Institute of Computer Architecture and Computer Engineering
University of Stuttgart
Pfaffenwaldring 47
D–70569 Stuttgart

Studienarbeit Nr. 2306

CUDA-accelerated Delay Fault Simulation

Eric Schneider

Course of Study: Computer Science

Examiner: Prof. Dr. Hans-Joachim Wunderlich

Supervisor: Dipl.-Inf. Stefan Holst

Commenced: November 1, 2010

Completed: May 3, 2011

CR-Classification: B.7.1, B.8.1, C.1.4, C.4, D.1.3, J.6

Contents

1 Introduction 7
1.1 Fundamental Definitions . . . 7
1.2 Common Delay Fault Models . . . 9
1.3 The NVIDIA CUDA Architecture . . . 13
1.4 Parallel Logic and Fault Simulation . . . 14
1.5 The `wave` Time Simulator . . . 15
1.6 POINTER Pattern Analysis . . . 15

2 Gate Delay Fault-List Generation 17
2.1 Determining Defect Size Boundaries . . . 17
2.2 Computing Fault Sets . . . 19
2.2.1 Classifying Faults . . . 20
2.2.2 Computing Path Slacks . . . 21
2.3 Complexity . . . 24
2.3.1 Modification A: Transition-Fault Model . . . 25
2.3.2 Modification B: Defect Range Quantization . . . 26
2.3.3 Fault Equivalences . . . 27
2.4 Relation between Arbitrary Defect Sizes and SmallDelays . . . 28

3 Delay Fault Simulator 29
3.1 Test Pattern Generation . . . 29
3.2 Fault List Generation . . . 30
3.3 Fault Simulation . . . 31
3.3.1 Initialization . . . 31
3.3.2 Tasks . . . 31
3.3.3 Remarks on Circuit Levelization . . . 34
3.3.4 Parallelism . . . 35
3.4 Fault Injection . . . 35
3.5 Response Evaluation . . . 35

4 Diagnosis 37
4.1 Setup . . . 37
4.2 Delay Fault Diagnosis using POINTER . . . 38
4.2.1 Fine-grained Resimulation . . . 39

4.2.2 Pattern Analysis 40
4.3 Suspect Ranking 41

5 Experimental Results 43
5.1 Fault Set Generation 43
5.1.1 Counting Delay Faults 43
5.1.2 Evaluation 45
5.2 Fault Simulation 46
5.3 Diagnosis 49
5.3.1 GSI Ranking 49
5.3.2 SIG Ranking 52
5.3.3 Reasoning with Evidences and Representatives 54
5.4 Summary 55

6 Conclusion 61

A ADAMA `fsample` 63

B ADAMA `wave/diagnose dfcuda` 65

C The Internals of `wave` 67

List of Symbols 73

Bibliography 75

Preface

In today's nano-scale technology, small variations (e.g. in gate-oxide thickness) during design manufacturing are inevitable. In some chips, these variations may manifest as defects, that alter the behaviour of the circuit. In some cases, these defects do not change the logic structure itself, but merely affect the timing of certain sites by increasing the signal propagation time. These so called *delay defects* go undetected by standard stuck-at tests in *high-speed integrated circuits*, as they require at-speed test conditions in order to make the fault visible.

The main goal of this thesis was to extend the **Adaptive Diagnosis of Arbitrary Manifold Artifacts (ADAMA)** framework in order to support simulation of small delay faults and to apply the **Partially Overlapping Impact counTER (POINTER)** pattern analysis algorithm to diagnose such defects.

Since fault simulation is inherently parallelizable, the simulator will make use of **NVIDIA's Compute Unified Device Architecture (CUDA)** by utilizing **General Purpose Graphics Processing Units (GPGPU)** in order to exploit parallelism and to improve the overall simulation performance.

Contributions

The main contributions of this thesis are:

- Introduction and implementation of a delay fault model.
- Extension of a simulation engine in order to perform delay fault simulation on CUDA devices.
- Integration of the delay fault simulator into an existing diagnosis framework.
- Evaluation and discussion of the diagnosis results.

Organization

This document is organized as follows:

Section 1 – Introduction: The first section serves as a brief introduction for definitions and background. It offers some useful information about CUDA, parallel fault-simulation, common delay fault models among others in order to recall the basic ideas and the necessary steps, that have to be taken.

Section 2 – Gate Delay Fault-List Generation: Introduces implementation of a path-based method for generation of reasonable gate delay defect sizes in combinatorial circuits with unit and nominal propagation delay.

Section 3 – Delay Fault Simulator: This section describes the implementation of the delay fault simulator.

Section 4 – Diagnosis: Explains how the implemented delay fault simulator was integrated into the diagnosis framework.

Section 5 – Experimental Results: Here, some results of a various range of experiments are shown and discussed.

Finally, everything will be summarized in conclusion.

Acknowledgements

I would like to thank my supervisor Stefan Holst for his encouragement and the fruitful discussions throughout this work.

1 Introduction

1.1 Fundamental Definitions

Netlist

A **netlist** serves as a description of circuits. For combinatorial logic, it can be viewed as a directed acyclic graph $G(V, E)$, where each node $z \in V$ represents a standard boolean logic gate. Each edge $(x, y) \in E$ corresponds to a direct connection between two gates, where the output of x is connected to an input of y. A node can be evaluated by applying the values of its direct predecessors to its implemented function.

For a proper evaluation of a gate, it must be ensured that all its predecessor values have already been computed in an earlier step, or else the computation has to be stalled. This can be achieved by *levelizing* the graph a priori. Thus, the goal of circuit **levelization** is to generate a schedule of partitions of the node set, such that the nodes can be processed in topological order and take these data dependencies into account. Nodes of the same level are independent and can be evaluated concurrently. Some typical levelization schemes are **as-soon-as-possible (ASAP)** and **as-late-as-possible (ALAP)**.

Defects, Faults and Fault Models

A **defect** is a distortion of the circuit's physical structure and might be caused by impurities or variations during the manufacturing process. In general, the term **fault** will be used to describe the deviations of the specified circuit behaviour due to particular defects. It serves as an abstraction and usually represents a class of various defects that might appear in different size or variation, but which would all cause the same faulty behaviour.

The faults are further specified according to a certain **fault model**, that constrains the possible defect scenarios to a *finite* set: e.g. the well known *stuck-at* model with a *stuck-at-0*, *stuck-at-1* and, of course, the *fault-free* case for each fault-site. It is also distinguished between *single-fault models*, which allow only one fault at a time, and *multiple-fault models*, that consider multiple fault sites at once and have a higher complexity. Thus, the choice of the fault model will determine the precision of the defect modeling as well as the overall modeling complexity.

A comprehensive collection of many proposed fault models is listed in [BA02].

Fault Simulation

The purpose of **fault simulation** is the observation of a circuit's behaviour in the presence of faults. This will serve as a facility to test whether a certain fault is detected by some input patterns or detectable at all. An example application would be the measuring of the *fault coverage* and the effectiveness of *Automatic Test Pattern Generator (ATPG)* test-sets, or the generation of fault dictionaries for diagnosis purposes.

Figure 1.1 shows an schematic setup of an example fault simulator, where test-patterns are applied to the **primary inputs (PI)** of a specification model G and a *modified* version G'. The latter was modified in order to match its behaviour according to a certain fault. This modification step is called **fault injection**. After a simulation of the circuit, the responses of both devices' **primary outputs (PO)** are compared. Here, the good simulation responses are used as reference values and help searching for any errors, that might have been caused by the injected fault.

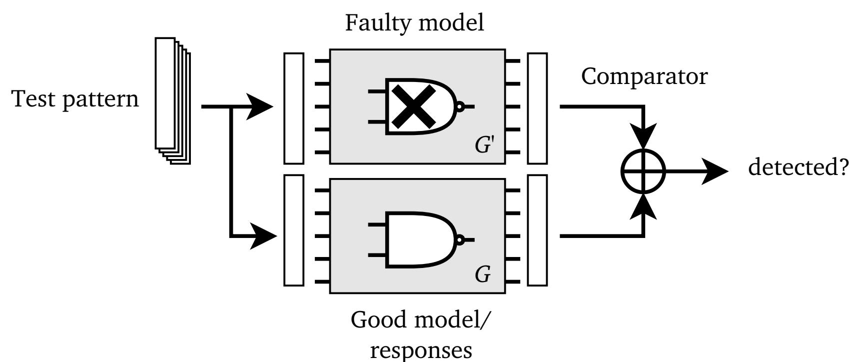

Figure 1.1: Schematics of a fault simulation. – The responses of the faulty device G' are compared with the responses of a specification model G.

Faults can only be observed for some input pattern if the following two conditions are met:

- The patterns have to activate the faults at the corresponding fault sites (**fault excitation**).
- The fault effect has to be propagated to the POs to become visible (**fault propagation**).

ATPG patterns can be used to enhance fault activation and detection, whereas fault detection with random generated patterns will generally show a bad performance, as many activation- and propagation-paths may be blocked by controlling off-path signals.

1.2 Common Delay Fault Models

Delay fault models are mainly found in literature about testing or simulation. According to [KC98] and [MA98] there are five popular delay fault models: The *transition fault model* [WLRI87], the *gate delay fault model* [CIR87], the *path delay fault model* [Smi85], the *line delay fault model* [MAJP00] and the *segment delay fault model* [HPA96]. Each has its own characteristics regarding defects sizes and defect distribution. In the following, these models will be briefly introduced and discussed.

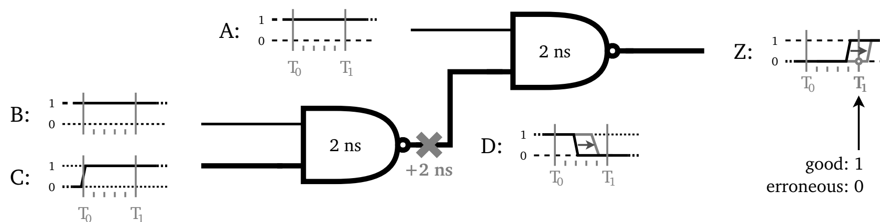

Figure 1.2: Example of a delay defect at a line. – Although the logic structure of the circuit has not been changed, it does eventually produce an erroneous response when the output is sampled. Input and output signals are latched at T_0 and T_1.

Transition Fault Model

The transition fault model [WLRI87] considers two types of transition faults per fault site. The **slow-to-rise (STR)** fault for defects that causes an *infinite* delay in a rising transition and the **slow-to-fall (STF)** for falling transitions respectively. Only a single gate input or output may be affected at the same time.

The **transition fault** itself is independent of the actual circuit timing and particular defect sizes. They are also referred to as *gross delay faults*, since the effects are visible at any sensitized POs of its output cone. For simplicity, a transition fault can be modeled as a *temporary* or *conditional stuck-at* [WLRI87] by triggering the corresponding stuck-at faults for the transitions. The resulting behaviour of a fault-site can then be expressed like shown in Table 1.1 [KC98].

Since all defects are assumed to be sufficiently large, the transition fault model is neither very accurate nor realistic: Delay defects can also be of smaller size and might violate only a small subset of paths in their sensitized output cones. However, the transition fault model is not precise enough to model this kind of small delays.

Transition	behaviour	
	STR	STF
$0 \to 0$	–	–
$0 \to 1$	$0 \to 0$	–
$0 \to x$	$0 \to 0$	–
$1 \to 0$	–	$1 \to 1$
$1 \to 1$	–	–
$1 \to x$	–	$1 \to 1$
$x \to 0$	–	$x \to x$
$x \to 1$	$x \to x$	–
$x \to x$	–	–

Table 1.1: Behaviour of a fault-site in the *transition fault model* using a three-valued algebra („–“ represents no change).

Gate Delay Fault Model

In contrast to the transition fault model, the gate delay fault model by Carter *et al.* [CIR87] does consider circuit timing information. In fact, it assumes that the delay times of each gate in the circuit is known by a *bounded interval*.

A gate suffering from a defect causes a **gate delay fault**, if a defect increases the signal propagation time of a rising or a falling transition through the gate. The size of the defect is merely the actual increase in the propagation time at the fault-site. A fault is detected, if the timing constraints of any sensitized path is violated. So, for each test only those faults are visible, whose associated defect size is greater than some threshold. The threshold depends on the minimum positive slack of all sensitized paths (typically caused by the longest) through the output-cone of the fault-site [PR88].

It has been proposed, that the timing of signal propagations is computed by using simplified waveforms to reduce computation complexity. All events at a gate are merged into a single wave, which simply denotes the **earliest arrival (EA)** and **latest stabilization (LS)** time to summarize the qualitative behaviour. An example is shown in Figure 1.3 [IRW90].

Like in the transition fault model, at most of one fault at the same time is assumed.

Path Delay Fault Model

While the location of transition and gate delay faults are restricted to a single site, the path delay fault model [Smi85] was proposed to model the cumulative delay effect along complete

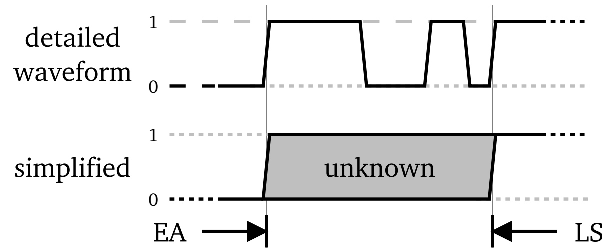

Figure 1.3: Summarized waveform for a rising transition.

paths from PIs to POs. Due to statistical variations during the manufacturing process, many gates can be affected simultaneously by smaller delay defects, each of which for itself might go undetected if tested separately. Yet, all these small defects may accumulate to create a more severe fault along a certain path and cause the **circuit under test (CUT)** to fail.

In this model, a path p in a CUT from PI to PO is said to have a **path delay fault**, if the signal propagation time along p exceeds the desired clock rate [Smi85]. The clock rate may be exceeded because of a single gate delay or a distributed defect. For each physical path, a *rising* and a *falling path* is introduced, where the rising (falling) path is tested with a rising (falling) transition at the input.

The path delay fault model allows detection of any delay fault, that is caused by defects of arbitrary size and distribution throughout the CUT. A major disadvantage of the model is its complexity. Since the number depends on the actual number of paths, the complexity may rise exponentially with the circuit size (worst case). An enumeration of all possible paths might become impractical and it is not guaranteed that all paths are easily testable or even sensitizable at all (i.e. false paths).

Line Delay Fault Model

The line delay fault model by Majhi *et al.* [MA98, MAJP00] combines both transition and path delay fault model. It allows testing for delay defects at a *line* by propagating a transition along the longest sensitizable path, which traverses the fault site. Here, not only transition faults, but also the cumulative delay effects caused by smaller and distributed defects on the sensitized path might be detected.

Each line delay fault has to be tested with a rising and a falling transition at its corresponding fault site, so the total number of faults to be tested is twice the number of lines in the circuit.

Segment Delay Fault Model

In the segment delay fault model proposed by Heragu *et al.* [HPA96] defects are distributed on a chain of gates, whose path length is less than or equal to a specified value l. This chain is called a segment. It is assumed that a segment delay fault causes delay faults on all sensitized paths from PI to PO, which contain the faulty segment as a sub-path. The idea behind the segmentation is, for example, that a defect at a line may affect surrounding neighbours. Analogous to other delay fault models, there is a rising and a falling delay fault for each segment.

The choice of the length l offers a trade-off between the benefits and the drawbacks of both transition and path delay fault models [MA98, KC98], e.g. while avoiding the exponential path count, the defects can still be modeled in a distributed and thus more *realistic* fashion.

Summary

In this section, the most common delay faults found in literature [KC98, BA02, WST08] were introduced. The key differences are the detectable defect sizes, their scope (whether local or globally distributed) and the number of fault-sites, that have to be considered. Table 1.2 summarizes the major advantages and limitations of the presented models.

Obviously, the path delay fault model, as well as the line- and segment delay fault model are not appropriate for logic level delay fault simulation, since fault injection into the circuit cannot be directly applied. Their purpose is also mainly focused on testing and test generation for CUTs in order to validate timings at paths, without making any assumptions about the actual number of physical delay defects, their size or location in particular. Thus, circuit simulation would become complex.

The remainder seems to be more appropriate: Both transition- and gate delay fault model have $\mathcal{O}(N)$ fault-sites and allow relatively simple fault-injection, because each fault is already associated with a single location (gate, wire) in the circuit. However, only the gate delay fault model considers actual timing information. It is also worth to mention that transition faults are a subset of the gate delay model, because each transition fault might correspond to a gate delay defect with, for example, infinite size.

Therefore, focus will be directed towards the **gate delay fault model** type, allowing a precise evaluation of timing behaviour. In order to simulate such small delays, reasonable defect sizes have to be determined.

Fault model	advantages	disadvantages
Transition [WLRI87]	simple	only gross defects (local)
Gate [CIR87]	timing considered, models local defects of any size	no distributed defects
Path [Smi85]	models globally distributed delay defects	complexity problem, not useful for gate simulation
Line [MAJP00]	models gross and globally distributed delay defects	not all paths considered, not useful for gate simulation
Segment [HPA96]	models gross and distributed delay effects (local and global, depending on segment length)	not useful for gate simulation

Table 1.2: Comparison of the presented delay fault models.

1.3 The NVIDIA CUDA Architecture

NVIDIA's **Compute Unified Device Architecture (CUDA)** [NVI10] is an architecture for parallel computing on highly threaded **general purpose graphics processing unit (GPGPU)** devices, which has been introduced in 2007. The architecture itself consists of both the underlying hardware (CUDA-capable GPU devices) as well as the software required to utilize the GPUs (driver, API, runtime). Whereas common CPUs have been optimized for single thread performance, GPGPUs are specialized for parallel data processing purposes with many threads and high computational throughput. A CUDA GPGPU consists of multiple **Streaming Multiprocessors (SM)**, which are able to create, schedule and execute threads. Each SM executes threads concurrently in groups of size 32 (16), called **warps** (*half-warps*). All threads contained within a warp will process the same instruction of the execution path, unless they diverge. In that case, all branches will be executed serially until the threads converge again.

CUDA programs are generally separable into two parts: The host code running on the CPU and the **kernel** code, which is called by the host to be executed on the CUDA device. A typical host program allocates data on the device, calls the kernel and fetches the results after computation. This is also called '*heterogeneous computing*'. A host program can make use of different kernels. However, the kernels have to be processed in a sequential order, since only one kernel can be active at a time. When a kernel is launched, it spawns a set of threads on the device that

will all execute the implemented kernel function. The set is arranged in a two-dimensional grid of three-dimensional blocks of threads. All dimensions have to be provided at the kernel function call and are denoted as parameters in '**<<< ... >>>**'–brackets. These are typically chosen depending on the type and size of the given problem, which is split into subproblems accordingly in order to exploit parallelism.

Besides some local registers, all threads have access to a large, but slow **global device memory** (typically 1–4 GB, depending on the device). Furthermore, a small *read-only* **constant memory** (64 KB) can be used. If a thread accesses the constant memory, the data is distributed to all other threads in its half-warp and is also cached for quick access, thereby reducing global memory accesses. Also, a small low-latency *shared memory* is available, that allows cooperation of threads within a block.

The utilization of highly parallel GPUs for general purpose issues has become more and more popular, since it offered the opportunity to deal with highly complex computational problems [OHL$^+$08]. Yet, a major difficulty remains for the programmer to carefully distribute thread resources and optimize memory access management as this has a large impact on the overall performance.

1.4 Parallel Logic and Fault Simulation

Logic-level and fault simulators generally have to deal with large designs, that consist of many gates and are also exposed to many patterns. In addition to the circuit gates and patterns, fault simulation does add another dimension of complexity by the number of simulated faults. Hence, there are many opportunities to exploit parallelism.

With the introduction of CUDA, first approaches of parallelized fault simulation with GPGPUs have been reported [GK08]. Here, a stuck-at fault simulator has been implemented, that uses both fault- and pattern-parallelism. It performs level-wise simulation as well as fault-detection by separate kernels on the GPGPU.

