization of the resources. However, also for the larger circuits, the speedup decreases again, due to the increasing calibration management, which is handled by the (serial) host system as well as due to the serialization of the stimuli set processing.

In the *best case* simulation scenario (Col. 9–11), the simulation runtimes (Col. 9) ranged from milliseconds to few minutes only even for the multi-million node designs. The speedups obtained by the GPU-acceleration (Col. 11) ranged from 260× (circuit s38417) up to 2110× (p81k) and the maximum simulation throughput (Col. 10) increased up to 1804.2 MEPS (p89k). The average throughput over all circuits was 1034.7 MEPS. Again,

due to under-utilization of the computing resources, the simulation throughput of the smaller circuits is much lower compared to the larger ones.

10.1.2 Simulation Throughput

The simulation data of Table 10.1 suggests that the throughput saturates the larger the circuits and test sets get. Thus, for any larger simulation problem the simulation throughput is expected to be constant. The general trend of the simulation performance is illustrated in Fig. 10.1, which shows the simulation throughput with respect to the size of the circuit in number of nodes. Each point in the graph reflects the measured throughput in MEPS for the *worst-* and *best-case* simulation scenarios. The lines show the average of the data points for the two simulation scenarios (dashed line for *worst-* and full stroke for *best-case*). While the trend of the *worst-case* simulation is strongly decreasing for increasing circuit size due to increased waveform reallocation overhead, the trend line of the *best case* simulation is flattening and expected to saturate, due to the limited computing resources available on the GPU devices.

At this point, further increase in throughput can generally be achieved by either using GPU devices with an increased number of parallel computing resources and memory, or by using multiple GPU devices in parallel on the host system. Since the *stimuli parallelism* assumes that the individual test stimuli in the provided test sets are mutually independent, and for the larger circuits the stimuli are processed in many consecutive simulation runs (cf. Table 10.1), the test sets can simply be divided and distributed evenly among the different devices for a *stimuli-parallel* execution across GPUs.

The simulation throughput of the proposed high-throughput logic level simulation was further explored for different GPU architectures. For this, the simulation was re-run on a host system equipped different accelerators with all executable binaries having been compiled and executed for each device architecture respectively. Fig. 10.2 depicts the general simulation performance across the devices.

The left graph depicts the throughput performance in the proposed MEPS metric for both *worst-case* (dotted lines) as well as *best-case* simulation runs (full strokes). The devices on the X-axis are sorted in increasing order of the compute capability. As shown, the simulation throughput rises with each new architecture, which relates to the increasing

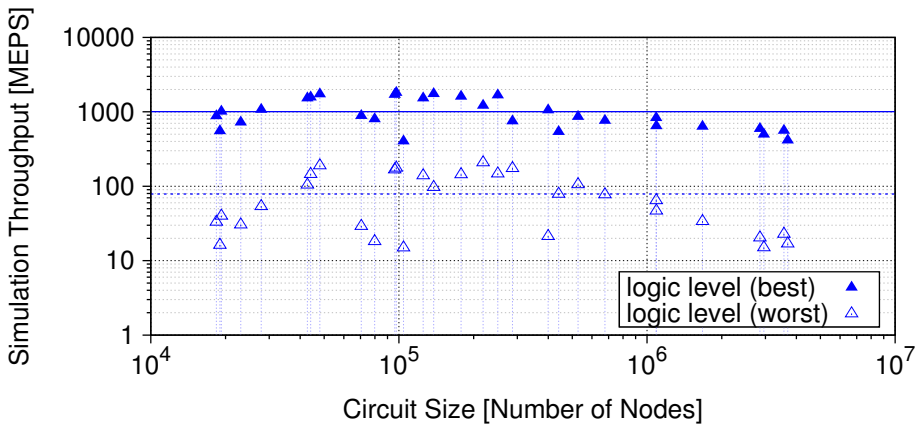


Figure 10.1: Logic level simulation throughput for *worst-* and *best-case* simulation. Both, the X and Y axis are log-scale. Full lines denote averages.

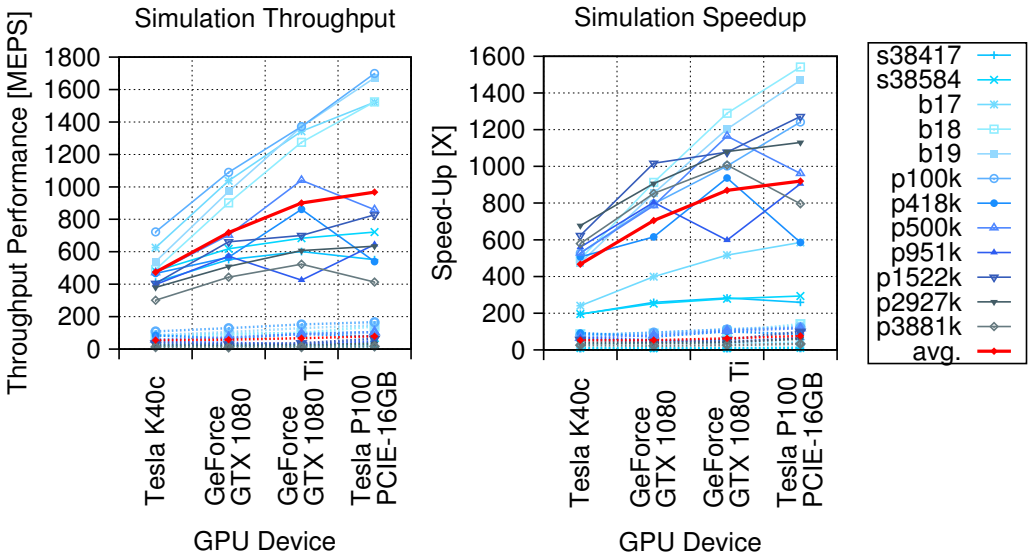


Figure 10.2: Simulation throughput (left) and speedup (right) of the logic level simulation executed on individual GPU devices with different specifications (cf. Table 9.1). Full strokes represent *best-case* and dashed lines represent *worst-case* simulation results.

numbers of available cores and device memory as well as the increasing clock speed. The more recent Pascal™ architecture outperforms its Kepler™ predecessor allowing for a much higher degree of parallelization and thus for higher simulation throughput. Yet, regarding the calibration runs, the simulation time is dominated by memory calibration and allocation resulting in lower speedups. In average, a throughput increase of over 505 MEPS (double the base speedup) was observed from the oldest to the newest device used in the experiment, with a maximum throughput gain of roughly 1133 MEPS for

circuit b19. For the worst case simulation runs, the throughput is seemingly constant with little gain, due to the high waveform register calibration overhead.
The right-hand side of Fig. 10.2 shows the measured speedup of the simulation compared to the serial commercial logic level simulator. The improvement in speedup over the device architectures is similar.

10.2 Switch Level Timing Simulation Performance

Input to the switch level simulator is the synthesized netlist, the pre-characterized transistor parameters and annotations of electrical parameters from the layout stored in the *detailed standard parasitics format* (DSPF) [IEE10]. For the transistor model cards, the 45nm predictive technology model (PTM) was used [ZC06, Nan17] as utilized by the standard cell library cells [Nan10]. The resistive parameters and the threshold voltages of the transistors have been manually extracted via the IV-characteristics [WH11] from SPICE simulations using a sweep of the gate-source voltages $V_{gs} \in [0\text{V}, 1.1\text{V}]$ with the drain-source voltages set to $V_{ds} := 1.1\text{V}$.
The resulting transistor descriptions with *threshold voltage*, *blocking* and *conducting* drain-source resistance were $D^P := (-0.3021\text{V}, 12.81\text{M}\Omega, 1145.1\Omega)$ for the PMOS and $D^N := (0.3220\text{V}, 28.51\text{M}\Omega, 1550.6\Omega)$ for the NMOS-type transistors respectively, which were used throughout the experiments with the presented switch level simulation. From the DSPF file the electrical parameters of the interconnections (both resistances and capacitances) were extracted. These parameters were assumed as lumped at the RRC-cell node outputs.

10.2.1 Runtime Performance

The measured runtimes are compared to simulation at logic level as shown in Table 10.2. In the table, column 2 to 3 contain the size of the input stimuli set in pairs as well as the maximum pattern pairs able to process concurrently on the GPU device (16GB global device memory). Column 4 and 5 show the simulation runtime at logic level of the serially executed commercial event-driven approach and the presented GPU-accelerated parallel simulation. As previously, runtimes are split into worst and best case over column 6

through 11 with the simulation runtime, the throughput performance in MEPS and the speedup over the event-driven approach given for each case.

As shown, the switch level simulation runtimes range from 513ms (circuit s38417) up to 5.5 hours (circuit p3847k) in the worst case, corresponding to a speedup of over 25× over the event-driven approach in average. The simulation throughput decreases for the larger designs due to the increased calibration overhead. In the best case simulation, the throughput performance increased by an average factor of over 5× thereby reducing the general runtimes to a maximum of few minutes.

For the million-node designs (circuits p951k–p3881k) the parallelization showed to be more effective, due to a better utilization of the accelerator hardware during computation. Despite the higher modeling accuracy, the simulation speed of the implemented switch level simulation is only one order of magnitude slower (average of roughly 9×) compared to the presented GPU-accelerated simulation at logic level.

Note that in the proposed high-throughput parallel simulation model, the amount of memory for storing a switch level event is three times higher compared to storing a logic level event. Hence, although the initial capacity for each waveform register was chosen the same ($\kappa = 10$ events), less stimuli pairs can be simulated in parallel (Col. 3).

10.2.2 Throughput performance

Fig. 10.3 illustrates the measured simulation throughput with respect to the circuit size. Again, each point reflects the obtained simulation performance in MEPS of a circuit. Bold points represent the best case and clear points represent the worst case. Similarly, the lines reflect the average trends of the simulation data.

As shown, the simulation throughput of the worst case scenario is in average 24.6 MEPS and quite scattered. However, it quickly decreases for the larger designs, due to increasing calibration overhead. In the best case simulation scenario, the throughput remains roughly constant around 128 MEPS. For the million-node designs, the simulations showed a simulation throughput of even 145 MEPS in average, leading to the highest speedups obtained in the experiments. Here, the throughput is expected to be saturated and constant due to a full utilization of the computing resources. In contrast to the logic level application, the runtime of the switch level algorithm is strongly dominated by floating point

Table 10.2: Switch level simulation performance (NVIDIA Tesla P100 accelerator).

Circuit[1]	Pattern-Pairs		Logic-Level		Switch-Level					
			Event-	GPU[4]	Worst (GPU)			Best (GPU)		
	source[2]	*max.*[3]	Driven[4]		Time[6]	MEPS[7]	X[8]	Time[9]	MEPS[10]	X[11]
s38417	348	17.7k	3.11s	12ms	513ms	12.9	7	52ms	127.1	60
s38584	563	14.9k	5.29s	18ms	646ms	20.1	9	87ms	149.2	61
b14	904	49.0k	5.30s	12ms	685ms	12.5	8	100ms	85.9	53
b17	2135	19.9k	35.13s	60ms	2.91s	31.4	13	671ms	136.1	53
b18	3174	6144	6:42m	261ms	15.24s	26.1	27	3.56s	111.9	114
b19	4651	3040	0:17h	696ms	39.61s	29.4	26	10.40s	111.9	99
b20	1097	30.1k	14.41s	23ms	1.34s	15.0	11	227ms	88.8	64
b21	1154	29.5k	16.60s	22ms	1.43s	15.5	12	256ms	86.8	65
b22	1190	21.0k	25.39s	31ms	1.83s	18.1	14	370ms	89.6	69
p35k	4096	7040	2:22m	114ms	4.29s	45.8	34	1.48s	132.6	96
p45k	2417	8416	49.96s	68ms	2.53s	42.2	20	869ms	122.7	58
p77k	1979	8128	3:02m	158ms	9.81s	14.2	19	1.96s	71.3	94
p81k	795	4512	2:12m	63ms	3.17s	34.6	42	711ms	154.4	187
p89k	2460	5088	2:26m	133ms	5.25s	45.7	28	1.51s	159.1	98
p100k	2809	4960	3:17m	159ms	7.40s	36.5	27	2.07s	130.4	96
p141k	2043	2688	5:34m	227ms	10.31s	35.3	33	2.67s	136.5	126
p267k	3181	2016	0:14h	573ms	15.15s	45.9	57	5.17s	134.4	166
p330k	5928	1856	0:38h	2.28s	43.01s	39.5	54	12.77s	133.2	180
p418k	3676	1120	0:29h	3.00s	50.74s	31.9	35	10.02s	161.6	176
p500k	5012	1056	0:49h	3.07s	1:32m	28.7	33	19.14s	138.0	155
p533k	3417	768	1:07h	3.04s	1:20m	28.7	51	18.55s	124.6	218
p951k	7063	352	3:00h	11.95s	6:22m	20.1	29	44.88s	171.6	241
p1522k	17980	512	8:21h	23.66s	0:18h	17.2	27	2:13m	146.9	226
p2927k	22107	256	18:17h	58.25s	0:45h	13.6	25	3:56m	156.6	279
p3188k	26502	192	42:02h	2:07m	2:54h	7.2	15	9:14m	136.2	273
p3726k	15512	160	39:24h	1:38m	2:07h	7.2	19	7:57m	115.7	297
p3847k	31653	160	40:12h	3:08m	5:24h	4.8	8	0:11h	133.9	208
p3881k	12092	128	23:53h	1:47m	1:18h	9.4	19	4:56m	150.6	291

operations rather than control flow or initialization overhead, resulting in a somewhat constant throughput performance over all circuits.

Fig. 10.4 shows the achieved switch level simulation throughput on different GPU devices for different circuits as well as the achieved speedup compared to the event driven logic level simulation. Points connected by dashed lines represent the initial worst case runtime and full lines represent the best case times with calibrated waveform registers, respectively. The red lines represent the average over the shown circuits.

Again, while for the worst case scenario, the simulation time is dominated by the waveform calibration overhead resulting in an average simulation throughput ranging from 10 to 45.9 MEPS for the circuits shown. In the best case simulation, an average performance gain of 100 MEPS was observed throughout the devices with the highest throughput measured for circuit p951k with 171.6 MEPS on the Tesla P100 accelerator. Throughout the

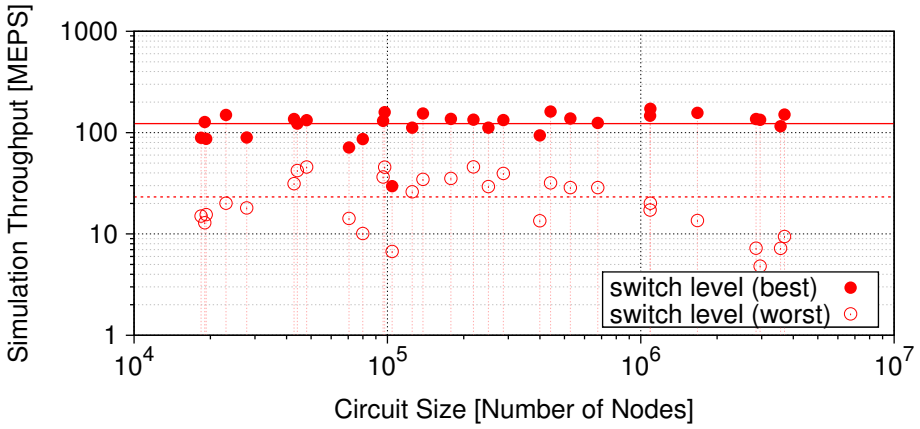


Figure 10.3: Switch level simulation throughput performance with average trend.

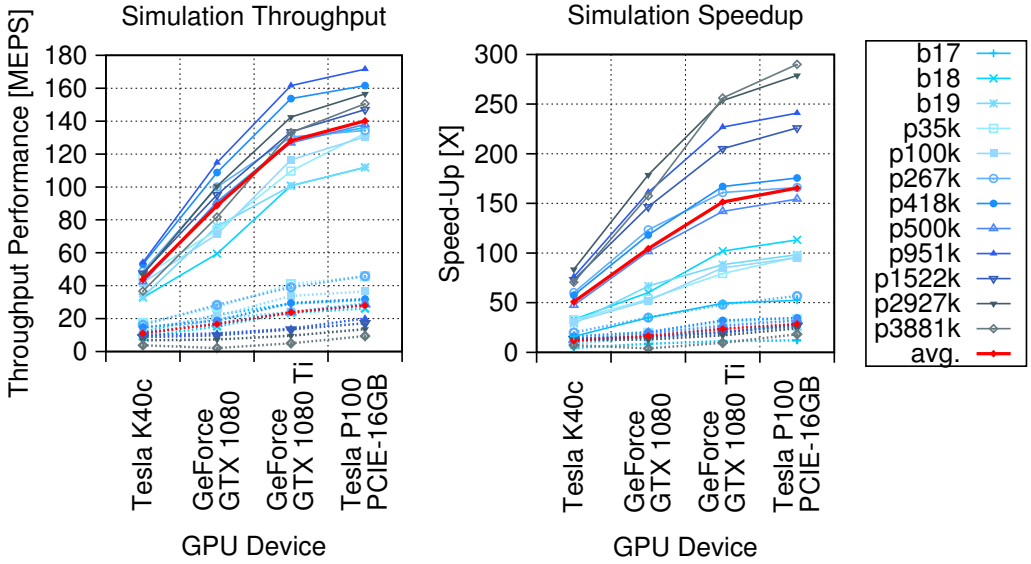


Figure 10.4: Switch level simulation throughput (left) and simulation speedup (right) on different GPU devices.

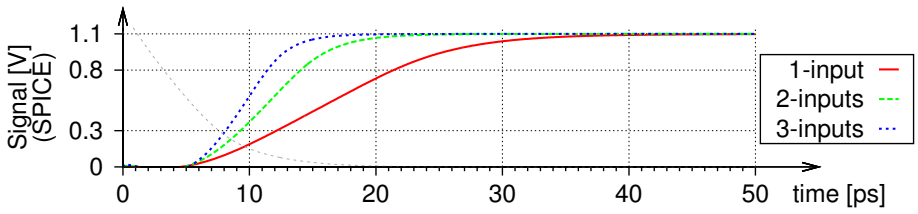
experiments, the Tesla P100 device showed the highest average performance of all devices under investigation. Although the maximum clock (1.33GHz) of the Tesla P100 accelerator is lower than that of the GeForce GTX 1080 Ti (1.67GHz), its larger global memory of 16GB (compared to 11GB) still allows for higher utilization and more parallelism.

Regarding the speedup, a similar behavior was observed with an average gain of roughly 110× over all devices. While the gain in speedup is in general lower for the smaller circuits due to under-utilization of the hardware, the larger circuits can profit from the larger amount of available global device memory and higher parallelism.

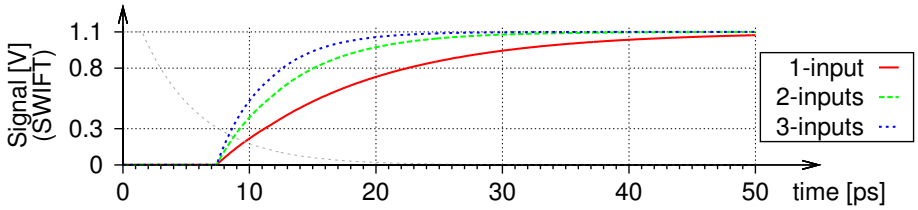
10.3 Switch Level Behavior

To assess the accuracy of the presented switch level simulation (SWIFT), the waveform results are compared to electrical simulation (SPICE) and logic level simulation (LOGIC). Here, the focus of the evaluation is solely the quality of the model for representing electrical behavior.

In the following, transition ramps and the impact of pattern-dependent delay effects due to multiple input switching (MIS) on the signal propagation are investigated on a small three-input NAND3_X1 cell as shown in Fig. 10.5. Initially, all cell inputs are *high* (non-controlling) and the output is *low*. In consecutive simulation runs, the input signals were set to *low* forming a falling transition at one, two and three input pins simultaneously. The switching at an input causes the associated PMOS transistors to conduct, which charges the output load of the cell (rising transition). The additional output transition delay resulting from Miller-effects (over/undershoot due to dynamic capacitances within the cell) is compensated by applying a constant delay offset to the output transition in the switch level waveform.



a) Output waveform of three-input NAND cell in SPICE.



b) Output waveform of three-input NAND cell in SWIFT.

Figure 10.5: Pattern-dependent slope of a three-input NAND3_X1 cell output [Nan10, ZC06] under single and multiple simultaneous (falling) transitions at the input pins.

Fig. 10.5 depicts the resulting output waveforms in SPICE and SWIFT. As shown, the slope of the output signal gets steeper, the more inputs are switching, since the PMOS are arranged in parallel and the effective conductance of the charging is proportional to

the number of conducting transistors. Since SWIFT always computes the resistance of the transistor networks based on all transistor states, it is able to reflect the pattern-dependent conductance, and hence, the aforementioned MIS-effect. Note that conventional simulation at logic level does not reflect this kind of behavior.

In the next experiment, the signal propagation over multiple states was observed. For this experiment, an artificial circuit as depicted in Fig. 10.6 was synthesized to layout. The circuit is composed of a chain of 16 two-input NAND cells (U1 to U16) with identical properties, which are connected in a way, such that the outputs of each cell are connected to both input pins of their respective successors. As input stimulus, a rising transition is provided, which propagates through the chain.

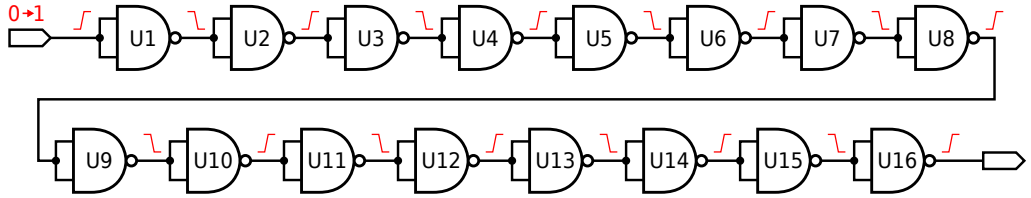
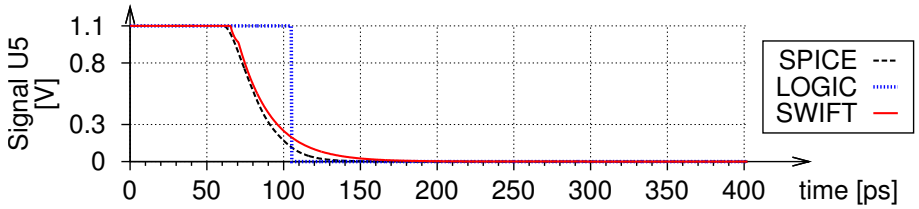
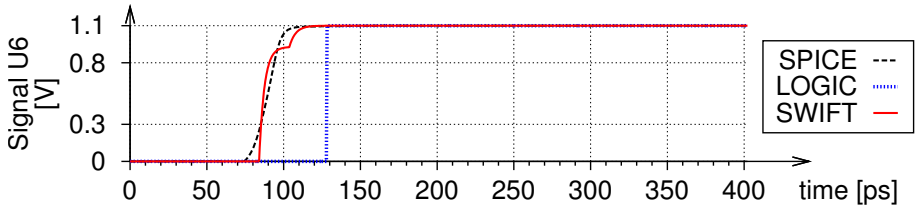


Figure 10.6: Test circuit comprising a chain of 16 two-input NAND cells (U1 to U16) for the observation of pattern-dependent delay effects during signal propagation.

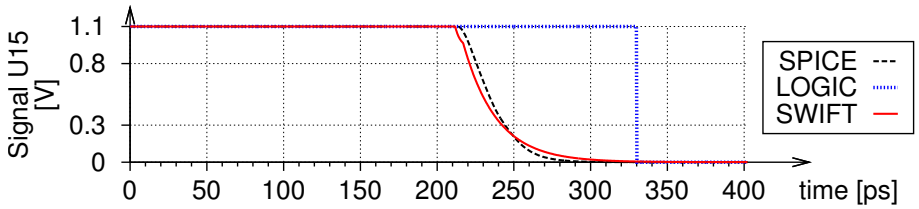
Fig. 10.7 shows the waveforms of intermediate signals obtained from electrical simulation (SPICE), logic level simulation (LOGIC) and the presented switch level simulation (SWIFT) [SW19a]. As shown, SWIFT is able to reflect the pattern-dependent delay effect, which is completely ignored during logic level simulation. These effects are observable in both SPICE and SWIFT simulation at the outputs of all even stages (signals U2, U4,..., U16). Since at these stages, both inputs of the cells have a falling transition, the output load of the cells is charged via two conducting PMOS transistors in parallel. Therefore, the slopes of the (rising) output signals is steeper (Fig. 10.7-b) and -d)) compared to a single input switch. By ignoring these effects, the logic level simulation introduces small errors at all (even) stages which accumulate during propagation through the design up to a delay deviation of roughly 30 percent of the reference delay from SPICE. As shown before, these deviations are even more severe for three- or even four-input cells. In SWIFT, the transition ramps as well as the transition points in time correlate well with the SPICE result also in presence multiple input switches.