A GPU-accelerated logic simulator has been proposed in [CDB09b]. This approach uses a preprocessing step, where cone-partitioning is used to extract the input-cones of all netlist outputs. These cones are merged into *clusters*, which run on separate thread-blocks. The clusters are further being balanced to allow concurrent evaluation of independent gates within through higher utilization of the resources. According to the report, nearly 20% of the circuit's gates have been duplicated during the preprocessing step, due to cone overlaps.

Another GPGPU gate-level simulator is introduced in [CDB09a]. This implementation is event-driven and cuts a circuit into layers (pools) of macro-gate blocks, whose execution is triggered by sensitivity lists. Each active macro-gate of a layer is assigned to a thread-block, which evaluates multiple independent gates at once. Similar to the previous approach, the macro segmentation is likely to cause gate duplication.

In [GK09], a method is presented to compute circuit delays and arrival times via a parallelized Monte-Carlo-based statistical static timing analysis on the GPU. Here, pseudo random-number generation and evaluation is performed concurrently for many samples to compute the delay distribution of single gates in parallel.

1.5 The `wave` Time Simulator

In this thesis, the core of the implemented delay fault simulator consists of a piece of CUDA/C code, called `wave`, that represents a gate-level time simulator for execution on CUDA capable GPGPU devices, which has formerly been used to determine the *switching activity* in circuits. It was written by Stefan Holst and has been provided for use with this thesis.

`Wave` stores intermediate signal values as **waveforms** and captures the complete timing history of a signal rather than storing a single discrete value. Their representations allow for an efficient and precise evaluation of the signal states. Since the computation of these signal timings causes a lot of arithmetic operations, CUDA is used in order to speed-up the simulation process: The simulator uses many-threading to evaluate large numbers of gates for multiple patterns at once, thus exploiting structural- and data-parallelism.

Further information about the mechanics of `wave` is found in Section C of the appendix.

1.6 POINTER Pattern Analysis

In logic diagnosis one tries to identify the location of defects in a failing chip. This can be of particular interest, as the results may uncover potential design flaws or regions with high probability of failing during manufacturing or aging processes. Knowing these flaws might, as well, help to correct or harden the circuit for future revisions. Two major diagnosis paradigms have arisen [WWW06]. A *cause-effect* paradigm, which is usually a fault dictionary-based approach. The fault dictionary is generated for the circuit w.r.t. a certain *known* fault model. The real culprit is identified by looking up the syndromes in the dictionary. If multiple fault locations have to be considered, the fault dictionary will grow substantially large.

Another approach is the **effect-cause** paradigm. Here, the syndromes of the **device under diagnosis (DUD)** are examined in order to confine the set of possible suspects and to find reasonable fault candidates. One particular effect-cause approach for adaptive response pattern analysis is the **Partially Overlapping Impact couNTER (POINTER)** algorithm [HW09]. It has been used with the generalized **conditional line flip (CLF)** calculus [Wun09] to show its effectiveness and fault model independence.

POINTER uses evidences of a fault in order to rank and constrain suspects. An **evidence** of a suspect f and a pattern π is defined as a tuple $e(f,\pi) = (\sigma_\pi, \iota_\pi, \tau_\pi, \gamma_\pi)$, where the components

are computed by comparing the syndromes of the DUD with the **fault machine (FM)** simulating fault f:

- σ_π is the number of faulty outputs detected by both FM and DUD simulation,
- ι_π is the number of faulty outputs, which are mispredicted by fault f,
- τ_π is the number of faulty outputs of the DUD, which cannot be explained by f, and
- γ_π is defined as $\min(\{\sigma_\pi, \iota_\pi\})$.

For a whole pattern set $\Pi = \pi_0, ..., \pi_k$, the individual evidence components of the patterns are further being summed up, forming the complete evidence of the fault:

$$e(f, \Pi) = (\sum \sigma_{\pi_i}, \sum \iota_{\pi_i}, \sum \tau_{\pi_i}, \sum \gamma_{\pi_i}).$$

Now, the resulting evidence collection is ordered to create a ranked list of suspect faults. The proposed ordering of the original paper sorts first in increasing order of γ to move single and conditional stuck-ats in front. Then, evidences with equal γ are ordered by decreasing σ to highlight the suspects by the most explaining bits. Minimizing ι helps to weed out suspects with too many mispredictions.

The relation between the ranking of two faults f_1 and f_2 and their corresponding evidences after **Gamma-Sigma-Iota (GSI)** ordering is described as follows:

1. $\gamma_\Pi^1 > \gamma_\Pi^2 \Rightarrow \text{rank}(f_1) > \text{rank}(f_2)$
2. $\sigma_\Pi^1 > \sigma_\Pi^2 \Rightarrow \text{rank}(f_1) < \text{rank}(f_2)$
3. $\iota_\Pi^1 > \iota_\Pi^2 \Rightarrow \text{rank}(f_1) < \text{rank}(f_2)$.

The lower the rank of a suspect, the better the result, and the more likely it will be a *perfect* candidate, that can explain the defect behaviour.

2 Gate Delay Fault-List Generation

When trying to model arbitrary gate delay faults for use with logic simulation, one has to face an additional dimension of complexity as compared to the common stuck-at model: The **defect size** (in the following denoted as δ_f) related to a fault f, or the actual *increase in delay* at a certain fault site. This defect size is — like time itself — in general a continuous and also unrestricted parameter.

The following section will provide a method to compute relevant defect sizes for circuit netlists with *unit* and *nominal gate delay*. First, it is shown how to restrict the defect sizes by a lower and an upper bound. A path-based fault generation will be defined afterwards in order to obtain all defects, that might cause a distinct fault behaviour.

To reduce the initial fault count and model complexity, the location of any fault has been constrained to single gates. Also, in contrast to other delay fault models ([WLRI87], [Smi85], [HPA96]), it is **not** distinguished between rising and falling transition faults during simulation for simplicity reasons, but the method will allow for extensions.

2.1 Determining Defect Size Boundaries

To specify the defect size boundaries for a given node, the **minimum (maximum) slack** at a certain node n has to be defined first. Let P_n denote the set of all paths p from PI to PO through node n and let $\delta(n)$ be its nominal propagation delay.

> **Assumption 1.** Defects only increase the propagation delay of gates and at most one is affected at a time.

Definition 4.1. The *slack* $\Delta^T(p)$ of a path p in a circuit G is defined as the difference $\Delta^T(p) = T - t_p$, where T is the sampling time of G and t_p is the signal propagation time along p. The propagation time $t_p = \sum_{\forall n \in p} \delta(n)$ is the cumulative delay of all nodes on p.

Thus, the slack of a path p can be described as the amount of time-units a signal propagated along p will arrive earlier at the output than required. By definition, all paths with a negative slack will violate the timing constraints.

Example: Consider a circuit G, which is sampled at $T = 8$ ns, a path p with $t_p = 6$ ns and a path q with $t_q = 10$ ns propagation time. The slacks will be $\Delta^T(p) = 8 - 6 = 2$ ns for path p and $\Delta^T(q) = 8 - 10 = -2$ ns for q respectively.

Definition 4.2. The *minimum* slack $\Delta_{min}(n)$ of node n in circuit G is defined as $\min\limits_{p \in P_n}\{\Delta^T(p)\}$. Analogous, the *maximum* slack $\Delta_{max}(n)$ of n is defined as $\max\limits_{p \in P_n}\{\Delta^T(p)\}$.

A delay defect of positive size $\delta_f > 0$, injected at a node n, will change the delay of n to $\delta'(n) = \delta(n) + \delta_f$ by increasing t_p for each $p \in P_n$ and causing all slacks $\Delta^T(p)$ to decrease by δ_f (see Def. 1+2). The resulting slack for each path p will be $\tilde{\Delta}^T(p) = \Delta^T(p) - \delta_f$. If the minimum slack of n is positive, then all paths $p \in P_n$ meet the timing constraints, since

$$\forall p \in P_n : (\Delta^T(p) \geq \Delta_{min}(n)) \wedge (\Delta_{min}(n) \geq 0) \Rightarrow \Delta^T(p) \geq 0.$$

However, if the minimum slack at n is below zero (that is $\Delta_{min}(n) < 0$), the circuit will fail for any path $p \in P_n$ with $\Delta^T(p) < 0$ that is statically sensitizable, such that the fault effects are propagated over p to the POs for some delay test pattern.

Definition 4.3. A circuit G is *fault-free*, if none of the paths in G violates the timing constraints, or equivalently:

$$\forall n \in G, \forall p \in P_n : \Delta^T(p) \geq 0.$$

Lower Bound

Using these definitions, one can now prove the following lemma in order to specify a lower bound for detectable defect sizes at a certain node n. This lower bound is given by $\Delta_{min}(n)$ and any defect of smaller size ($\delta_f \leq \Delta_{min}(n)$) will go undetected during simulation.

Lemma 2.1.1. *For each node n in a fault-free circuit G, an injection of a defect at n with defect size $\delta_f \leq \Delta_{min}(n)$ will not violate the circuit timing constraints.*

Proof. The circuit is assumed to be *fault-free* and $\delta_f \leq \Delta_{min}(n)$. Since $\Delta^T(p) \geq \Delta_{min}(n)$ for all paths p through n: $\tilde{\Delta}^T(p) = \Delta^T(p) - \delta_f \geq \Delta^T(p) - \Delta_{min}(n) \geq 0$. □

Upper Bound

The delay defect size at a node n can also be constrained by an upper boundary, such that all sizes beyond this boundary behave like the same fault. This type will have the maximum delay-impact on the circuit. Hence, it represents a gross-delay fault or a type of **transition fault** (see Section 1.2). The following lemma defines the upper bound as $\Delta_{max}(n)$, which is usually lower than the naïve solution T.

Lemma 2.1.2. *Consider a fault-free circuit G. Injecting a defect at node n of size $\delta_f > \Delta_{max}(n)$ has the maximum impact on the circuit output behaviour. Any defect $\delta'_f \geq \delta_f$ will cause no further changes.*

Proof. Assume that a *fault-free* circuit is sampled at T and $\Delta_{max}(p)$ is the maximum slack at node n. The slack for any path p' through n was defined as $\Delta^T(p') = T - t_{p'}$. Now, consider a defect at n of size $\delta_f > \Delta_{max}(p)$, which is $\delta_f = \Delta_{max}(p) + \epsilon$ for any $\epsilon > 0$.

After injection, the slack $\tilde{\Delta}^T(p')$ for any path $p' \in P_n$ is as follows:

$$\begin{aligned}\tilde{\Delta}^T(p') &= \Delta^T(p') - \delta_f = \Delta^T(p') - \Delta_{max}(p) - \epsilon \\ &= (T - t_{p'}) - (T - t_p) - \epsilon = t_p - t_{p'} - \epsilon.\end{aligned}$$

Since T is fixed and $t_p \leq t_{p'}$ by Def. 2, this implies $t_p - t_{p'} \leq 0$ and, therefore $\tilde{\Delta}(p') < 0$. □

In conclusion, the previous lemmas allow a restriction of the defect size δ_f for a node n without dropping any faults. The range D_n for a detectable defect size $\delta_f \in D_n$ is determined by

$$D_n = [\Delta_{min}(n) + \epsilon, \Delta_{max}(n) + \epsilon], \quad \epsilon > 0.$$

2.2 Computing Fault Sets

In order to perform the actual fault simulation, the simulator will require a list of delay defects to inject, which are represented by the classes SMALLDELAY and SMALLDELAYSET. Each SMALLDELAY consists of a gate identifier (*gidx*) and a defect size (*d*). It will increase the transition time at a node by its defect value, regardless of rising or falling transitions. To distinguish between those types, it would require additional changes to the data structure and the simulator in order to properly recognize rising and falling transitions and to trigger the fault accordingly — and individually — for each pattern. The SMALLDELAYSET simply provides a list of SMALLDELAYS and functions for computing and modification of the list.

Definition 4.4. A delay defect causes an *error*, if it alters the circuit's timing in a way, such that at least one path p from PI to PO violates the timing condition according to Definition 4.3.

For now, it is not required that the error or the erroneous behaviour becomes visible at the output immediately. In fact, it may happen that the fault effect cannot be observed by any delay test, because of blocking off-path signals that mask the transition propagation. It is then called *untestable*.

Each fault is a representative of a whole set of different defects, where the observed output is identical for each defect and any possible input patterns. Thus, defects that cause a different output behaviour cannot be represented by the same fault. Also, delay defects may have different levels of severity: A defect δ_{f_1} at a particular gate would probably cause a set of POs to fail, while larger defects $\delta_{f_2} > \delta_{f_1}$ would cause even more failing POs.

In the following, a method is described, that is able to obtain reasonable defect sizes for a circuit with unit and nominal gate delays. Each computed defect size δ_{f_i} will relate to a different fault f_i. The number of faults is *complete* w.r.t. to the defined model.

2.2.1 Classifying Faults

First, some assumptions and definitions have to be set up in order to explain the mechanics of the algorithm and the used primitives. The following assumption is important for the current configuration of the model, as it has a huge impact in model complexity:

> **Assumption 2.** All gates have single nominal propagation delay.

Based on this assumption, each path will have exactly one cumulative propagation delay. Recall that the **fan-in** (**fan-out**) of a gate n is defined as the set of gates that *directly feed* the inputs of n (*are fed* by n's output). The **input-cone** (**output-cone**) of a gate n is the portion of the circuit that is reached by *recursive traversal* of the fan-ins (fan-outs) of n.

Figure 2.1 illustrates the input-cone and the output-cone of a gate. It shows a defective gate n in a circuit G and two paths segments p and q in the output-cone. Assume, that the fault at n is excitated, and both p and q are sensitized. If the path length of q is shorter than p, it may happen, that a defect δ_f causes the PO of p to fail. However, the slack for q might still be positive. Imagine an increase in the defect size $\delta_{f'} = \delta_f + \epsilon$. The PO at p will still violate the circuit timings and the slack at q's PO is decreasing. Eventually, it will fall below zero for some $\delta_{f'}$ and the PO at q will fail, too. The moment, the slack of path q falls below zero, it will be classified $\delta_{f'}$ as a new fault f' because a new path has violated the timing conditions.

The algorithm classifies faults through a reverse approach by computing their defect sizes. For each node n in a graph G it generates the slacks for all paths $p \in P_n$. With these slacks, all relevant defect sizes for n are generated. Each defect δ_P is related to a set of unique paths $P \subseteq P_n$, such that all paths $p \in P$ begin to fail ($\delta_P > \Delta^T(p)$), hence it can be classified as separate fault. Since all SMALLDELAY defect sizes and circuit delays are implemented as SHORT data type, ϵ equals 1. The resulting defect size is $\delta_P = \max_{p \in P}\{\Delta^T(p)\} + 1$ for each set $P \subseteq P_n$.

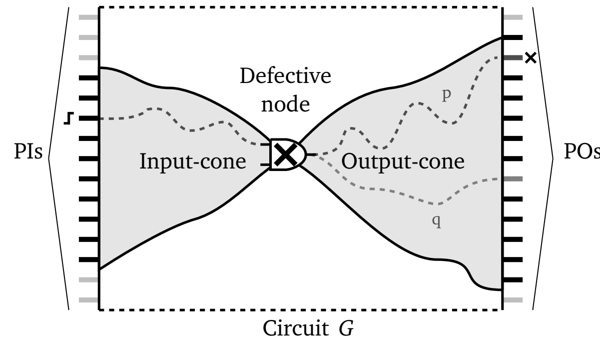

Figure 2.1: Input-cone and output-cone of a fault site.

2.2.2 Computing Path Slacks

The ADAMA framework, in which the procedures are called, provides a powerful class to model and operate on the circuit. The circuit itself is stored as a levelized graph (LEVELEDGRAPH) with access to node lists for all levels. Input pins are located exclusively on the lowest level, output pins at the highest. Each node or gate has a unique global identifier. There are functions to access its direct predecessors (fan-in) and direct successors (fan-out). The required path slacks for determining the defect sizes are computed in three phases:

In the *first phase* a forward traversal of the levelized circuit is performed from PI towards PO. For each node n at a level, a list of its signal arrival times is stored. The **arrival time** list is a set of all possible times signals will take on any PI of n's input-cone to reach n's output (including n's own gate delay). Algorithm 2.1 shows the outline of the procedure COMPUTEARRIVAL. The procedures uses the functions MERGEUNIQUE(*List,List*) and MERGEUNIQUEADD(*List,List,value*) — which simply are *union*-operators — in order to create and merge unique elements of lists. In addition, the latter procedure is able to add an additional delay directly to each value of the second list while merging at a gate. All arrival times for the inputs (at level 0) are initialized with 0. At any other level, each node n merges the arrival lists of its direct predecessors and adds its own delay (line 8–10). The timings of all nodes is provided by the $d[\,]$ parameter.

The *second phase* resembles a backward traversal through the circuit. It works like phase one, except for the reversed direction. Algorithm 2.2 shows the outline of the procedure COMPUTEPROPAGATION. Instead of arrivals, each node has to deal with *propagation delays* of each of its direct successors. A **propagation delay** list is similar to the arrival list, but everything holds for the output-cone: it is a list of all possible times it may take for a signal to reach any PO starting from n.

During the *third phase*, both arrival- and propagation delay lists are *combined* at each node n in a way, such that any element $a[i]$ of n's arrival list is added with any element $p[j]$ of the propagation delay list. All pairs (i, j) will form a path delay $t_{(i,j)} = a[i] + p[j]$, which is is

Algorithm 2.1 Computing arrival lists for all nodes (Phase 1)

```
 1: procedure computeArrival(G, d[ ])
 2:     Arrivals[0 .. N − 1][∗]        // keep a list of arrival times for each node in the graph
 3:     for all Levels L in G from PI to PO do
 4:         for all Nodes n in L do
 5:             if isInput(n) then
 6:                 Arrivals[n] ← mergeUnique(Arrivals[n], 0)          // initialize with zero
 7:             else
 8:                 for all Nodes p of FanIn(n) do        // output: predecessor + node delay
 9:                     Arrivals[n] ← mergeUniqueAdd(Arrivals[n], Arrivals[p], d[n])
10:                 end for
11:             end if
12:         end for
13:     end for
14:     return Arrivals
15: end procedure
```

Algorithm 2.2 Computing propagation lists for all nodes (Phase 2)

```
 1: procedure computePropagation(G, d[ ])
 2:     Props[0 .. N − 1][∗]      // keep a list of propagation times for each node in the graph
 3:     for all Levels L in G from PO to PI do
 4:         for all Nodes n in L do
 5:             if isOutput(n) then
 6:                 Props[n] ← mergeUnique(Props[n], 0)                // initialize with zero
 7:             else
 8:                 for all Nodes s of FanOut(n) do          // input: successor + node delay
 9:                     Props[n] ← mergeUniqueAdd(Props[n], Props[s], d[s])
10:                 end for
11:             end if
12:         end for
13:     end for
14:     return Props
15: end procedure
```

valid for some paths $p \in P_n$, since each element in $a[i]$ corresponds to some segment in the input-cone of n and the elements of $p[j]$ are related to paths of the output-cone.

The pairwise combination is done by the procedure COMBINELISTS(), which computes all *distinct* values of $t_{(i,j)}$ by dropping duplicate values, and it will eventually return a complete list of unique slacks for each node n used for estimating the set of defect sizes $D^T_\Delta(n) \subseteq D_n$. Thus, the number of elements found in $D^T_\Delta(n)$ is equal to the number of unique path lengths in P_n:

$$|D^T_\Delta(n)| = |\bigcup_{p \in P_n} \{\Delta^T(p)\}|.$$

Hence, $D^T_\Delta(n)$ covers all possible paths through n. Due to the double-coned shape of P_n (see Fig. 2.1), the model is called **Double-Cone Delay Model (DCD Model)**.

Figure 2.2 illustrates an example application of the defect size generation method on the circuit c17. Arrival times for each node are denoted in "{ }" and propagation delays are denoted in "( )". The final slacks are denoted in "[ ]". The thickened lines indicate the input and output cone of the highlighted center NAND2 node.