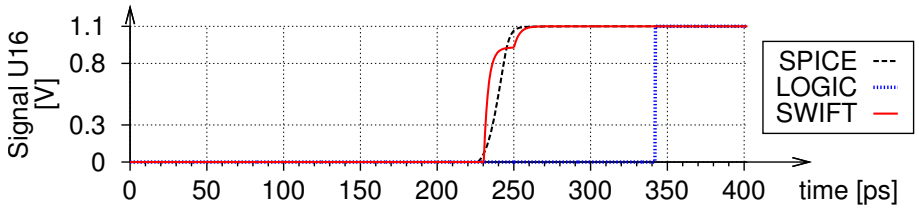
a) Signal waveforms at stage 5: Falling transition.



b) Signal waveforms at stage 6: Rising transition with MIS effect.



c) Signal waveforms at stage 15: Falling transition.



d) Signal waveforms at stage 16: Rising transition with MIS effect.

Figure 10.7: Waveforms compared for electrical (SPICE), logic (LOGIC) and the presented switch level timing simulation (SWIFT) of a signal transition propagating through a chain of two-input NAND cells [SW19a].

10.4 Summary

In this chapter, the proposed high-throughput parallel simulation was investigated for the application to logic level (cf. Chapter 5) and switch level timing simulation (cf. Chapter 6). On current GPU-accelerator devices, the presented simulation schemes allow to occupy the available streaming multi-processor computing resources and fully utilize the memories, providing constant simulation throughput and being able to scale even for designs with millions of nodes and thousands of test stimuli.

Experimental results have shown, that the proposed parallel logic level simulator allows for speedups of up to three orders of magnitude compared to a unparallelized commercial solution, which significantly reduces the runtime of timing-accurate simulations at logic level. Despite the higher modeling complexity, the proposed switch level model also allows for speedups of up to two orders of magnitude compared to unparallelized logic level simulation. Experiments have shown that the simulation model is able to capture significant electrical behavior, such as transition ramps and multiple input switching effects, in an efficient and accurate manner.

The timing-accurate evaluation is crucial in many fields, such as delay fault simulation, power estimation, low-power test and circuit diagnosis that heavily rely on simulations with accurate switching activity information. With the power of parallelization on GPU-architectures, the simulation throughput allows to vastly increase the simulation speed and to reduce the overall runtime of the evaluations. The parallel simulation is highly scalable and thus opens new opportunities for a wide range of applications that are applicable to simulation problems at a larger scale.

Chapter 11

Application: Fault Simulation

This chapter evaluates the extensions of the logic level and switch level simulation for the use in a fault simulation environment. Both small delay faults (at logic level) as well as parametric faults (at switch level) are investigated as presented in [SHK+15, SKH+17] and [SW16, SW19a].

Given an initial fault universe, equivalent faults are first removed using the *fault collapsing* instructions as provided in Section 7.4.1 for both small delay faults and transistor-level faults, respectively. The presented fault grouping algorithm (cf. Section 7.4.3) is then applied to partition the *collapsed* fault set into *fault groups* for parallel injection and simulation. The *effectiveness* (eff_{fsim}) of the fault grouping is defined as the fraction of the size of the fault set for simulation divided by the number of obtained fault groups:

$$eff_{fsim} := \frac{\#FaultsInFaultSet}{\#ObtainedFaultGroups}. \tag{11.1}$$

Except where otherwise mentioned, the fault universes of the simulations are chosen as follows. As fault universe for the small delay faults a spatially exhaustive fault set is defined that considers fault locations for each pin of a cell (input and output pin) [SKH+17]. For each fault location, two small delay faults are assumed, which affect either the rising or the falling transition. Given the nominal circuit clock frequency from topological timing analysis, the *size* of each small delay fault is finite and chosen *half* between the slack of the longest and shortest path through the associated fault site.

The fault universe of the switch level parametric faults considers a high resistive open fault at each transistor as well as a high capacitive fault in the output load capacitance of each cell in the circuit [SW19a].

11.1 Fault Grouping Results

Table 11.1 summarizes the results of the fault collapsing and fault grouping for all the benchmark circuits. Column 2 and 3 show the initial size of the small delay fault universe defined previously as well as the remaining number of faults after fault collapsing. Column 4 contains the number of fault groups obtained after fault grouping of the collapsed fault set with the fault grouping efficiency (eff_{fsim}) given in column 5. Similarly, column 6 through 9 summarize the results for the switch level parametric faults.

As shown in column 2 and 3, the fault collapsing was able to reduce the size of the initial small delay fault universe by 38 percent in the worst (p330k) to 49 percent in the best case (p35k) with an average reduction of approximately 41.7 percent for all the circuits listed. The grouping heuristic was able to partition the collapsed fault set into fault groups with a fault grouping efficiency ranging between 1.9 in the worst (p35k) and 219.8 in the best case (p533k) as shown in column 4 (48.9 in average). For each circuit, eff_{fsim} directly provides the average number of faults that are processed during the simulation of the fault groups. It also indicates the degree of the available fault parallelism as well as speedup of the proposed parallel fault simulation approach over a naïve serial simulation without fault collapsing. For the circuit p35k, the grouping seems to be less effective, possibly due to a high number of reconvergent fanout fault locations causing a large number of output cones from distinct fault sites to overlap. Yet, even in case of p35k with an average group size of 1.9 faults per fault group, almost 50 percent (≈54k) of the simulation runs can be saved compared to the naïve simulation. On the other hand, circuit p533k has fewer reconvergences and more mutually data-independent nodes allowing to reduce the number of overall simulation runs substantially by approximately 99.5 percent with a fault grouping efficiency of over 219. The runtime of the grouping heuristic for the small delay fault sets ranged from less than a second to tens of minutes for the million-node designs [SKH+17], which also includes the fault collapsing. This is a negligible amount of

Table 11.1: Fault collapsing and grouping statistics for small delay faults at logic level [SKH+17] and parametric faults at switch level [SW19a].

Circuit[1]	Small Delay Faults				Transistor-Level Faults			
	Faults		Groups[4]	eff_{fsim}[5]	Faults		Groups[8]	eff_{fsim}[9]
	initial[2]	*collapsed*[3]			*initial*[6]	*collapsed*[7]		
s38417	80.9k	46.1k	1744	26.5	68.7k	59.5k	2084	28.5
s38584	104.6k	59.5k	2010	29.6	88.0k	75.4k	2610	28.9
b17	215.1k	127.5k	15.6k	8.2	178.3k	150.3k	18.5k	8.1
b19	1.28M	770.1k	36.5k	21.1	1.06M	891.1k	39.6k	22.5
b20	94.6k	58.2k	11.1k	5.2	78.3k	65.7k	12.7k	5.2
b21	99.3k	60.9k	11.2k	5.5	82.1k	68.9k	12.7k	5.4
p35k	221.3k	113.9k	59.7k	1.9	183.6k	158.7k	82.4k	1.9
p45k	203.1k	118.6k	9272	12.8	171.5k	145.6k	13.4k	10.9
p77k	344.3k	205.5k	64.9k	3.2	287.5k	242.7k	72.3k	3.4
p81k	678.0k	380.4k	16.8k	22.7	555.9k	477.1k	20.3k	23.5
p89k	471.1k	266.6k	13.7k	19.4	391.9k	333.1k	16.1k	20.7
p100k	456.1k	275.9k	9272	29.8	383.4k	324.2k	13.4k	24.3
p141k	829.5k	472.6k	42.5k	11.1	695.0k	592.8k	55.8k	10.6
p267k	996.8k	591.9k	9710	61.0	846.2k	716.8k	14.2k	50.6
p330k	1.36M	842.7k	64.1k	13.1	1.15M	969.2k	85.4k	11.3
p418k	2.00M	1.12M	18.6k	60.2	1.68M	1.44M	22.7k	63.6
p500k	2.46M	1.41M	21.3k	66.0	2.05M	1.76M	24.3k	72.3
p533k	3.32M	2.03M	9248	219.8	2.77M	2.33M	10.7k	217.5
p951k	4.62M	2.53M	13.5k	187.9	3.92M	3.40M	20.2k	168.3
p1522k	5.13M	2.94M	65.7k	44.8	4.32M	3.65M	77.8k	47.0
p2927k	7.77M	4.46M	32.3k	137.9	6.49M	5.56M	37.0k	150.1
p3188k	13.73M	8.42M	1.29M	6.5	11.48M	9.72M	524.5k	18.5
p3726k	17.26M	9.88M	159.7k	61.9	14.31M	12.19M	201.8k	60.4
p3881k	16.66M	9.79M	82.1k	119.3	14.04M	12.02M	111.2k	108.1

time that is spent for pre-processing compared to the (dominating) simulation effort that is required to process the individual groups serially.

As for the switch level parametric fault set (Col. 6–9), the fault collapsing is less efficient, since fault equivalences are only identified within each individual cell, but not across different cells. As shown, the amount of faults in the initial fault universe could only be reduced by an average of 14.9 percent. The grouping heuristic was able to partition the faults into groups with a grouping efficiency between 1.9 to 217.5 (Col. 9) with an average efficiency of 48.4 over all circuits. Due to the topological dependencies of all fault locations involved, the grouping results and the obtained fault grouping efficiencies of the switch level fault set (Col. 9) are similar to those of the small delay fault set for the individual circuits. The runtimes of the fault grouping of the switch level fault set are larger than those for the small delay fault sets [SW19a, SKH+17]. Since more groups have to be checked for mutual data-independence of the faults, the runtimes now range from

seconds to a few hours. Yet, in comparison to the switch level simulation overhead, this time is still negligible.

11.2 Small Delay Fault Simulation

Table 11.2 compares the fault coverage of both *transition fault* (TF) and *small delay fault* (SD) from spatially-exhaustive fault simulation of the collapsed fault set and the fault groups from Table 11.1. Given the nominal circuit frequency, the size of the small delay fault at a particular location is finite and chosen half between the slacks of the longest and shortest paths through the associated fault site obtained from topological timing analysis. As test stimuli the ATPG-generated 10-detect transition delay fault pattern sets are used.

In the table, column 2 contains the number of collapsed faults in the fault set under investigation. All faults are evaluated for all stimuli in the test set, thus, no fault dropping is used during the process, which allows to obtain detection information for each fault and every stimuli. Columns 3 and 4 contain the number of detected transition faults (”*TD det.*”) as well as the amount of small delay faults (”⊇*SD und.*”) that have not been detected at these respective locations. Similarly, the number of total detected small delay faults (”*SD det.*”) and the portion of corresponding transition faults (”⊇*TD und.*”) that went undetected are reported in column 5 and 6.

As shown in the table, the transition fault coverage (Col. 3) is high in general with 98.2 percent coverage in average for the circuit listed. However, a significant portion

Table 11.2: Fault detection of transition delay (TD) and small delay (SD) faults at same fault locations [SKH+17].

Circuit[(1)]	Faults	Transition (TD)		Small Delay (SD)	
	collapsed[(2)]	TD *det.*[(3)]	⊇ SD *und.*[(4)]	SD *det.*[(5)]	⊇ TD *und.*[(6)]
s38417	46138	45965	12198	33777	10
s38584	59530	57838	20242	37734	138
b17	127490	124979	60557	64443	21
b19	770082	764557	216023	549854	1320
p35k	113946	111820	39172	72702	54
p45k	118608	118287	34138	84242	93
p77k	205478	188221	78187	110500	466
p89k	266576	264314	95266	169049	1
p100k	275948	274753	66758	208199	204

of small delay faults (up to 48 percent and over 31 percent in average) at the exact same locations could not be detected, which confirms the well-known fact, that transition faults over-estimate the small delay fault coverage since the small delays faults are harder to detect [IRW90]. As expected, the coverage of the investigated small delay faults (Col. 5) is with 67.2 percent in general much lower compared to the transition faults [YS04]. Interestingly, there are also numerous cases where a small delay fault (of finite size) was detected at a fault location, but a corresponding transition fault (with *infinite* delay [WLRI87]) went undetected. Here, the small delay faults propagated along reconvergent fanout structures, causing glitches to appear at the outputs. Along reconvergent paths these faults are sometimes only detectable for smaller (finite) fault sizes and only within a short time window due to test invalidation [FM91]. Although rare, these cases are especially important for diagnosis purposes, validation and failure analysis to maximize the diagnostic resolution, thereby emphasizing the importance of fast and accurate simulation of small delay faults.

11.2.1 Variation Impact

The small delay fault simulation has been evaluated under *random variation* in a Monte-Carlo experiment for a population of 100 distinct circuit instances [SKH+17]. In the experiment, node delays are modeled using a Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$ with mean μ set to the nominal node delay d_{nom} and standard deviation $\sigma = 0.2 \cdot \mu$. The sample time of the circuit instances has been set to 1.5$\times$ of the longest path delay of the nominal instance avoid excessive timing failures due to close margins. Again, spatially exhaustive and collapsed small delay fault sets are assumed, with the sizes of the small delay faults chosen at half of the slack interval based on the nominal instance timing.

Table 11.3 reports the impact of the random variation on the small delay fault detection compared to the report of the nominal instance simulation. For each circuit, the faults are classified as either *gain* types or *loss* types (Col. 2). In column 3 a fault detection is considered a *gain*, if the fault is detected in one of the random instances, but not in the nominal case. Analogously, a fault detection is *loss* type, if the fault is detected in the nominal case, but went undetected in at least one of the random instances. Column 4 through 13 further categorize all gains and losses based on their impact, or amount of occurrences, through-

out the circuit population from occurrences in at least ten percent (Col. 4) to occurrences in all of the random circuit instances (Col. 13).

As shown, fault detection gains and losses occur for all circuits throughout the entire circuit population. For example, in circuit s38417 a total of 1802 faults have been reported as *undetected* in the nominal case, but were *detectable* in at least one random instance. The detection cases stated in each column are super-sets of the cases stated in the columns to their right. Hence, out of the 1802 gain cases in s38417, 1124 (872) cases occurred in more than 10 percent (20 percent) of the circuit population instances and so on. Since the sizes of the faults were chosen according to the nominal instance delays, more faults showed losses compared to gains. On the other hand, detection gains showed more impact throughout the entire population, possibly due to hazards and glitches in the nominal instances which caused the faults to be closely missed.

The variation impact on the fault detection has been observed on almost 10 percent of the faults in the collapsed fault universe in average over all the circuits which ranges up to 16 percent (circuit s38584). Therefore, in order to properly reason about the small delay faults as well as their robustness of the detection, variation should be taken into account during evaluation [SPI+14]. Note, that the runtime of the parallel delay calculation of each gate negligible [SKH+17], which allows to generate and evaluate many variation instances in parallel on the GPU without noticable overhead. The commercial solution on the other hand does not support changes in the delay and the simulation instance must be reset and newly re-built with instance-specific SDF files for each instance, which results in costly initialization overhead.

11.3 Switch Level Fault Simulation

To provide reasonable parametric faults for the proposed *switch level fault simulation* (in the following referred to as *SWIFT* [SW19a]), all fault sets were generated from topological timing analysis of the netlist. During the process the latest arriving transition time over all test patterns was extracted for each output and the nominal clock period was extracted from the highest latest transition time to which a safety margin of 10 percent was added. Then, all outputs showing transitions with a slack of less than 25 percent were selected

Table 11.3: Fault detection losses and gains in the nominal circuit instance due to random variation (Monte-Carlo experiment with 100 random circuit instances) [SKH+17].

Circuit[1]	Det. Type[2]	Affected Variation Instances (% of Total)										
		>0%[3]	>10%[4]	>20%[5]	>30%[6]	>40%[7]	>50%[8]	>60%[9]	>70%[10]	>80%[11]	>90%[12]	all[13]
s38417	gain	1802	1124	872	688	514	358	232	132	50	12	0
	loss	3246	1764	930	292	80	6	0	0	0	0	0
s38584	gain	3322	1680	1124	776	444	216	108	62	32	6	0
	loss	6392	3792	2720	1762	710	96	2	0	0	0	0
b17	gain	5254	3150	2344	1744	1240	708	404	176	76	28	2
	loss	6806	3460	2180	1216	564	164	52	12	0	0	0
p35k	gain	3752	2052	1392	980	682	430	258	140	58	26	10
	loss	4274	1844	1098	684	448	260	134	64	14	0	0
p45k	gain	3068	1928	1452	1068	718	404	158	54	18	8	2
	loss	7404	2564	1514	692	264	46	2	2	0	0	0
p89k	gain	11532	6242	4486	3162	2040	1042	506	264	136	46	6
	loss	16716	7226	4458	1944	798	260	14	4	0	0	0
p100k	gain	8838	5662	4134	3074	2130	1120	510	218	102	56	24
	loss	10904	5058	3132	1736	660	120	16	0	0	0	0

and traced back to the circuit inputs. Out of all nodes residing in the traced input cones 1000 RRC-cells were randomly selected as fault locations for resistive and capacitive parametric faults [SW19a]. To investigate the impact of the parameter size, the faults at each location were simulated for multiple sizes. Again, as input stimuli the 10-detect transition delay test set is used and no fault dropping is performed.

11.3.1 Resistive Open-Transistor Faults

In a first experiment, resistive-open faults were assumed at all transistors in the set of extracted fault locations and switch level fault collapsing and grouping was performed. Four fault sizes were chosen and simulated at each location: 10kΩ, 50kΩ, 100kΩ and 1MΩ. These fault sizes lie within the range of the conducting and blocking resistance of the associated transistors, which were extracted from SPICE experiments [ZC06, Nan17]. Fig 11.1 summarizes the detection results for the simulated resistive open faults and shows the coverage of *detected* (DT, full strokes) as well as *possibly detected* faults (PD, dashed lines) in percent of total for each fault size simulated. The figure is split into two parts, with the righthand side representing a magnification of the left side 0–5% area for better comprehensibility. The total number of faults simulated for each fault size is shown above. The threshold interval $[V_{thL}, V_{thH}]$ for the PD-classification was chosen with boundaries

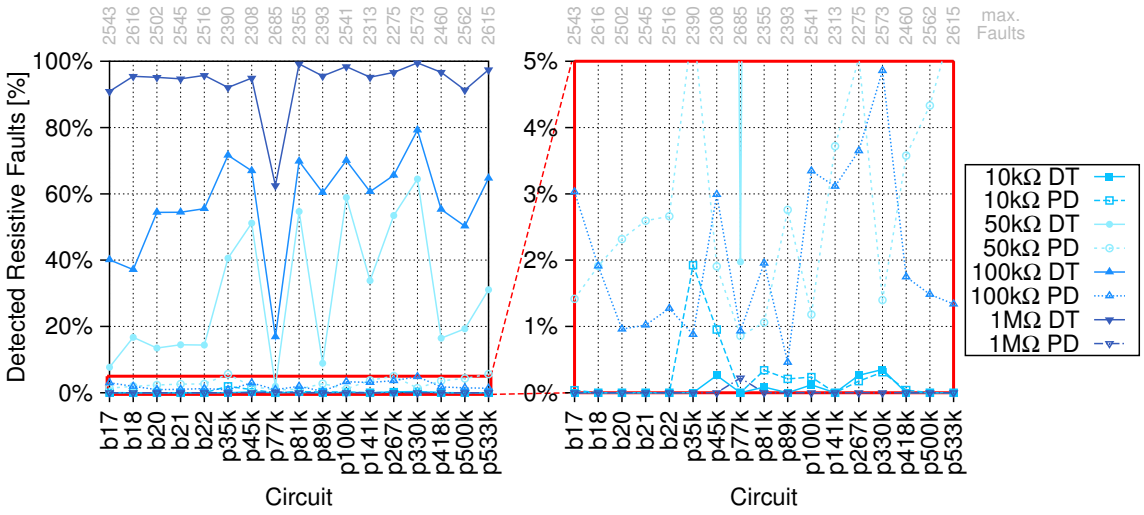


Figure 11.1: Resistive open-transistor fault simulation results for 1000 cells.

$V_{thL} := 0.3\text{V}$ and $V_{thH} := 0.8\text{V}$. Faults that are neither shown as DT or PD are considered as *undetected* (UD) by the test set.

As shown, faults of size 10kΩ were rarely *detected* (below 1 percent in average) and are only shown in the magnified part. Here, the fault effect falls within the range of a few cell delays only, which is usually covered by the clock margin. Yet, many faults are *possibly detected*, which indicate imminent timing violations. For larger fault sizes (50kΩ and 100kΩ) many faults previously being possibly detected turn detectable and even more faults become visible. As expected, for the largest fault size (1MΩ) investigated, the ratio of detected faults is the highest. These faults severely impact the timing, but do not affect the functional behavior of the cells because the fault size is still much lower than the critical resistance that would turn the fault into a stuck-open transistor [VMG93].

Given the clock margin, the behavior of the 1MΩ faults in time is similar to transition faults, since all affected signal transitions within the clock interval are being suspended. Similar to the small delay fault simulation, the average runtime for simulating a fault group in SWIFT is close to the best-case simulation runtime.

To assess the fault modeling capability of SWIFT, the output signals of faulty RRC-cells are investigated and compared against electrical simulation. In Fig. 11.2 the output signals of a two-input NOR-cell before and after injection of parametric faults with varying sizes are visualized. Initially, the input signals of the cell are set to (*high*, *low*), which produces an

initial output of $0V$ of the cell. Then all inputs are set to (*low*, *low*), causing the output load to charge to $1.1V$. Eventually, the inputs are switched to (*low*, *high*), which causes the output load to discharge again to $0V$.

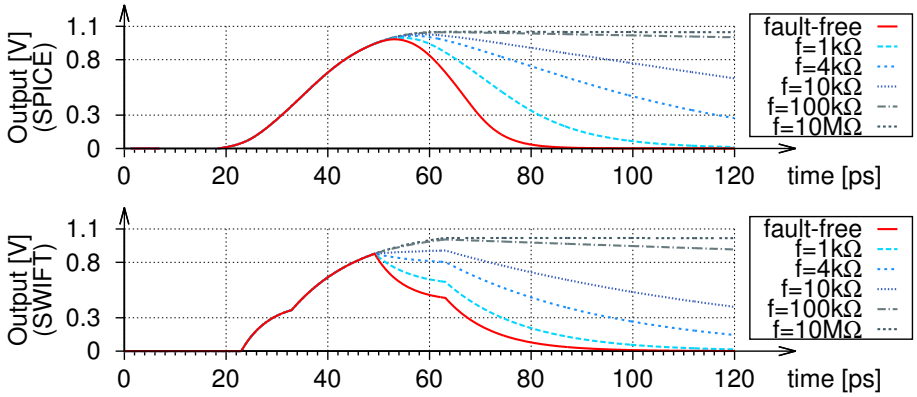
In the first case of Fig. 11.2 (a), a resistive open-transistor fault was injected into the NMOS transistor of the second input the parallel pull-down network, which causes a slow falling transition upon the second input switch. As expected, the drain current through the affected transistor becomes smaller for higher ohmic resistances, until the transistor eventually becomes unable to effectively discharge the load (10MΩ), such that the output level sustains *high*. In the second case (b), the faults were injected into the corresponding PMOS transistor the serial pull-up network. These faults strongly affect affect the rising transition of the hazard at the output of the cell. The larger the injected fault size becomes, the slower the output transition is performed and the pull-up network becomes incapable of charging the output load capacitance in time, before the second input switch occurs.

For the electrical simulation reference (SPICE), the NOR-cell description of the standard cell library [Nan10] was modified by adding a parameterizable resistor at the drain-pin of the affected transistor in order to model the open faults. As shown, the behavior of the faults modeled in SWIFT shows a fairly high similarity compared to SPICE.

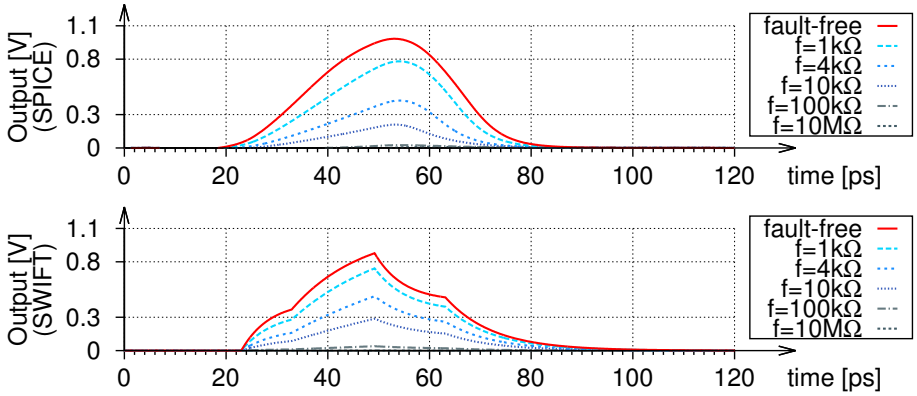
11.3.2 Capacitive Faults

Capacitive faults were assumed at each extracted fault location. The fault sizes have been chosen as multiples of the typical net load average found in the DSPF files from synthesis (i.e., 10pF for the circuits used). Again, four fault sizes were simulated at each location: 50fF, 100fF, 250fF and 1pF, which are injected into the output load capacitance of the RRC-cells. As opposed to the previous resistive faults, the capacitive faults were grouped without performing structural collapsing. The results of the capacitive fault simulation is summarized in Fig 11.3 similar to the previous figure.

As shown, the fault detections gradually increase for larger fault sizes. For the largest size (1pF), the faults were either DT or UD for most of the circuits, except for b18 and p77k. Compared to the others, those circuits have a comparably higher depth and therefore larger slacks at the fault locations. Hence, few faults still remain PD.



a) Injection at an NMOS-transistor in the *parallel* pull-down net.



b) Injection at a PMOS-transistor in the *serial* pull-up net.

Figure 11.2: Behavior of a resistive open-transistor fault in a) NMOS- and b) PMOS-transistors of a NOR-cell in presence of an input hazard [SW19a].

The fault behavior of the capacitive faults in simulation is visualized in Fig. 11.4. For any finite fault size, the capacitive faults strongly affect the timing behavior of the cells. While transitions at the outputs of the cells can be severely slowed, the faults do not alter the functional behavior (stationary voltage), since the affected output capacitor gets still charged over the pull-up and pull-down nets and eventually provides the correct output level after a certain time.

As shown in the figure, the capacitive faults simultaneously affect both rising and falling transitions at the cell output, which was observed in both SWIFT and SPICE simulations. Although the fault size is a discrete constant value, the delay deviation introduced by the capacitive faults can strongly vary in presence of *pattern dependent delays*, since the *time constant* τ that describes the output slope still depends on the driving resistance (cf.

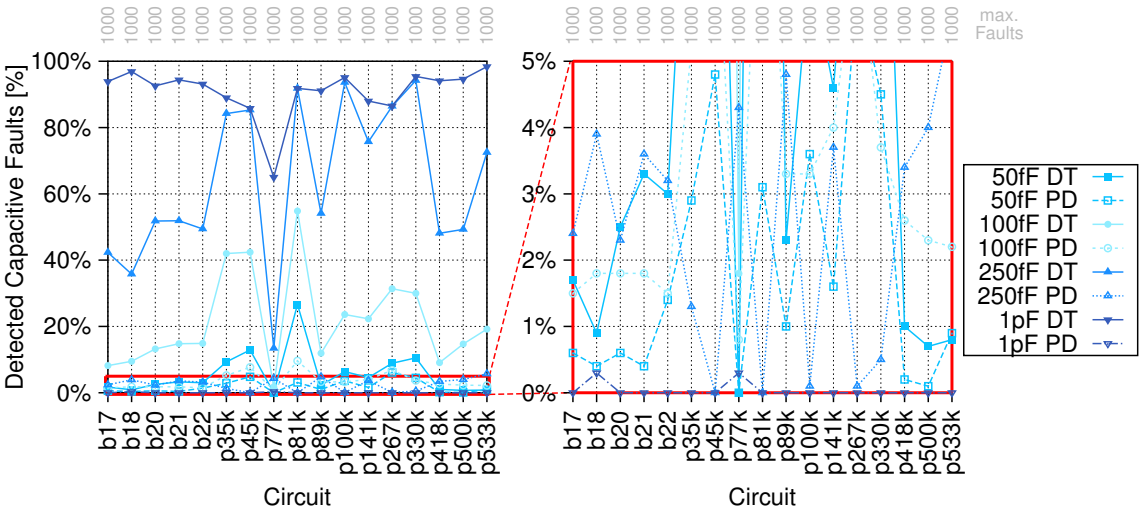


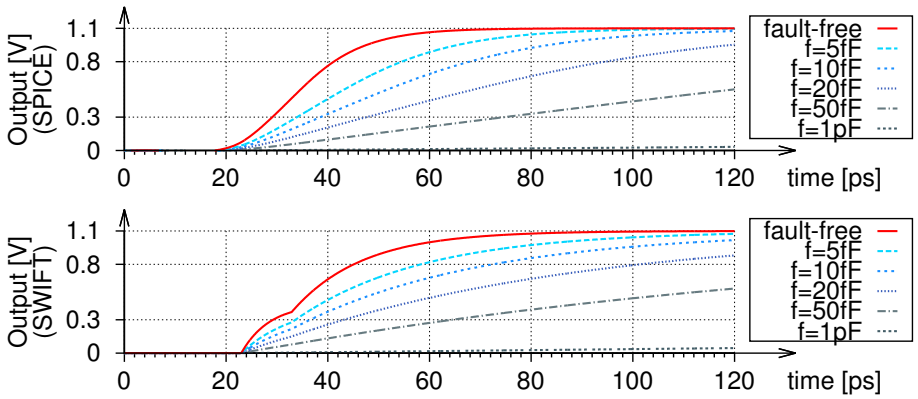
Figure 11.3: Capacitive fault simulation results for 1000 cells.

Eq. (6.9)). This *"dynamic"* fault effect is not captured by conventional logic level time simulation approaches based on SDF descriptions appropriately, therefore demanding for evaluation using lower level simulation approaches.

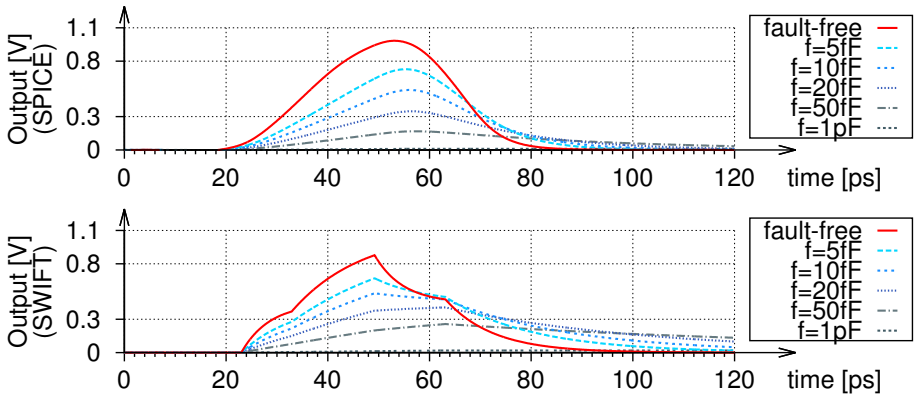
11.4 Summary

In this chapter, both high-level timing-accurate small delay fault simulation at logic level [SHK+15] as well as low-level parametric fault simulation at switch level [SW16] was presented. It was shown that conventional and simpler fault models that rely on untimed simulation, such as transition delay faults [WLRI87], cannot capture the subtle effects of small gate delay and parametric faults. As a consequence, these untimed models typically result in a pessimistic overestimation of the fault coverage. With newer technology nodes and increasing design complexity it has become necessary to cover even smallest delay deviations as accurately as possible [HIK+14, KKS+15]. Yet, timing-accurate evaluation involves high computational complexity, which needs to be tackled using data-parallel GPU architectures.

Both presented high and low-level fault simulation approaches utilize the presented high-throughput time simulation model with node- and stimuli parallelism in combination with a fast fault grouping heuristic to exploit additional parallelism from faults. In addition to the significant speedup of the simulation approaches of up two to three orders of mag-



a) Rising signal transition affected by capacitive faults.



b) Signal hazard affected by a capacitive fault.

Figure 11.4: Behavior of capacitive faults at the NOR-cell output affecting a) a single rising transition, b) a hazard [SW19a].

nitude, the fault grouping heuristic showed a reduction of the simulation overhead by 98% in average compared to a naïve serial simulation of faults. Therefore the presented approaches enable exhaustive fault simulation without the need of fault dropping even for larger designs. In addition, instance-parallelism was exploited in Monte-Carlo experiments allowing to investigate and compare the impact of random variation on the fault detection in detail [SKH+17, SW19a] which is important for fault grading and variation-aware test pattern generation [SPI+14].

Chapter 12

Application: Power Estimation

In this chapter, the presented high-throughput time simulation (cf. Chapter 5) is used to analyze the circuit switching activity for the application in power estimation.

Excessive power consumption from non-functional switching activity can lead to reliability issues (such as high thermal design power and IR-drop) when not considered during functional operation and test [YYHI12, WYM+05, WMK+08]. Typical validation approaches to tackle power-related problems often rely on untimed (zero-delay) logic simulation, since full timing-aware simulation has a high runtime complexity. However, untimed simulation misses the switching activity caused by hazards and glitches in the designs and therefore cannot deliver accurate results.

The presented high-throughput GPU-accelerated parallel timing simulation allows to reveal the full switching activity including hazards and glitches with unprecedented simulation speed. In the following, a comprehensive analysis of the switching activity in the circuits is performed, by comparing results from untimed and timing-accurate simulation to emphasize the necessity of more accurate simulations.

12.1 Circuit Switching Activity Evaluation

Fig. 12.1 summarizes the switching activity results of the benchmark circuits for the provided 10-detect transition fault test pattern set. Each node evaluation corresponds to the computation of a waveform or a given test stimuli which is considered a simulation case. All simulation cases were categorized based on the transition count tc of the respective

signal waveforms. The categories are divided into glitch-free cases comprised of static constant signals with $tc = 0$ and robust single transitions with $tc = 1$, signals with glitches comprising static-0 and static-1 glitches with $tc = 2$ as well as dynamic glitches with multiple transitions for $tc > 2$. The percentage of the occurrences with respect to the overall node evaluations are given in parentheses.

Roughly 50 percent of the overall node evaluations show constant signals without any transitions. The static-0/1 and dynamic glitches make up almost 20 percent of the overall cases in average for all the circuits investigated. For these cases, untimed logic level simulation will report either zero switching activity (for all static cases and dynamic cases with even transition count and $tc > 0$) or a single transition (dynamic cases with odd transition count and $tc > 2$ only), as only the information of the initialization and propagation vector responses are available.

As shown, the glitch-accurate timing simulation results reveal a large gap between untimed simulation. This difference in the computed switching activity information can have severe impact on the results of power-related applications as shown in the following.

12.1.1 Weighted Switching Activity for Power Estimation

The *weighted switching activity* (WSA) of a circuit and a given test stimuli set was calculated as a measure for estimating test power consumption [GNW10, YYHI12]. In this work, the WSA $wsa(n, p)$ corresponding to a node n and test stimuli p was computed as

$$wsa(n, p) := tc_{n,p} \cdot fanout(n), \tag{12.1}$$

where $tc_{n,p}$ is the transition count of the obtained waveform at the node in the timing-accurate simulation of the stimuli and $fanout(n)$ is the number of direct successor nodes in the fanout of n.

The resulting WSA values were compared for both untimed and timing-accurate simulation of the generated 10-detect stimuli test set [HSW+16]. Fig. 12.2 shows the WSA average difference ($\Delta avg.$) from timed and untimed simulation per stimuli as well as the maximum difference ($\Delta max.$) observed among all stimuli applied.

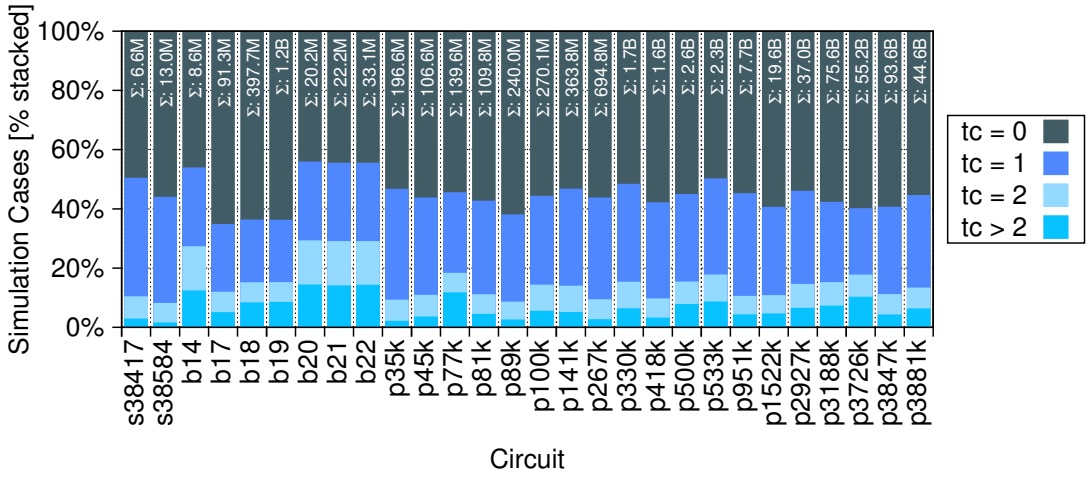


Figure 12.1: Switching activity in waveforms obtained by timing-accurate logic level simulation classified as constant signals ($tc = 0$), single transitions ($tc = 1$), static ($tc = 2$) and dynamic glitches ($tc > 2$). The total amount of simulation cases is given by Σ.

As shown, the average WSA difference obtained from timing-accurate simulation exceeded 130 percent per stimuli in average over all circuits. For circuit p3726k this difference is even as high as 454.6 percent mainly because of reconvergent signal propagation in the circuit. As for the maximum WSA difference of the applied test stimuli, the average over all circuits exceeds more than 200 percent with a global maximum of 857.5 percent difference for circuit p77k. Thus, in order to obtain reasonably accurate results a timing-accurate evaluation is mandatory.

12.1.2 Distribution of Switching Activity

Fig. 12.3 illustrates the distribution of switching activity over time after each test pattern has been applied. The time interval (x-axis) of the evaluation has been chosen with respect to the latest stabilization time of each circuit. The interval was divided into 32 equidistant slices in each of which the time-slice evaluation kernel was called to count the signal events contained. The resulting distribution of the events is shown on the left y-axis (stroked lines), which is given in percent with respect to the overall event count in the circuit. The right y-axis (dashed lines) indicates the cumulative distribution of the events until including each time slice.

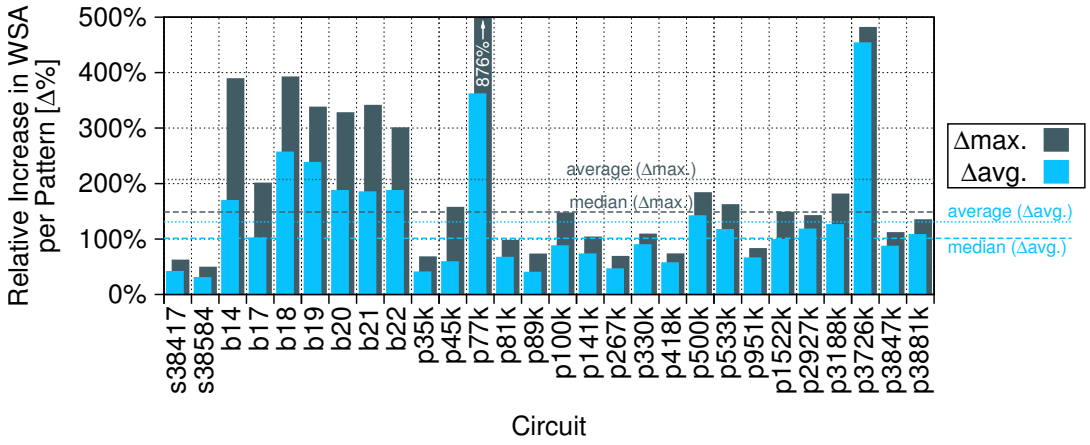


Figure 12.2: Relative increase in weighted switching activity (WSA) of timing-accurate logic level simulation compared to untimed simulation.

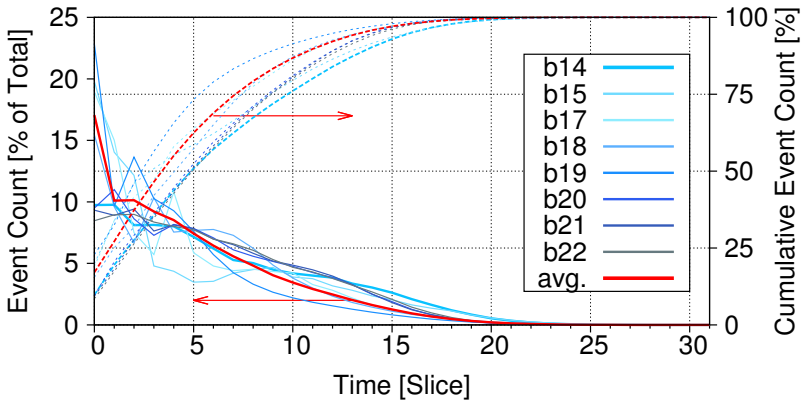


Figure 12.3: Distribution of the switching activity in a time-sliced evaluation over a clock interval with 32 slices [HSW+16].

As shown, for all the circuits more than 50% of the overall events do occur within the first five time slices of the clock interval. After that, most signals stabilize and little switching activity sustains mostly only in the regions of the longest paths. This emphasizes that power should be determined with higher granularity rather than averaging over clock intervals, since accurate peak power information is especially important for IR-drop analyses [YYHI12, JAC+13] and power-related test applications [AWH+15].

12.1.3 Correlation of Clock Skew Estimates

In the following, the difference between the use of timing-accurate and untimed (zero-delay) simulation for estimating the impact of regional switching activity on the clock

distribution was investigated. Fig. 12.4 shows an example of a scan chain of three *scan flip flops* (SFFs) in a circuit with a part of the clock distribution tree in the lower section. The clock distribution tree forwards the *global clock* signal from a clock pin along branches of clock buffers to the clock inputs clk of the individual SFFs in the scan chain. During *scan shift* cycles, when the clock signal reaches a SFF, the SFF copies the state of its predecessor from the value at its own input and forwards it to the input of the succeeding element in the chain as well as to the functional logic paths. During the shift, the state changes of all scan flip flops cause switching activity at *aggressor nodes* along the functional logic paths of the circuit. This switching activity is assumed to cause delays of the clock signals associated to the SFFs due to IR-drop which eventually might lead to scan-shift errors. In the following, the experiment will be briefly described, a more detailed description of the setup and the modeling is given in [HSK+17].

Any clock buffer B in the clock distribution tree will experience an *extra* delay $wsa\_imp(B)$ that is expressed as a function of the sum of the *weighted switching activity* (WSA) caused by its surrounding nodes [HSK+17]. These so-called *aggressor nodes* represent all cells in the layout that fall within a specified range of the clock buffer, each of which can be weighted by an influence factor. The overall delay impact $imp(i) := \sum_{B \in P} wsa\_imp(B)$ on the clock distribution to a SFF i then modeled as the sum of the extra delays of each individual clock buffer $B \in P$ on the path P from the *global clock* pin to the clock input clk of the associated SFF. For a given shift cycle, the *skew* between two consecutive SFFs $(i-1)$ and i in the scan chain is then estimated by computing the difference of the delay impact of both SFFs [HSK+17] using

$$skew(i) := imp(i) - imp(i-1), \tag{12.2}$$

where SFF i is the receiving and SFF $(i-1)$ is the sending scan flip-flop. In case the computed $skew(i)$ has a large positive (negative) value, the clock delay of the receiving flip-flop is assumed higher (lower), which might eventually cause *hold-time* (*setup-time*) violations leading to scan-shift errors. If the skew estimate is close to zero ($skew(i) \approx 0$), the clock distribution to both flip-flops SFF i and SFF $(i-1)$ is assumed to be evenly balanced [HSK+17].

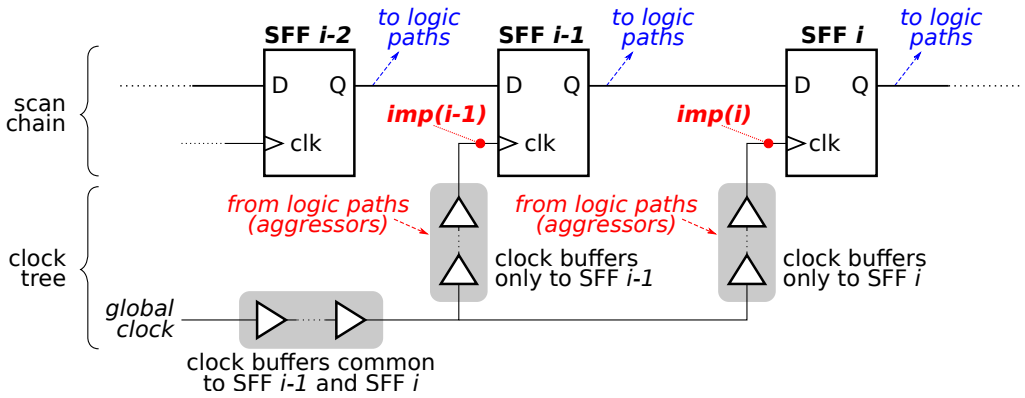


Figure 12.4: Clock-skew in the clock tree caused by accumulation of unbalanced regional switching activity in the circuit.

For the experiments, a commercial synthesis flow was used to synthesize the ITC'99 benchmark designs down to layout using the Synopsys SAED 90nm library [GBW+09] and a commercial ATPG tool was used to generate transition fault test patterns. From the layout data, the clock tree was extracted with all nodes of aggressor regions for each clock buffer in the design leading to scan flip-flops. Based on this, the skew estimate was calculated for all shift cycles required to process the generated test pattern sets. Details about the synthesis and simulation results as well as the algorithm can be found in [HSK+17].

Fig. 12.5 illustrates the correlation of the obtained skew estimate from timing-accurate and untimed (zero-delay) simulation at logic level for circuit b14. While the x-axis shows the computed skew of a scan flip-flop obtained from a simulation with full timing information (including all static and dynamic hazards) and the y-axis shows the respective result from untimed simulation (without information about hazards). The evaluation was performed for 406 test stimuli applied within 87,505 shift cycles, which resulted in roughly 13.7 million update events out of which 3.0 million showed non-zero skew [HSK+17]. The color of each point in the graph indicates the number of simulation cases (log-scale) that occurred with the associated correlation. The point at $(0, 0)$ has been omitted in order to improve visibility. As shown in the example of circuit b14, some of the cases show a very high skew in the timing simulation, while they have close to zero skew in the untimed simulation. Any setup- and hold-time violations associated with these cases hence go undetected in the untimed evaluation [HSK+17]. Vice versa, any simulation case that

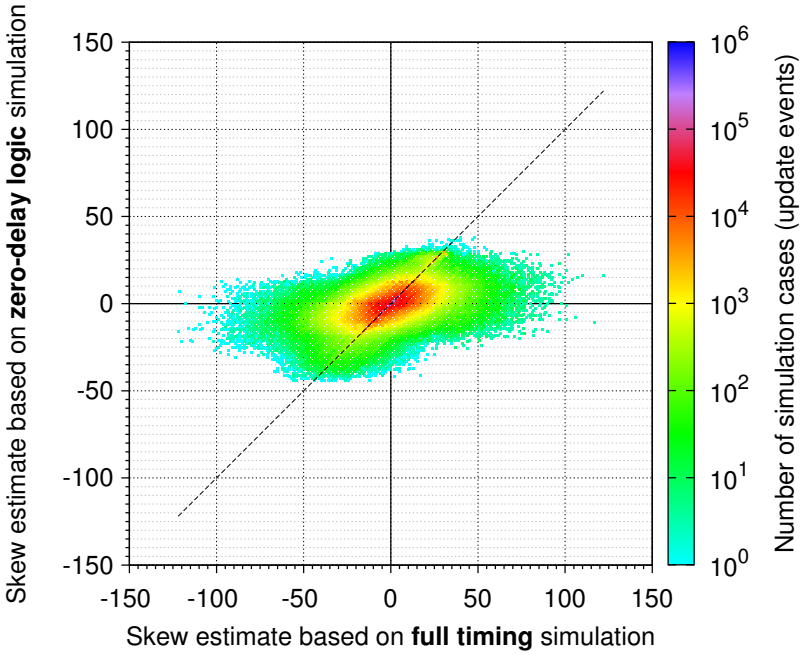


Figure 12.5: Correlation of the computed skew from timing-accurate (x-axis) and untimed (zero-delay, y-axis) simulation for circuit b14.

shows a high skew in the untimed but zero skew in the timing-accurate simulation, might over-estimate the presence of violations.

Similarly, Fig. 12.6 shows the skew correlation for circuit b18. Note, that the general number of simulation cases is much larger. For b18, 1563 test stimuli were applied in 4,319,768 shift cycles. During the shifts, 9.23 billion update events were evaluated for the scan flip flops out of which 1.68 billion showed non-zero skew [HSK+17]. Although many simulation cases are also clustered along the diagonal indicating a high correlation, many simulation cases show near to zero skew in untimed simulation while in the skew obtained from timing-accurate simulation is spread in a broader fashion.

Hence, static and dynamic hazard-aware timing-accurate simulation is crucial for more accurate estimations of IR-drop and the clock skew [HSK+17]. However, without the help of GPU-acceleration and the possible high-throughput parallelization with up to three orders of magnitude speedup, obtaining and evaluating such a vast amount of data points would be infeasible for these large designs, due to the high runtime complexity of the mandatory timing simulation.