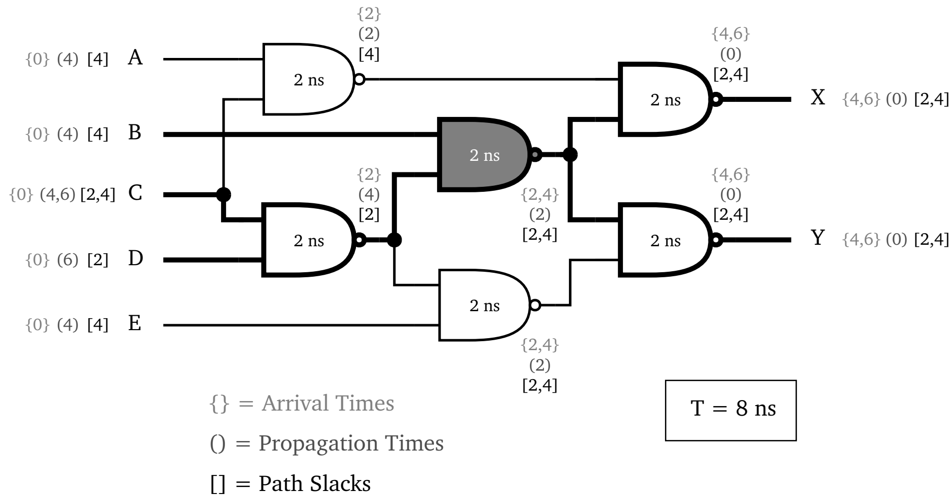

Figure 2.2: Computation of *arrival times*, *propagation delays* and *path slacks* for each node of c17 with sampling time $T = 8$ ns.

Finally, Algorithm 2.3 lists the necessary steps to compute all relevant defect sizes of the underlying model. Since each computed SmallDelay δ_f of a node n causes a unique set of paths $P \subseteq P_n$ to fail, it is directly referred to it as a **fault** f. Different faults for a node n will produce unique syndromes according to Definition 4.4, when applying all possible input patterns.

Algorithm 2.3 Determining defect sizes

1: **procedure** DOUBLECONEMODEL(G, $d[\,]$, T) // T: circuit sampling time
2: $Arrivals \leftarrow$ COMPUTEARRIVALS(G, d)
3: $Proptimes \leftarrow$ COMPUTEPROPAGATION(G, d)
4: **for all** Nodes n in G **do**
5: $Slacks \leftarrow$ COMBINELISTS($Arrivals[n]$, $Proptimes[n]$, T) // combine unique path segments to a path delay $t_p = t_{(a_n[i], p_n[j])}$ and compute the corresponding slack ($= T - t_p$)
6: **for all** Values v in $Slacks$ **do**
7: ADDDEFECT(n, $v + 1$) // defect size = slack + ϵ
8: **end for**
9: **end for**
10: **end procedure**

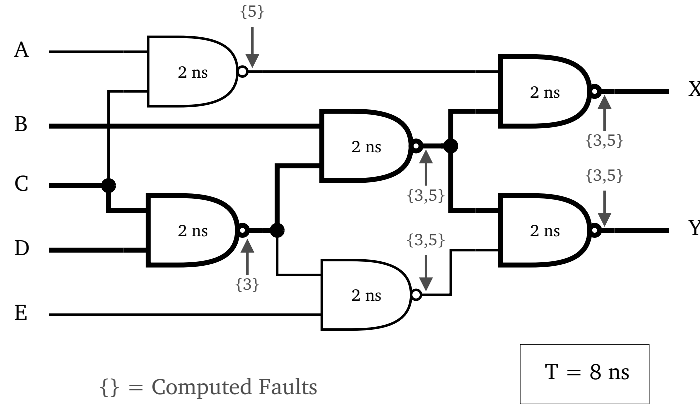

Figure 2.3: Computed faults of the interior nodes of c17. The circuit sampling time is $T = 8$ ns.

Figure 2.3 shows the resulting fault lists for each gate of the example circuit. For illustration purposes, the fault sites were restricted to the interior nodes. The centered NAND2 node n is traversed by a total of six paths, two of whom — the ones initiating at B — have a path delay of 4 ns, while the others have a delay of 6 ns. The computed slacks are {2} and {4}, hence the resulting defect sizes $D^T_\Delta(n) = \{3, 5\}$.

2.3 Complexity

The major drawback of the DCD fault generation is its complexity: The fault count as well as the computation depend on the number of possible paths through each double-cone P_n, which sums up to roughly $\mathcal{O}(N \cdot 2^N)$ faults in the theoretical worst case for circuits with N two-input

gates. Although, this indicates exponential nature, experiments showed that the average fault count scales linearly with the circuit size, thus being not that worse.

However, to make the simulation model more viable w.r.t. to the fault count and computation, some faults may be dropped from the list. Any fault, that is dropped, will cause a loss of precision of the model, since the related defects would produce a faulty output behaviour in the real design, which could not be reproduced within the simulation model.

In the following, two modifications are presented in order to create fault-lists with reduced fault count and reduced computational complexity of the fault-list generation.

2.3.1 Modification A: Transition-Fault Model

Algorithm 2.4 Modification of DCD for usage as transition-fault model

```
1: procedure TRANSITIONFAULTMODEL(G, d[ ], t)
2:     Arrivals ← COMPUTEMINARRIVALS(G, d)          // keep minimum values only
3:     Proptimes ← COMPUTEMINPROPAGATION(G, d)
4:     for all Nodes n in G do
5:         slack ← COMBINELISTS(Arrivals[n], Proptimes[n], t)
6:         ADDDEFECT(n, slack + 1)
7:     end for
8: end procedure
```

The first modification (see Alg. 2.4) resembles a subtype of the well-known transition-fault model. The fault set computation is similar to the DCD model, but instead of merging whole lists to super-lists, each node will only have to store the minimum arrival and minimum propagation time. With each node n holding the values for a_{min} and p_{min}, one can compute the cumulative delay $t_{p'} = a_{min} + p_{min}$ of the shortest path p' of P_n and its slack $\Delta^T(p')$ to determine a defect size. Since $\Delta^T(p') = \Delta_{max}(n)$, the related defect size $\Delta_{max}(n) + \epsilon$ will violate the timing conditions for any path of P_n, due to:

$$\forall p \in P_n : \Delta^T(p) - \Delta_{max}(n) - \epsilon < 0.$$

These faults are classified as gross-delay-faults and the resulting set will have a size of N ($\Rightarrow \mathcal{O}(N)$), with one fault per node. Unlike the original transition-fault model, a fault at a node n currently represents both slow-to-rise (STR) and slow-to-fall (STF) faults simultaneously, since it is not distinguished between rising and falling transitions.

2.3.2 Modification B: Defect Range Quantization

The transition-fault modification of the algorithm showed, that the fault count an be reduced dramatically at the expense of model precision, since several small-delay faults of intermediate size are not considered in the simulation set.

The following modification is a simple trade-off to improve the defect model precision, while keeping the fault count and complexity low. The method is similar to the transition fault modification: every node computes the maximum *in addition* to the minimum arrival and propagation times. Using the maximum and minimum for each node n, we define the slack range $[\Delta_{min}(n), \Delta_{max}(n)]$ and the resulting defect range:

$$D_n = [\Delta_{min}(n) + \epsilon, \Delta_{max}(n) + \epsilon].$$

A **quantization parameter** q is used to fetch intermediate values of D_n, in order to recover some small-delay faults, that might have intermediate severity levels. Quantization could be done by using:

- equally distributed values, or
- adaptive computation of intermediate values by regarding defect size distribution.

Here, a quantization with equally distributed values was implemented as follows:

If $q = 0$, a single defect $\delta_f = (\Delta_{max} - \Delta_{min})/2$ is added for each gate. The mean is intended to have a noticeable impact on the circuit behaviour, but it should not be as severe as a transition fault.

For $q > 0$, $q + 1$ equally distributed defect sizes out of D_n were chosen, such that each two neighbouring defects δ_{f_i} and $\delta_{f_{i+1}}$ have maximum *distance*. Hence, the defects of the interval endpoints $\delta_{f_0} = \{\Delta_{min}(n) + \epsilon\}$ and $\delta_{f_q} = \{\Delta_{max}(n) + \epsilon\}$ will always be in the computed set. They serve as a representatives for the faults at n, to observe the *least* and *worst* possible impact on the circuit behaviour. All other δ_{f_i} $(0 < i < q)$ are distributed over D_n.

Figure 2.4 gives an example for the defect size distribution of different qs. Assuming a quantization factor of q, the fault count can be estimated by $\mathcal{O}((q + 1) \cdot N) \approx \mathcal{O}(q \cdot N)$. However, the generated fault list is not *directly compatible* with the prior definition of *fault*, since the computation of intermediate defect sizes might generate values, that cannot be derived from any path in the circuit. Yet, these defects δ_x are related to some known faults $\delta_{f_1}, \delta_{f_2} \in D^T_\Delta(n)$ of the model through $\delta_{f_1} \leq \delta_x < \delta_{f_2}$. The generated δ_x is a *valid* representative for fault f_1, but it may happen, that multiple representatives of one and the same fault are generated. Thus, inserting a duplicate fault to the fault set is imminent.

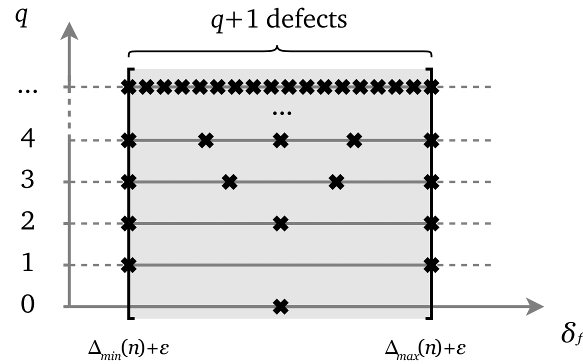

Figure 2.4: Quantized D_n for different qs. The marks represent the resulting defect sizes.

2.3.3 Fault Equivalences

Reducing the fault count by exploiting equivalence classes is much more complex and restricted, than compared to the stuck-at fault model. Since, all computed defect sizes for a node n in the model are related to paths in P_n, the faults of n and g will be equivalent only if $P_n = P_g$, because each defect has to be excited and propagated over the same paths in order to produce the same output behaviour for any input pattern.

According to Waicukauski *et. al* [WLRI87], transition faults are considered as equivalent, if at least one of the following rules applies:

- A gate has one input — then both input and output transition faults are equivalent.
- A gate has only one fan-out — in this case, both output faults and faults at the fan-out inputs are equivalent.

However, the underlying model only considers fault sites at gate outputs, so, these rules cannot be applied directly. An extension could be easily *simulated* by simple insertion of buffers at each input, but this is not very helpful, as the faults introduced by the insertion buffers are merely the ones, which are collapsed in the step afterwards.

Yet, recalling that $P_n = P_g$ implies fault equivalence for n and g, this idea lead to the following rule in order to perform fault collapsing:

> **Rule:** Assume a gate n with a single fan-out g. The fault lists of n and g are equivalent, if g is either a single input gate, or a k-chain of single input gates $g_0, \dots, g_{k-1}$, since $P_n = P_{g/g_i}$. Then, the fault lists of g respective g_i for $i = 0, \dots, k-1$ can be dropped.

This is fault collapsing in its simplest form (Fig. 2.5). Although, there *might* be more fault equivalences, finding these classes would require complex methods and special (global) knowledge about the circuit's structural behaviour, which is neither given nor subject of this thesis.

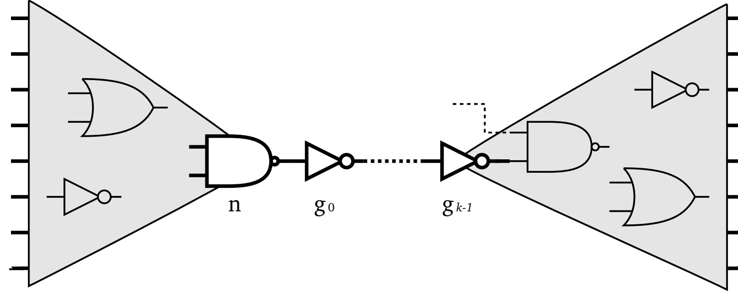

Figure 2.5: Illustration of $P_n = P_{g_i}$. – Fault equivalences of a single fan-out gate n and a succeeding chain of k single-input gates g_i.

2.4 Relation between Arbitrary Defect Sizes and SmallDelays

Arbitrary delay defect sizes $\delta_x > 0$ at any gate n can be mapped to a defect size $\delta_f \in D^T_\Delta(n)$ of a SMALLDELAY fault f by

$$\delta_x \mapsto \begin{cases} \max(\{\delta_{f_i} \in D^T_\Delta(n) : \delta_{f_i} \leq \delta_x\}) & \text{if } \delta_x \geq \min(D^T_\Delta(n)) \\ 0 & \text{else } (\textit{fault-free}). \end{cases}$$

For two consecutive defect sizes $\delta_1, \delta_2 \in D^T_\Delta(n)$ at a node n, all defects $\delta \in [\delta_1, \delta_2)$ behave the same and thus belong to the same *class*. Given a particular defect size δ_x, the above mapping simply determines a representative of the defect class it belongs to. Since, P_n is finite and the underlying model considers integer-valued defect sizes only, each representative is simply chosen by the lowest bound of the defect class (which is δ_1). If defects are below the minimum detectable size, they are mapped to the fault-free case ($\delta_f = 0$), because the circuit itself is assumed to be fault-free and the defect will cause no noticeable changes. Hence, any defect size can be mapped to a unique SMALLDELAY representative.

3 Delay Fault Simulator

This section describes the implementation of our delay fault simulator and its integration into the ADAMA framework. The core consists of a *data parallel* timing simulator, called `wave`, which was written by Stefan Holst. It is implemented in CUDA/C in order to compute many circuit signals concurrently on CUDA-capable graphics devices.

3.1 Test Pattern Generation

The general structure for delay fault tests differs from normal stuck-at tests. Finite delay defects cannot be detected by using a single pattern during simulation. These defects only affect signal transitions at the corresponding sites, but not the logic behaviour in a stable state. Each transition requires at least two input patterns: the **initialization vector (IV)**, which sets the circuit into the stable state, and a **propagation vector (PV)**, that spawns transitions at all PIs, where the values of IV and PV differ. If a transition reaches the defective site, it triggers the fault effect. Unless further propagation of the affected signal is masked by controlling off-path signals, the fault effect will become visible at the outputs.

The ADAMA framework generates either deterministic or random pattern sequences, which have to be converted into delay tests by combining pairs of patterns.

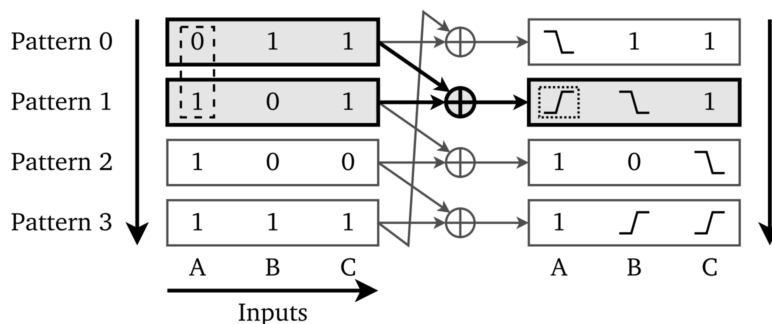

Figure 3.1: Test-pattern conversion. – Each delay test-pattern is computed by using two consecutive patterns of a random generated pattern-block.

Figure 3.1 illustrates the conversion of a PATTERNBLOCK into a series of delay tests. One should note, that the signal values of resulting patterns do not only consist of 0s and 1s, but also of *rising* and *falling* transitions. This is because the timing simulator uses a data structure, that describes signal *waveforms* rather than plain signal values. The current conversion scheme was chosen to ease pattern evaluation and fault detection, since the *i*-th pattern of a block is also used as PV in the delay-test vector. Thus, the responses of the *i*-th pattern will represent the stabilized state of the good simulation, which can be acquired through plain logic simulation, instead of wasting an additional complex time-simulation for the fault-free case.

3.2 Fault List Generation

As presented in the preceeding section, the double-cone delay model (DCD) and its modifications were implemented in order to obtain the fault-lists for the delay fault simulation. The fault-lists can be provided in two ways. They can either be generated at runtime, or being provided by external files, that contain, for example, pre-computed sets of faults.

For **runtime generation**, the simulator is started normally by choosing a particular fault model modification, or by using the default setting. After instantiation of the circuit and generation of the gate delays a SMALLDELAYSET will be setup, that uses a fault-list generation procedure according to the chosen modification. Once the final fault-list has been computed, it is stored within the SMALLDELAYSET, where each fault can be directly accessed through an index. Additional functions, such as CROP(*from*, *to*) or CROPRANDOM(*number*), allow to pick a range or a certain number of random faults out of the computed set and join them into a new list. The original set can always be restored by calling RESTOREFAULTLIST(), which resets the reference.

Since fault-list generation is very compute intensive, it might also be recommended to generate the list prior to the actual simulation. For example, if fault simulation of a circuit has to be repeated different times for experimental issues, a computation of the full fault set would be required each time the simulation is started. This would cause an increase in simulation time and prolong the experiments.

To avoid this, fault-list **file generation** has been implemented in ADAMA `fsample` for the use with SMALLDELAY-sets (see Appendix A). The fault files generated for a circuit use a fixed sampling time, which does currently depend on the circuit depth. Any modifications will likely cause the fault files to become invalid and require a re-computation of the sets.

3.3 Fault Simulation

In order to capture the effect of arbitrary small delay faults, a time simulator called `wave` is used, which allows precise logic-level timing simulation to be computed on CUDA-capable devices. For further references and detailed mechanics of the `wave` process it is referred to the `wave` section (see Appendix C).

The implemented delay fault simulator consists of two parts: The simulator task itself as a part of the ADAMA framework, and the `wave` executable. The framework part is responsible for the pattern generation and conversion, computing register and memory mappings, fault injection as well as the final response evaluation, whereas the `wave` executable handles all the CUDA-related accesses and performs the gate evaluations. Both processes communicate together via input/output streams using implemented state-machines and memory maps for exchanging data.

3.3.1 Initialization

When the simulator is started, the LEVELEDGRAPH of the circuit is converted by the FRAME-FORMATTER into a scheduled frame-map. The frame-map holds the circuit information for all existing gates including the register mappings, which are used to identify cells, where signals of certain gates are or have to be stored in the memory maps.

The `wave` process is started once the FRAMEFORMATTER has finished its work in order to initialize the device and the memory mappings properly. It selects a certain CUDA device and the required amount of device memory for allocation. The implemented state machine then awaits orders from the ADAMA framework. If either the frame count of the circuit is too large or the amount of required memory is not sufficient, the initialization fails and the simulator will terminate.

3.3.2 Tasks

In each **simulation run** an individual fault is simulated for all provided input stimuli. For a large number of patterns it may happen that the pattern-set has to be split into smaller disjoint subsets, which each is simulated in a separate *pass*, since the limited memory on the GPGPU only allows a finite set of patterns to be simulated at a time. When using a fixed amount of global device memory, the number of passes strongly depends on the number of used registers and their respective maximum size or **wavecap**, as well as the amount of provided patterns. The relationship between these parameters can be described as follows:

$$\#PatternsPerPass = \left\lfloor \frac{MemorySize}{\#Registers \times MaxRegisterSize} \right\rfloor, \quad \#Passes = \left\lceil \frac{\#Patterns}{\#PatternsPerPass} \right\rceil.$$

A **pass** is performed in a level-wise manner starting from PI to PO. Each time, the `wave` simulator will be involved by executing three *major tasks*:

1. The input stimuli (waveforms) of all patterns to be processed in the pass are transferred to the GPGPU device.
2. The level-wise evaluation of the gates is performed by repeatedly calling the kernel for each frame index in increasing order.
3. At last, the responses are transferred back to the host (where they can be evaluated by comparing to the good simulation in order to find any failing bits and patterns).

In its simplest form, the delay fault simulator can be described as three nested loops: for every fault (*run*), for every pattern-pass (*pass*), for each level (*frame*). Figure 3.2 and 3.3 show the outline of the major tasks that are performed during a fault simulation. The whole flowchart was split into two parts to provide a better comprehensibility.