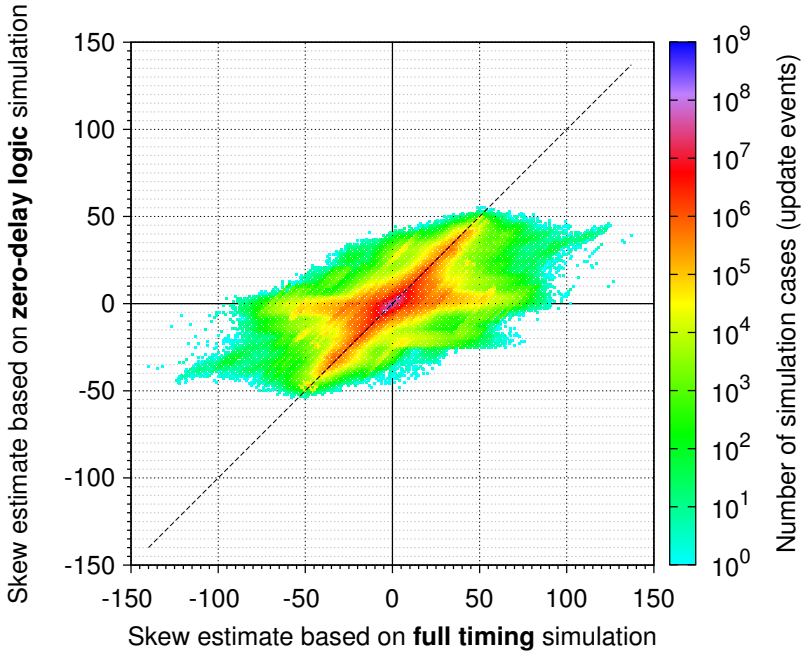


Figure 12.6: Correlation of the computed skew from timing-accurate (x-axis) and untimed (zero-delay, y-axis) simulation for circuit b18.

12.2 Summary

Fast timing-accurate simulation at logic level is crucial for accurate validation of the circuit timing as well as many power-related applications. For these applications, the proper capturing of glitches and hazards provides detailed insights of the internal switching behavior and contributes to a large portion to the overall simulation accuracy [HSK+17, HSW+16]. The experiments have shown that glitches and hazards occur frequently in the investigated circuits, whose impact on the simulation results cannot be captured using traditional untimed (zero-delay) logic level simulation approaches.

Chapter 13

Application: Multi-Level Simulation

This chapter evaluates the multi-level simulation approach presented in Chapter 8 similar to [SKW18, SW19b] and further investigates the proposed high-throughput simulation in terms of scalability and simulation efficiency.

In order to evaluate the multi-level simulation, the used benchmark designs have been simulated for varying mixed-abstraction scenarios with different amounts of active *regions of interest* (ROIs). Throughout the experiments, only single nodes are considered as activation points for ROIs. For each node in the circuit, both logic level description and switch level description are available. As input stimuli, the well-known ATPG-generated 10-detect transition delay test set is used.

13.1 Multi-Level Simulation Runtime

Fig. 13.1 summarizes the simulation performance of the multi-level simulation for the different circuit sizes. On the left side, the simulation throughput in MEPS is reported, while the right side presents the average time (milliseconds) spent per pattern pair.

As shown, the multi level simulation achieved more than 1000 MEPS for a full logic level simulation with 740 MEPS in average. The peak simulation throughput for simulations at full switch level was 174 MEPS with an average of over 132 MEPS for all circuits investigated, thereof being roughly one magnitude slower compared to the logic level case. Regarding the average per-pattern runtimes, the simulation times ranged from less than 0.1ms to 25ms for the largest design investigated. As indicated by the trend lines,

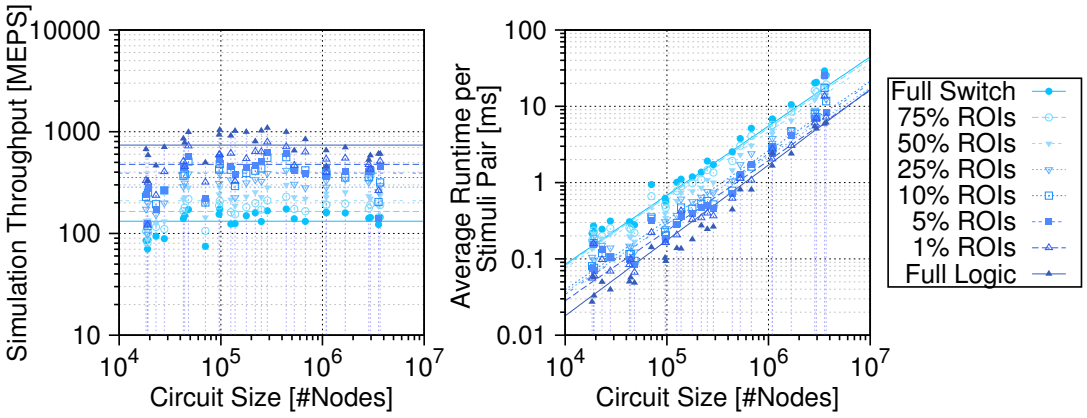


Figure 13.1: Multi-level simulation throughput in MEPS (left) and average simulation time per stimuli (right) for different ROI scenarios [SW19b].

the average runtimes per pattern scale with the size of the circuit, while the throughput performance shows a constant trend since the GPU is already fully utilized.

As shown, the observed runtimes of the multi-level timing simulation range from 20ms (for circuit s38417) up to 10 minutes (p3874k). Also, for the medium-sized circuits the simulations were finished within few seconds only. For all experiments, the obtained runtimes were faster than the commercial event-driven solution.

However, compared to the native logic level simulation (cf. Table 10.1), a full logic level simulation using the mixed-level abstraction simulator was observed to be 40 percent slower in average. One explanation is the overhead of the additional checks for abstraction types as well as the waveform conversion mechanics during simulation. As a second reason for the runtime increase in logic level simulation is the increased amount of multiprocessor registers determined by the compiler, that are required by the evaluation kernel threads. In contrast to the pure logic level simulation with 47 processor registers required per execution thread, the mixed-abstraction level simulation needed 86 registers per thread, which is as much as the evaluation kernel in the switch level simulation. As a consequence, the thread block dimensions of the kernel had to be adapted, which also lowered the thread-group occupancy[1] of the streaming multi-processors [NVI18b]. Yet, for the million-node designs the average runtimes of the simulators deviate by approximately 2 percent, thus being virtually equal.

[1] Obtained from CUDA Occupancy Calculator (Version 7.5) [NVI17a].

As for the full switch level simulation, the mixed-abstraction implementation showed to be 5 percent slower in average compared to its native counterpart (cf. Table 10.2). For the million node-circuits, the average runtimes in the mixed-abstraction simulator were even reported to be 2 percent faster. Hence, at this magnitude the overhead can be considered as negligible, since it falls within the range of the random runtime fluctuations.

13.2 Multi-Level Trade-Off

With the mixed abstraction descriptions, the multi-level simulation allows for a flexible trade-off in simulation efficiency based on the amount of active ROIs. In the following, the efficiency of multi-level simulation is investigated for mixed switch and logic level scenarios.

13.2.1 Speedup

In Fig. 13.2 the left y-axis illustrates the speedup of the multi-level simulation over the unparallelized commercial event-driven solution for all circuits and ROI scenarios in more detail. The logic level simulation speedup ranges from 155× to over 1600×, while the speedups of the switch level simulation are between 33× (s38417) and 314× (p3726k). In general, the obtained speedups tend to be higher for larger circuit designs since the available computing resources on the GPUs can be better utilized. The red line is associated with the right y-axis of the figure, which indicates the ratio of the speedups obtained from full switch level simulation to full logic level simulation in the multi-level simulator. As shown the ratio ranges from 3× up to 8× with an average of 6× for all circuits.

13.2.2 Runtime Savings

In the following, the runtime savings of the multi-level simulation are investigated. For each ROI a single node location is assumed. In each simulation run ROIs are activated at random nodes in the circuit. The corresponding multi-level simulation scenario $x := |\#\text{active\_ROIs}|/N \in [0, 1]$ is thus determined by the fraction of active ROIs among the N circuit nodes which ranges from full logic- (zero nodes $x = 0$) to full switch level simulation (all nodes $x = 1$). Thus, given a certain ROI-scenario x, the respective savings

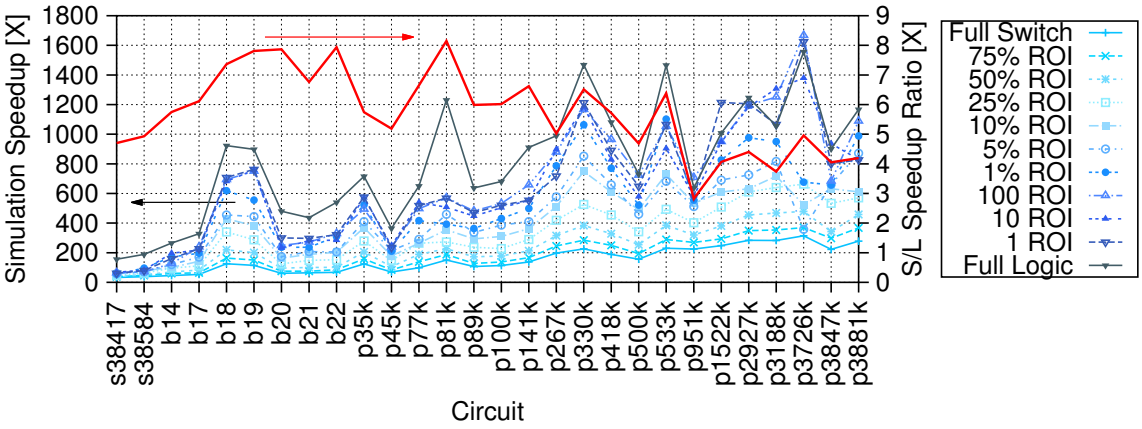


Figure 13.2: Multi-level simulation speedup (left-axis) and *Full-Switch*-to-*Full-Logic* level (S/L) simulation speedup ratio (right axis) [SW19b].

are calculated as

$$savings(x) := 100\% \cdot \left(1 - \frac{T_{ML}(x)}{T_{swl}}\right), \tag{13.1}$$

where $T_{ML}(x) \in \mathbb{R}$ corresponds to the mixed-abstraction simulation and $T_{swl} := T_{ML}(1) \in \mathbb{R}$ corresponds to the respective reference runtime of the full switch level simulation.

Fig. 13.3 illustrates the savings in runtime for the mixed-abstraction simulation compared to a full simulation at switch level. The x-axis represents the different ROI scenarios with decreasing amount of active low-level nodes. The runtime savings $savings(x)$ are given in percent on the y-axis for different mixed-abstraction scenarios x.

As shown, the achieved runtime savings scale linearly with the active number of ROIs between the full switch and full logic level scenario. The average savings of the mixed-abstraction simulations range from 20 percent with 75 percent of active nodes to over 70 percent in the single active node case in average. Yet, for certain simulation tasks, such as timing validation, a sparse activation of ROI nodes is already sufficient. For small amounts of ROIs, the runtime savings fluctuate based on the random location chosen. Especially for the smaller circuits s38417 and s38584 these fluctuations showed a higher runtime impact, due to an increasing overhead for the generation and propagation of waveforms with grown complexity.

Given a mixed-abstraction simulation scenario $x \in [0, 1]$ as described above, the efficiency of the multi-level simulation $eff_{ML}(x)$ will be described according to Eq. (8.1) with the targeted-abstraction reference simulation time corresponding full switch level

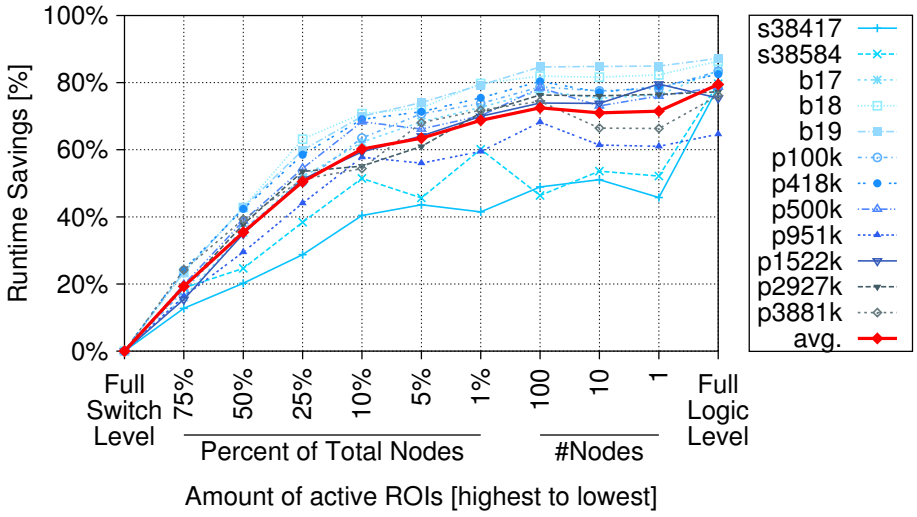


Figure 13.3: Runtime savings of the multi-level approach compared to a full switch level simulation. ROIs are activated at random nodes with varying sparsity [SW19b].

simulation T_{swl}. Hence, the resulting formula represents to the speedup of the multi-level approach over a full switch level simulation.

In [SKW18], an experiment was performed in which ROIs were activated on nodes along circuit paths with a slack of less than 20 percent of the nominal delay. The resulting amount of active ROIs in each circuit ranged from as little as 0.08 percent (p3881k) up to 48.5 percent (p35k). In average only 9.9 percent of the nodes in each circuit needed to be activated for the simulation at switch level during the timing validation, while in more than 50 percent of the cases the portion of nodes that needed to be activated was only 6.7 percent. Especially for all the million-node designs, the ratio of ROI activation was less than 1 percent. Therefore, the application greatly benefits from the runtime savings of the sparse ROI activation in the mixed-abstraction simulation [SKW18].

13.2.3 Simulation Efficiency

The implemented multi-level simulation approach allows to gain additional efficiency during parallel fault simulation of a grouped fault set through sparse simultaneous ROI activation. Fig. 13.4 shows the distribution of the sizes of the fault groups obtained in Table 11.1-Col. 9 as well as the cumulative count of faults processed with respect to the collapsed switch level fault set [SW19a].

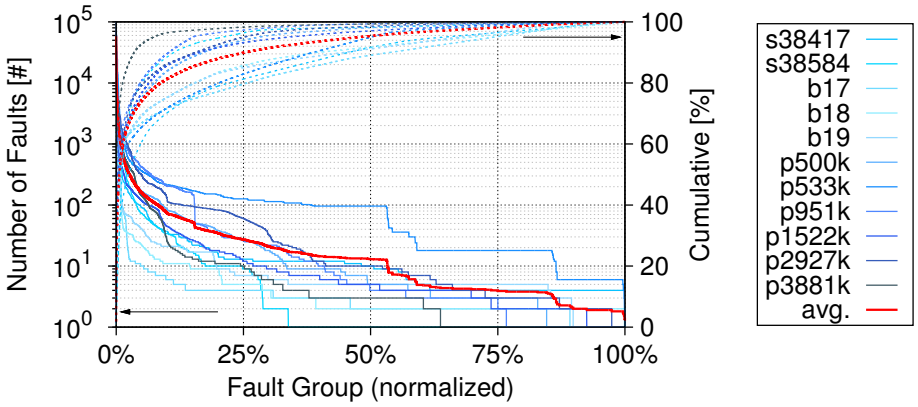


Figure 13.4: Size of groups in faults (left axis, log-scale) and cumulative amount of faults processed (dotted, right axis) [SW19a].

The early leftmost groups contain the most faults, most of which are located directly or close to the circuit outputs. These faults have a high probability of mutual output-independence, since each output itself is output-independent to others, and the output cones of fault sites in these regions are small. By processing 25 percent of the overall fault groups in simulation, already 90 percent of the total faults in each circuit can be evaluated. As the grouping progresses, the fault groups get smaller due to increasing overlap of the output cones of the fault sites. For the first 50 percent of the fault groups at least ten faults are processed in average *per* group. The sizes of the fault groups following after drop to a few faults only. The faults of these groups are typically associated with locations along the structurally longest paths, which cannot be grouped efficiently, since the faults have a high probability of sharing common outputs. Yet, the size of each of the obtained fault groups was always less than 10% of the nodes in the respective circuits.

In this work, a *single-fault-single-ROI* activation is assumed, where single node ROIs are activated at all fault locations of the fauls in a fault group. The efficiency of a multi-level fault simulation using the mixed abstraction simulator now becomes evident from the runtime savings of the ROI-scenario previously shown in Fig. 13.3. For each fault group, the savings obtained in the mixed abstraction simulation generally correspond to the savings obtained from the simulation of the corresponding ROI-scenario of the same sparsity. This is due to the fact that the fault injection scheme itself introduces no additional costs [SKH+17, SW19a], while the transfers of the node descriptions of the active ROIs needs to be performed nonetheless.

The *multi-level fault simulation efficiency* (eff_{MLfsim}) for processing a grouped fault set $\mathcal{F}$ is then defined enclosed as

$$\mathit{eff}_{MLfsim}(\mathcal{F}) := \frac{1}{|\mathcal{F}|} \cdot \sum_{FG_i \in \mathcal{F}} \left(\mathit{eff}_{ML}\left(\frac{|FG_i|}{N} \right) \right), \tag{13.2}$$

where $|\mathcal{F}|$ the size of the fault set in fault groups, and $|FG_i|$ the size of the i-th fault group in number of faults contained over the size N of the circuit in number of nodes. Hence, with the number of ROIs needed to be activated for each fault group being less than 10% of the circuit nodes, average savings ranging from 60% for the larger and 70% for the smaller groups (i.e., down to a minimum of 1 fault) are possible.

13.3 Summary

This chapter investigated the simulation performance and simulation efficiency of mixed-abstraction scenarios in the presented multi-level timing simulator [SKW18].
The simulator uses switch- and logic-level descriptions concurrently for a flexible trade-off in terms of simulation speed and accuracy. The abstraction of the design is lowered on demand in so-called *regions of interest* (ROIs), by swapping the associated node descriptions. Multi-level waveform data structures and waveform transformations allow for simultaneous use of switch and logic level waveforms at the same time throughout the evaluation of the circuit. Experiments showed runtime savings of over 80 percent while using the mixed-abstraction simulation compared to full evaluation at switch level.
The multi-level simulation allows designers to validate their specific designs with the desired timing-accuracy at lower levels in a higher-level environment independent of the availability and accessibility of low-level implementations of the other parts in the system. Also, the use of mixed-abstractions during simulation allows for highly efficient and flexible low-level fault simulation by activating ROIs for providing low-level injection points for parametric faults. This way a more efficient and faster evaluation is enabled, allowing to aid in everyday design and test validation tasks.

Conclusion

The simulation of circuits as well as of faults therein is an essential tool in today's design and test validation tasks of nano-scaled integrated digital CMOS circuits. The increasing proneness to manufacturing process variations and the growing complexity of low level parametric and parasitic defect mechanisms of the steadily shrinking technology nodes severely tamper with the performance and the reliability of the designs. Thus, thorough validation under consideration of accurate timing has become a necessity, not only to validate the functional performance, but also to investigate switching-activity-related non-functional properties and to observe the impact of variation on faults. Yet, conventional timing simulation even at logic level is a compute-intensive task.

Data-parallel *graphics-processing unit* (GPU) hardware accelerators provide a large array of multi-processing cores and are able to process thousands of threads concurrently. With the enormous computing throughput in the teraFLOP-range on a single die, affordable high performance computing for scientific applications is enabled, which has already been utilized to accelerate many EDA applications.

In this thesis, highly parallel simulation models were developed that enable fast and efficient timing-accurate simulation of digital CMOS designs on GPU architectures. During simulation, parallelism is exploited from nodes, stimuli, faults and circuit instances under variation instances to maximize the simulation throughput. The presented modeling was applied to logic level, which enabled fast and timing-accurate simulation of small delay faults under variation with access to full signal switching histories. Furthermore, the modeling was applied to switch level to provide a more accurate modeling of the functional and time behavior of CMOS-cells based on first-order electrical effects as well as to enable low-level parametric fault simulation.

Experimental results have shown speedups of two to three orders of magnitude over a commercial event-driven time simulation at logic level, and showed that the presented method scales for million node designs. Ultimately, additional simulation efficiency was achieved by utilizing multi-level fault simulation on the GPU with mixed-abstraction scenarios with variable trade-off in simulation speed and modeling accuracy.

Ongoing and Future Research

Accurate timing simulation allows for a large variety of applications and can support the development of novel test schemes and algorithms for evaluation of functional and non-functional design aspects.

Delay faults and low-level parametric fault models can be explored to guide and validate test-pattern generation [EFD07, ED12, SCPB12] under consideration of parameter variations [CIJ+12, SPI+14]. The evaluation of delay deviations can be used to investigate aging mechanisms and techniques for performance monitoring, early-life and wear-out failure prediction [LGS09, KCK+10, LKW17] and recently FAST [HIK+14, KKS+15].

The high-throughput time simulation on GPUs allows to execute large-scale power simulations for guiding test schemes to ensure high test reliability and confidence [HSK+17, HSW+16]. Regions with excessive switching activity can be effectively identified [ETD10, DED+17b] that might otherwise exceed the power specifications of the design, and efficient IR-drop simulations can be implemented [YYHI12, MSB+15] that heavily rely on switching activity information. The impact of these non-functional properties on the functional properties could be investigated in a closed-loop to provide more accurate results.

Further applications can be explored in the area of circuit debug and diagnosis. Here, extensive simulation can be utilized to investigate machine-learning supported fault classification approaches [RGW16] in combination with location-based diagnosis [HW07] for a better understanding of faulty syndromes and quick identification of their root-causes.

The ongoing development of current GPU architectures (upcoming NVIDIA V100 GPU with more than 5000 cores [NVI17d]) provides a promising foundation for exploring larger simulation problems and more complex evaluations. The dimensions of parallelism can be extended (e.g., execution of test-programs or tasks in parallel), such that new application can benefit from more and individual dimensions to achieve the best speedup.

Bibliography

[ABF90] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design, Revised Printing*. IEEE Press, NY, 1st edition, 1990. ISBN: 0-7803-1062-4.

[ABZ03] A. Agarwal, D. Blaauw, and V. Zolotov. Statistical Timing Analysis for Intra-Die Process Variations with Spatial Correlations. In *Proc. Int'l Conf. on Computer Aided Design (ICCAD)*, pages 900–907, Nov. 2003. doi:10.1109/ICCAD.2003.1257914.

[AKM+88] M. Abramovici, B. Krishnamurthy, R. Mathews, B. Rogers, M. Schulz, S. Seth, and J. Waicukauski. What is the Path to Fast Fault Simulation? In *Proc. Int'l Test Conf. (ITC)*, pages 183–192, Sep. 1988. doi:10.1109/TEST.1988.207796.

[AMM84] M. Abramovici, P. R. Menon, and D. T. Miller. Critical Path Tracing: An Alternative to Fault Simulation. *IEEE Design & Test of Computers*, 1(1):83–93, Feb. 1984. doi:10.1109/MDT.1984.5005582.

[APZM07] M. Agarwal, B. C. Paul, M. Zhang, and S. Mitra. Circuit Failure Prediction and Its Application to Transistor Aging. In *Proc. IEEE 25th VLSI Test Symp. (VTS)*, pages 277–286, May 2007. doi:10.1109/VTS.2007.22.

[AWH+15] K. Asada, X. Wen, S. Holst, K. Miyase, S. Kajihara, M. A. Kochte, E. Schneider, H.-J. Wunderlich, and J. Qian. Logic/Clock-Path-Aware At-Speed Scan Test Generation for Avoiding False Capture Failures and Reducing Clock Stretch. In *Proc. IEEE 24th Asian Test Symp. (ATS)*, pages 103–108, Nov. 2015. doi:10.1109/ATS.2015.25.

[AYY+14] Y. Ali, Y. Yamato, T. Yoneda, K. Hatayama, and M. Inoue. Parallel Path Delay Fault Simulation for Multi/Many-Core Processors with SIMD Units. In *Proc. IEEE 23rd Asian Test Symp. (ATS)*, pages 292–297, Nov. 2014. doi:10.1109/ATS.2014.61.

[BA04] M. Bushnell and V. Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Frontiers in Electronic Testing. Springer US, 2004. ISBN: 9780792379911.

[BAS97] S. Bose, V. D. Agrawal, and T. G. Szymanski. Algorithms for Switch Level Delay Fault Simulation. In *Proc. Int'l Test Conf. (ITC)*, pages 982–991, Nov. 1997. doi:10.1109/TEST.1997.639714.

[BB12] M. Beckler and R. D. Blanton. On-Chip Diagnosis for Early-Life and Wear-Out Failures. In *Proc. IEEE Int'l Test Conf. (ITC)*, pages 1–10, Paper 15.1, Nov. 2012. doi:10.1109/TEST.2012.6401580.

[BB17] M. Beckler and R. D. Blanton. Fault Simulation Acceleration for TRAX Dictionary Construction using GPUs. In *Proc. IEEE Int'l Test Conf. (ITC)*, pages 1–9, Paper A.3, Oct. 2017. doi:10.1109/TEST.2017.8242078.