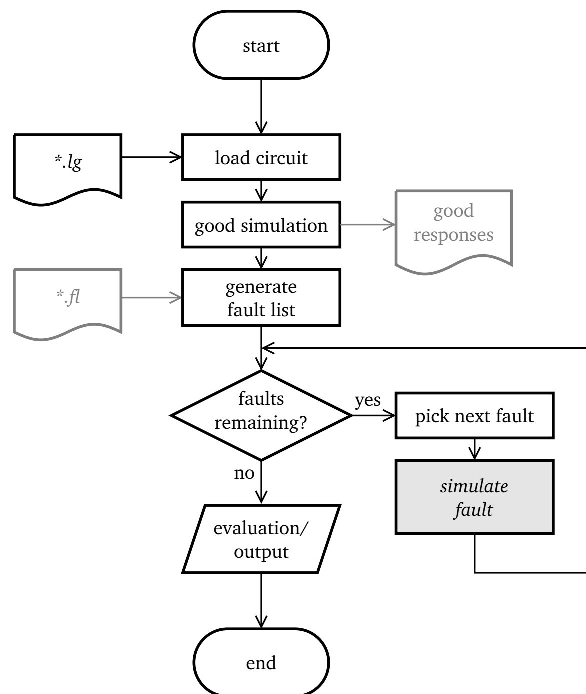

Figure 3.2: Outline of the major steps done by the fault simulation framework indicating the outer loop of the fault simulation.

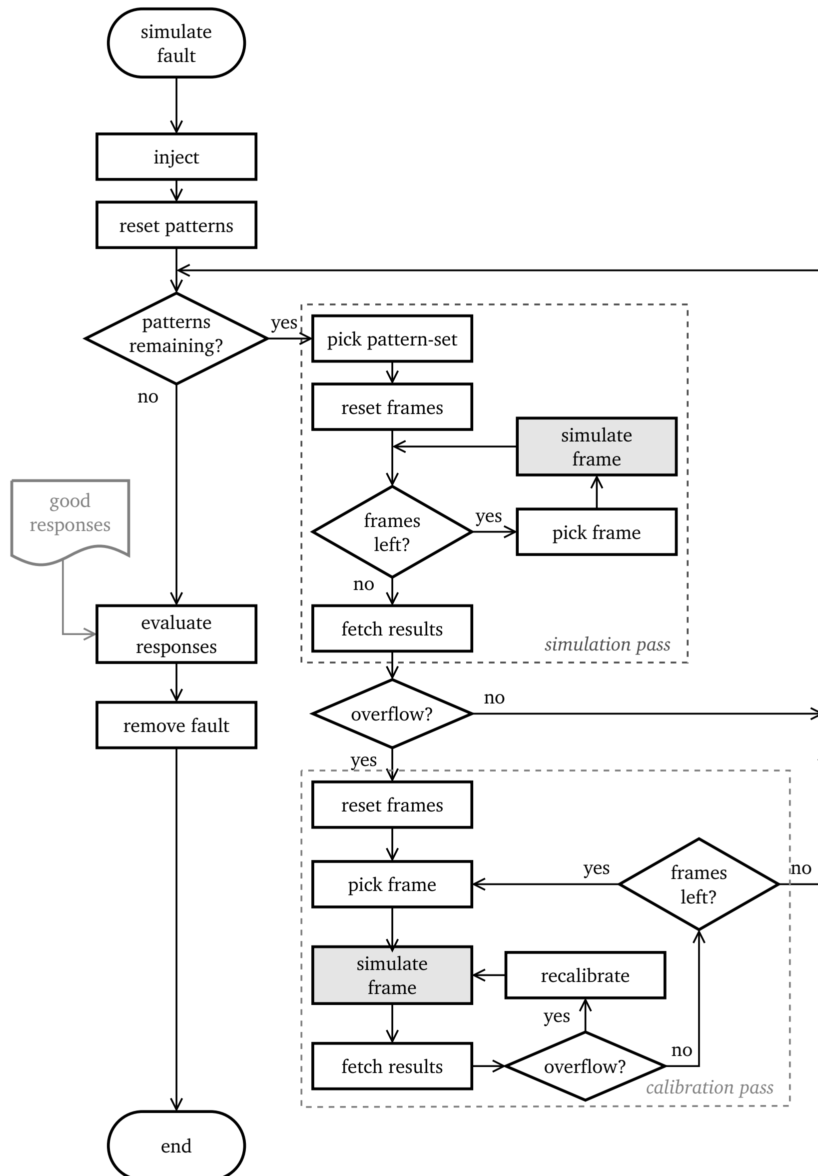

Figure 3.3: Fault simulation. – Major tasks performed during a run. The GPGPU device is being occupied at the highlighted stages, where a single frame is executed. The separate calibration pass has been added for completion.

3.3.3 Remarks on Circuit Levelization

The `wave` simulator transfers the gate data of a level L_i to the CUDA device constant memory prior to calling the simulation kernel. The data structure, used for representing a single gate, restricts the number of gates per level to a **maximum** of 4000. So, all circuits have to be levelized in a way that each level does not exceed that maximum allowed gate count. If a level meets this requirement, it is referred to as **a frame**. The **frame-depth** will be the number of levels of a *frame-compliant* circuit.

Most of the NXP benchmark circuits did not meet this frame property and had to be re-levelized in order to meet the maximum allowed gate count per level. This came along with an increase in frame-depth, although the actual logic depth remained unchanged.

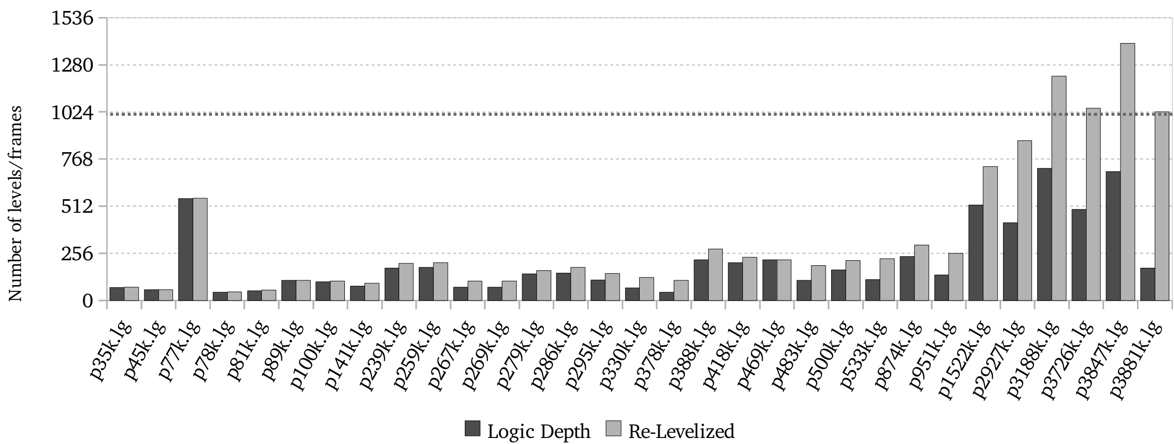

Figure 3.4: Frame-depth of the NXP benchmark circuits after limiting the maximum allowed number of gates per level to 4000. The thick dotted line indicates the current frame-limit.

Figure 3.4 shows the number of frames after re-levelization of the NXP circuits. One should notice, that the process had a much larger impact for bigger, than for the smaller ones. This is due to the nature of ASAP levelization, which caused the width of the first levels often to exceed hundreds of thousands of gates. For some unknown reasons, p874k.lg could not be re-levelized. Thus, the missing frame-depth in the evaluation. The number of maximum allowed frames is currently capped at 1024.

3.3.4 Parallelism

During simulation, both data and structural parallelism are exploited by evaluating multiple patterns in *wavesets* for multiple gates at once on the GPGPU. Depending on the circuit structure, the ratio between those types of parallelism may vary.

3.4 Fault Injection

All faults of the provided fault-list are processed serially in a loop with the currently indexed fault being passed via an argument to the simulator for the injection process. Injection takes place prior to the actual logic simulation run.

The FrameFormatter class is responsible for creating and allocating the aligned gate data structures for the simulator. It maps the ADAMA internal LeveledGraph representation to the memory. So, the FrameFormatter seemed thus to be a viable entry point for the fault injection by simply adding the defect size at the corresponding gate before the memories are synchronized and the simulation commences. In addition, this mechanism will also allow for injecting multiple faults. In the simulation experiments only one defect at a time is considered.

When the simulation run has been completed for a certain fault, the original delay of the affected gate has to be restored explicitly by calling clear().

> **Remarks:** Fault injection by modifying the frame-data will only affect interior nodes of the circuit. Without changing the circuit structure, faults located at either input or output pins have to be handled differently (which is not yet implemented): Any delay defect at an input pin could be simulated by *delaying* transition at the corresponding locations, while converting the input stimuli. For outputs, one could simply adjust the sampling time at the pins when evaluating the response waves.

3.5 Response Evaluation

After the device has finished the computation of the last frame, the results are being synchronized and stored in the mapped memory. Before the evaluation takes place, all results are checked for overflows, which are also encoded in the waves. If none of them were detected, the registers designated as outputs are evaluated by the **wave evaluation algorithm**. Otherwise, the calibration procedure will be called.

For the wave evaluation, the sampling time has to be identical to the one, which was used to generate the fault lists. This will ensure that the intended fault behaviour is captured. The

default capture time for fault generation and wave sampling is currently set to a multiple of the circuit depth: $T = 2 \cdot (CircuitDepth + 1)$ [ns], since the maximum delay of any specified gate in the underlying model was 2 ns. The final results are stored as common two-valued logic at the output-offset of the corresponding patterns.

By comparing all output bits with the expected values of the good simulation responses any failing bits can be marked in a syndrome vector. The wave evaluation scheme does not detect all path timing violations caused by signal hazards. For example, any delay fault, that pushes a hazard beyond the sampling point, might go undetected if the sampled signal value equals the stabilized *good* value (see Fig. 3.5). In fact, this is one of the big advantages of `wave`, since the waveform evaluation reflects the behaviour of in-the-field devices, where signals are being sampled at discrete points of time. In addition, extensions can be made to further increase modeling accuracy, e.g. by considering setup- and hold times of latches or flip-flops.

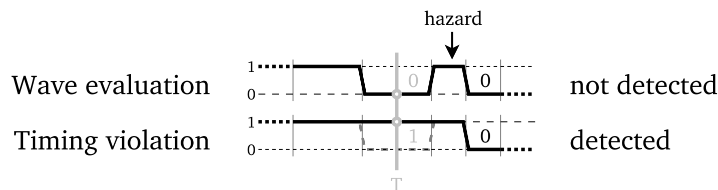

Figure 3.5: Some violations caused by hazards go undetected during simulation when using wave evaluation with the circuit sampling time. General timing violations can be detected by examining the latest transitions only.

> **Remarks:** Another way to evaluate the simulation results could be checking for pure signal timing violations along paths. These can be obtained by simply checking whether the latest signal transition at an output pin lies beyond the sampling point or not (except for the ∞ terminal). On the one hand, this evaluation scheme would simplify the structure of the simulator and increase the simulation performance, as the amount of memory per register could be drastically reduced. However, on the other hand, it would also cause a huge loss in modeling accuracy.

All results of the simulator experiments, regarding performance on common ISCAS'85 benchmark circuits and larger industrial designs from NXP, are shown in Section 5.2 in "Experimental Results".

4 Diagnosis

This section will describe the integration of the delay fault simulator into the ADAMA framework for the use with delay fault diagnosis.

The main purpose of the diagnosis is to find and locate the cause of errors in defective circuits by analyzing their responses. As explained in Section 1.6, the POINTER [HW09] algorithm uses evidences of each fault in order to establish a ranking of possible fault candidates. For experimental results regarding the diagnosibility of delay faults with POINTER, it is referred to Section 5.3.

4.1 Setup

Figure 4.1 shows the general setup of the diagnosis of a defect by using POINTER. Here, the output of a **device under diagnosis (DUD)** is compared with the responses of the fault simulated in a **fault machine (FM)** to compute the evidence components σ_π, ι_π, as well as τ_π and γ_π. By adding up these components for every simulated pattern π, the global evidence of a suspect fault is obtained. During the diagnostic run, the FM will simulate a collapsed set of all possible faults of the circuit and create a collection of evidences. This collection is sorted or *ranked* afterwards by its evidence components in order to create a list of reasonable candidates, that might explain the DUD defect.

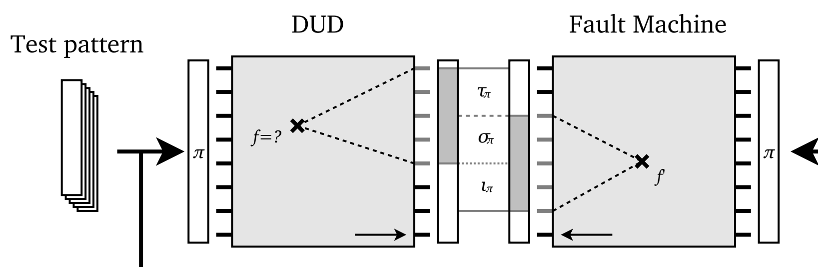

Figure 4.1: Evidences of a fault f for a pattern π in the POINTER diagnosis. The erroneous output pins are highlighted. The marked fault in the DUD is usually *unknown*.

4.2 Delay Fault Diagnosis using POINTER

The delay fault diagnosis has been implemented as follows: At first, a fault list is generated or loaded externally to create a set of delay faults, which each is injected into a DUD and diagnosed in a separate run. During each diagnostic run, a delay fault simulation of a DUD is performed in order to create the faulty responses for a provided set of test-patterns. The responses are stored and will be used to compute the evidences later.

Mapping Delay Faults to Stuck-Ats

After the DUD simulation, the delay fault (SMALLDELAY) is mapped to regular **stuck-at faults** by spawning a *stuck-at-0* and *stuck-at-1* at the corresponding location as an intermediate step. These stuck-ats are further mapped to their respective *class representatives*, which are merely predetermined representatives of their equivalence classes used for fault set collapsing. The DUD defects are in general not known, but the *artificial knowledge* will allow for an evaluation of the diagnosed candidates.

The FM will now simulate the whole collapsed stuck-at set of the circuit. Since, delay fault simulation is more time-consuming than compared to stuck-at simulation, due to the increased simulation-complexity and higher fault counts, processing a complete delay fault set for a circuit would become infeasible. The POINTER algorithm does also try to diagnose *independently* of the applied fault models.

Now, both the responses of the FM and the DUD are compared in order to compute the evidences of all stuck-at faults according to POINTER. These evidences are ranked to obtain a **suspect list** of possible candidates, that might cause a similar behaviour as the DUD. The resulting suspect list is further split into **rank-groups**, which contain suspects with identical evidence. The index of a rank group is determined by the rank of its first (best) suspect. Since all suspects of the group have to be treated equally for evaluation due to equal evidences, the *middle-rank* is used instead. The **middle-rank** of a suspect is the *average* rank of all suspects in its rank-group. For example, let $RG(s)$ be the rank-group of a suspect s and let $Index(RG)$ be the rank of the first suspect of the group, then the middle-rank of s is computed by:

$$MiddleRank(s) = Index(RG(s)) + \left\lfloor \frac{|RG(s)|}{2} \right\rfloor .$$

With this suspect list, one can observe how well a certain SMALLDELAY can be explained by a stuck-at fault at that particular location. The success of the diagnosis is determined by the rank of the actual suspects, which are the stuck-at substitutes of the delay fault.

Rank	Evidence	GroupRank	RankGroup	MiddleRank
1	e_0	1	1	2
2	e_0	1	1	2
3	e_1	2	3	4
4	e_1	2	3	4
5	e_1	2	3	4
6	e_2	3	6	6

Table 4.1: Example. – RankGroup and MiddleRank of a sorted collection of evidences.

4.2.1 Fine-grained Resimulation

Recall that the stuck-at diagnosis step has not yet determined the actual defect location, as the POINTER algorithm has only compared the equivalence class representatives so far. Although, the equivalence class can contain multiple stuck-at faults located at different gates, they will all produce the same evidence. Since this does not necessarily hold for delay faults at the exact same locations, the real defect might still be hidden somewhere within. Therefore, an additional fine-grained delay fault diagnosis step is performed for the top-ranked suspects to identify the actual culprit and to weed out the unrelated faults. The resimulation is only done for a DUDs in case that the defect has been detected. If a DUD response shows no failing bits, resimulation is skipped and the actual rank will be set to 100000, whose literal equivalent is "*not diagnosed*". As a result, this does not allow a ranking of suspects by *exclusion*.

Mapping Stuck-Ats to Delay Faults

Beginning from the top, the equivalence classes of the ranked stuck-at collection will be unrolled for any failing DUD. The location of each fault is then used to generate multiple representative **delay faults**. For each entry, a whole set of appropriate SMALLDELAYS should be **re-simulated**, since varying defect sizes can have a strong influence on the output behaviour. Experiments showed that the choice of representatives also has a huge impact on the diagnostic resolution. Here, the currently chosen SMALLDELAYS are the *mean* and the *maximum* defect size, but any arbitrary combination can be implemented. All representatives are obtained *on-the-fly* during diagnosis by the SMALLDELAYSET class. Again, due to the time-consuming delay fault simulation, the maximum number of resimulations should be restricted. The choice of the number of resimulations is thus a trade-off between speed and diagnostic performance.

Also, depending on the number of delay fault representatives per stuck-at for a fixed amount of resimulations, the diagnosis will be able to examine more stuck-at suspects and probably

more ranks in a coarse-grained fashion, or to look at less suspects but more carefully in order to consider large variations in the output behaviour for some defect sizes.

> **Example** Let the number of maximum resimulations be 20 and let the number of delay fault representatives be 4. Then, 5 stuck-ats (or up to 5 ranks) can be resimulated. If the number of representatives is lowered to 2, then 10 stuck-ats (up to 10 ranks) can be investigated.

After each resimulation step, the responses are used to compute the new evidences. This second evidence collection is usually smaller than the first, since the number is restricted by the resimulations. Once again, the new collection is sorted and checked for presence and rank of the actual suspect by comparing the fault location. If the actual suspect has been cut off due to the resimulation, its exact rank cannot be determined and it will be declared as "*not diagnosed*". Thus, the coarse restriction to the top-ranked stuck-at candidates assumes a well-ranked stuck-at diagnosis.

An outline of the different steps in the new two-level delay fault diagnosis approach is depicted in Figure 4.2. All stages, where actual delay fault simulation is performed, are highlighted.

4.2.2 Pattern Analysis

The original implemented stuck-at POINTER analyzer in ADAMA had a highly efficient procedure for analyzing response pattern-blocks and computing their evidences. However, the pattern representation in the delay fault simulator (PATTERNLIST) differs from the one used for stuck-at faults (PATTERNBLOCKLIST). Since it was most likely that an additional conversion of the data types would nullify its effective gain in speed, evidence computation was implemented specifically for comparing PATTERNs with PATTERNBLOCKs during resimulation. Also, the coarse time consumption of the resimulation step is still determined by the communication overhead of the simulator rather than the pattern analysis part.

The procedure for the pattern evaluation simply compares each output value of the response of one representation with its respective counterpart and counts the σ_π and ι_π for each pattern π. All partial evidences are added up, which eventually provides access to the σ_Π, ι_Π and γ_Π $(= \min(\{\sigma_\pi, \iota_\pi\}))$ of the fault w.r.t. the complete simulated pattern set. The evidence will be associated with the corresponding fault and added to the global evidence collection for ranking and evaluation.

> **Remarks:** The DUD responses are converted into PATTERNBLOCKs once for the stuck-at analysis, since the FM will do a block-wise exhaustive comparison of a large set of responses. A PATTERNBLOCK always consists of up to 64 patterns. Even if less patterns have to be *generated*, the random generator will fill **all** positions,

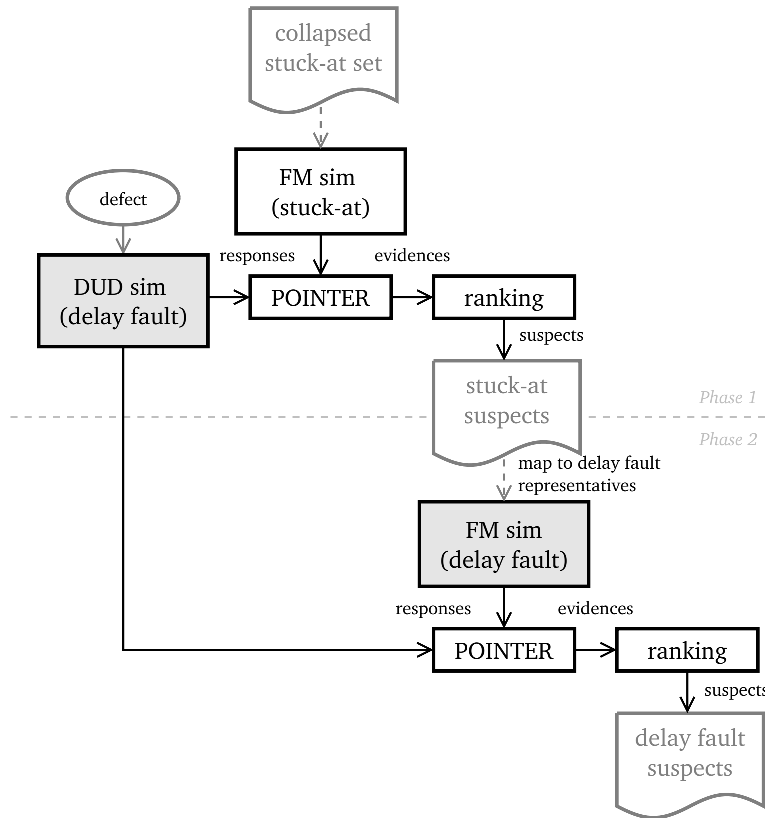

Figure 4.2: Outline of the two-phased delay fault diagnosis approach. Data dependencies are shown.

each of which is examined by the stuck-at analyzer. The delay fault simulator currently converts the wave responses back into a set of PATTERN via the procedure PACKPATTERNS( ), but it should also be able to produce whole PATTERNBLOCKS, since it does currently accept multiples of 64 patterns only to avoid possible *unknown* patterns during conversion into blocks.

4.3 Suspect Ranking

Both rankings of the two phases in the diagnosis task are independent and allow generic comparators for sorting the evidence collection. The currently implemented comparators are the original proposed **Gamma-Sigma-Iota (GSI)** and a modification **Sigma-Iota-Gamma (SIG)**, which sort according to the given order of evidence components. The evidences, that occurred during experiments, suggest that additional *weighting* of individual components or even *ranking functions* might be an alternative to adapt and improve the stuck-at evidence evaluation.

Candidate Evaluation

Finally, the comparators will provide a ranked list of suspects, which suggests likely candidates for the DUD defect. By searching the candidates for the actual culprit, the success of the diagnosis and the effectiveness of the ranking is determined. The lower the reported middle-rank, the better the result.

Regarding the experiments, a diagnosis task is said to be "*successful*" if the **actual suspect** (defect location of the DUD) is found among the Top-10 ranks. Otherwise, it is considered as "*failed*". The defect is *perfectly identified*, if the corresponding candidate is located at the first rank of the list.

5 Experimental Results

The current section presents the results, that emerged from experiments throughout this work. It is organized in three parts: An evaluation of the presented fault generation procedure, followed by experiments regarding the performance of the simulator and, finally, an *experimental journey* of the proposed delay fault diagnosis approach using the POINTER pattern analysis.

5.1 Fault Set Generation

In the following, delay fault set generation for the DCD model, the transition fault and the quantized modification with equally distributed values (with $q = 2$ and $q = 9$) were applied to a set of benchmark circuits. The timing information of the circuit has been generated automatically:

- Single-input gates (INV, BUF) have a nominal propagation delay of 1 ns.
- Two-input gates (NAND, NOR, AND, OR, XOR, XNOR) the delay was set to 2 ns.
- Input and output pins (INPUT, OUTPUT) have zero delay.

Other types of gates do not appear in the underlying circuit model. Also, faults at either inputs or outputs have been excluded from the evaluation.

5.1.1 Counting Delay Faults

Figure 5.1 shows general circuit information about the ISCAS'85 benchmark circuits and the results after applying the DCD fault generation procedure. The right side is of particular interest, as it hides information about the fault count as well as the approximate number of faults per gate. The modifications were listed separately in Figure 5.2. Here, all numbers are w.r.t. to the complete DCD model above.

Obviously, the quantized model scales linear in faults as the parameter q increases. If each D_n consists of a finite number of discrete defect sizes, the fault count will saturate for higher q, since all defect sizes will eventually be covered. However, a high q will make the model impractical due to the increasing number of redundant faults. These side-effects usually occur for smaller circuits structures or small circuits, such as c17 and c499, when q exceeds the

Circuit Data										DCD Model		
Name	Inputs	Outputs	Gates	Depth	Avg Path	Width	FO Stems	Avg FO	Max FO	[F]aults	~F/G	Max
c17.lg	5	2	6	3	3	2	3	2.00	2	10	2	2
c432.lg	36	7	216	29	22	27	89	2.65	9	6270	29	42
c499.lg	41	32	246	14	14	40	59	4.34	12	1792	7	14
c880.lg	60	26	435	30	11	58	125	3.50	8	7180	17	44
c1355.lg	41	32	590	27	27	64	259	2.97	12	15668	27	36
c1908.lg	33	25	1057	44	37	97	385	2.58	16	31272	30	57
c2670.lg	233	140	1476	39	4	169	454	2.74	11	29857	20	54
c3540.lg	50	22	1983	56	31	216	579	3.15	16	89218	45	87
c5315.lg	178	123	2973	52	15	335	806	3.51	15	65015	22	75
c6288.lg	32	32	2416	124	78	256	1456	2.64	16	407999	169	234
c7552.lg	207	108	4043	45	12	445	1300	2.95	15	87894	22	64

Figure 5.1: General information of the ISCAS'85 benchmark circuits and DCD fault count. The circuit data has been extracted from ADAMA `stat`.

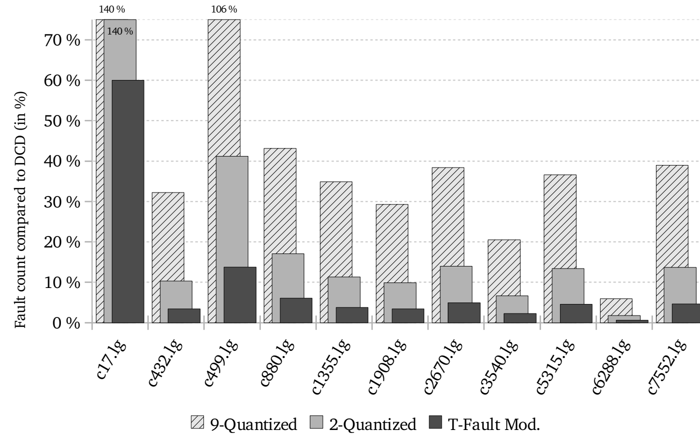

Figure 5.2: ISCAS'85 circuits. – Fault counts of transition fault, 2-Quantized and 9-Quantized modifications compared to the DCD model (in %).

average of the faults per node and causes the fault count to overshoot. Recall that overshooting does not imply a full fault coverage, which in return can only be ensured if q is saturating.

Defects of smaller size produce less errors and are also less likely to be detected by random patterns, since the fault effects have to be propagated along the longest paths in the circuit. Experiments have shown that the DCD defect size distribution for a node n is often clustered in dense groups of defects with occasional small distance of $\epsilon = 1$. Although a defect δ_f should violate a unique set of paths in P_n, most of the paths might never get sensitized by test-sets, due to **random pattern resistance** — unless trying ATPG for path delay faults, of course. Thus, defects of such a cluster will often cause a similar behaviour for random generated test-sets.

Circuit Data										DCD Model		
Name	Inputs	Outputs	[G]ates	Depth	Avg Path	Width	FO Stems	Avg FO	Max FO	[F]aults	~F/G	Max
p35k.lg	2912	2229	41443	71	14	7072	6823	4.56	70	1447267	35	106
p45k.lg	3739	2550	38811	58	13	3903	7467	4.31	2048	615875	16	88
p77k.lg	3487	3400	65015	554	33	3869	12378	4.62	589	13012996	200	868
p78k.lg	3148	3484	68263	44	23	5278	18436	4.26	245	2972333	44	67
p81k.lg	4029	3952	106450	53	24	7324	29110	3.71	1593	3027057	28	78
p89k.lg	4632	4557	80963	109	23	5760	14507	5.05	400	2160841	27	146
p100k.lg	5902	5829	84356	102	15	5919	18486	4.20	2048	2068392	25	164
p141k.lg	11290	10502	152808	79	31	14810	31083	4.26	1238	5852552	38	117
p239k.lg	18692	18495	224597	177	16	32566	46465	4.50	2286	8244936	37	269
p259k.lg	18713	18495	298796	180	19	33582	65450	4.36	12162	10470416	35	273
p267k.lg	17332	16621	238697	72	26	21946	38705	4.36	451	4397442	18	96
p269k.lg	17333	16621	239771	72	26	21946	38706	4.39	1077	4439515	19	96
p279k.lg	18074	17827	257736	145	28	20011	52393	4.37	1536	9557864	37	251
p286k.lg	18351	17835	332726	149	38	20720	71449	4.28	2392	14219139	43	256
p295k.lg	18508	18521	249747	111	20	19394	41705	5.30	1986	4353646	17	146
p330k.lg	18010	17468	312666	69	24	22653	56441	4.53	1164	8965730	29	106
p378k.lg	15732	17420	341315	44	23	26390	92172	4.26	245	14861665	44	67
p388k.lg	25005	24065	433331	221	31	28259	94439	4.27	7019	24815827	57	329
p418k.lg	30430	29809	382633	205	27	36937	72481	4.33	1104	13159788	34	315
p469k.lg	635	403	96408	220	79	1467	14286	5.88	618	29548270	306	377
p483k.lg	33264	32610	444664	109	15	43206	106598	4.02	1159	20793512	47	170
p500k.lg	30768	30840	431439	167	20	29713	92303	4.23	1847	22125750	51	284
p533k.lg	33373	32610	586819	113	19	47781	141450	4.08	13621	24953268	43	174
p874k.lg	42897	42243	717288	239	39	49740	108022	4.34	126	33518299	47	359
p951k.lg	91994	104714	816072	139	12	124850	168494	4.17	8176	24380764	30	232
p1522k.lg	71392	68035	1104085	518	35	72077	140840	5.70	816	51598247	47	781
p2927k.lg	101844	95159	2408328	423	25	155113	708825	2.39	2178	82204260	34	630
p3188k.lg	154899	143395	2948057	718	73	194255	729155	3.54	6868	344905611	117	1155
p3726k.lg	160486	147661	3975023	495	30	257698	741291	4.29	1068	272612812	69	799
p3847k.lg	179317	174150	3200906	701	35	230256	601979	3.93	2371	143023091	45	1275
p3881k.lg	243000	294803	3788054	175	18	315257	718168	4.00	34120	126657414	33	273

Figure 5.3: General information of the NXP industrial benchmark circuits and DCD fault count. The number in each circuit's name serves as an approximate gate count.

Fault counts were also obtained for some industrial benchmark designs provided by NXP (see Figure 5.3). One can see that for most of the circuits the fault count per gate is rather constant as complexity rises, thus the fault count is assumed to scale linear with the circuit size — in contrary to the expected exponential behaviour.

> **Remarks:** Obtaining the results for the p3188k circuit required special treatment. The number of DCD faults of p3188k was computed, but the corresponding fault list was not stored due to memory constraints.

5.1.2 Evaluation

The double-cone delay model implements a path-based method to determine reasonable delay defect sizes for unit- and nominal-delay circuits by using slacks. Arbitrary gate delay defect sizes can be mapped to a unique SMALLDELAY fault. According to the results for ISCAS'85 and

NXP benchmark circuits, the presented DCD model produces an overwhelming amount of gate delay faults ranging from small- to gross-delay with linear fault count. In section 2 it has been shown that the computed set of faults is complete and that each fault is unique w.r.t. to the underlying model. However, clusters may occur, which make the faults sometimes hard to distinguish when using smaller sets of random patterns.

The transition fault and the quantized modification seem to be fast alternatives to produce smaller fault counts at variable precision, which are also less prone to clustering. While the transition fault modification is only capable of modeling gross-delay faults, the quantized model has its drawbacks regarding the computation of redundant sets and occasional overshooting for smaller circuits. Nevertheless, both modifications have been used in the following delay fault simulation experiments, as they represent different delay fault scenarios. For each of the tested circuits, three fault sets were generated in order to observe the observability and diagnosability of different fault categories: arbitrary delay defects (DCD), transition faults (TF) and the quantized model with $q = 0$ (Q0). The latter serves as *mean defect* size. Up to 10k faults were randomly extracted out of each set and have been stored in external files for later use in the diagnosis experiments.

5.2 Fault Simulation

To evaluate the performance of the simulator in measures of speed and throughput, a series of experiments was run on the ISCAS'85 circuits. Up to 100 sample faults were picked out of the generated sets, with each fault being simulated with 5120 random generated patterns. All simulations were performed on different machines equipped with commercial NVIDIA GeForce GTX 480 graphics cards (clocked at about 1.4 GHz). A maximum of 1000 MB device memory was used, which is about two-thirds of the GTX 480 capacity, and the number of transitions stored per signal has been restricted to 48. This transition cap turned out to be rather high and could have been lowered afterwards in order to increase the number of concurrent pattern bundles.

Figure 5.4 shows the average time to simulate a single fault, which was obtained by dividing the total required time for pattern conversion, simulation and pattern evaluation by the number of simulated faults. A selection of smaller NXP benchmark circuits has also been simulated using the same configuration as for the ISCAS'85 circuits above. Here, it was observed that — in contrary to the smaller ISCAS'85 circuits — only a few pattern bundles could be simulated per pass, due to the device memory restriction of 1 GB. Thus, providing more GPGPU memory would likely result in a linear gain in speed-up. In Figure 5.5 one can also see, where most overflows have occurred. Almost 500 calibrations had to be performed for simulating p77k and after caching the register sizes, the simulation speed has been more than doubled.

With an average fault simulation time of 0.22 s for ISCAS'85 and 32 s in average for the NXP circuits, processing the whole fault sets of the designs would become infeasible. According to

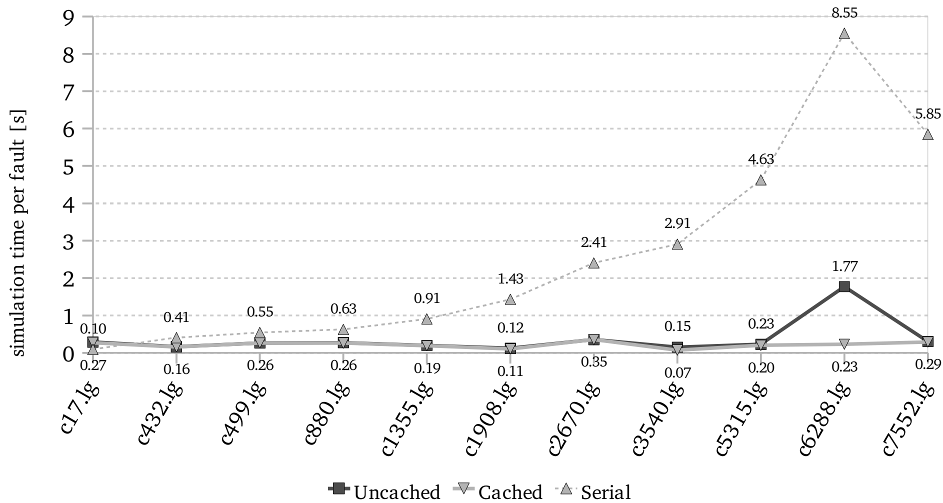

Figure 5.4: ISCAS'85 circuit simulation time. – Average simulation time per fault for 5120 patterns with and without cached register sizes.

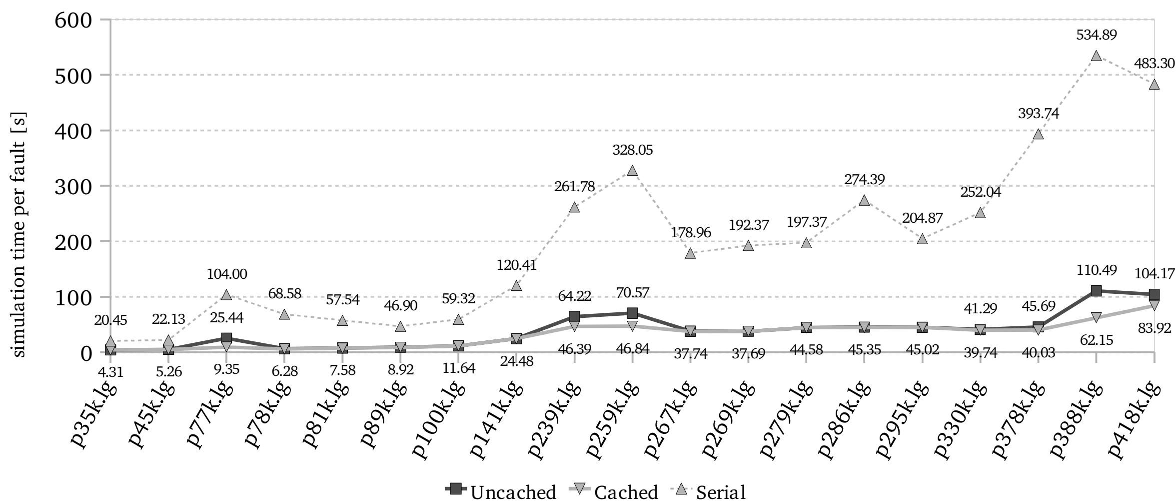

Figure 5.5: NXP circuit simulation time. – Average simulation time per fault for 5120 patterns with and without cached register sizes.

measurements, an average of 50% of the time is consumed by the framework doing pattern conversion and synchronizing the memory maps, another 40% is used by the GPGPU device for transferring memory. Only about 10% of the time is actually spent by the kernel. So, the communication overhead is a major drawback, but as a result, this seems to be a promising entry point for further optimizations regarding simulation speed.

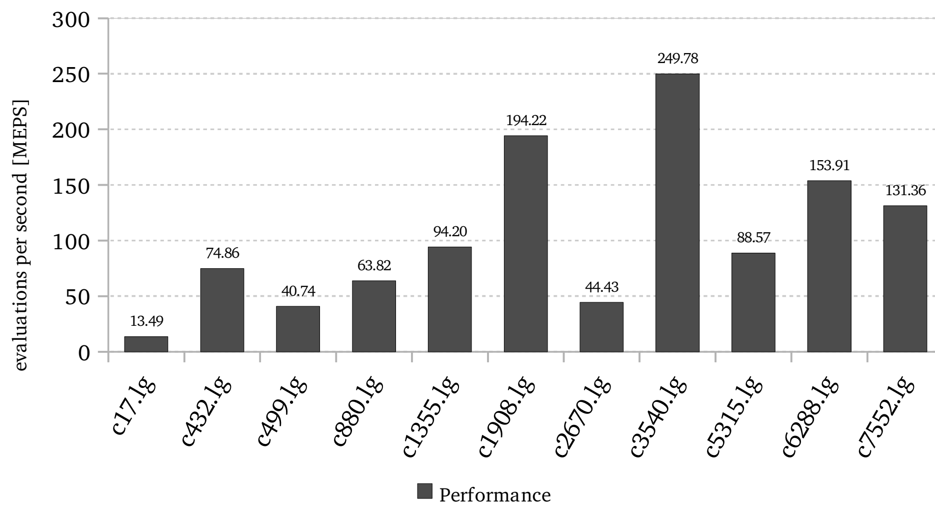

Figure 5.6: Average performance of the fault simulation for ISCAS'85 circuits measured in millions of evaluations per second (MEPS).