[BBC94] M. L. Bailey, J. V. Briner, Jr., and R. D. Chamberlain. Parallel Logic Simulation of VLSI Systems. *ACM Computing Surveys*, 26(3):255–294, Sep. 1994. doi:10.1145/185403.185424.

[BBH+88] Z. Barzilai, D. K. Beece, L. M. Huisman, V. S. Iyengar, and G. M. Silberman. SLS - A Fast Switch-Level Simulator. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 7(8):838–849, Aug. 1988. doi:10.1109/43.3214.

[BBK89] F. Brglez, D. Bryan, and K. Kozminski. Combinational Profiles of Sequential Benchmark Circuits. In *Proc. IEEE Int'l Symp. on Circuits and Systems (ISCAS)*, volume 3, pages 1929–1934, May 1989. doi:10.1109/ISCAS.1989.100747.

[BC13] A. Biddle and J. S.T. Chen. FinFET Technology – Understanding and Productizing a New Transistor, White Paper, 2013. [Last Access: Feb. 26, 2019].

[BCSS08] D. Blaauw, K. Chopra, A. Srivastava, and L. Scheffer. Statistical Timing Analysis: From Basic Principles to State of the Art. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(4):589–607, Apr. 2008. doi:10.1109/TCAD.2007.907047.

[BD15] K. Bhanushali and W. R. Davis. FreePDK15: An Open-Source Predictive Process Design Kit for 15Nm FinFET Technology. In *Proc. Int'l Symp. on Physical Design (ISPD)*, pages 165–170. ACM, Mar. 2015. doi:10.1145/2717764.2717782.

[BFG12] N. Bombieri, F. Fummi, and V. Guarnieri. FAST-GP: An RTL Functional Verification Framework based on Fault Simulation on GP-GPUs. In *Proc. Conf. on Design, Automation Test in Europe (DATE)*, pages 562–565, Mar. 2012. doi:10.1109/DATE.2012.6176532.

[BGA07] S. Bose, H. Grimes, and V. D. Agrawal. Delay Fault Simulation with Bounded Gate Delay Model. In *Proc. Int'l Test Conf. (ITC)*, pages 1–10, Paper 26.3, Oct. 2007. doi:10.1109/TEST.2007.4437637.

[BJV06] M. J. Bellido, J. Juan, and M. Valencia. *Logic-timing Simulation and the Degradation Delay Model*. Imperial College Press, 2006. ISBN: 1-86094-589-9.

[BKK+93] S. Buchner, K. Kang, D. Krening, G. Lannan, and R. Schneiderwind. Dependence of the SEU Window of Vulnerability of a Logic Circuit on Magnitude of Deposited Charge. *IEEE Trans. on Nuclear Science*, 40(6):1853–1857, Dec. 1993. doi:10.1109/23.273470.

[BKL+82] A. K. Bose, P. Kozak, C. Y. Lo, H. N. Nham, E. Pacas-Skewes, and K. Wu. A Fault Simulator for MOS LSI Circuits. In *Proc. 19th Design Automation Conf. (DAC)*, pages 400–409, Jun. 1982. doi:10.1109/DAC.1982.1585530.

[BKN+03] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter Variations and Impact on Circuits and Microarchitecture. In *Proc. Design Automation Conf. (DAC)*, pages 338–342, Jun. 2003. doi:10.1145/775832.775920.

[BM09] A. H. Baba and S. Mitra. Testing for Transistor Aging. In *Proc. IEEE 27th VLSI Test Symp. (VTS)*, pages 215–220, May 2009. doi:10.1109/VTS.2009.56.

[BMA88] L. M. Brocco, S. P. McCormick, and J. Allen. Macromodeling CMOS circuits for timing simulation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 7(12):1237–1249, 12 1988. doi:10.1109/43.16802.

[BMC96] W. I. Baker, A. Mahmood, and B. S. Carlson. Parallel event-driven logic simulation algorithms: tutorial and comparative evaluation. *Proc. IEE - Circuits, Devices and Systems*, 143(4):177–185, Aug. 1996. doi:10.1049/ip-cds:19960477.

[BPSB00] T. D. Burd, T. A. Pering, A. J. Stratakos, and R. W. Brodersen. A Dynamic Voltage Scaled Microprocessor System. *IEEE Journ. of Solid-State Circuits*, 35(11):1571–1580, Nov. 2000. doi:10.1109/4.881202.

[Bry77] R. E. Bryant. Simulation of packet communication architecture computer systems. Technical report, Cambridge, MA, USA, 1977.

[Bry84] R. E. Bryant. A Switch-Level Model and Simulator for MOS Digital Systems. *IEEE Transactions on Computers (TC)*, C–33(2):160–177, Feb. 1984. doi:10.1109/TC.1984.1676408.

[Bry87] R. E. Bryant. A Survey of Switch-Level Algorithms. *IEEE Design & Test of Computers*, 4(4):26–40, Aug. 1987. doi:10.1109/MDT.1987.295146.

[CCCW16] H. H. Chen, S. Y. H. Chen, P. Y. Chuang, and C. W. Wu. Efficient Cell-Aware Fault Modeling by Switch-Level Test Generation. In *Proc. IEEE 25th Asian Test Symp. (ATS)*, pages 197–202, Nov. 2016. doi:10.1109/ATS.2016.33.

[CDB09a] D. Chatterjee, A. DeOrio, and V. Bertacco. Event-Driven Gate-Level Simulation with GP-GPUs. In *Proc. ACM/IEEE 46th Design Automation Conf. (DAC)*, pages 557–562, Jul. 2009. doi:10.1145/1629911.1630056.

[CDB09b] D. Chatterjee, A. DeOrio, and V. Bertacco. GCS: High-Performance Gate-Level Simulation with GP-GPUs. In *Proc. Conf. on Design, Automation Test in Europe (DATE)*, pages 1332–1337, Apr. 2009. doi:10.1109/DATE.2009.5090871.

[Cel91] F. E. Cellier. *Continuous System Modeling*. Springer NY, 1st edition, 1991. ISBN: 978-1-4757-3922-0.

[CGB01] L.-C. Chen, S. K. Gupta, and M. A. Breuer. A New Gate Delay Model for Simultaneous Switching and Its Applications. In *Proc. ACM/IEEE 38th Design Automation Conf. (DAC)*, pages 289–294, Paper 19.2, Jun. 2001. doi:10.1109/DAC.2001.156153.

[CGK75] B. Chawla, H. Gummel, and P. Kozak. MOTIS - An MOS Timing Simulator. *IEEE Trans. on Circuits and Systems*, 22(12):901–910, Dec. 1975. doi:10.1109/TCS.1975.1084003.

[CHE+08] A. Czutro, N. Houarche, P. Engelke, I. Polian, M. Comte, M. Renovell, and B. Becker. A Simulator of Small-Delay Faults Caused by Resistive-Open Defects. In *Proc. 13th European Test Symp. (ETS)*, pages 113–118, May 2008. doi:10.1109/ETS.2008.19.

[Che18] H. H. Chen. Beyond Structural Test, the Rising Need for System-Level Test. In *Proc. Int'l Symp. on VLSI Design, Automation and Test (VLSI-DAT)*, pages 1–4, Apr. 2018. doi:10.1109/VLSI-DAT.2018.8373238.

[CIJ+12] A. Czutro, M. E. Imhof, J. Jiang, A. Mumtaz, M. Sauer, B. Becker, I. Polian, and H.-J. Wunderlich. Variation-Aware Fault Grading. In *Proc. IEEE 21st Asian Test Symp. (ATS)*, pages 344–349, Nov. 2012. doi:10.1109/ATS.2012.14.

[CK06] F. E. Cellier and E. Kofman. *Continuous System Simulation*. Springer NY, 1st edition, 2006. ISBN: 978-0-387-26102-7.

[CKG13] J. F. Croix, S. P. Khatri, and K. Gulati. Using GPUs to Accelerate CAD Algorithms. *IEEE Design Test*, 30(1):8–16, Feb. 2013. doi:10.1109/MDAT.2013.2250053.

[Cle01] J. J. Clement. Electromigration Modeling for Integrated Circuit Interconnect Reliability Analysis. *IEEE Trans. on Device and Materials Reliability*, 1(1):33–42, Mar. 2001. doi:10.1109/7298.946458.

[CM79] K. M. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Trans. on Software Engineering (TSE)*, SE-5(5):440–452, Sept 1979. doi:10.1109/TSE.1979.230182.

[CRWY15] X. Chen, L. Ren, Y. Wang, and H. Yang. GPU-Accelerated Sparse LU Factorization for Circuit Simulation with Performance Modeling. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 26(3):786–795, Mar. 2015. doi:10.1109/TPDS.2014.2312199.

[CS96] V. Chandramouli and K. A. Sakallah. Modeling the Effects of Temporal Proximity of Input Transitions on Gate Propagation Delay and Transition Time. In *Proc. 33rd Design Automation Conf. (DAC)*, pages 617–622, Paper 40.2, Jun. 1996. doi:10.1109/DAC.1996.545649.

[CUS19] Ngspice on GPU - CUSPICE, 2019. http://ngspice.sourceforge.net/cuspice.html, [Last Access: Feb. 16, 2019].

[CWC17] P. Chuang, C. Wu, and H. H. Chen. Cell-Aware Test Generation Time Reduction by Using Switch-Level ATPG. In *Proc. Int'l Test Conf. in Asia (ITC-Asia)*, pages 27–32, Sep. 2017. doi:10.1109/ITC-ASIA.2017.8097105.

[DAJ+11] B. P. Das, B. Amrutur, H. S. Jamadagni, N. V. Arvind, and V. Visvanathan. Voltage and Temperature-Aware SSTA Using Neural Network Delay Model. *IEEE Trans. on Semiconductor Manufacturing (TSM)*, 24(4):533–544, Nov. 2011. doi:10.1109/TSM.2011.2163532.

[DED17a] H. Dhotre, S. Eggersglüß, and R. Drechsler. Identification of Efficient Clustering Techniques for Test Power Activity on the Layout. In *Proc. 26th IEEE Asian Test Symp. (ATS)*, pages 108–113, Nov. 2017. doi:10.1109/ATS.2017.31.

[DED+17b] H. Dhotre, S. Eggersglüß, M. Dehbashi, U. Pfannkuchen, and R. Drechsler. Machine Learning Based Test Pattern Analysis for Localizing Critical Power Activity Areas. In *Proc. IEEE Int'l Symp. on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6, Oct 2017. doi:10.1109/DFT.2017.8244464.

[Den10] Y. Deng. GPU Accelerated VLSI Design Verification. In *Proc. IEEE 10th Int'l Conf. on Computer and Information Technology (CIT)*, pages 1213–1218, Jun. 2010. doi:10.1109/CIT.2010.219.

[DG05] R. Drechsler and D. Grosse. System level validation using formal techniques. *IEE Proceedings - Computers and Digital Techniques*, 152(3):393–406, May 2005. doi:10.1049/ip-cdt:20045073.

[DGB+98] R. Degraeve, G. Groeseneken, R. Bellens, J. L. Ogier, M. Depas, P. J. Roussel, and H. E. Maes. New Insights in the Relation Between Electron Trap Generation and the Statistical Properties of Oxide Breakdown. *IEEE Trans. on Electron Devices*, 45(4):904–911, Apr. 1998. doi:10.1109/16.662800.

[DJA+08] B. P. Das, V. Janakiraman, B. Amrutur, H. S. Jamadagni, and N. V. Arvind. Voltage and Temperature Scalable Gate Delay and Slew Models Including Intra-Gate Variations. In *Proc. 21st Int'l Conf. on VLSI Design (VLSID)*, pages 685–691, Jan. 2008. doi:10.1109/VLSI.2008.92.

[DM08] Y. Deng and S. Mu. The Potential of GPUs for VLSI Physical Design Automation. In *Proc. 9th Int'l Conf. on Solid-State and Integrated-Circuit Technology (ICSICT)*, pages 2272–2275, Oct. 2008. doi:10.1109/ICSICT.2008.4735023.

[DOR87] P. Debefve, F. Odeh, and A. E. Ruehli. Waveform Techniques. In A. E. Ruehli, editor, *Circuit Analysis, Simulation and Design*, volume 3 of *Advances in CAD for VLSI*, chapter 8, pages 41–127. Elsevier Science B.V., 1987.

[DPA+17] C. K. Dabhi, S. S. Parihar, H. Agrawal, N. Paydavosi, T. H. Morshed, D. D. Lu, W. Yang, M. V. Dunga, X. Xi, J. He, W. Liu, K. M. Cao, X. Jin, J. J. Ou, M. Chan, Y. S. Chauhan, A. M. Niknejad, and Hu. C. *BSIM4 4.8.1 MOSFET Model - User's Manual*, Feb. 2017. [Last Access: Feb. 26, 2019].

[DWB+10] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge. Near-Threshold Computing: Reclaiming Moore's Law Through Energy Efficient Integrated Circuits. *Proceedings of the IEEE*, 98(2):253–266, Feb. 2010. doi:10.1109/JPROC.2009.2034764.

[DWM09] Y. Deng, B. D. Wang, and S. Mu. Taming Irregular EDA Applications on GPUs. In *Proc. IEEE/ACM Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 539–546, Nov. 2009. ISBN: 978-1-60558-800-1.

[ED11] S. Eggersglüß and R. Drechsler. As-Robust-As-Possible Test Generation in the Presence of Small Delay Defects using Pseudo-Boolean Optimization. In *Proc. Conf. on Design, Automation & Test in Europe (DATE)*, pages 1–6, Mar. 2011. doi:10.1109/DATE.2011.5763207.

[ED12] S. Eggersglüß and R. Drechsler. *High Quality Test Pattern Generation and Boolean Satisfiability*. Springer New York, 2012. ISBN: 1-86094-589-9.

[EFB81] B. Eitan and D. Frohman-Bentchkowsky. Hot-Electron Injection into the Oxide in n-Channel MOS Devices. *IEEE Trans. on Electron Devices*, 28(3):328–340, Mar. 1981. doi:10.1109/T-ED.1981.20336.

[EFD07] S. Eggerglüß, G. Fey, and R. Drechsler. SAT-based ATPG for Path Delay Faults in Sequential Circuits. In *Proc. IEEE Int'l Symp. on Circuits and Systems (ISCAS)*, pages 3671–3674, May 2007. doi:10.1109/ISCAS.2007.378639.

[Elm48] W. C. Elmore. The Transient Response of Damped Linear Networks with Particular Regard to Wideband Amplifiers. *Journal of Applied Physics*, 19(1):55–63, Jan. 1948. doi:10.1063/1.1697872.

[ETD10] S. Eggersglüß, D. Tille, and R. Drechsler. Efficient Test Generation with Maximal Crosstalk-Induced Noise using Unconstrained Aggressor Excitation. In *Proc. IEEE Int'l Symp. on Circuits and Systems (ISCAS)*, pages 649–652, May 2010. doi:10.1109/ISCAS.2010.5537503.

[FD17] D. Foley and J. Danskin. Ultra-Performance Pascal GPU and NVLink Interconnect. *IEEE Micro*, 37(2):7–17, Mar. 2017. doi:10.1109/MM.2017.37.

[FGRC17] F. Forero, J. Galliere, M. Renovell, and V. Champac. Analysis of Short Defects in FinFET Based Logic Cells. In *Proc. IEEE 18th Latin American Test Symp. (LATS)*, pages 1–6, Mar. 2017. doi:10.1109/LATW.2017.7906755.

[FM91] P. Franco and E. J. McCluskey. Delay Testing of Digital Circuits by Output Waveform Analysis. In *Proc. Int'l Test Conf. (ITC)*, pages 798–807, Paper 29.3, Oct. 1991. doi:10.1109/TEST.1991.519745.

[FS88] F. J. Ferguson and J. P. Shen. A CMOS Fault Extractor for Inductive Fault Analysis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 7(11):1181–1194, Nov. 1988. doi:10.1109/43.9188.

[GBRC09] A. Ghosh, R.B. Brown, R.M. Rao, and Ching-Te Chuang. A Precise Negative Bias Temperature Instability Sensor using Slew-Rate Monitor Circuitry. In *Proc. IEEE Int'l Symp. on Circuits and Systems (ISCAS)*, pages 381–384, May 2009. doi:10.1109/ISCAS.2009.5117765.

[GBW+09] R. Goldman, K. Bartleson, T. Wood, K. Kranen, C. Cao, V. Melikyan, and G. Markosyan. Synopsys' Open Educational Design Kit: Capabilities, Deployment and Future. In *Proc. IEEE Int'l Conf. on Microelectronic Systems Education (MSE)*, pages 20–24, Jul. 2009. doi:10.1109/MSE.2009.5270840.

[GCKS09] K. Gulati, J. F. Croix, S. P. Khatri, and R. Shastry. Fast Circuit Simulation on Graphics Processing Units. In *Proc. 14th Asia and South Pacific Design Automation Conf. (ASP-DAC)*, pages 403–408, Paper 4C–6s, Jan. 2009. doi:10.1109/ASPDAC.2009.4796514.

[GK08] K. Gulati and S. P. Khatri. Towards Acceleration of Fault Simulation using Graphics Processing Units. In *Proc. ACM/IEEE 45th Design Automation Conf. (DAC)*, pages 822–827, Paper 45.1, Jun. 2008. doi:10.1145/1391469.1391679.

[GK09] K. Gulati and S. P. Khatri. Accelerating Statistical Static Timing Analysis Using Graphics Processing Units. In *Proc. Asia and South Pacific Design Automation Conf. (ASP-DAC)*, pages 260–265, Paper 3B–1, Jan. 2009. doi:10.1109/ASPDAC.2009.4796490.

[GK10a] K. Gulati and S. P. Khatri. Fault Table Computation on GPUs. *Journal of Electronic Testing*, 26(2):195–209, Apr. 2010. doi:10.1007/s10836-010-5147-x.

[GK10b] K. Gulati and S. P. Khatri. *Hardware Acceleration of EDA Algorithms*. Springer, Boston, MA, 2010. ISBN: 978-1-4419-0944-2.

[GKET16] M. S. Golanbari, S. Kiamehr, M. Ebrahimi, and M. B. Tahoori. Variation-Aware Near Threshold Circuit Synthesis. In *Proc. Conf. on Design, Automation Test in Europe (DATE)*, pages 1237–1242, Mar. 2016.

[GMS88] S. Gai, P. L. Montessoro, and F. Somenzi. MOZART: A Concurrent Multilevel Simulator. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 7(9):1005–1016, Sep. 1988. doi:10.1109/43.7799.

[GNW10] P. Girard, N. Nicolici, and X. Wen, editors. *Power-Aware Testing and Test Strategies for Low Power Devices*. Springer New York, 2010. ISBN: 978-1-4419-0928-2.

[GSR+14] K. U. Giering, C. Sohrmann, G. Rzepa, L. Heiß, T. Grasser, and R. Jancke. NBTI modeling in analog circuits and its application to long-term aging simulations. In *Proc. IEEE Int'l Integrated Reliability Workshop (IIRW)*, pages 29–34, Oct. 2014. doi:10.1109/IIRW.2014.7049501.

[Hay86] J. P. Hayes. Digital Simulation with Multiple Logic Values. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 5(2):274–283, Apr. 1986. doi:10.1109/TCAD.1986.1270196.

[Hay87] J. P. Hayes. An Introduction to Switch-Level Modeling. *IEEE Design & Test of Computers*, 4(4):18–25, Aug. 1987. doi:10.1109/MDT.1987.295145.

[HBH+10] F. Hopsch, B. Becker, S. Hellebrand, I. Polian, B. Straube, W. Vermeiren, and H.-J. Wunderlich. Variation-aware fault modeling. In *Proc. IEEE 19th Asian Test Symp. (ATS)*, pages 87–93, Dec. 2010. doi:10.1109/ATS.2010.24.

[HBPW14] N. Hatami, R. Baranowski, P. Prinetto, and H.-J. Wunderlich. Multilevel Simulation of Nonfunctional Properties by Piecewise Evaluation. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 19(4):37:1–37:21, Aug. 2014. doi:10.1145/2647955.

[HF16] L. Han and Z. Feng. TinySPICE Plus: Scaling Up Statistical SPICE Simulations on GPU Leveraging Shared-memory Based Sparse Matrix Solution Techniques. In *Proc. 35th Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 1–6, Paper 99. ACM, Nov. 2016. ISBN: 978-1-4503-4466-1.

[HGV+06] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M. R. Stan. HotSpot: A Compact Thermal Modeling Methodology for Early-Stage VLSI Design. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems (TVLSI)*, 14(5):501–513, May 2006. doi:10.1109/TVLSI.2006.876103.

[HIK+14] S. Hellebrand, T. Indlekofer, M. Kampmann, M. A. Kochte, C. Liu, and H.-J. Wunderlich. FAST-BIST: Faster-than-At-Speed BIST Targeting Hidden Delay Defects. In *Proc. IEEE Int'l Test Conf. (ITC)*, pages 1–8, Paper 29.3, Oct. 2014. doi:10.1109/TEST.2014.7035360.

[HIW15] S. Holst, M. E. Imhof, and H.-J. Wunderlich. High-Throughput Logic Timing Simulation on GPGPUs. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 20(3):1–22, Article 37, Jun. 2015. doi:10.1145/2714564.

[HKBG+09] F. Hapke, R. Krenz-Baath, A. Glowatz, J. Schloeffel, H. Hashempour, S. Eichenberger, C. Hora, and D. Adolfsson. Defect-Oriented Cell-Aware ATPG and Fault Simulation for Industrial Cell Libraries and Designs. In *Proc. IEEE Int'l Test Conf. (ITC)*, pages 1–10, Paper 1.2, Nov. 2009. doi:10.1109/TEST.2009.5355741.

[HLK+00] D. Hisamoto, W.-C. Lee, J. Kedzierski, H. Takeuchi, A. Asano, C. Kuo, E. Anderson, T.-J. King, J. Bokor, and C. Hu. FinFET – A Self-Aligned Double-Gate MOSFET Scalable to 20 nm. *IEEE Trans. on Electron Devices*, 47(12):2320–2325, Dec. 2000. doi:10.1109/16.887014.

[HM91] H. Hao and E. J. McCluskey. "Resistive Shorts" within CMOS Gates. In *Proc. Int'l Test Conf. (ITC)*, pages 292–301, Oct. 1991. doi:10.1109/TEST.1991.519521.

[HP12] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 5th edition, 2012. ISBN: 978-0-123838728.

[HPA96] K. Heragu, J. H. Patel, and V. D. Agrawal. Segment Delay Faults: A New Fault Model. In *Proc. 14th VLSI Test Symp. (VTS)*, pages 32–39, Apr. 1996. doi:10.1109/VTEST.1996.510832.

[HRG+14] F. Hapke, W. Redemund, A. Glowatz, J. Rajski, M. Reese, M. Hustava, M. Keim, J. Schloeffel, and A. Fast. Cell-Aware Test. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 33(9):1396–1409, Sep. 2014. doi:10.1109/TCAD.2014.2323216.

[HS14] C. Han and A. D. Singh. Improving CMOS Open Defect Coverage Using Hazard Activated Tests. In *Proc. IEEE 32nd VLSI Test Symp. (VTS)*, pages 1–6, Apr. 2014. doi:10.1109/VTS.2014.6818740.

[HS15] C. Han and A.D. Singh. Testing Cross Wire Opens within Complex Gates. In *Proc. IEEE 33rd VLSI Test Symp. (VTS)*, pages 1–6, Apr. 2015. doi:10.1109/VTS.2015.7116301.

[HSK+17] S. Holst, E. Schneider, K. Kawagoe, M. A. Kochte, K. Miyase, H.-J. Wunderlich, S. Kajihara, and X. Wen. Analysis and Mitigation of IR-Drop Induced Scan Shift-Errors. In *Proc. IEEE 48th Int'l Test Conf. (ITC)*, pages 1–7, Paper 3.4, Oct. 2017. doi:10.1109/TEST.2017.8242055.

[HSW12] S. Holst, E. Schneider, and H.-J. Wunderlich. Scan Test Power Simulation on GPGPUs. In *Proc. IEEE 21st Asian Test Symp. (ATS)*, pages 155–160, Nov. 2012. doi:10.1109/ATS.2012.23.

[HSW+16] S. Holst, E. Schneider, X. Wen, S. Kajihara, Y. Yamato, H.-J Wunderlich, and M. A. Kochte. Timing-Accurate Estimation of IR-Drop Impact on Logic- and Clock-Paths During At-Speed Scan Test. In *Proc. IEEE 25th Asian Test Symp. (ATS)*, pages 19–24, Nov. 2016. doi:10.1109/ATS.2016.49.

[HTWS16] K. He, S. X. D. Tan, H. Wang, and G. Shi. GPU-Accelerated Parallel Sparse LU Factorization Method for Fast Circuit Analysis. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 24(3):1140–1150, Mar. 2016. doi:10.1109/TVLSI.2015.2421287.

[HTZ15] G. Harutyunyan, G. Tshagharyan, and Y. Zorian. Impact of Parameter Variations on FinFET Faults. In *Proc. IEEE 33rd VLSI Test Symp. (VTS)*, pages 1–4, April 2015. doi:10.1109/VTS.2015.7116276.

[Hu11] C. Hu. New Sub-20nm Transistors – Why and How. In *Proc. ACM/EDAC/IEEE 48th Design Automation Conf. (DAC)*, pages 460–463, Jun. 2011. ISBN: 978-1-4503-0636-2.

[HVC97] U. Hubner, H. T. Vierhaus, and R. Camposano. Partitioning and Analysis of Static Digital CMOS Circuits. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 16(11):1292–1310, Nov. 1997. doi:10.1109/43.663819.

[HW07] S. Holst and H.-J. Wunderlich. Adaptive Debug and Diagnosis without Fault Dictionaries. In *Proc. IEEE European Test Symp. (ETS)*, pages 7–12, May 2007. doi:10.1109/ETS.2007.9.

[HZF13] L. Han, X. Zhao, and Z. Feng. TinySPICE: A Parallel SPICE Simulator on GPU for Massively Repeated Small Circuit Simulations. In *Proc. ACM/EDAC/IEEE 50th Design Automation Conf. (DAC)*, pages 1–8, Article 89, May 2013. doi:10.1145/2463209.2488843.

[IEE01a] IEEE Computer Society. IEEE Standard for Standard Delay Format (SDF) for the Electronic Design Process. *IEEE Std 1497-2001*, Dec. 2001. doi:10.1109/IEEESTD.2001.93359.

[IEE01b] IEEE Computer Society. IEEE Standard Verilog Hardware Description Language. *IEEE Std 1364-2001*, pages 1–856, Sep. 2001. doi:10.1109/IEEESTD.2001.93352.

[IEE08] IEEE Computer Society. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug. 2008. doi:10.1109/IEEESTD.2008.4610935.

[IEE09] IEEE Computer Society. IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008*, Jan. 2009. doi:10.1109/IEEESTD.2009.4772740.

[IEE10] IEEE Computer Society. IEEE Standard for Integrated Circuit (IC) Open Library Architecture (OLA). *IEEE Std 1481-2009*, pages c1–658, Mar. 2010. doi:10.1109/IEEESTD.2009.5430852.

[IEE12] IEEE Computer Society. IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011*, pages 1–638, Jan. 2012. doi:10.1109/IEEESTD.2012.6134619.

[IRW90] V. S. Iyengar, B. K. Rosen, and J. A. Waicukauski. On Computing the Sizes of Detected Delay Faults. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 9(3):299–312, Mar. 1990. doi:10.1109/43.46805.

[IT88] V. S. Iyengar and D. T. Tang. On Simulating Faults in Parallel. In *Proc. 18th Int'l Symp. on Fault-Tolerant Computing (FTCS)*, pages 110–115, Jun. 1988. doi:10.1109/FTCS.1988.5307.

[ITC99] ITC99 Benchmark Home Page, 1999. http://www.cerc.utexas.edu/itc99-benchmarks/bench.html, [Last Access: Feb. 26, 2019].

[ITR15] ITRS. International Technology Roadmap for Semiconductors 2.0: 2015, 2015. http://www.itrs2.net, [Last Access: May 30, 2018].

[JAC+13] J. Jiang, M. Aparicio, M. Comte, F. Aza is, M. Renovell, and I. Polian. MIRID: Mixed-Mode IR-Drop Induced Delay Simulator. In *Proc. 22nd Asian Test Symp. (ATS)*, pages 177–182, Nov. 2013. doi:10.1109/ATS.2013.41.

[JRW14] A. Jutman, M. S. Reorda, and H.-J. Wunderlich. High Quality System Level Test and Diagnosis. In *Proc. IEEE 23rd Asian Test Symp. (ATS)*, pages 298–305, Nov 2014. doi:10.1109/ATS.2014.62.

[Kao92] R. Kao. Piecewise Linear Models for Switch-level Simulation. Technical report, Departments of Electrical Engineering and Computer Science, Stanford University, 1992.

[Kar72] R. M. Karp. Reducibility Among Combinatorial Problems. In *Proc. Symp. on Complexity of Computer Computations*, pages 85–103, Mar. 1972. doi:10.1007/978-1-4684-2001-2_9.

[KCK+10] Y. M. Kim, T. W. Chen, Y. Kameda, M. Mizuno, and S. Mitra. Gate-Oxide Early-Life Failure Identification using Delay Shifts. In *Proc. 28th VLSI Test Symp. (VTS)*, pages 69–74, Apr. 2010. doi:10.1109/VTS.2010.5469615.

[KET16] S. Kiamehr, M. Ebrahimi, and M. Tahoori. Temperature-aware Dynamic Voltage Scaling for Near-Threshold Computing. In *Proc. Int'l Great Lakes Symp. on VLSI (GLSVLSI)*, pages 361–364, May 2016. doi:10.1145/2902961.2902997.

[Khr17] Khronos OpenCL Working Group. The OpenCL Specification - Version: 2.2, Mar. 2017. https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf, [Last Access: Feb. 26, 2019].

[Kin05] T.-J. King. FinFETs for Nanoscale CMOS Digital Integrated Circuits. In *Proc. IEEE/ACM Int'l Conf. on Computer-aided Design (ICCAD)*, pages 207–210. IEEE Computer Society, Nov. 2005. doi:10.1109/ICCAD.2005.1560065.

[KKK+10] Y. M. Kim, Y. Kameda, H. Kim, M. Mizuno, and S. Mitra. Low-Cost Gate-Oxide Early-Life Failure Detection in Robust Systems. In *Proc. IEEE Symp. on VLSI Circuits (VLSIC)*, pages 125–126, Jun. 2010. doi:10.1109/VLSIC.2010.5560326.

[KKL+18] M. Kampmann, M. A. Kochte, C. Liu, E. Schneider, S. Hellebrand, and H.-J. Wunderlich. Built-in Test for Hidden Delay Faults. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, pages 1–13, Aug. 2018. doi:10.1109/TCAD.2018.2864255.

[KKS+15] M. Kampmann, M. A. Kochte, E. Schneider, T. Indlekofer, S. Hellebrand, and H.-J. Wunderlich. Optimized Selection of Frequencies for Faster-Than-at-Speed Test. In *Proc. IEEE 24th Asian Test Symp. (ATS)*, pages 109–114, Nov. 2015. doi:10.1109/ATS.2015.26.

[Kno65] K. C. Knowlton. A Fast Storage Allocator. *Communications of the ACM*, 8(10):623–624, Oct. 1965. doi:10.1145/365628.365655.

[Knu97a] D. E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 3rd edition, 1997. ISBN: 0-201-89683-4.

[Knu97b] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Publishing Co., Inc., 3rd edition, 1997. ISBN: 0-201-89684-2.

[Kon00] H. Konuk. On Invalidation Mechanisms for Non-Robust Delay Tests. In *Proc. Int'l Test Conf. (ITC)*, pages 393–399, Paper 14.3, Oct. 2000. doi:10.1109/TEST.2000.894230.

[KSWZ10] M. A. Kochte, M. Schaal, H.-J. Wunderlich, and C. G. Zoellin. Efficient Fault Simulation on Many-Core Processors. In *Proc. ACM/IEEE 47th Design Automation Conf. (DAC)*, pages 380–385, Paper 23.4, Jun. 2010. doi:10.1145/1837274.1837369.

[KZB+10] M. A. Kochte, C. G. Zöllin, R. Baranowski, M. E. Imhof, H.-J. Wunderlich, N. Hatami, S. Di Carlo, and P. Prinetto. Efficient Simulation of Structural Faults for the Reliability Evaluation at System-Level. In *Proc. IEEE 19th Asian Test Symp. (ATS)*, pages 3–8, Dec. 2010. doi:10.1109/ATS.2010.10.

[Lam05] W. K. Lam. *Hardware Design Verification*. Prentice Hall, 1st edition, 2005. ISBN: 0-13-143347-4.

[LGS09] D. Lorenz, G. Georgakos, and U. Schlichtmann. Aging Analysis of Circuit Timing Considering NBTI and HCI. In *Proc. 15th IEEE Int'l On-Line Testing Symp. (IOLTS)*, pages 3–8, Jun. 2009. doi:10.1109/IOLTS.2009.5195975.

[LH91] H. K. Lee and D. S. Ha. An Efficient, Forward Fault Simulation Algorithm based on the Parallel Pattern Single Fault Propagation. In *Proc. Int'l Test Conf. (ITC)*, pages 946–955, Paper 35.1, Oct. 1991. doi:10.1109/TEST.1991.519760.

[LH10] M. Li and M. S. Hsiao. FSimGP^2: An Efficient Fault Simulator with GPGPU. In *Proc. IEEE 19th Asian Test Symp. (ATS)*, pages 15–20, Dec. 2010. doi:10.1109/ATS.2010.12.

[LH11] M. Li and M. S. Hsiao. 3-D Parallel Fault Simulation With GPGPU. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 30(10):1545–1555, Oct. 2011. doi:10.1109/TCAD.2011.2158432.

[Lin87] S. M. Lin, C. J.and Reddy. On Delay Fault Testing in Logic Circuits. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 6(5):694–703, Sep. 1987. doi:10.1109/TCAD.1987.1270315.

[LKW17] C. Liu, M. A. Kochte, and H.-J. Wunderlich. Aging Monitor Reuse for Small Delay Fault Testing. In *Proc. 35th VLSI Test Symp. (VTS)*, pages 1–6, Apr. 2017. doi:10.1109/VTS.2017.7928921.

[LNB83] C.-Y. Lo, H. N. Nham, and A. K. Bose. A Data Structure for MOS Circuits. In *Proc. 20th Design Automation Conf. (DAC)*, pages 619–624, Jun. 1983. doi:10.1109/DAC.1983.1585719.

[LQB08] X. Li, J. Qin, and J. B. Bernstein. Compact Modeling of MOSFET Wearout Mechanisms for Circuit-Reliability Simulation. *IEEE Trans. on Device and Materials Reliability (TDMR)*, 8(1):98–121, Mar. 2008. doi:10.1109/TDMR.2008.915629.

[LSK+18] C. Liu, E. Schneider, M. Kampmann, S. Hellebrand, and H.-J. Wunderlich. Extending Aging Monitors for Early Life and Wear-Out Failure Prevention. In *Proc. IEEE 27th Asian Test Symp. (ATS)*, pages 92–97, Oct. 2018. doi:10.1109/ATS.2018.00028.

[LTL18] L. Lai, H. Tsai, and H. Li. GPGPU-based ATPG System: Myth or Reality? *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, pages 1–9, Dec. 2018. doi:10.1109/TCAD.2018.2884992.

[LX12] Y. Liu and Q. Xu. On Modeling Faults in FinFET Logic Circuits. In *Proc. IEEE Int'l Test Conf. (ITC)*, pages 1–9, Paper 11.3, Nov. 2012. doi:10.1109/TEST.2012.6401565.

[LXH+10] H. Li, D. Xu, Y. Han, K. T. Cheng, and X. Li. *nGFSIM* : A GPU-Based Fault Simulator for 1-to-n Detection and its Applications. In *Proc. IEEE Int'l Test Conf. (ITC)*, pages 1–10, Paper 12.1, Nov. 2010. doi:10.1109/TEST.2010.5699235.

[MAJP00] A. K. Majhi, V. D. Agrawal, J. Jacob, and L. M. Patnaik. Line Coverage of Path Delay Faults. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 8(5):610–614, Oct. 2000. doi:10.1109/92.894166.

[MC93] W. Meyer and R. Camposano. Fast, Accurate, Integrated Gate- and Switch-Level Fault Simulation. In *Proc. 3rd European Test Conf. (ETC)*, pages 194–199, Apr. 1993. doi:10.1109/ETC.1993.246517.

[MC95] W. Meyer and R. Camposano. Active Timing Multilevel Fault-Simulation with Switch-Level Accuracy. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 14(10):1241–1256, Oct. 1995. doi:10.1109/43.466340.

[MMGA19] J. Mahmod, S. Millican, U. Guin, and V. Agrawal. Delay Fault Testing: Present and Future. In *Proc. IEEE 37th VLSI Test Symp. (VTS) [to appear]*, pages 1–10, Apr. 2019.

[MMR+15] M. Martins, J. M. Matos, R. P. Ribas, A. Reis, G. Schlinker, L. Rech, and J. Michelsen. Open Cell Library in 15nm FreePDK Technology. In *Proc. Int'l Symp. on Physical Design (ISPD)*, pages 171–178, Mar. 2015. doi:10.1145/2717764.2717783.

[Moo65] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, Apr. 1965. doi:10.1109/N-SSC.2006.4785860.

[Moo75] G. E. Moore. Progress In Digital Integrated Electronics. In *Proc. IEEE International Electron Devices Meeting*, volume 21, pages 11–13, Sep. 1975. doi:10.1109/N-SSC.2006.4804410.

[MRD92] E. Melcher, W. Röthig, and M. Dana. Multiple input transitions in CMOS gates. *Microprocessing and Microprogramming*, 35(1–5):683–690, Sep. 1992. doi:10.1016/0165-6074(92)90387-M.

[MSB+15] K. Miyase, M. Sauer, B. Becker, X. Wen, and S. Kajihara. Identification of High Power Consuming Areas with Gate Type and Logic Level Information. In *Proc. IEEE 20th European Test Symp. (ETS)*, pages 1–6, May 2015. doi:10.1109/ETS.2015.7138773.

[MV91] W. Meyer and H. T. Vierhaus. Switch-Level Fault Simulation for Non-Trivial Faults Based on Abstract Data Types. In *Proc. 5th Annual European Computer Conference (CompEuro)*, pages 233–237, May 1991. doi:10.1109/CMPEUR.1991.257388.

[Nan10] Nangate Inc. NanGate 45nm Open Cell Library, 2010. http://www.nangate.com/, [Last Access: Aug. 4, 2017], Retrievable via https://www.silvaco.com [Feb. 26, 2019].

[Nan14] Nangate Inc. NanGate 15nm Open Cell Library, 2014. http://www.nangate.com/, [Last Access: Aug. 4, 2017], Retrievable via https://www.silvaco.com [Feb. 26, 2019].

[Nan17] Nanoscale Integration and Modeling (NIMO) Group. Predictive Technology Model (PTM), 2017. http://ptm.asu.edu/, [Last Access: Feb. 26, 2019].

[Nas01] S. R. Nassif. Modeling and Analysis of Manufacturing Variations. In *Proc. IEEE Custom Integrated Circuits Conf. (CICC)*, pages 223–228, May 2001. doi:10.1109/CICC.2001.929760.

[ND10] J. Nickolls and W. J. Dally. The GPU Computing Era. *IEEE Micro*, 30(2):56–69, Mar. 2010. doi:10.1109/MM.2010.41.

[NG00] P. Nigh and A. Gattiker. Test Method Evaluation Experiments & Data. In *Proc. Int'l Test Conf. (ITC)*, pages 454–463, Paper 17.1, Oct. 2000. doi:10.1109/TEST.2000.894237.

[NP73] L. 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.

[NPJS10] M. Nanjundappa, H. D. Patel, B. A. Jose, and S. K. Shukla. SCGPSim: A fast SystemC simulator on GPUs. In *Proc. 15th Asia and South Pacific Design Automation Conf. (ASP-DAC)*, pages 149–154, Jan. 2010. doi:10.1109/ASPDAC.2010.5419903.

[NVI10] NVIDIA Corporation. NVIDIA GF100. Whitepaper, Version 1.5, 2010. http://www.nvidia.com, [Last Access: May 03, 2018].

[NVI14] NVIDIA Corporation. NVIDIA Kepler GK110/210. Whitepaper, Version 1.1, 2014. http://www.nvidia.com, [Last Access: Feb. 05, 2018].

[NVI17a] NVIDIA Corporation. CUDA Zone | NVIDIA Developer, 2017. https://developer.nvidia.com/cuda-zone, [Last Access: Feb. 26, 2019].

[NVI17b] NVIDIA Corporation. High Performance Computing (HPC) and Supercomputing | NVIDIA Tesla | NVIDIA, 2017. http://www.nvidia.com/object/tesla-supercomputing-solutions.html, [Last Access: Aug. 4, 2017].

[NVI17c] NVIDIA Corporation. NVIDIA Tesla P100. Whitepaper, Version 1.2, 2017. http://www.nvidia.com/object/pascal-architecture-whitepaper.html, [Last Access: Aug. 4, 2017].

[NVI17d] NVIDIA Corporation. NVIDIA Tesla V100. Whitepaper, Version 1.1, 2017. http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, [Accessed: Aug. 4, 2018].

[NVI18a] NVIDIA Corporation. CUDA C Best Practices Guide, Mar. 2018. http://www.nvidia.com, [Last Access: Feb. 26, 2019].

[NVI18b] NVIDIA Corporation. CUDA C Programming Guide, Mar. 2018. http://www.nvidia.com, [Last Access: Feb. 26, 2019].

[NVI18c] NVIDIA Corporation. GPU-Accelerated Applications Catalog. Whitepaper, Mar. 2018. https://www.nvidia.com/content/gpu-applications/PDF/gpu-applications-catalog.pdf, [Last Access: Feb. 26, 2019].

[NVI18d] NVIDIA Corporation. NVIDIA DGX-2, Apr. 2018. http://www.nvidia.com/DGX-2, [Last Access: May. 4, 2018].

[NVI18e] NVIDIA Corporation. Tuning CUDA Applications for Pascal. Application Note, Mar. 2018. http://www.nvidia.com, [Last Access: Feb. 26, 2019].

[OHL+08] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008. doi:10.1109/JPROC.2008.917757.

[PB07] O. Poku and R. D. Blanton. Delay Defect Diagnosis using Segment Network Faults. In *Proc. IEEE Int'l Test Conf. (ITC)*, pages 1–10, Paper 15.1, Oct. 2007. doi:10.1109/TEST.2007.4437602.

[PBH+11] I. Polian, B. Becker, S. Hellebrand, H.-J. Wunderlich, and P. Maxwell. Towards Variation-Aware Test Methods. In *Proc. IEEE 16th European Test Symp. (ETS)*, pages 219–225, May 2011. doi:10.1109/ETS.2011.51.

[PR08] I. Pomeranz and S. M. Reddy. Transition Path Delay Faults: A New Path Delay Fault Model for Small and Large Delay Defects. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 16(1):98–107, Jan. 2008. doi:10.1109/TVLSI.2007.909796.

[QD11] H. Qian and Y. Deng. Accelerating RTL Simulation with GPUs. In *Proc. IEEE/ACM Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 687–693, Nov. 2011. doi:10.1109/ICCAD.2011.6105404.

[QS10] X. Qian and A. D. Singh. Distinguishing Resistive Small Delay Defects from Random Parameter Variations. In *Proc. IEEE 19th Asian Test Symp. (ATS)*, pages 325–330, Dec. 2010. doi:10.1109/ATS.2010.62.

[RAH+14] P. G. Ryan, I. Aziz, W. B. Howell, T. K. Janczak, and D. J. Lu. Process Defect Trends and Strategic Test Gaps. In *Proc. Int'l Test Conf. (ITC)*, pages 1–8, Paper 1.1, Oct. 2014. doi:10.1109/TEST.2014.7035276.

[RGA87] W. A. Rogers, J. F. Guzolek, and J. A. Abraham. Concurrent Hierarchical Fault Simulation: A Performance Model and Two Optimizations. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 6(5):848–862, Sep. 1987. doi:10.1109/TCAD.1987.1270328.

[RGW16] L. Rodríguez Gómez and H.-J. Wunderlich. A Neural-Network-Based Fault Classifier. In *Proc. IEEE 25th Asian Test Symp. (ATS)*, pages 144–149, Nov. 2016. doi:10.1109/ATS.2016.46.

[RK08] M. Radetzki and R. Salimi Khaligh. Accuracy-adaptive Simulation of Transaction Level Models. In *Proc. Conf. on Design, Automation and Test in Europe (DATE)*, pages 788–791, Mar. 2008. ISBN: 978-3-9810801-3-1.

[RMdGV02] R. Rodríguez Montañés, J. P. de Gyvez, and P. Volf. Resistance Characterization for Weak Open Defects. *IEEE Design & Test of Computers*, 19(5):18–26, Sep. 2002. doi:10.1109/MDT.2002.1033788.

[RPH83] J. Rubinstein, P. Penfield, and M. A. Horowitz. Signal Delay in RC Tree Networks. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2(3):202–211, Jul. 1983. doi:10.1109/TCAD.1983.1270037.

[RS11] K. Rupp and S. Selberherr. The Economic Limit to Moore's Law. *IEEE Trans. on Semiconductor Manufacturing*, 24(1):1–4, Feb. 2011. doi:10.1109/TSM.2010.2089811.

[SABM10] A. Sen, B. Aksanli, M. Bozkurt, and M. Mert. Parallel Cycle Based Logic Simulation Using Graphics Processing Units. In *Proc. 9th Int'l Symp. on Parallel and Distributed Computing (ISPDC)*, pages 71–78, Jul. 2010. doi:10.1109/ISPDC.2010.26.

[SAKF08] S. Soleimani, A. Afzali-Kusha, and B. Forouzandeh. Temperature Dependence of Propagation Delay Characteristic in FinFET Circuits. In *Proc. Int'l Conf. on Microelectronics (ICM)*, pages 276–279, Dec. 2008. doi:10.1109/ICM.2008.5393513.

[SBJ+03] J. Saxena, K. M. Butler, V. B. Jayaram, S. Kundu, N. V. Arvind, P. Sreeprakash, and M. Hachinger. A Case Study of IR-Drop in Structured At-Speed Testing. In *Proc. Int'l Test Conf. (ITC)*, volume 1, pages 1098–1104, Paper 42.2, Sep. 2003. doi:10.1109/TEST.2003.1271098.

[SCPB12] M. Sauer, A. Czutro, I. Polian, and B. Becker. Small-Delay-Fault ATPG with Waveform Accuracy. In *Proc. IEEE/ACM Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 30–36, Nov. 2012. doi:10.1145/2429384.2429391.

[SDC94] S. Sun, D. H.C. Du, and H.-C. Chen. Efficient Timing Analysis for CMOS Circuits Considering Data Dependent Delays. In *Proc. 1994 IEEE Int'l Conf. on Computer Design (ICCD)*, pages 156–159, Oct. 1994. doi:10.1109/ICCD.1994.331878.

[SG91] L. Soulé and A. Gupta. An Evaluation of the Chandy-Misra-Bryant Algorithm for Digital Logic Simulation. *ACM Trans. on Modeling and Computer Simulation*, 1(4):308–347, Oct. 1991. doi:10.1145/130611.130613.

[SGYW05] M. Shao, Y. Gao, L.-P. Yuan, and M. D. R. Wong. IR drop and Ground Bounce Awareness Timing Model. In *Proc. IEEE Computer Society Annual Symposium on VLSI: New Frontiers in VLSI Design (ISVLSI)*, pages 226–231, May 2005. doi:10.1109/ISVLSI.2005.44.

[SHK+15] E. Schneider, S. Holst, M. A. Kochte, X. Wen, and H.-J. Wunderlich. GPU-Accelerated Small Delay Fault Simulation. In *Proc. ACM/IEEE Conf. on Design, Automation Test in Europe (DATE)*, pages 1174–1179, Mar. 2015. doi:10.7873/DATE.2015.0077.

[SHM+05] Y. Sato, S. Hamada, T. Maeda, A. Takatori, and S. Kajihara. Evaluation of the Statistical Delay Quality Model. In *Proc. Asia and South Pacific Design Automation Conf. (ASP-DAC)*, volume 1, pages 305–310, Jan. 2005. doi:10.1109/ASPDAC.2005.1466179.

[Sho86] M. Shoji. Elimination of Process-Dependent Clock Skew in CMOS VLSI. *IEEE Journ. of Solid-State Circuits (JSSC)*, 21(5):875–880, Oct. 1986. doi:10.1109/JSSC.1986.1052620.

[SHWW14] E. Schneider, S. Holst, X. Wen, and H.-J. Wunderlich. Data-Parallel Simulation for Fast and Accurate Timing Validation of CMOS Circuits. In *Proc. IEEE/ACM 33rd Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 17–23, Nov. 2014. doi:10.1109/ICCAD.2014.7001324.

[Sin16] A. D. Singh. Cell Aware and Stuck-Open Tests. In *Proc. IEEE 21st European Test Symp. (ETS)*, pages 1–6, Paper 15.1, May 2016. doi:10.1109/ETS.2016.7519316.

[SJN94] R. A. Saleh, S.-J. Jon, and A. R. Newton. *Mixed-Mode Simulation and Analog Multilevel Simulation*, volume 279 of *The Springer International Series in Engineering and Computer Science*. Springer US, 1st edition, 1994. ISBN: 978-0-7923-9473-0.

[SKH+17] E. Schneider, M. A. Kochte, S. Holst, X. Wen, and H.-J. Wunderlich. GPU-Accelerated Simulation of Small Delay Faults. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 36(5):829–841, May 2017. doi:10.1109/TCAD.2016.2598560.