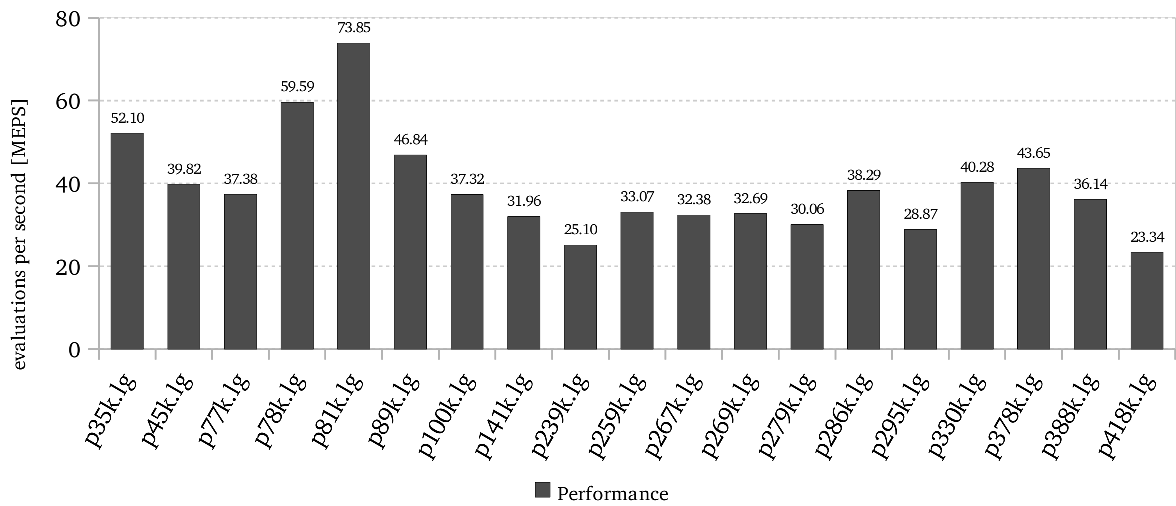

Figure 5.7: Average performance of the fault simulation for some NXP circuits measured in millions of evaluations per second (MEPS).

The general performance of the delay fault simulator has been evaluated as the amount of gate evaluations processed per second and it is measured in **millions evaluations per second (MEPS)**. The results for the simulated designs are shown in Figure 5.6 and 5.7.

5.3 Diagnosis

For evaluating the diagnosis, the experiments were set up as follows: Up to 100 gate delay defects of the previously generated sets were extracted, each being injected into a DUD and diagnosed separately. All DUDs have been exposed to the same 5120 random patterns to create faulty responses. If a device did not produce any erroneous outputs, it was excluded from the evaluation.

5.3.1 GSI Ranking

In the first diagnosis experiment, the DUD responses were analyzed with the stuck-at fault machine and the two stuck-at representatives. The suspects were ranked by **GSI (Gamma-Sigma-Iota)** according to the original paper [HW09]. Figure 5.8 shows the middle-ranks of the actual representatives in the ISCAS'85 circuits. It depicts how well the SMALLDELAYS of a certain model could be explained by any stuck-at fault at their locations. The lower the rank, the better the result. Suspects with a rank beyond or greater than 9 are considered as *not diagnosed*. The results show that transition faults could be well described via stuck-at representatives, whereas the performance of the DCD and the Q0 model diagnosis is lower. At this point, DCD and Q0 even behave similar. It was also observed that gradually increasing the defect size of a particular location caused the stuck-at suspects to ascend the rank list. Sometimes, minor fluctuations did occur, that caused a suspect to descend again for a moment.

Now, to further confine the location of the DUD defect with the resimulation, the top-suspects were unrolled (equivalence classes) and two SMALLDELAY fault representatives with **maximum** and **mean** defect size were generated for each stuck-at. Due to the time-consuming delay fault simulation, the maximum number of resimulations had been restricted to 100, and thus allowed a resimulation of up to 50 distinct stuck-at fault locations.

The resulting rankings after the delay fault resimulation are given in Figure 5.9. They show that the actual defect locations could be found well within the first top-ranked suspects by investigation of the equivalence classes. Nearly all detected transition faults could be diagnosed and about 90% of the suspects were located at the first rank. Again, the results of DCD and Q0 are similar as the *coarse* diagnostic performance did not change, but more actual suspects could be distinguished from the surrounding candidates and could even be identified at first level. Sometimes, the diagnosis of a DUD failed after the resimulation when the equivalence classes were too large and the actual suspect was cut off.

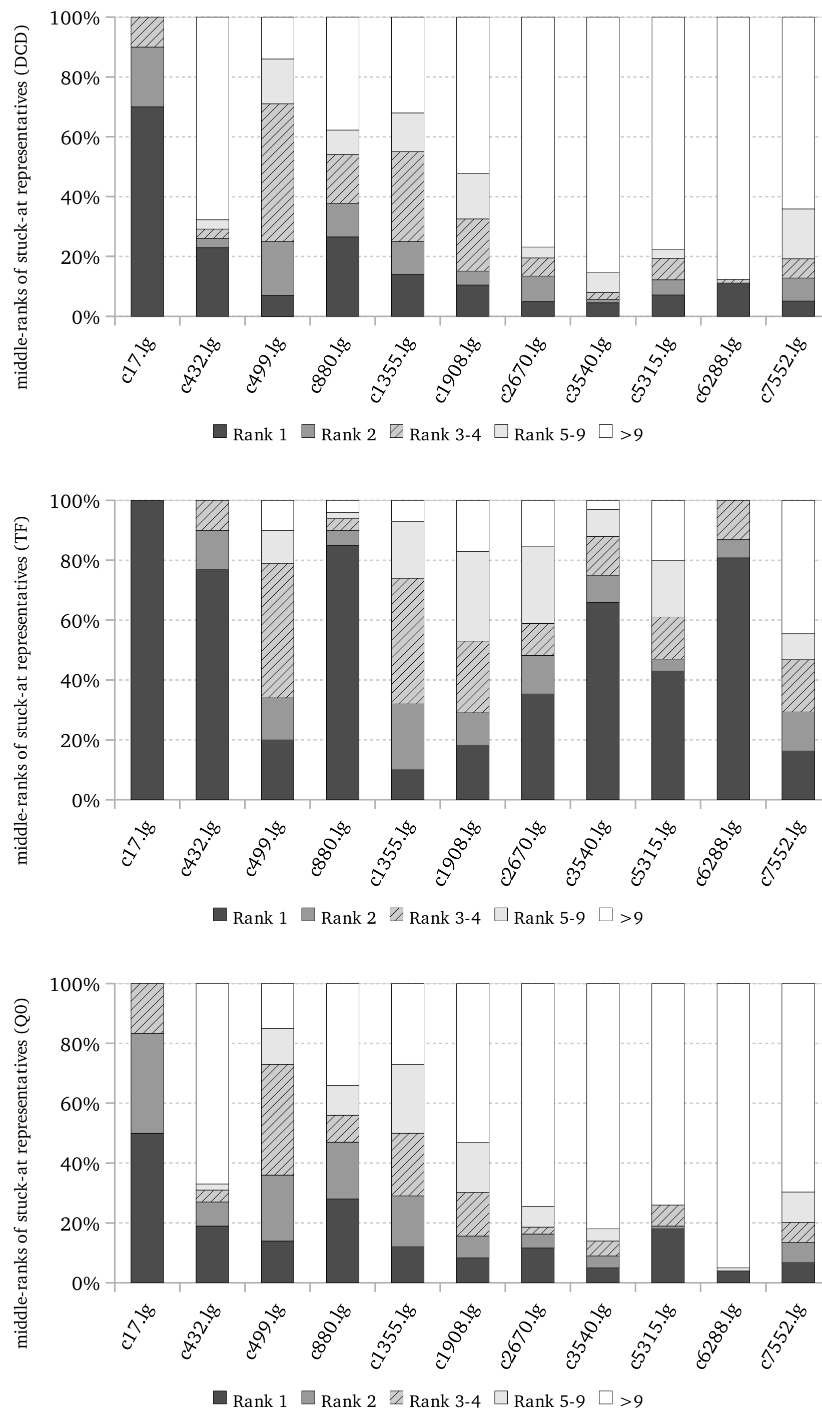

Figure 5.8: ISCAS'85 – Phase 1: GSI middle rank distribution of the stuck-at representatives for DCD, TF and Q0. Yet, these suspects represent equivalence classes and have to be further investigated.

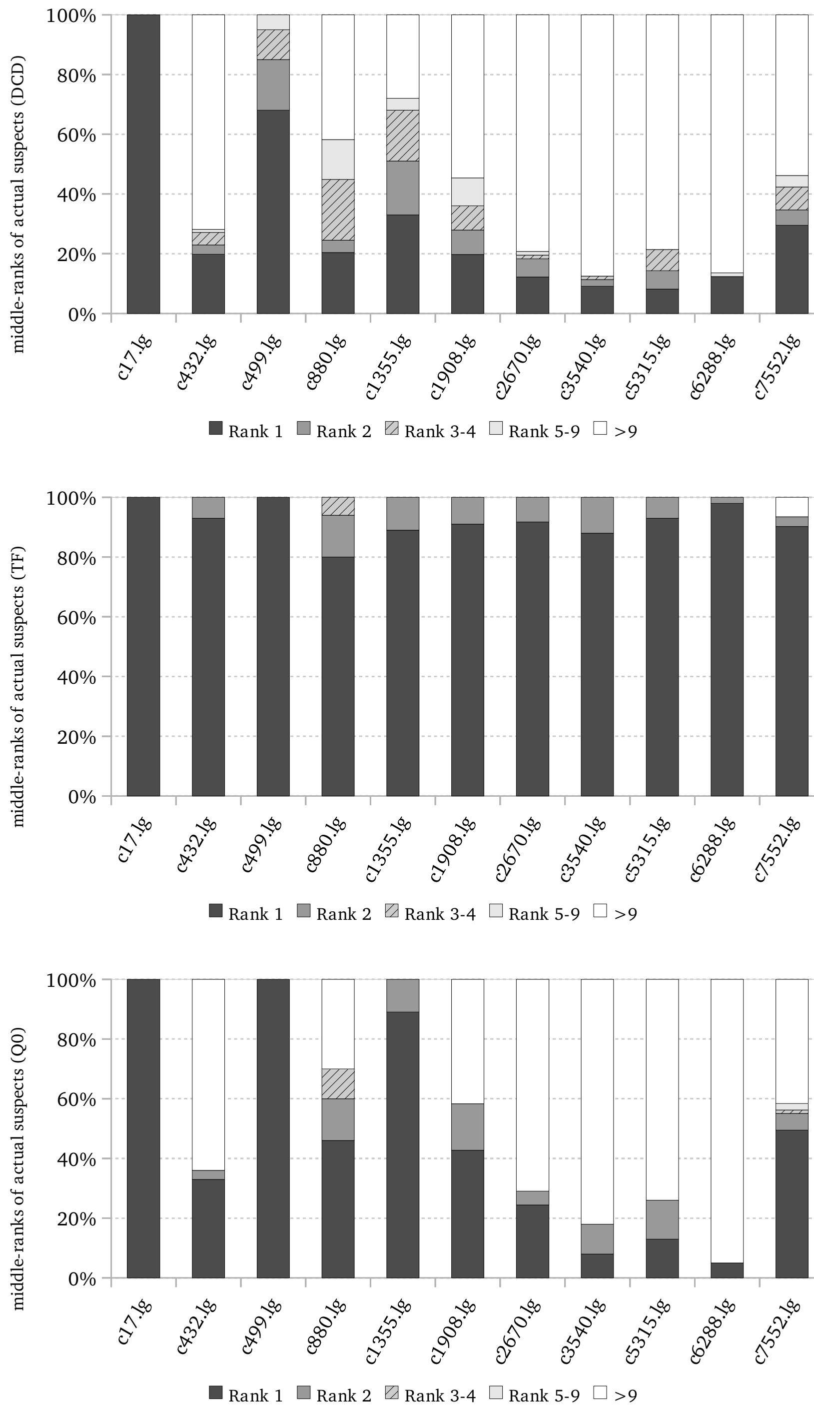

Figure 5.9: ISCAS'85 – Phase 2: GSI rank distribution of the exact locations after resimulation of the top stuck-at suspects equivalence classes.

5.3.2 SIG Ranking

By taking a closer look at the evidences, it has been observed that the γ_Π $(= \sum \gamma_\pi)$ components of about 50% of all actual stuck-at suspects in either the DCD or the Q0 model were greater than zero (in the TF model, all suspects still had $\gamma_\Pi = 0$). However, the chosen GSI ranking sorts the evidences in increasing order of γ_Π.

Now, if a pattern π causes $\gamma_\pi \neq 0$, then there are both explained ($\sigma_\pi > 0$) and mispredicted pins ($\iota_\pi > 0$), due to $\gamma_\pi = \min(\sigma_\pi, \iota_\pi)$. Figure 5.10 depicts the responses, that cause gamma to increase. During the experiments, this usually happened for delay faults of smaller size when only a subset of the sensitized POs in the output cone has been violated.

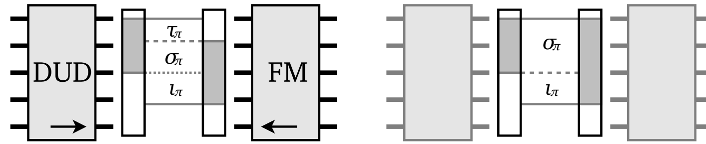

Figure 5.10: Pattern responses of stuck-at suspects with $\gamma > 0$ occurred for smaller and medium DUD defects.

By having a closer look at the evidence components, one can conclude that for $\gamma_\Pi = 0$ the following *special cases* of suspects can appear:

- If $\sigma_\Pi \neq 0$ and $\iota_\Pi = 0$, this will be a possible candidate worthwhile to investigate (e.g. perfect match).
- If $\sigma_\Pi = 0$ and $\iota_\Pi \neq 0$, the faults have no failing bits in common and are unlikely or even bad candidates.
- If $\sigma_\Pi = 0$ and $\iota_\Pi = 0$, then the suspect has either not been activated, or it is located at a disjunct location. In the experiments, it was assumed that the latter is more likely to appear.

It is obvious that all possible and actual candidates with $\gamma_\Pi \neq 0$ will get cut out by these special cases when using GSI order for ranking, no matter how good the failing-bits of DUD and FM have actually matched. Thus, in the next step the focus was set on σ_Π and ι_Π rather than γ_Π by changing the order to **SIG (Sigma-Iota-Gamma)**. The decision was also based on the idea that prioritizing higher σ_Π yields suspects with more matching failing-bits, and lower ι_Π lead to more constrained sets of suspects with less mispredicted POs.

After the SIG ranking has been applied, the results of the first stuck-at phase did only show slight differences. Some suspects were ranked lower, but were still too similar with other *unrelated* faults. However, the final resimulation phase (see Fig. 5.11) could now capture many

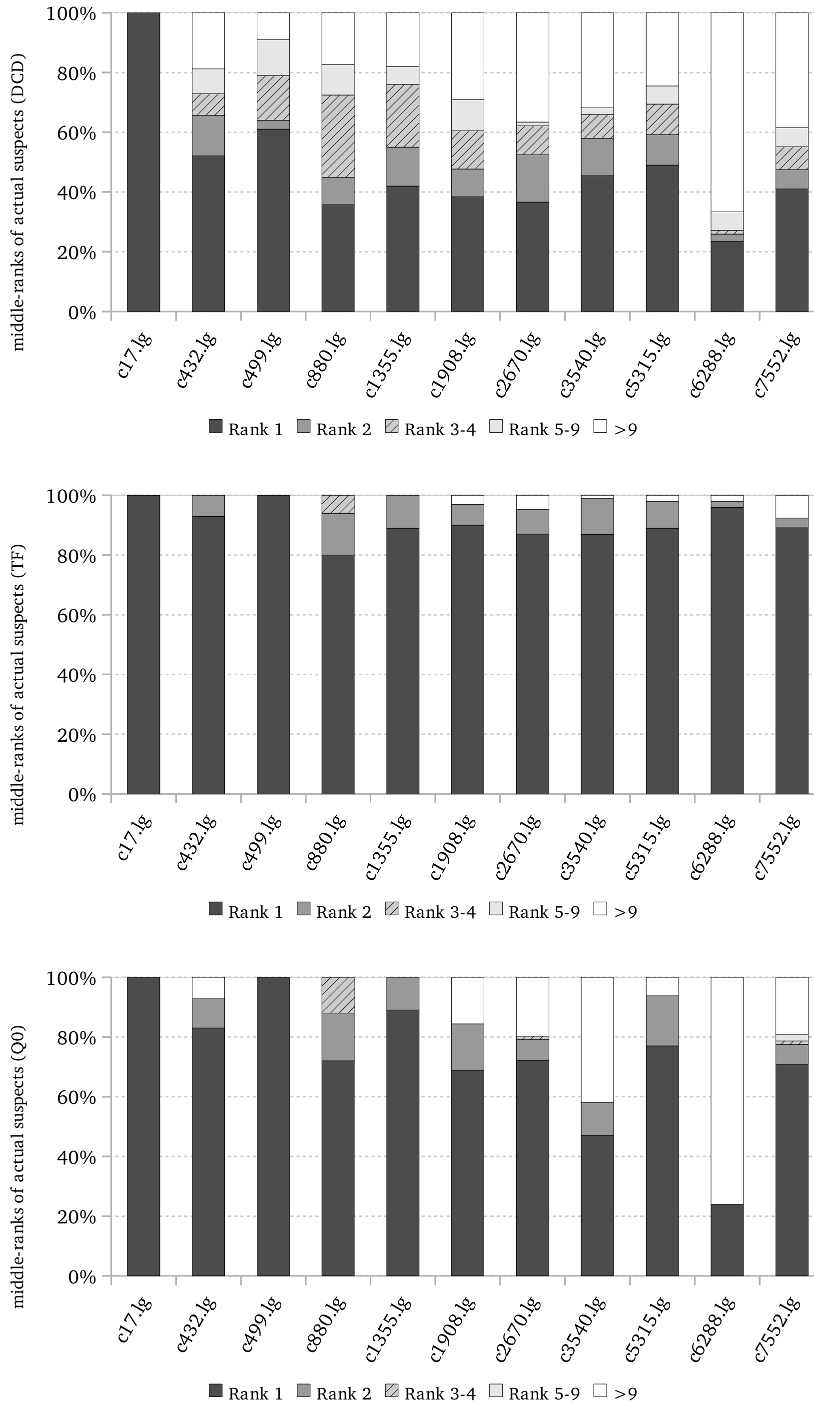

Figure 5.11: ISCAS'85 – Phase 2: SIG rank distribution of the exact locations after resimulation. Now, the actual culprits could be better distinguished from other faults.

of the actual culprits and the resulting evidences were better distinguished from each other. Compared to the stuck-at phase, the average middle-rank has been lowered, especially for the TF and Q0 model. The high ratio of first-rank suspects for these models can be explained by the choice of the delay defect **representatives** during resimulation, since in the experiments they caused perfect matches at the appropriate stuck-at locations ($\iota_{\Pi} = 0$, $\gamma_{\Pi} = 0$). Thus, a variation of these reference sizes will most likely lead to different results and may sometimes even highlight the wrong suspects. Since during resimulation all processed representatives were computed *on-the-fly*, their number and their defect size could also be adapted according to more complex functions in order to create better matches. According to the results of the DCD and the Q0 model, the SIG ranking has improved the overall diagnosability of these small delays.

5.3.3 Reasoning with Evidences and Representatives

In general, the outcome of the comparison between a delay fault and a stuck-at fault during diagnosis strongly depends on the observability of both faults and their individual **activation conditions**, as the resulting evidence is influenced by the active failing-bit overlap of the output-cones at both sides. If the fault of neither DUD nor FM got activated for a pattern, the overall evidences will not improve. In fact, the only case where suspect evidences actually *can* improve is when a transition at the fault-site is triggered (which is already rare enough) **AND** the corresponding stuck-at fault with the correct *polarity* is activated at the same time in the PV of the pattern sequence. Both conditions have to be met in order to maximize the failing-bits, that are in common. However, if these conditions are fulfilled less often, σ_{Π} of the actual suspects will stay low and it may easily happen that unrelated faults will be placed in front of the ranking — e.g. by activating stuck-at suspects with huge sensitized output-cones, that would increase σ_{Π} dramatically. If this happens more frequently, the real suspect would not stand a chance in getting ranked better and it would be hard to distinguish from false candidates.