[SKR10] R. Salimi Khaligh and M. Radetzki. Modeling Constructs and Kernel for Parallel Simulation of Accuracy Adaptive TLMs. In *Proc. Conf. on Design, Automation and Test in Europe (DATE)*, pages 1183–1188, Mar. 2010. doi:10.1109/DATE.2010.5456987.

[SKW18] E. Schneider, M. A. Kochte, and H.-J. Wunderlich. Multi-Level Timing Simulation on GPUs. In *Proc. ACM/IEEE 23rd Asia and South Pacific Design Automation Conf. (ASP-DAC)*, pages 470–475, Jan. 2018. doi:10.1109/ASPDAC.2018.8297368.

[SMF85] J. P. Shen, M. Maly, and F. J. Ferguson. Inductive Fault Analysis of MOS Integrated Circuits. *IEEE Design & Test of Computers*, 2(6):13–26, Dec. 1985. doi:10.1109/MDT.1985.294793.

[Smi85] G. L. Smith. Model for Delay Faults Based Upon Paths. In *Proc. Int'l Test Conf. (ITC)*, pages 342–351, 1985.

[SN90] R. A. Saleh and A. R. Newton. *Mixed-Mode Simulation*. The Springer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1990. ISBN: 9781461306955.

[SPI+14] M. Sauer, I. Polian, M. E. Imhof, A. Mumtaz, E. Schneider, A. Czutro, H.-J. Wunderlich, and B. Becker. Variation-Aware Deterministic ATPG. In *Proc. IEEE 19th European Test Symp. (ETS)*, pages 1–6, May 2014. doi:10.1109/ETS.2014.6847806.

[SRG+11] L. Suresh, N. Rameshan, M. S. Gaur, M. Zwolinski, and V. Laxmi. Acceleration of Functional Validation Using GPGPU. In *Proc. IEEE 6th Int'l Symp. on Electronic Design, Test and Application (DELTA)*, pages 211–216, Jan. 2011. doi:10.1109/DELTA.2011.46.

[SSB89] M. H. Schulz, B. H. Seiss, and F. Brglez. Hierarchical Fault Simulation in Combinational Circuits. In *Proc. 1st European Test Conf. (ETC)*, pages 33–40, Apr. 1989. doi:10.1109/ETC.1989.36217.

[SSB05] A. Srivastava, D. Sylvester, and D. Blaauw. *Statistical Analysis and Optimization for VLSI: Timing and Power*. Springer, 1st edition, 2005. ISBN: 978-0-38-726528-5.

[STKS17] S. K. Sadasivam, B. W. Thompto, R. Kalla, and W. J. Starke. IBM Power9 Processor Architecture. *IEEE Micro*, 37(2):40–51, Mar. 2017. doi:10.1109/MM.2017.40.

[STO95] C. Sebeke, J. P. Teixeira, and M. J. Ohletz. Automatic Fault Extraction and Simulation of Layout Realistic Faults for Integrated Analogue Circuits. In *Proc. European Conf. on Design and Test (EDTC)*, pages 464–468, Mar. 1995. ISBN: 0-8186-7039-8.

[SW19a] E. Schneider and H.-J. Wunderlich. SWIFT: Switch Level Fault Simulation on GPUs. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 38(1):122–135, Jan. 2019. doi:10.1109/TCAD.2018.2802871.

[SW19b] E. Schneider and H.-J. Wunderlich. Multi-level timing and fault simulation on GPUs. *Integration*, 64:78–91, Jan. 2019. doi:10.1016/j.vlsi.2018.08.005.

[SW16] E. Schneider and H.-J. Wunderlich. High-Throughput Transistor-Level Fault Simulation on GPUs. In *Proc. IEEE 25th Asian Test Symp. (ATS)*, pages 151–156, Nov. 2016. doi:10.1109/ATS.2016.9.

[Syn11] Synopsys Inc. SAED 90nm Digital Standard Cell Library, 2011.

[Syn15] Synopsys Inc. SAED 32/28nm Digital Standard Cell Library, 2015.

[Ter83] C. J. Terman. Simulation Tools for Digital LSI Design, PhD. Thesis, Massachusetts Institute of Technology, Sep. 1983.

[TKD+07] J. Tschanz, N. S. Kim, S. Dighe, J. Howard, G. Ruhl, S. Vangal, S. Narendra, Y. Hoskote, H. Wilson, C. Lam, M. Shuman, C. Tokunaga, D. Somasekhar, S. Tang,

D. Finan, T. Karnik, N. Borkar, N. Kurd, and V. De. Adaptive Frequency and Biasing Techniques for Tolerance to Dynamic Temperature-Voltage Variations and Aging. In *Proc. IEEE Int'l Solid-State Circuits Conf. (ISSCC)*, pages 292–604, Feb. 2007. doi:10.1109/ISSCC.2007.373409.

[TPC11] M. Tehranipoor, K. Peng, and K. Chakrabarty. *Test and Diagnosis for Small-Delay Defects*. Springer New York, 2011. ISBN: 978-1-4419-8297-1.

[TTC+15] H. Tang, T. Tai, W. Cheng, B. Benware, and F. Hapke. Diagnosing Timing Related Cell Internal Defects for FinFET Technology. In *Proc. VLSI Design, Automation and Test (VLSI-DAT)*, pages 1–4, Apr. 2015. doi:10.1109/VLSI-DAT.2015.7114547.

[Ulr69] E. G. Ulrich. Exclusive Simulation of Activity in Digital Networks. *Communications of the ACM*, 12(2):102–110, Feb. 1969. doi:10.1145/362848.362870.

[VH99] R. Vollertsen and R. Hijab. Burn-In. In *IEEE Int'l Integrated Reliability Workshop Final Report*, pages 132–133, Oct. 1999. doi:10.1109/IRWS.1999.830574.

[VMG93] H. T. Vierhaus, W. Meyer, and U. Glaser. CMOS Bridges and Resistive Transistor Faults: IDDQ versus Delay Effects. In *Proc. IEEE Int'l Test Conf. (ITC)*, pages 83–91, Oct. 1993. doi:10.1109/TEST.1993.470715.

[VR91] C. Visweswariah and R.A. Rohrer. Piecewise Approximate Circuit Simulation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 10(7):861–870, Jul. 1991. doi:10.1109/43.87597.

[vSAH18] V. M. van Santen, H. Amrouch, and J. Henkel. Reliability Estimations of Large Circuits in Massively-Parallel GPU-SPICE. In *Proc. IEEE 24th Int'l Symp. on On-Line Testing And Robust System Design (IOLTS)*, pages 143–146, Jul. 2018. doi:10.1109/IOLTS.2018.8474096.

[Wad78] R. L. Wadsack. Fault Modeling and Logic Simulation of CMOS and MOS Integrated Circuits. *The Bell System Technical Journal*, 57(5):1449–1474, May 1978. doi:10.1002/j.1538-7305.1978.tb02106.x.

[WEF+85] J.A. Waicukauski, E.B. Eichelberger, D.O. Forlenza, E. Lindbloom, and T. McCarthy. Fault Simulation for Structured VLSI. *VLSI Systems Design*, 6(12):20–32, Dec. 1985.

[WH11] N. H. E. Weste and D. Money Harris. *CMOS VLSI Design – A Circuits and Systems Perspective*. Addison-Wesley, 4th edition, 2011. ISBN: 978-0-321-54774-3.

[WLHW14] H. H. W. Wang, L. Y. Z. Lin, R. H. M. Huang, and C. H. P. Wen. CASTA: CUDA-Accelerated Static Timing Analysis for VLSI Designs. In *Proc. 43rd Int'l Conf. on Parallel Processing (ICPP)*, pages 192–200, Sep. 2014. doi:10.1109/ICPP.2014.28.

[WLRI87] J. A. Waicukauski, E. Lindbloom, B. K. Rosen, and V. S. Iyengar. Transition Fault Simulation. *IEEE Design & Test of Computers*, 4(2):32–38, Apr. 1987. doi:10.1109/MDT.1987.295104.

[WMK+08] X. Wen, K. Miyase, S. Kajihara, H. Furukawa, Y. Yamato, A. Takashima, K. Noda, H. Ito, K. Hatayama, T. Aikyo, and K. K. Saluja. A Capture-Safe Test Generation Scheme for At-Speed Scan Testing. In *Proc. 13th European Test Symp. (ETS)*, pages 55–60, May 2008. doi:10.1109/ETS.2008.13.

[Wun91] H.-J. Wunderlich. *Hochintegrierte Schaltungen: Prüfgerechter Entwurf und Test*. Springer-Verlag Berlin Heidelberg, 1st edition, 1991. ISBN: 978-3-540-53456-3.

[Wun10] H.-J. Wunderlich, editor. *Models in Hardware Testing*, volume 43. Springer-Verlag Heidelberg, 2010. ISBN: 978-90-481-3281-2.

[WW13] M. Wagner and H.-J. Wunderlich. Efficient Variation-Aware Statistical Dynamic Timing Analysis for Delay Test Applications. In *Proc. Conf. on Design, Automation Test in Europe (DATE)*, pages 276–281, Mar. 2013. doi:10.7873/DATE.2013.069.

[WWX06] L.-T. Wang, C.-W. Wu, and Wen X., editors. *VLSI Test Principles and Architectures: Design for Testability*. Morgan Kaufmann, 1st edition, 2006. ISBN: 978-0-12-370597-6.

[WYM+05] X. Wen, Y. Yamashita, S. Morishima, S. Kajihara, L.-T. Wang, K. K. Saluja, and K. Kinoshita. Low-Capture-Power Test Generation for Scan-Based At-Speed Testing. In *Proc. IEEE Int'l Test Conf. (ITC)*, pages 1–10, Paper 39.2, Nov. 2005. doi:10.1109/TEST.2005.1584068.

[WZD10] B. Wang, Y. Zhu, and Y. Deng. Distributed Time, Conservative Parallel Logic Simulation on GPUs. In *Proc. Design Automation Conf. (DAC)*, pages 761–766, Paper 45.2, Jun. 2010. doi:10.1145/1837274.1837467.

[YMO15] Y. Yuan, C. G. Martin, and E. Oruklu. Standard Cell Library Characterization for FinFET Transistors using BSIM-CMG Models. In *Proc. IEEE Int'l Conf. on Electro/Information Technology (EIT)*, pages 494–498, May 2015. doi:10.1109/EIT.2015.7293388.

[YS04] H. Yan and A. D. Singh. On the Effectiveness of Detecting Small Delay Defects in the Slack Interval. In *Proc. of IEEE Int'l Workshop on Current and Defect Based Testing (DBT)*, pages 49–53, Apr. 2004. doi:10.1109/DBT.2004.1408955.

[YWK+11] Y. Yamato, X. Wen, M. A. Kochte, K. Miyase, S. Kajihara, and L. Wang. A Novel Scan Segmentation Design Method for Avoiding Shift Timing Failure in Scan Testing. In *Proc. IEEE Int'l Test Conf. (ITC)*, pages 1–8, Paper 12.1, Sep. 2011. doi:10.1109/TEST.2011.6139162.

[YYHI12] Y. Yamato, T. Yoneda, K. Hatayama, and M. Inoue. A Fast and Accurate Per-Cell Dynamic IR-drop Estimation Method for At-Speed Scan Test Pattern Validation. In *Proc. IEEE Int'l Test Conf. (ITC)*, pages 1–8, Paper 6.2, Nov. 2012. doi:10.1109/TEST.2012.6401549.

[ZBK+13] H. Zhang, L. Bauer, M. A. Kochte, E. Schneider, C. Braun, M. E. Imhof, H.-J. Wunderlich, and J. Henkel. Module Diversification: Fault Tolerance and Aging Mitigation for Runtime Reconfigurable Architectures. In *Proc. IEEE 44th Int'l Test Conf. (ITC)*, pages 1–10, Paper 14.1, Sep. 2013. doi:10.1109/TEST.2013.6651926.

[ZC06] W. Zhao and Y. Cao. New Generation of Predictive Technology Model for Sub-45 nm Early Design Exploration. *IEEE Trans. on Electron Devices*, 53(11):2816–2823, Nov. 2006. doi:10.1109/TED.2006.884077.

[ZHHO04] P. S. Zuchowski, P. A. Habitz, J. D. Hayes, and J. H. Oppold. Process and Environmental Variation Impacts on ASIC Timing. In *Proc. IEEE/ACM Int'l Conf. on Computer Aided Design (ICCAD)*, pages 336–342, Nov. 2004. doi:10.1109/ICCAD.2004.1382597.

[ZKAH13] S. Zhong, S. Khursheed, and B. M. Al-Hashimi. Impact of PVT variation on Delay Test of Resistive Open and Resistive Bridge Defects. In *IEEE Int'l Symp. on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, pages 230–235, Oct. 2013. doi:10.1109/DFT.2013.6653611.

[ZKS+15] H. Zhang, M. A. Kochte, E. Schneider, L. Bauer, H.-J. Wunderlich, and J. Henkel. STRAP: Stress-Aware Placement for Aging Mitigation in Runtime Reconfigurable Architectures. In *Proc. IEEE/ACM 34th Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 38–45, Nov. 2015. doi:10.1109/ICCAD.2015.7372547.

[ZWD11] Y. Zhu, B. Wang, and Y. Deng. Massively Parallel Logic Simulation with GPUs. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 16(3):1–20, Article 29, Jun. 2011. doi:10.1145/1970353.1970362.

[ZWH+18] Y. Zhang, X. Wen, S. Holst, K. Miyase, S. Kajihara, H.-J. Wunderlich, and J. Qian. Clock-Skew-Aware Scan Chain Grouping for Mitigating Shift Timing Failures in Low-Power Scan Testing. In *Proc. IEEE 27th Asian Test Symp. (ATS)*, pages 149–154, 2018. doi:http://dx.doi.org/10.1109/ATS.2018.00037.

Appendix A

Benchmark Circuits

An overview of all the benchmark circuits considered throughout the evaluation is given in Table A.1.

In the table, the first two columns list the name of the benchmark set as well as the name of the circuit design. The size of the circuit in total number of nodes (cells and input/output pins) after synthesis is given in the third column. The number of input and output nodes (primary and pseudo-primary) are listed separately in column 4 and 5. In the sixth column, the maximum *combinational depth* of the circuit is shown, which corresponds to number of levels in the topologically ordered netlist. The last column provides the number of test pattern pairs that have been applied to the benchmark circuits, except where otherwise mentioned.

For the test patterns, n-detect transition delay test pattern pairs were chosen ($n = 10$), which are commonly used for timing-related applications and delay fault simulation that depend on switching activity in the circuit. These pattern sets have been generated for the circuits using a commercial *automated test pattern generation* (ATPG) tool. The transition fault coverages of the obtained test sets are shown in the last column ("*TF-Cov.*"). For all the circuits listed, the transition fault coverage was over 99.6% in average.

Table A.1: Benchmark circuit and test set statistics.

Benchmark[1]	Circuit Name[2]	Nodes			Comb. Depth[6]	Test Set Stimuli	
		Total[3]	Inputs[4]	Outputs[5]		Pattern-Pairs[7]	*TF-Cov.*[8]
ISCAS'89	s38417	19.0k	1664	1742	48	348	100.0%
	s38584	23.1k	1464	1730	70	563	99.80%
ITC'99	b14	9506	277	299	79	904	100.0%
	b17	42.8k	1452	1512	120	2135	99.11%
	b18	125.3k	3357	3364	195	3174	99.91%
	b19	250.2k	6666	6713	203	4651	99.69%
	b20	18.4k	522	512	88	1097	99.89%
	b21	19.3k	522	512	88	1154	99.94%
	b22	27.8k	767	757	88	1190	99.90%
NXP	p35k	48.0k	2912	2229	74	4096	99.92%
	p45k	44.1k	3739	2550	57	2417	99.93%
	p77k	70.5k	3487	3400	466	1979	95.61%
	p81k	138.1k	4029	3952	51	795	99.95%
	p89k	97.5k	4632	4557	85	2460	99.99%
	p100k	96.2k	5902	5829	108	2809	99.85%
	p141k	178.1k	11290	10502	79	2043	99.85%
	p267k	218.4k	17332	16621	55	3181	99.98%
	p330k	286.9k	18010	17468	61	5928	99.93%
	p418k	440.3k	30430	29809	174	3676	99.96%
	p500k	527.0k	30768	30840	179	5012	99.69%
	p533k	676.6k	33373	32610	112	3417	99.87%
	p951k	1.09M	91994	104714	153	7063	99.84%
	p1522k	1.09M	71392	68035	508	17980	99.90%
	p2927k	1.67M	101844	95159	388	22107	99.36%
	p3188k	2.85M	154897	143393	618	26502	99.27%
	p3726k	3.56M	160485	147660	438	15512	98.99%
	p3847k	2.96M	179316	174149	913	31653	99.21%
	p3881k	3.69M	242983	294796	178	12092	99.82%

Appendix B

Result Tables

The following pages contain additional tables with detailed raw simulation data obtained from the performed experiments.

Logic Level Switching Activity Results

Table B.1 investigates the switching activity of the benchmark circuits for the provided 10-detect transition fault test pattern set as summarized in Fig. 12.1.
Column 2 (*"Node Evals."*) shows the total number of node evaluations of the circuit ($\#nodes \times \#stimuli$). Each node evaluation corresponds to the computation of a waveform at a node for a given test stimuli. The resulting simulation cases were categorized based on the transition count tc in the respective signal waveforms obtained. The number of total occurrences in each category have been summarized in column 3 through 6. The categories are divided into glitch-free cases comprised of static constant signals with $tc = 0$ (Col. 3) and robust single transitions with $tc = 1$ (Col. 4), signals with glitches comprising static-0 and static-1 glitches with $tc = 2$ (Col. 5) as well as dynamic glitches with multiple transitions for $tc > 2$ (Col. 6). The percentage of the occurrences with respect to the overall node evaluations are given in parentheses.

Weighted Switching Activity Results

Table B.2 compares the WSA values from both untimed and timing-accurate simulation with the generated 10-detect stimuli test set as summarized in Fig. 12.2.

Table B.1: Characterization of the switching activity from each waveform obtained during timing-accurate simulation at logic level.

Circuit[1]	Node-Evals.[2]	Simulation Cases							
		Glitch-Free				with Glitches			
		Static[3]		Transition[4]		Static[5]		Dynamic[6]	
s38417	6.61M	3.27M	(49.5%)	2.65M	(40.0%)	498.5k	(7.5%)	194.5k	(2.9%)
s38584	12.98M	7.26M	(55.9%)	4.65M	(35.8%)	855.7k	(6.6%)	210.4k	(1.6%)
b14	8.59M	3.96M	(46.0%)	2.29M	(26.6%)	1.28M	(14.9%)	1.07M	(12.5%)
b17	91.33M	59.47M	(65.1%)	20.91M	(22.9%)	6.25M	(6.8%)	4.70M	(5.1%)
b18	397.72M	252.95M	(63.6%)	84.30M	(21.2%)	26.93M	(6.8%)	33.54M	(8.4%)
b19	1.16B	740.70M	(63.6%)	245.37M	(21.1%)	78.22M	(6.7%)	99.54M	(8.6%)
b20	20.17M	8.87M	(44.0%)	5.37M	(26.6%)	3.01M	(14.9%)	2.92M	(14.5%)
b21	22.22M	9.87M	(44.4%)	5.89M	(26.5%)	3.32M	(14.9%)	3.15M	(14.2%)
b22	33.14M	14.72M	(44.4%)	8.76M	(26.4%)	4.90M	(14.8%)	4.76M	(14.4%)
p35k	196.60M	104.69M	(53.3%)	73.61M	(37.4%)	13.96M	(7.1%)	4.34M	(2.2%)
p45k	106.58M	59.88M	(56.2%)	35.01M	(32.8%)	7.81M	(7.3%)	3.88M	(3.6%)
p77k	139.57M	75.92M	(54.4%)	37.99M	(27.2%)	9.19M	(6.6%)	16.48M	(11.8%)
p81k	109.78M	62.84M	(57.2%)	34.66M	(31.6%)	7.30M	(6.6%)	4.98M	(4.5%)
p89k	239.96M	148.44M	(61.9%)	70.76M	(29.5%)	14.44M	(6.0%)	6.32M	(2.6%)
p100k	270.15M	150.11M	(55.6%)	81.18M	(30.0%)	23.80M	(8.8%)	15.05M	(5.6%)
p141k	363.78M	193.49M	(53.2%)	119.19M	(32.8%)	32.18M	(8.8%)	18.93M	(5.2%)
p267k	694.84M	390.26M	(56.2%)	238.76M	(34.4%)	46.90M	(6.7%)	18.93M	(2.7%)
p330k	1.70B	876.46M	(51.5%)	562.45M	(33.1%)	152.17M	(8.9%)	109.68M	(6.4%)
p418k	1.62B	935.28M	(57.8%)	524.80M	(32.4%)	104.88M	(6.5%)	53.51M	(3.3%)
p500k	2.64B	1.45B	(55.0%)	780.44M	(29.5%)	200.70M	(7.6%)	207.50M	(7.9%)
p533k	2.31B	1.15B	(49.7%)	750.15M	(32.4%)	211.67M	(9.2%)	201.47M	(8.7%)
p951k	7.70B	4.21B	(54.7%)	2.67B	(34.7%)	479.95M	(6.2%)	336.07M	(4.4%)
p1522k	19.57B	11.61B	(59.3%)	5.83B	(29.8%)	1.21B	(6.2%)	918.60M	(4.7%)
p2927k	36.97B	19.92B	(53.9%)	11.63B	(31.4%)	2.99B	(8.1%)	2.44B	(6.6%)
p3188k	75.56B	43.54B	(57.6%)	20.48B	(27.1%)	6.00B	(7.9%)	5.53B	(7.3%)
p3726k	55.25B	33.03B	(59.8%)	12.37B	(22.4%)	4.15B	(7.5%)	5.70B	(10.3%)
p3881k	44.64B	24.69B	(55.3%)	13.96B	(31.3%)	3.15B	(7.1%)	2.84B	(6.4%)

The total WSA obtained over all stimuli in the test set are given in column 2 for untimed and column 3 for timing-accurate simulation, respectively. Column 4 ("$\Delta avg.$") shows the average difference of the computed WSA values from timed and untimed simulation per stimuli. The maximum difference observed among all stimuli applied is listed in the last column ("$\Delta max.$").

Resistive Open Transistor Fault Simulation Results

Table B.3 summarizes the detection results of the simulated resistive-open fault simulation shown in Fig. 11.1.

A total of four different fault sizes ΔR_f were investigated throughout the experiment: 10kΩ, 50kΩ, 100kΩ and 1MΩ. The number of faults of the simulation is given in column 2.

Table B.2: Weighted Switching Activity (WSA) from untimed (zero-delay) and timing-accurate simulation at logic level in comparison.

Circuit[1]	Total WSA		Per Stimuli WSA Difference	
	Untimed[2]	Timed[3]	$\Delta avg.$[4]	Δmax[5]
s38417	4.32M	6.17M	+42.5%	+62.9%
s38584	7.96M	10.51M	+31.4%	+50.1%
b14	5.36M	14.50M	+170.3%	+389.8%
b17	42.56M	86.46M	+102.8%	+201.7%
b18	187.86M	672.53M	+257.7%	+393.1%
b19	548.25M	1.86B	+239.1%	+338.8%
b20	12.93M	37.32M	+188.4%	+328.8%
b21	14.17M	40.58M	+186.2%	+341.8%
b22	21.10M	60.91M	+188.4%	+301.3%
p35k	118.23M	167.28M	+41.4%	+68.7%
p45k	61.30M	97.74M	+59.9%	+157.9%
p77k	82.59M	383.59M	+362.8%	+857.5%
p81k	61.22M	103.00M	+68.1%	+98.7%
p89k	131.87M	186.11M	+41.0%	+73.9%
p100k	157.18M	296.76M	+88.8%	+148.0%
p141k	217.97M	379.76M	+74.0%	+104.5%
p267k	403.05M	593.48M	+47.1%	+69.5%
p330k	1.06B	2.01B	+90.6%	+109.8%
p418k	903.24M	1.43B	+58.2%	+74.1%
p500k	1.49B	3.61B	+142.8%	+184.4%
p533k	1.54B	3.34B	+118.0%	+162.8%
p951k	4.40B	7.35B	+66.8%	+83.6%
p1522k	10.86B	21.71B	+100.0%	+149.3%
p2927k	20.80B	45.54B	+119.0%	+143.1%
p3188k	40.94B	92.69B	+126.8%	+181.9%
p3726k	26.16B	145.07B	+454.6%	+482.4%
p3881k	25.72B	53.77B	+109.1%	+135.8%

The total runtime as well as the average simulation runtime per group for processing the four fault sets for all provided test stimuli are presented in column 3 and 4. Column 5 and 6 report the number of cases of *detected* (DT) and *possibly detected* (PD) faults for the fault size of 10kΩ. Similarly, the detection results for the sizes 50kΩ, 100kΩ and 1MΩ are given in column 7 to 8, 9 to 10 and 11 to 12, respectively, The percentage of the detection cases with respect to the number of simulated faults is given below.