To further tighten the set of matching candidates, POINTER introduced ι_{Π} as the number of mispredictions produced by the suspects. As it could be observed by the evidences during the experiments, a higher ι_{Π} is not a criterion for weeding out, since its behaviour differs from fault to fault and typically increases for lower defect sizes.

> **Remarks:** The stuck-at evidences suggest further investigations of the suspects regarding the weight of individual evidence components. For example: Finding an appropriate function for ranking the evidences with respect to the overall failing pin count. Besides, the use of ATPG for deterministic test-pattern generation should greatly assist in fault activation and could help to improve suspect evidences.

Fault Polarity and Evidence Decomposition

Regarding the activation conditions of the stuck-at representatives of a delay fault, it was noticed that for any pattern either the *stuck-at-0* or the *stuck-at-1* was active — or none of them — but never both faults at the same time. Also, the delay fault activation is not tied to a single pattern, but to certain pattern sequences of IVs and PVs, where the PV can pull the node's signal value to either 0 or 1. The fault activation still depends on its previous signal value, that was set by the IV. As a consequence, the polarity of the stuck-ats caused the evidence of each fault-site to **decompose** and split into two smaller and more *insignificant* evidences. These are harder to distinguish, as the probability of being overruled by outliers is higher if their total evidence is not evenly distributed over the two stuck-at representatives. Also, recall that these stuck-at suspects are still *classes*, where the actual location is revealed only after an examination with resimulation.

Therefore, all representative evidence pairs for each fault-site have been added in order to form *merged stuck-at evidences*. A merged stuck-at pair can be viewed as an **unconditional flip**, that eliminates the activation condition of the fault polarity, but still maintains the actual location information (intersection of the equivalence classes). Thus, by merging the representatives of each location their evidences will be strengthened — especially the ones of the actual suspects. Experiments have shown that all suspect evidences of the first phase improved greatly compared to the previous analysis with the decomposed evidences.

Figure 5.12 shows the final outcome after resimulation of the merged evidences. The diagnosability has further been increased — especially for the TF and Q0 faults, which is presumed due to the choice of the SMALLDELAY representatives.

Anyway, the decomposed stuck-at evidences might still be useful in order to distinguish between **slow-to-rise (STR)** and **slow-to-fall (STF)** delay faults by using the approach as proposed in the transition fault model of Waicukauski *et. al* [WLRI87], where slow rising or falling transitions are represented by stuck-at-0 or stuck-at-1 faults respectively. All suspect evidences would look accordingly: For any SMALLDELAY with a certain transition polarity, the activating PVs of the pattern sequences will also enable the corresponding stuck-at representative, whereas the other stuck-at would not get activated, since good simulation and faulty value are equal (Thus leaving all faulty DUD pins unexplained). However, by using the decomposed stuck-at representatives, all reported suspects are equivalence classes again and have to be further investigated by using resimulation to find the actual defect location.

5.4 Summary

This section has outlined the experiments, that were performed throughout this work. It showed that the proposed DCD defect size generation method produces a huge, but still linear fault count for the circuits. Regarding the current performance of the delay fault simulator,

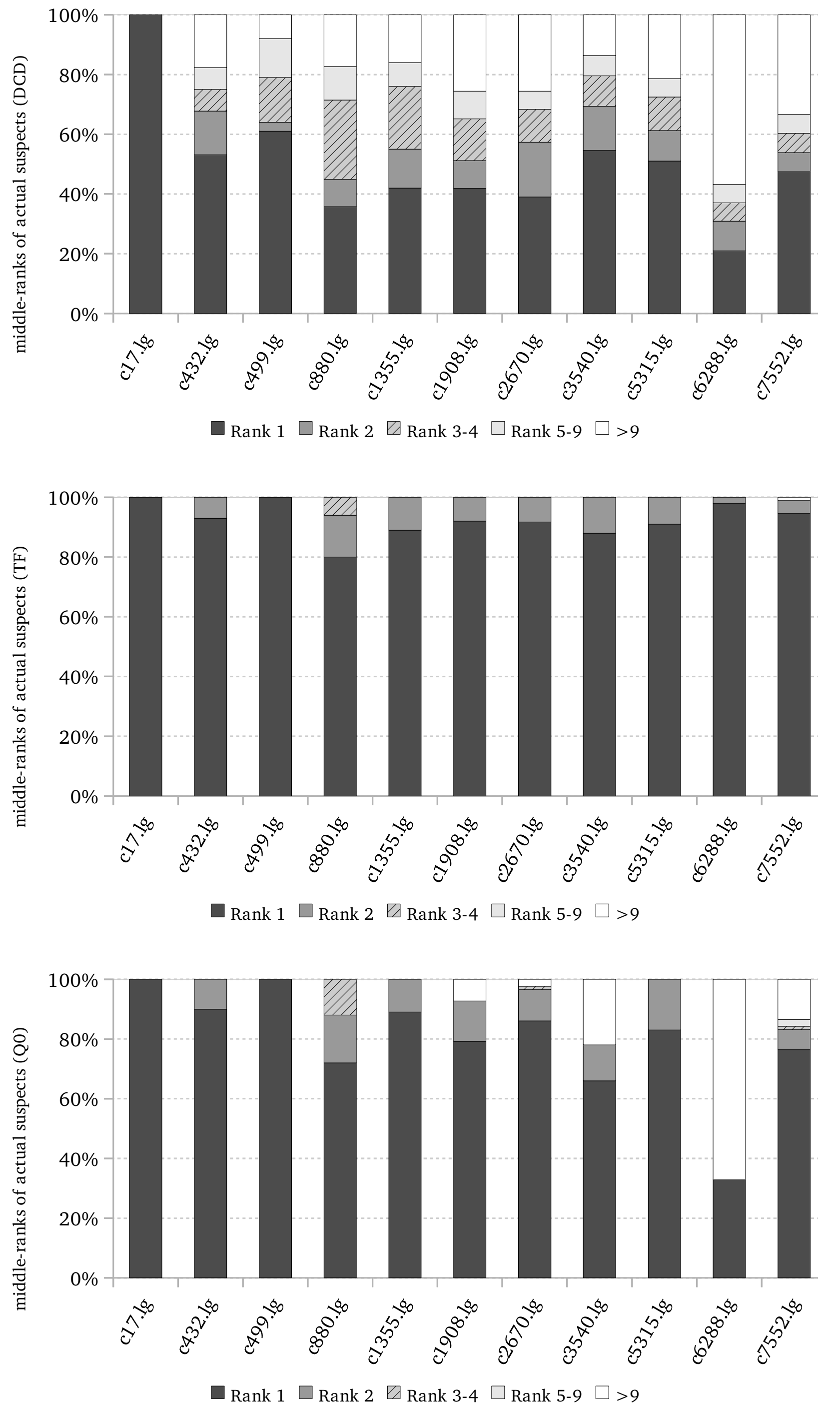

Figure 5.12: ISCAS'85: Diagnosis results after resimulation of the merged stuck-at representatives.

an exhaustive processing of the whole fault sets becomes infeasible for larger circuits. Hence, the use of simulation had to be reduced as much as possible. Although the simulator showed a large communication and memory transfer overhead during the elapsed time, it has still offered a significant speed-up than compared to the serial execution.

During the diagnosis experiments, the behaviour of the POINTER analysis was observed for different defect generation models — from arbitrary small delays to transition faults. The results of the presented delay fault analysis approach have indicated that larger defect sizes are easier to detect and better to diagnose than smaller ones. After confining the set of candidates, the resimulation could be used in order to enable a more fine grained location of equivalent suspects. The general outcome, however, does still depend on the choice of representatives as well as the applied ranking scheme of the evidences. For the average SMALLDELAY fault, the SIG ranking has proven best. The latest experiments have shown that merged evidences fit better to the properties of the SMALLDELAY than single stuck-ats, due to the dropped activation conditions of the fault polarity.

Figure 5.13 summarizes the results of the different approaches for arbitrary defect sizes (DCD). By using the **merged evidence resimulation** approach, the best average results were achieved: Nearly 78% of all DUD defects were *successfully* diagnosed, 59% of which could be identified *perfectly* at the first rank. A comparison of the approaches on the different defect size models is given in Table 5.1 below, which sums up all diagnosed DUDs of the experiments.

Approach	DCD	TF	Q0
GSI-Resim	43.73% (24.65%)	99.39% (91.45%)	50.56% (41.56%)
SIG-Resim	72.41% (43.62%)	98.07% (90.12%)	81.58% (70.52%)
SIG-Merged	**77.75**% (41.44%)	97.45% (79.74%)	77.99% (42.89%)
SIG-Merged-Resim	77.54% (**45.69**%)	**99.90**% (**91.75**%)	**88.74**% (**77.48**%)

Table 5.1: ISCAS'85 diagnosis results of the experiments showing successfully diagnosed DUDs and first-ranks (in brackets). All numbers are with respect to the total DUD count.[a]

[a]Faulty devices only. – DCD: 917 out of 1010, TF: 982 out of 1006, Q0: 977 out of 1006.

To demonstrate the scalability and the effect of the resimulation diagnosis approach (with decomposed evidences) on larger designs, it has been successfully applied to the NXP circuits as seen in Figure 5.14. Here, the coarse diagnostic performance is similar to the smaller ISCAS'85 circuits (using decomposed evidences). However, it was observed that the average rankgroup was larger, causing the middle-ranks to increase. This might be avoided by selecting more representatives per suspect to distinguish better between candidates with similar evidences and to find perfect matches.

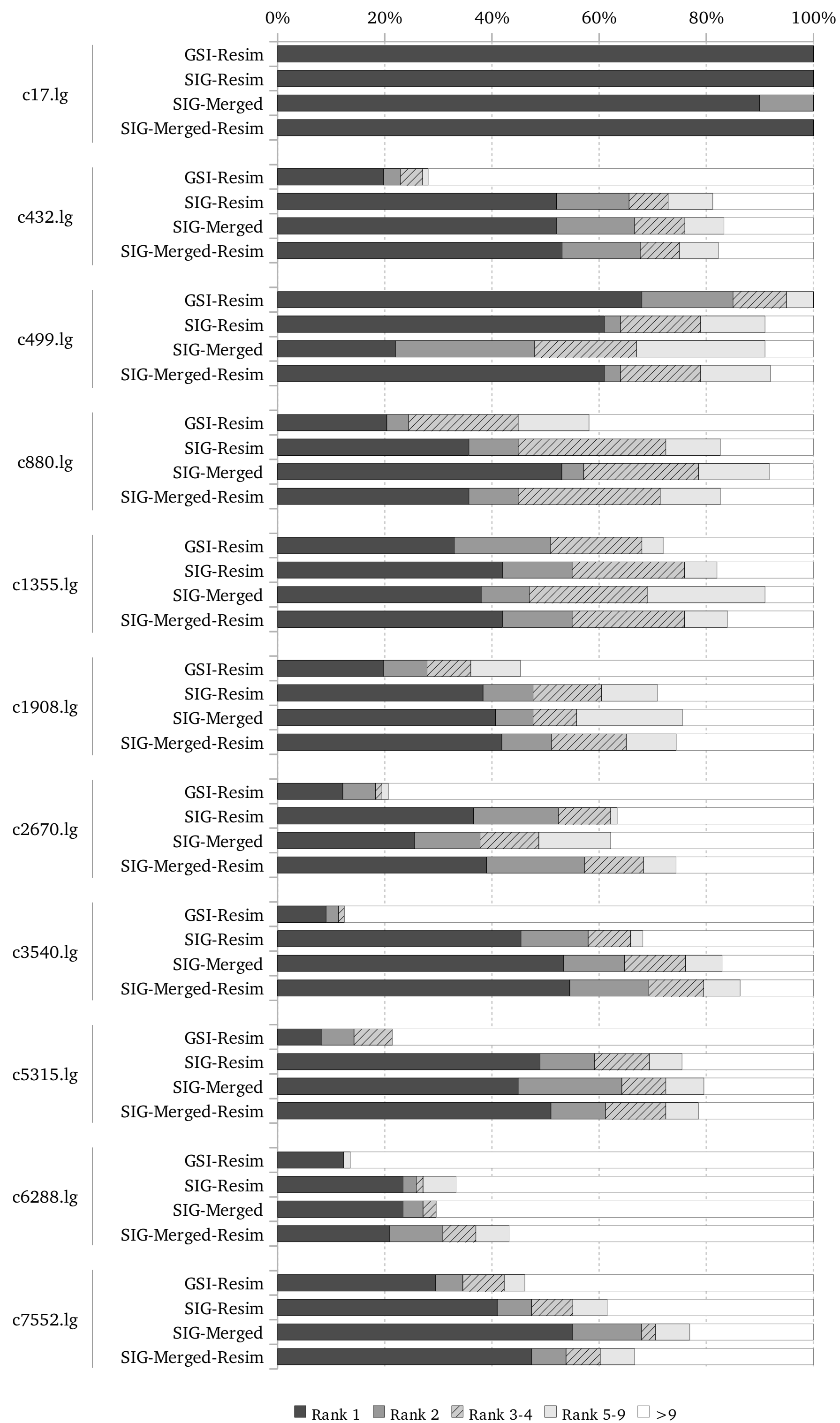

Figure 5.13: Summary. – All results of the different diagnosis approaches put in juxtaposition for the DCD model.

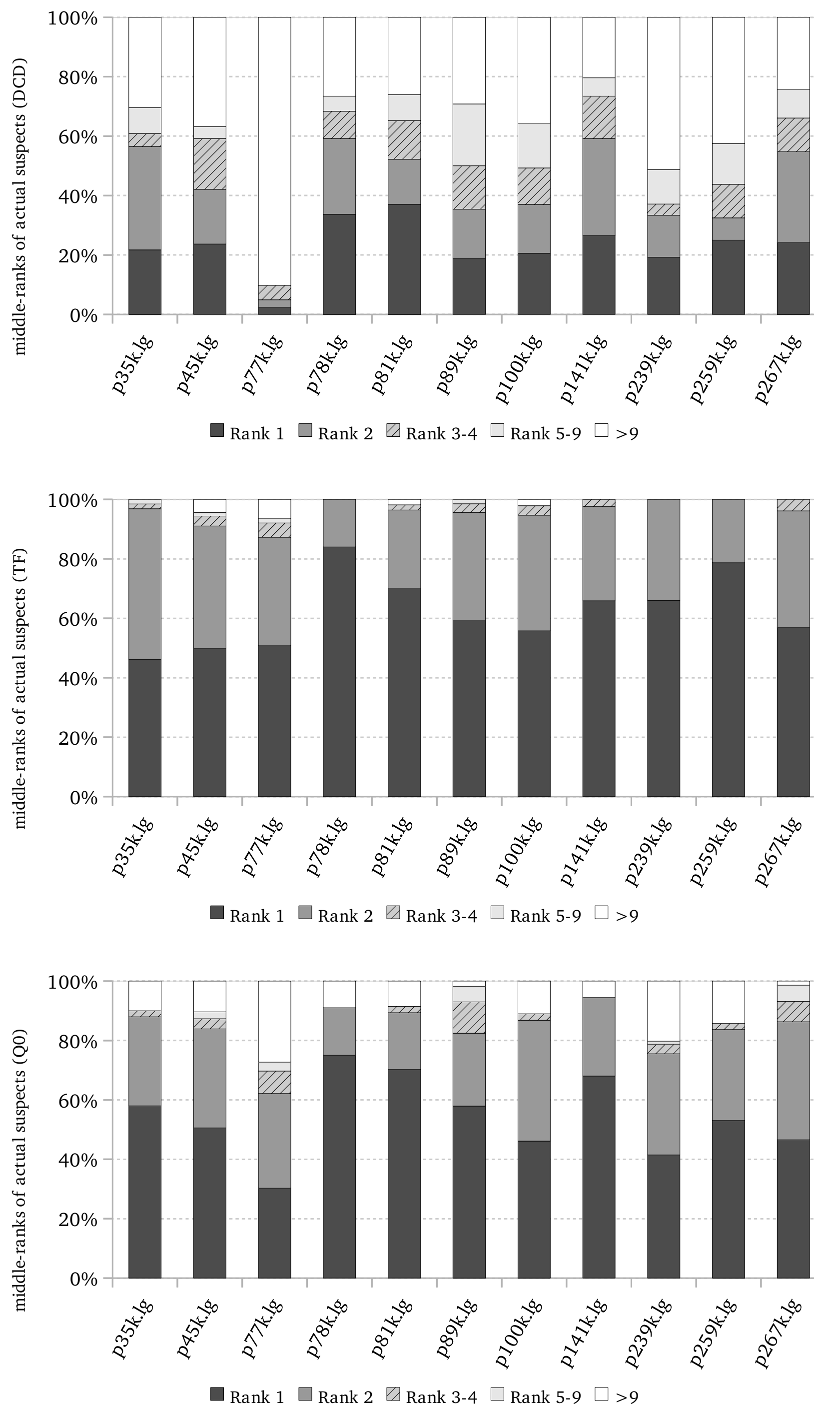

Figure 5.14: Scalability. – The decomposed resimulation approach demonstrated on larger circuits (SIG-rankings applied). The size of the individual rank groups has grown, but the coarse diagnosis result looks similar to ISCAS'85.

This page is intentionally left blank.

6 Conclusion

The goal of this thesis was to implement a fault simulator for simulating small delay faults on CUDA-capable devices, and its integration into a diagnosis framework for evaluation of the POINTER pattern analysis approach. Further, a method has been proposed, that is able to compute the sets of all small delay faults, which may occur in circuits with nominal- and unit-delay, by constraining the defect sizes through lower and upper bounds.

The implemented simulator performs a gate-level time simulation for each small delay fault, that processes the circuits in a level-wise fashion. It is able to cope with arbitrary large designs and uses waveforms as signal representations to model the precise timing behaviour of the circuit logic. By exploiting the available structural- as well as data-parallelism, many gates are concurrently evaluated for multiple patterns at once by different threads on the GPGPU. Thus, simulation with many utilized threads on the GPGPU showed a significant speed-up than compared to the serial execution. According to measurements, a large fraction of the elapsed simulation time was still spent during pattern data transfers between processes and the device. Further improvements to minimize this overhead should fully unlock its potential. Current CUDA devices with up to 6 GB of memory[1] will be able to store even more waveforms for concurrent simulation.

Diagnosis experiments with random input stimuli have shown that medium-sized and larger delay faults could very well be diagnosed by the POINTER analysis approach using simple stuck-at faults as references. Additional resimulation improves the average diagnosis results and further confines or identifies the actual suspects. Yet, the analyzer gets rather uncertain when defects of smaller sizes occur, as they produce less failures and cause the candidates to be harder to distinguish. Thus, for future diagnosis experiments, deterministic pattern generation and weighted rankings could be used in order to support the evidence collection of those smaller defect sizes.

Since delay faults are becoming a more and more important topic in VLSI design, due to the ever increasing circuit speed and reliability requirements, further optimization and evaluation of delay fault simulators and diagnosis are worthwhile to investigate.

[1]NVIDIA Tesla M2070

This page is intentionally left blank.

A ADAMA `fsample`

In the following, the basic functions and parameters are explained how to generate fault lists with `fsample`. The plain execution of `fsample` is done by typing:

```
> adama fsample <model.lg> {options}
```

Below, all SMALLDELAY-related **options** are given, that are required to compute the fault sets.

-m dfault: invokes the fault-list generation for SMALLDELAY sets.

-fm <*value*>: Specifies the fault model:

0 Double-Cone Delay model (default),

1 Transition Fault modification, or

2 Quantized modification.

-s <*value*>: Restricts the final fault-list to a number of random faults, specified by *value*. Useful, if just a subset of faults is required.

-q <*value*>: Quantization parameter for use with the quantized DCD-modification. The default value is 0.

-o <*filename*>: Specifies the name of the output fault-file. The file extension is by convention '*.fl*'. If the option is omitted, the fault-file will be stored as '*model.lg.fl*' in the current execution directory.