Capacitive Fault Simulation Results

The results of the capacitive fault simulation as shown in Fig. 11.3 is summarized in Table B.4 similar to the previous Table B.3. Again a total of four fault sizes ΔC_f were investigated: 50fF, 100fF, 250fF and 1pF.

Table B.3: Resistive open-transistor fault simulation results for max. 1000 cells [SW19a].

Circuit[1]	Faults *collapsed*[2]	Runtime		Fault Size (ΔR_f)							
				10kΩ		50kΩ		100kΩ		1MΩ	
		Total[3]	Group[4]	DT[5]	PD[6]	DT[7]	PD[8]	DT[9]	PD[10]	DT[11]	PD[12]
s38417	2259×4	2:29m	85ms	130 (5.8%)	140 (6.2%)	1434 (63.5%)	230 (10.2%)	1890 (83.7%)	45 (2.0%)	2199 (97.3%)	0 (0.0%)
s38584	619×4	1:48m	97ms	0 (0.0%)	67 (10.8%)	179 (28.9%)	18 (2.9%)	356 (57.5%)	12 (1.9%)	557 (90.0%)	0 (0.0%)
b14	2526×4	13:50m	127ms	0 (0.0%)	20 (0.8%)	571 (22.6%)	97 (3.8%)	1403 (55.5%)	24 (1.0%)	2341 (92.7%)	0 (0.0%)
b17	2543×4	1:14h	930ms	0 (0.0%)	1 (0.0%)	196 (7.7%)	36 (1.4%)	1020 (40.1%)	77 (3.0%)	2312 (90.9%)	0 (0.0%)
b18	2616×4	8:19h	6.80s	0 (0.0%)	0 (0.0%)	437 (16.7%)	50 (1.9%)	972 (37.2%)	50 (1.9%)	2497 (95.5%)	0 (0.0%)
b19	2629×4	11:39h	17.90s	0 (0.0%)	0 (0.0%)	102 (3.9%)	19 (0.7%)	847 (32.2%)	44 (1.7%)	2481 (94.4%)	0 (0.0%)
b20	2502×4	38:28m	298ms	0 (0.0%)	0 (0.0%)	338 (13.5%)	58 (2.3%)	1361 (54.4%)	24 (1.0%)	2381 (95.2%)	0 (0.0%)
b21	2545×4	26:20m	368ms	0 (0.0%)	0 (0.0%)	369 (14.5%)	66 (2.6%)	1386 (54.5%)	26 (1.0%)	2410 (94.7%)	0 (0.0%)
b22	2516×4	34:59m	554ms	0 (0.0%)	0 (0.0%)	362 (14.4%)	67 (2.7%)	1397 (55.5%)	32 (1.3%)	2408 (95.7%)	0 (0.0%)
p35k	2390×4	4:25h	2.30s	0 (0.0%)	46 (1.9%)	970 (40.6%)	134 (5.6%)	1712 (71.6%)	21 (0.9%)	2201 (92.1%)	0 (0.0%)
p45k	2308×4	1:04h	1.15s	6 (0.3%)	22 (1.0%)	1181 (51.2%)	44 (1.9%)	1545 (66.9%)	69 (3.0%)	2191 (94.9%)	0 (0.0%)
p77k	2685×4	6:47h	2.51s	0 (0.0%)	0 (0.0%)	53 (2.0%)	23 (0.9%)	453 (16.9%)	25 (0.9%)	1678 (62.5%)	6 (0.2%)
p81k	2355×4	32:13m	1.60s	2 (0.1%)	8 (0.3%)	1288 (54.7%)	25 (1.1%)	1644 (69.8%)	46 (2.0%)	2336 (99.2%)	0 (0.0%)
p89k	2393×4	3:20h	2.62s	0 (0.0%)	5 (0.2%)	213 (8.9%)	66 (2.8%)	1445 (60.4%)	11 (0.5%)	2287 (95.6%)	0 (0.0%)
p100k	2541×4	2:07h	4.26s	3 (0.1%)	6 (0.2%)	1497 (58.9%)	30 (1.2%)	1780 (70.1%)	85 (3.3%)	2500 (98.4%)	0 (0.0%)
p141k	2313×4	3:20h	4.48s	0 (0.0%)	0 (0.0%)	782 (33.8%)	86 (3.7%)	1403 (60.7%)	72 (3.1%)	2202 (95.2%)	0 (0.0%)
p267k	2275×4	2:55h	7.36s	6 (0.3%)	4 (0.2%)	1216 (53.5%)	115 (5.1%)	1492 (65.6%)	83 (3.6%)	2198 (96.6%)	0 (0.0%)
p330k	2573×4	11:42h	14.88s	9 (0.3%)	8 (0.3%)	1659 (64.5%)	36 (1.4%)	2038 (79.2%)	125 (4.9%)	2560 (99.5%)	0 (0.0%)
p418k	2460×4	16:48h	12.92s	0 (0.0%)	1 (0.0%)	405 (16.5%)	88 (3.6%)	1361 (55.3%)	43 (1.7%)	2378 (96.7%)	0 (0.0%)
p500k	2562×4	24:32h	27.53s	0 (0.0%)	0 (0.0%)	494 (19.3%)	111 (4.3%)	1288 (50.3%)	38 (1.5%)	2340 (91.3%)	0 (0.0%)
p533k	2615×4	10:43h	55.41s	0 (0.0%)	0 (0.0%)	813 (31.1%)	147 (5.6%)	1692 (64.7%)	35 (1.3%)	2548 (97.4%)	0 (0.0%)

Multi-Level Simulation Results

In Table B.5 the impact of ROI activations on the runtime of the multi-level simulation is summarized. Again, column 2 lists the runtime of the serial event-driven time simulation at logic level using the commercial simulation. The columns 3 through 13 then report the runtimes of the presented GPU-accelerated mixed-abstraction simulation for different ROI scenarios. Starting with zero active ROIs (Col. 3), which corresponds to a simulation at logic level, the amount of ROIs is increased gradually over the consecutive

Table B.4: Capacitive fault simulation results for max. 1000 cells [SW19a].

Circuit[1]	Faults *collapsed*[2]	Runtime		Fault Size (ΔC_f)							
				50fF		100fF		250fF		1pF	
		Total[3]	Group[4]	DT[5]	PD[6]	DT[7]	PD[8]	DT[9]	PD[10]	DT[11]	PD[12]
s38417	1000×4	1:24m	103ms	280 (28.0%)	137 (13.7%)	657 (65.7%)	71 (7.1%)	849 (84.9%)	9 (0.9%)	858 (85.8%)	0 (0.0%)
s38584	302×4	1:03m	110ms	69 (22.8%)	41 (13.6%)	155 (51.3%)	13 (4.3%)	198 (65.6%)	2 (0.7%)	245 (81.1%)	0 (0.0%)
b14	1000×4	6:15m	143ms	48 (4.8%)	22 (2.2%)	225 (22.5%)	30 (3.0%)	654 (65.4%)	23 (2.3%)	943 (94.3%)	0 (0.0%)
b17	1000×4	34:01m	1.08s	17 (1.7%)	6 (0.6%)	82 (8.2%)	15 (1.5%)	423 (42.3%)	24 (2.4%)	939 (93.9%)	0 (0.0%)
b18	1000×4	3:42h	8.00s	9 (0.9%)	4 (0.4%)	95 (9.5%)	18 (1.8%)	358 (35.8%)	39 (3.9%)	969 (96.9%)	3 (0.3%)
b20	1000×4	17:37m	338ms	25 (2.5%)	6 (0.6%)	132 (13.2%)	18 (1.8%)	518 (51.8%)	23 (2.3%)	926 (92.6%)	0 (0.0%)
b21	1000×4	12:20m	436ms	33 (3.3%)	4 (0.4%)	148 (14.8%)	18 (1.8%)	519 (51.9%)	36 (3.6%)	944 (94.4%)	0 (0.0%)
b22	1000×4	16:54m	652ms	30 (3.0%)	14 (1.4%)	149 (14.9%)	15 (1.5%)	494 (49.4%)	32 (3.2%)	931 (93.1%)	0 (0.0%)
p35k	1000×4	2:03h	2.53s	93 (9.3%)	29 (2.9%)	420 (42.0%)	52 (5.2%)	842 (84.2%)	13 (1.3%)	890 (89.0%)	0 (0.0%)
p45k	1000×4	35:07m	1.25s	129 (12.9%)	48 (4.8%)	424 (42.4%)	77 (7.7%)	853 (85.3%)	0 (0.0%)	858 (85.8%)	0 (0.0%)
p77k	1000×4	2:41h	2.68s	0 (0.0%)	0 (0.0%)	18 (1.8%)	8 (0.8%)	134 (13.4%)	43 (4.3%)	650 (65.0%)	3 (0.3%)
p81k	1000×4	14:52m	1.78s	266 (26.6%)	31 (3.1%)	548 (54.8%)	96 (9.6%)	918 (91.8%)	0 (0.0%)	918 (91.8%)	0 (0.0%)
p89k	1000×4	1:30h	2.86s	23 (2.3%)	10 (1.0%)	119 (11.9%)	33 (3.3%)	541 (54.1%)	48 (4.8%)	911 (91.1%)	0 (0.0%)
p100k	1000×4	54:53m	4.90s	64 (6.4%)	36 (3.6%)	236 (23.6%)	33 (3.3%)	936 (93.6%)	1 (0.1%)	951 (95.1%)	0 (0.0%)
p141k	1000×4	1:38h	5.07s	46 (4.6%)	16 (1.6%)	223 (22.3%)	40 (4.0%)	757 (75.7%)	37 (3.7%)	880 (88.0%)	0 (0.0%)
p267k	1000×4	1:27h	8.53s	89 (8.9%)	61 (6.1%)	314 (31.4%)	68 (6.8%)	864 (86.4%)	1 (0.1%)	865 (86.5%)	0 (0.0%)
p330k	1000×4	4:43h	16.10s	105 (10.5%)	45 (4.5%)	300 (30.0%)	37 (3.7%)	942 (94.2%)	5 (0.5%)	954 (95.4%)	0 (0.0%)
p418k	1000×4	7:49h	14.31s	10 (1.0%)	2 (0.2%)	91 (9.1%)	26 (2.6%)	481 (48.1%)	34 (3.4%)	941 (94.1%)	0 (0.0%)
p500k	1000×4	12:37h	37.47s	7 (0.7%)	1 (0.1%)	147 (14.7%)	23 (2.3%)	493 (49.3%)	40 (4.0%)	946 (94.6%)	0 (0.0%)
p533k	1000×4	4:54h	1:05m	8 (0.8%)	9 (0.9%)	192 (19.2%)	22 (2.2%)	725 (72.5%)	57 (5.7%)	984 (98.4%)	0 (0.0%)

columns: First in absolute numbers (Col. 4–6) followed by relative amounts with respect to all nodes in the design (Col. 7–12). Eventually, ROIs at all nodes in the design are activated corresponding to a simulation at full switch level (Col. 13). All runtimes shown are given as average over three repeated simulations with randomly activated ROIs in each run [SKW18].

Table B.5: Multi-level simulation runtime for evaluating different amounts of regions of interest (ROIs) active at random nodes from lowest to highest.

Circuit[1]	Comm. Event-Driven[2]	Full Logic-Level[3]	Mixed Abstraction									Full Switch-Level[13]
			Active ROIs (#Nodes)			Active ROIs (% of total Nodes)						
			1[4]	10[5]	100[6]	1%[7]	5%[8]	10%[9]	25%[10]	50%[11]	75%[12]	
s38417	3.11s	20ms	51ms	46ms	48ms	55ms	53ms	56ms	67ms	75ms	82ms	94ms
s38584	5.29s	28ms	66ms	64ms	74ms	55ms	75ms	67ms	85ms	104ms	112ms	138ms
b14	5.30s	20ms	33ms	27ms	42ms	31ms	44ms	53ms	55ms	76ms	89ms	115ms
b17	35.13s	107ms	155ms	161ms	147ms	179ms	207ms	247ms	315ms	423ms	549ms	654ms
b18	6:42m	436ms	568ms	590ms	580ms	650ms	884ms	939ms	1.18s	1.84s	2.46s	3.21s
b19	0:17h	1.14s	1.34s	1.35s	1.36s	1.84s	2.30s	2.67s	3.57s	5.09s	6.80s	8.89s
b20	14.41s	30ms	48ms	62ms	58ms	61ms	83ms	88ms	116ms	151ms	192ms	236ms
b21	16.60s	38ms	56ms	59ms	61ms	70ms	80ms	88ms	112ms	165ms	219ms	257ms
b22	25.39s	47ms	79ms	89ms	77ms	82ms	124ms	125ms	166ms	251ms	300ms	373ms
p35k	2:22m	199ms	247ms	249ms	288ms	272ms	347ms	386ms	509ms	694ms	901ms	1.14s
p45k	49.96s	137ms	212ms	203ms	210ms	236ms	233ms	279ms	339ms	472ms	592ms	711ms
p77k	3:02m	281ms	352ms	339ms	367ms	438ms	629ms	692ms	693ms	967ms	1.31s	1.87s
p81k	2:12m	108ms	234ms	262ms	233ms	338ms	289ms	375ms	487ms	587ms	710ms	880ms
p89k	2:26m	230ms	311ms	328ms	304ms	403ms	435ms	499ms	619ms	852ms	1.15s	1.38s
p100k	3:17m	290ms	379ms	392ms	371ms	458ms	511ms	637ms	866ms	1.11s	1.43s	1.75s
p141k	5:34m	367ms	603ms	604ms	508ms	670ms	815ms	924ms	1.16s	1.57s	2.00s	2.43s
p267k	0:14h	867ms	1.20s	957ms	976ms	1.09s	1.49s	1.68s	2.05s	2.73s	3.50s	4.36s
p330k	0:38h	1.56s	1.89s	1.94s	1.97s	2.16s	2.69s	3.06s	4.36s	5.98s	8.06s	10.18s
p418k	0:29h	1.63s	1.98s	2.13s	1.82s	2.28s	2.67s	2.88s	3.86s	5.37s	7.04s	9.31s
p500k	0:49h	4.05s	4.55s	5.11s	4.09s	5.67s	6.41s	6.01s	8.64s	11.61s	15.21s	18.97s
p533k	1:07h	2.75s	3.79s	4.47s	3.83s	3.66s	5.92s	5.54s	8.19s	10.50s	13.94s	17.58s
p951k	3:00h	16.99s	18.72s	18.54s	15.27s	19.46s	21.13s	20.28s	26.87s	33.85s	40.20s	48.03s
p1522k	8:21h	29.87s	24.79s	31.78s	31.67s	36.46s	43.58s	49.32s	59.23s	1:18m	1:42m	2:01m
p2927k	18:17h	52.79s	54.63s	55.59s	55.06s	1:07m	1:30m	1:44m	1:47m	2:24m	3:08m	3:52m
p3188k	42:02h	2:23m	2:22m	1:55m	2:00m	2:39m	3:05m	3:31m	3:56m	5:23m	7:08m	8:55m
p3726k	39:24h	1:31m	1:27m	1:42m	1:25m	3:29m	6:36m	4:30m	3:33m	4:56m	6:24m	7:31m
p3847k	40:12h	2:40m	3:00m	2:34m	3:31m	3:39m	3:39m	3:49m	4:31m	7:03m	8:06m	0:10h
p3881k	23:53h	1:13m	1:43m	1:43m	1:19m	1:27m	1:38m	2:20m	2:30m	3:07m	3:54m	5:08m

List of Symbols

$G := (V, E)$ circuit graph representation of netlist

I, O inputs/output nodes of a circuit

n, a, b, z circuit nodes

L, L_i level in a circuit graph

ϕ Boolean node function

i, j, k common indices for counting and indexing

D transistor device

D_A^P, D_A^N PMOS (D_A^P) and NMOS (D_A^N) transistor device associated with gate A

R resistor/resistance in Ohms

R^D, R_{on}, R_{off} resistor/resistance of a transistor device; on/off resistance

$R^{A,P}, R^{A,N}$ resistance of PMOS ($R^{A,P}$) or NMOS ($R^{A,N}$) transistor at A

R_u, R_d RRC-cell internal pull-up resistance, pull-down resistance

C_{load} output load capacitance in Farad

t, t_i time, time point

t_{samp} signal sampling time, clock period

TS, TS_n time spec delay description of a node n

ts, ts_i delay description of a node pin i

Δt elapsed time, time offset

δ, δ_f (small) delay fault size; fault parameter

d, d_i^j (gate) delay; i-th delay parameter of the j-th pin of a gate

τ time constant

$\mathcal{P}$ circuit population

P, P_s circuit instance (parameter vector)

p, p_i single circuit instance parameter

e, e_i event, parameter set describing a signal switch; the i-th event in a waveform

K total waveform event capacity

κ waveform base register capacity

s simulation slot

$\mathcal{F}$ fault set

FG fault group

f, f_i fault

loc fault location

S, S_i state (node output)

$\mathcal{T}$ test set

$v, v_o, v(t)$ (logic or switch level) signal value; output voltage, voltage level at time t

$\overline{v}$ stationary voltage

V, V_{th} voltage potential; transistor threshold voltage

VDD power supply voltage potential

GND ground voltage potential

w logic level waveform, (general) waveform

$\tilde{w}$ switch level waveform

$\boldsymbol{w}(t), \tilde{\boldsymbol{w}}(t)$ waveform evaluation function

$x \in [0, 1]$ mixed-abstraction scenario, i.e., logic ($x = 0$) to full switch level ($x = 1$)

$T, T(x)$ simulation runtime; based on scenario x

X unknown or undefined value

Index

Publications of the Author

In the following, all former publications of the author are listed in chronological order and individually for each type of publication.

Book Chapters

[BZK+19] L. Bauer, H. Zhang, M.A. Kochte, E. Schneider, H.-J. Wunderlich, and J. Henkel. *Advances in Hardware Reliability of Reconfigurable Many-core Embedded Systems*, chapter 16, pages 395–416, In B. M. Al-Hashimi and G. V. Merrett, editors, *Many-Core Computing: Hardware and software*. Institution of Engineering and Technology (IET), 2019. ISBN: 978-1-78561-582-5.

Journal Publications

[BBI+13] L. Bauer, C. Braun, M. E. Imhof, M. A. Kochte, E. Schneider, H. Zhang, J. Henkel, and H.-J. Wunderlich. Test Strategies for Reliable Runtime Reconfigurable Architectures. *IEEE Trans. on Computers (TC)*, 62(8):1494–1507, Aug. 2013.

[SKH+17] E. Schneider, M. A. Kochte, S. Holst, X. Wen, and H.-J. Wunderlich. GPU-Accelerated Simulation of Small Delay Faults. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 36(5):829–841, May 2017.

[ZBK+17] H. Zhang, L. Bauer, M. A. Kochte, E. Schneider, H.-J. Wunderlich, and J. Henkel. Aging Resilience and Fault Tolerance in Runtime Reconfigurable Architectures. *IEEE Trans. on Computers (TC)*, 66(6):957–970, Jun. 2017.

[KKL+18] M. Kampmann, M. A. Kochte, C. Liu, E. Schneider, S. Hellebrand, and H.-J. Wunderlich. Built-in Test for Hidden Delay Faults. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, pages 1–13, Aug. 2018. doi:10.1109/TCAD.2018.2864255.

[SW19a] E. Schneider and H.-J. Wunderlich. SWIFT: Switch Level Fault Simulation on GPUs. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 38(1):122–135, Jan. 2019.

[SW19b] E. Schneider and H.-J. Wunderlich. Multi-level timing and fault simulation on GPUs. *Integration*, 64:78–91, Jan. 2019.

Conference Proceedings

[HSW12] S. Holst, E. Schneider, and H.-J. Wunderlich. Scan Test Power Simulation on GPGPUs. In *Proc. IEEE 21st Asian Test Symp. (ATS)*, pages 155–160, Nov. 2012.

[ZBK+13] H. Zhang, L. Bauer, M. A. Kochte, E. Schneider, C. Braun, M. E. Imhof, H.-J. Wunderlich, and J. Henkel. Module Diversification: Fault Tolerance and Aging Mitigation for Runtime Reconfigurable Architectures. In *Proc. IEEE 44th Int'l Test Conf. (ITC)*, pages 1–10, Paper 14.1, Sep. 2013.

[SPI+14] M. Sauer, I. Polian, M. E. Imhof, A. Mumtaz, E. Schneider, A. Czutro, H.-J. Wunderlich, and B. Becker. Variation-Aware Deterministic ATPG. In *Proc. IEEE 19th European Test Symp. (ETS)*, pages 1–6, May 2014. **Best Paper Award**.

[SHWW14] E. Schneider, S. Holst, X. Wen, and H.-J. Wunderlich. Data-Parallel Simulation for Fast and Accurate Timing Validation of CMOS Circuits. In *Proc. IEEE/ACM 33rd Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 17–23, Nov. 2014.

[SHK+15] E. Schneider, S. Holst, M. A. Kochte, X. Wen, and H.-J. Wunderlich. GPU-Accelerated Small Delay Fault Simulation. In *Proc. ACM/IEEE Conf. on Design, Automation Test in Europe (DATE)*, pages 1174–1179, Mar. 2015. **Nominated for Best Paper Award**.

[ZKS+15] H. Zhang, M. A. Kochte, E. Schneider, L. Bauer, H.-J. Wunderlich, and J. Henkel. STRAP: Stress-Aware Placement for Aging Mitigation in Runtime Reconfigurable Architectures. In *Proc. IEEE/ACM 34th Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 38–45, Nov. 2015.

[KKS+15] M. Kampmann, M. A. Kochte, E. Schneider, T. Indlekofer, S. Hellebrand, and H.-J. Wunderlich. Optimized Selection of Frequencies for Faster-Than-at-Speed Test. In *Proc. IEEE 24th Asian Test Symp. (ATS)*, pages 109–114, Nov. 2015.

[AWH+15] K. Asada, X. Wen, S. Holst, K. Miyase, S. Kajihara, M. A. Kochte, E. Schneider, H.-J. Wunderlich, and J. Qian. Logic/Clock-Path-Aware At-Speed Scan Test Generation for

Avoiding False Capture Failures and Reducing Clock Stretch. In *Proc. IEEE 24th Asian Test Symp. (ATS)*, pages 103–108, Nov. 2015. **ATS 2015 Best Paper Award**.

[SW16] E. Schneider and H.-J. Wunderlich. High-Throughput Transistor-Level Fault Simulation on GPUs. In *Proc. IEEE 25th Asian Test Symp. (ATS)*, pages 151–156, Nov. 2016.

[HSW+16] S. Holst, E. Schneider, X. Wen, S. Kajihara, Y. Yamato, H.-J Wunderlich, and M. A. Kochte. Timing-Accurate Estimation of IR-Drop Impact on Logic- and Clock-Paths During At-Speed Scan Test. In *Proc. IEEE 25th Asian Test Symp. (ATS)*, pages 19–24, Nov. 2016.

[HSK+17] S. Holst, E. Schneider, K. Kawagoe, M. A. Kochte, K. Miyase, H.-J. Wunderlich, S. Kajihara, and X. Wen. Analysis and Mitigation of IR-Drop Induced Scan Shift-Errors. In *Proc. IEEE 48th Int'l Test Conf. (ITC)*, pages 1–7, Paper 3.4, Oct. 2017. **Distinguished Paper**.

[SKW18] E. Schneider, M. A. Kochte, and H.-J. Wunderlich. Multi-Level Timing Simulation on GPUs. In *Proc. ACM/IEEE 23rd Asia and South Pacific Design Automation Conf. (ASP-DAC)*, pages 470–475, Jan. 2018.

[LSK+18] C. Liu, E. Schneider, M. Kampmann, S. Hellebrand, and H.-J. Wunderlich. Extending Aging Monitors for Early Life and Wear-Out Failure Prevention. In *Proc. IEEE 27th Asian Test Symp. (ATS)*, pages 92–97, Oct. 2018.

[HSK+19] S. Holst, E. Schneider, M. A. Kochte, X. Wen, and H.-J. Wunderlich. Variation-Aware Small Delay Fault Diagnosis on Compacted Failure Data. In *Proc. IEEE 50th Int'l Test Conf. (ITC) [to appear]*, Nov. 2019.

Workshop Contributions

[SKW15] E. Schneider, M. A. Kochte, and H.-J. Wunderlich. Hochbeschleunigte Simulation von Verzögerungsfehlern unter Prozessvariationen. In *Proc. GI/GMM/ITG 27th Workshop Testmethoden und Zuverlässigkeit von Schaltungen und Systemen (TuZ)*, pages 32–34, Mar. 2015.

Other Publications

[PGSU14] E. Plödereder, L. Grunske, E. Schneider, and D. Ull, editors. *44. Jahrestagung der Gesellschaft für Informatik, Informatik 2014, Big Data - Komplexität meistern, 22.-26.*

September 2014 in Stuttgart, Deutschland, volume P-232 of *Lecture Notes in Informatics (LNI)*. GI, 2014.

Declaration

All the work contained within this thesis,
except where otherwise acknowledged, was
solely the effort of the author.
At no stage was any collaboration entered into
with any other party.

Eric Schneider