Executing `fsample` for the example circuit c17, in order to get the full DCD fault-set, could look like:

```
> adama fsample c17.lg -fm 0 -o c17.lg.fl
```

The resulting fault file is shown in listing A.1. Each line contains a fault, stored as <*Gate.name*>/<*Port*>␣<*Size*>. One can easily identify some of the listed faults, with the locations given in Figure 2.3 of section 2. The file allows manual changes of defect sizes for experimental purposes. However, one should verify after changing, that each listed gate does also reside in the circuit or the simulator will not be able to inject the fault.

Listing A.1 Content of *c17.lg.fl*.

```
SDF { NAND_2_0/0 5 }
SDF { NAND_2_1/0 3 }
SDF { NAND_2_2/0 5 }
SDF { NAND_2_2/0 3 }
SDF { NAND_2_3/0 3 }
SDF { NAND_2_3/0 5 }
SDF { NAND_2_4/0 5 }
SDF { NAND_2_4/0 3 }
SDF { NAND_2_5/0 5 }
SDF { NAND_2_5/0 3 }
```

Although the **port** value will have no effect for the simulation — since the implemented simulator only considers faults at gate outputs (port 0 is gate output) — it was retained for completion. The port will be ignored during parsing.

All fault files can be loaded by the SMALLDELAYSET class with the provided PARSEFAULTS(*filename*) function.

B ADAMA `wave/diagnose dfcuda`

Assuming that the `wave` executable (`./wave`) is available in the current execution directory, the basic call of either the delay fault simulator or the delay fault diagnostic tool is done by simply typing:

```
> adama (wave|diagnose) dfcuda <model.lg> {options}
```

Without any options, the command will start the chosen task and instantly begin to generate the full fault list according to DCD and perform a fault simulation or a fault diagnosis of the design (*model.lg*) using 64 random patterns for each fault. Therefore all relevant options, that can be used to adjust the settings, are listed below:

-fm <*value*>: Specifies the delay defect generation model:

- *0* Double-Cone Delay model (default),
- *1* Transition Fault modification, or
- *2* Quantized modification.

-q <*value*>: Quantization parameter for use with the quantized DCD-modification. The default value is 0. Other models will not be affected by this parameter.

-e <*filename*>: Specify the path and the name of the CUDA `wave` executable. By default `./wave` is assumed to be in the current directory.

-d <*value*>: Select CUDA device. By default, device 0 is chosen.

-m <*value*>: Adjust the maximum amount of GPGPU memory to be allocated.

-r <*value*>: The number of random patterns that have to be simulated for each fault.

-fl <*filename*>: Specifies the name of the input fault-file. This option makes the choice of a fault model via **-fm** void.

-cl <*value*>: Crops the fault set (either generated or loaded) to a number of faults. A plain number x will choose x random faults of the set. A range *from a to b* (excluding b) of faults in the set can be expressed by `-cl a:b`.

-c <*value*>: Specifies the maximum transition count. The number should be greater than 6. By default the cap is 34.

-cs <*reg_file*>: Use input file for caching register sizes.

-mr <*value*>: Specify the maximum number of suspect resimulations per DUD (default 0). *Diagnosis only*.

Examples

A diagnosis of 100 defects of a pre-computed fault-set for the model *c3540.lg* with 5120 random patterns and 50 resimulations per DUD would look like this:

```
> adama diagnose dfcuda c3540.lg -e ./wave -fl dcd3540.fl -cl 100 -r 5120 -mr 50
```

Listing B.1 Sample output of a DUD diagnosis (c3540).

```
2019 ...
2020 0000465.457 [--] Defect #66 of 100: SDF { AND_2_1554/0 42 } (11%) // DUD defect (% size)
2021 0000465.457 [--] SA Representatives StuckAt { AND_2_1554/0 0 } StuckAt { AND_2_1554/0 1 }
2022 0000465.573 [--] FailingPatterns 4 // DUD failing bits and patterns
2023 0000465.573 [--] FailingBits 8
2024 0000465.573 [--] StuckAt Analysis (de.uni_stuttgart.iti.adama.analysis.SIGCompare)
2025 0000465.574 [DD] Representatives evidences:
2026 0000465.574 [DD] Suspect 9: StuckAt { AND_2_1554/0 0 }, gam 6 sig 8 iot 23 det 21
2027 0000465.574 [DD] Suspect 3125: StuckAt { AND_2_1554/0 1 }, gam 0 sig 0 iot 93 det 90
2028 0000465.574 [--] BestEvidence gamma 6 sigma 8 iota 6 tau 0 (StuckAt { BUF_1_1697/0 0 })
2029 0000465.574 [--] Classification Complex
2030 0000465.574 [--] RankGroup 1 ... 5 : StuckAt { BUF_1_1697/0 0 } StuckAt { NAND_2_1732/2 1 }
         StuckAt { NAND_2_1723/2 1 } StuckAt { NOR_2_1601/1 0 } StuckAt { BUF_1_1629/0 0 }
2031 0000465.574 [--] RankGroup 6 ... 6 : StuckAt { BUF_1_1631/0 0 }
2032 0000465.574 [--] RankGroup 7 ... 7 : StuckAt { AND_2_1734/1 1 }
2033 0000465.574 [--] RankGroup 8 ... 9 : StuckAt { NAND_2_1691/2 1 } StuckAt { NAND_2_1674/2 1 }
2034 0000465.574 [--] RankGroup 10 ... 10 : *StuckAt { AND_2_1554/0 0 } // real culprit is marked
2035 0000465.578 [DD] Merged StuckAt Evidences (2033). // (total evidences after merging)
2036 0000465.578 [DD] SmallDelay Resim (Top-50 Suspects)
2037 0000472.498 [--] SDFGroup 1 ... 2 : SDF { AND_2_1553/0 68 } *SDF { AND_2_1554/0 67 }
2038 0000472.498 [--] SDF_BestEvidence gamma 1 sigma 8 iota 79 tau 0 (SDF { AND_2_1553/0 68 })
2039 0000472.498 [--] SDF_ActualEvidence gamma 1 sigma 8 iota 79 tau 0 (SDF { AND_2_1554/0 67 })
2040 0000472.498 [--] ActualRank 10    // single stuck-at suspect ranking
2041 0000472.498 [DD] GroupRank 5
2042 0000472.498 [--] RankGroupSize 1
2043 0000472.498 [--] MiddleRank 10
2044 0000472.498 [--] SDF_ActualRank 2 // final suspect rankings
2045 0000472.498 [DD] SDF_GroupRank 1
2046 0000472.498 [--] SDF_RankGroupSize 2
2047 0000472.498 [--] SDF_MiddleRank 2
2048 ...
```

C The Internals of `wave`

The `wave` simulator mainly consists of two parts: a state machine and a **kernel** code. The state machine is used for communicating with the ADAMA framework, from which it is controlled by sending and receiving commands through standard input/output streams between the processes. The kernel code itself represents a single-frame simulation, which spawns a number of threads in the device grid. The number of threads depends on the amount of patterns processed in the simulation pass as well as the size of the current frame. The memory required to store intermediate signals will also limit the maximum allowed patterns per pass. In order to simulate a complete circuit, the kernel will be executed for each frame.

Registers

Intermediate results will be stored as **registers**, which are merely placeholders for signal values. Each gate-ID is mapped to a register-ID according to a scheduling algorithm, that minimizes the maximum amount of required registers. An allocation persists until its signal value is not required anymore. The register scheduling is done once at the beginning by the class FRAMEFORMATTER.

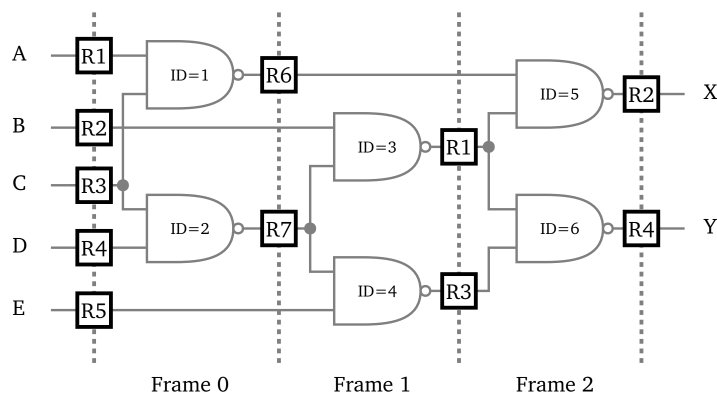

Figure C.1: Example register allocation for our c17 circuit. The number of used registers in this schedule is 7.

Figure C.1 shows an example scheduling for c17. All input signals are assigned to the first level. Preceding signals overwrite the registers of those, which are no longer required.

The purpose of the register translation is to minimize the amount of memory required to store intermediate results, without wasting too much storage. Since we simulate multiple patterns at once, the total count of registers as well as their respective size have significant impact on the number of patterns that can be simulated per pass, due to the fixed size of memory.

Memories

The wave simulator uses two types of mapped memories: a *frame memory* and a *wave memory*. The complete *circuit data* is stored in the **frame memory**, as depicted in Figure C.2. It is currently divided into the maximum allowed frame count of 1024 frames. Each frame offers space for 4000 gates. The limitation of the gate count is explained by the very limited device constant memory of 65536 bytes, where we store the gates during simulation, since our gate data structure is 16 bytes large ($16 \times 4000 = 64000$).

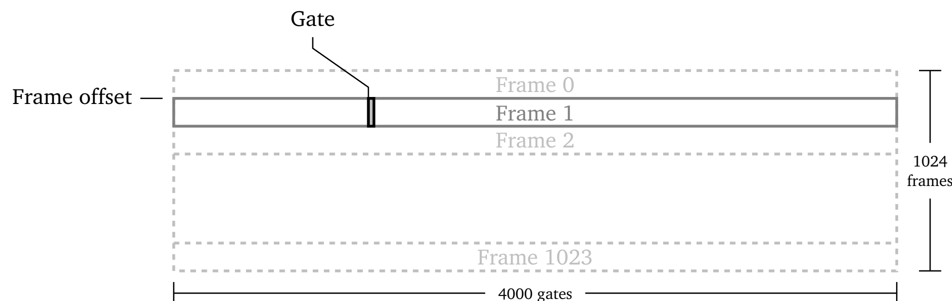

Figure C.2: Frame information stored in memory. – Up to 4000 gates are stored per frame. The final required gate memory is $65,536,000$ bytes large, due to the maximum frame count of 1024.

A more detailed view of the data structure is shown in Figure C.3. Each gate n_Z belongs to a signal Z, which is used as a 4 byte identifier for its corresponding register. Its fan-in register IDs (maximum of two) are stored at byte positions 4 and 8. A gate has a delay *d* stored as SHORT and a type *op*, which simply expresses the gate function ϕ_Z according to Table C.1. The *cap* value indicates the associated register size with the number of maximum transitions, that can be stored in the waveforms for that particular gate.

Every time the kernel is called to process a certain frame, the corresponding frame data will be copied into the constant memory via CUDAMEMCPYTOSYMBOL() prior to the actual kernel execution.

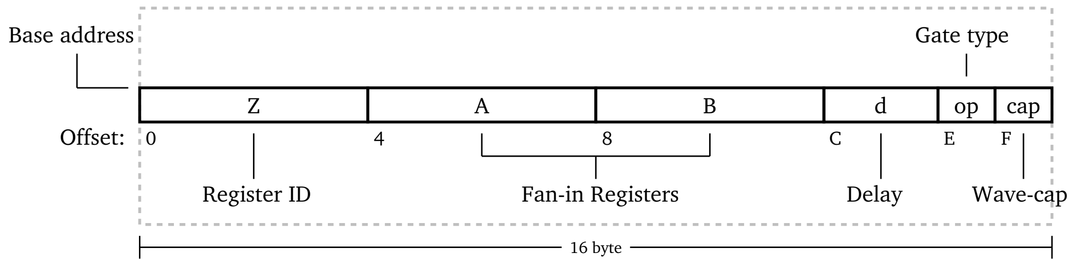

Figure C.3: Gate data structure in memory. The base address is a multiple of 16.

Gate type	AND	NAND	OR	NOR	XOR	XNOR	BUF	NOT
Op-value	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9

Table C.1: Mapping gate type to *op*-value.

The **wave memory** is used to store the signal information. In contrast to the frame memory, it is fully read, but only once at the beginning of each simulation pass in order to get the input stimuli. It is written to at the final end to store the output responses. The provided stimuli are first arranged in **bundles**, with each bundle consisting of up to 32 patterns. Depending on the available memory and the number of registers used, a certain number of bundles will be taken in each simulation pass and converted to *waveforms* and then stored in the memory map. So, the provided pattern list may require multiple runs in order to simulate all of its patterns.

Before the first kernel call is executed, the simulator will load the memory map into the CUDA device **global memory** to store the input stimuli at locations determined by the signal's corresponding *register ID* and the current input pattern index.

Figure C.4 illustrates the mapping of *register* $\times$ *pattern* to a memory location. The mapping is equal for both the global device memory and the memory map.

When the kernel is finally executed, each spawned thread will first look-up its fan-in register IDs, which in turn are used to compute the locations of the stimuli values that correspond to the thread. After evaluation, the resulting signal information is stored at the gate's indexed location and the kernel call will be repeated for the next frame. Eventually, the simulation pass is over and all the responses are written back to the memory.

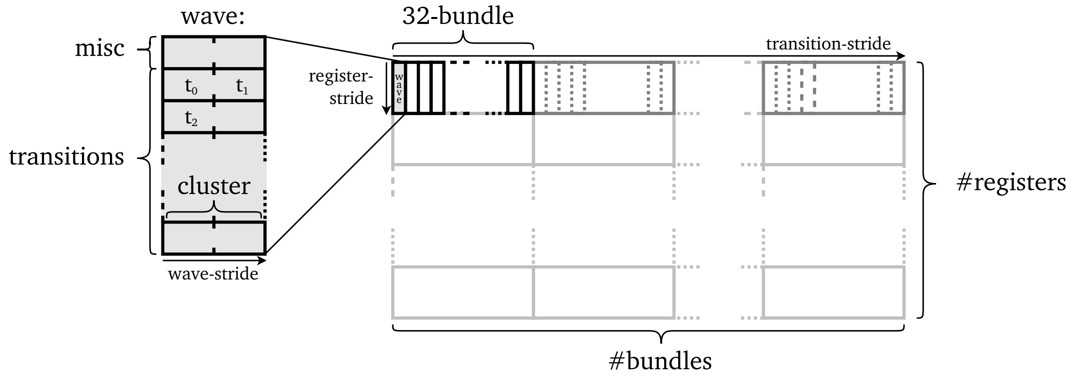

Figure C.4: Waveform representations stored in global memory.

Waveform Evaluation

All signals in `wave` are represented as *waves* or *waveforms*. A **waveform representation** w_S for a signal S is an ordered list of distinct times t_i with $\infty \geq t_i \geq 0$. Each $w_S(i)$ corresponds to a transition at S at time t_i, except for $t_i = 0$, which serves as a delimiter. The t_i are stored in descending order, so $w_S(i) > w_S(j) \Rightarrow i < i$. By definition, all signals S will have to stabilize at 0 eventually. Thus, if the last transition of S is a rising one, an additional $\{\infty\}$ will have to be inserted in the waveform in order to *fake* an additional final transition to 0. This will later help to evaluate the signal.

Figure C.5 illustrates some examples of signal waveforms. All events at a signal line will be stored in the data structure in order to sustain most event information. The value of S at a certain time t is determined by traversing its corresponding waveform w_S with an index i starting from $i = 0$. The index is increased until $t \geq w_S(i)$. The final value $S(t)$ is then computed by $S(t) = (i \mod 2)$, or to be more precise:

$$S(t) = |\{w_S(i) : w_S(i) > t\}| \mod 2.$$

The waveform format allows to store a *complete signal history*, instead of one discrete final value. If a signal S is evaluated in parallel by using multiple waveforms $w_S^0, w_S^1, \ldots, w_S^{k-1}$, we refer to the w_S^i as a **waveset** W_S. The reversed ordering of the t_i in w_S allows for an efficient evaluation of input waveforms at a gate. Let ϕ_Z be the function of a gate n with output Z and input signals A and B, then the output function for $Z = \phi_Z(A, B)$ can be computed in a single pass using Algorithm C.1.

The memory organization for the wavesets is also depicted in Figure C.4. All pattern bundles of a row, indexed by *gate*, are evaluated for one and the same corresponding gate in parallel by many threads. Each single wave of the wavesets consists of a number of **clusters**, which are merely pairs of floats (FLOAT2 C data structures). Their purpose is to reduce the overall

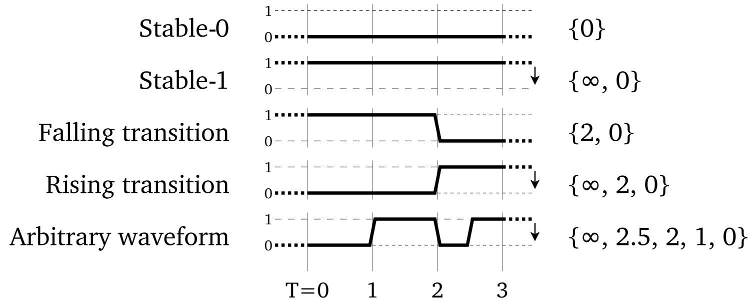

Figure C.5: Examples for waveform representations. "0" and "∞" are used as delimiters.

Algorithm C.1 Gate evaluation using waveforms

```
 1: procedure EVALUATE(A, B, Z)                       // input signals A, B; output signal Z
 2:     i ← (w_A(0) = ∞)                              // stable signal state of input values:
 3:     j ← (w_B(0) = ∞)          // if the last transition occurs at '∞': initialize with '1'
 4:     k ← 0
 5:     if φ_Z(i mod 2, j mod 2) = 1 then
 6:         w_Z(0) ← ∞                       // add '∞' if stabilized signal value of gate is '1'
 7:         k ← k + 1
 8:     end if
 9:     while max{w_A(i), w_B(j)} > 0 do                        // evaluation in reversed order
10:         if (k mod 2) ≠ φ_Z(i mod 2, j mod 2) then
11:             w_Z(k) ← max{w_A(i), w_B(j)}              // output transition (add delay here)
12:             k ← k + 1
13:         end if
14:         if w_A(i) ≥ w_B(j) then                     // next input change, proceed evaluation
15:             i ← i + 1
16:         else
17:             j ← j + 1
18:         end if
19:     end while
20:     w_Z(k) ← 0
21: end procedure
```

amount of memory accesses by threads through coalescing, by reading 64 bit instead of 32 bit per access. Except for the first, all clusters of a wave w_S will hold the transition information of the signal S. The first cluster is primarily used for memory recalibration purposes in case of transition overflows, which may occur during evaluation, since the transition count per wave is capped. The cap depends on the current register sizes.

After a simulation pass has finished and the final results have been fetched, the simulator checks for overflows. If any overflows did occur, a separate recalibration procedure will be called. It repeats the whole simulation task by repeatedly processing frame-by-frame and increasing register sizes where needed, until no more overflows are produced. This procedure is rather slow, since the registers have to be fetched and checked for each frame. For a fixed wavecap, the number of overflows in a simulation run strongly depends on the circuit complexity and the initial register size. Overflows may also vary from pattern to pattern. Once a circuit has been simulated for some patterns, the final configuration of the current registers can be saved. In case that the simulation has to be repeated, the state can be restored in order to prevent the initial calibrations.

Kernel Grid Structure

When the kernel is executed on a two-dimensional array of threads on the GPGPU device. Each spawned thread in the grid will compute a register value (Y-dimension) for a certain pattern (X-dimension) of a bundle. Table C.2 summarizes the different grid dimensions that have to be specified for the kernel call.

Dimension	**Description**
BLOCKDIM_X	Maximum number of patterns per bundle (32)
BLOCKDIM_Y	Maximum number of registers per block (8)
GRIDDIM_X	Maximum number of bundles
GRIDDIM_Y	(Maximum number of registers) / (Registers per block)

Table C.2: Description of the grid dimensions used by the kernel.

The simulator will always process whole bundles of 32 patterns each. The actual number of bundles, that can be simulated simultaneously, depends on the device memory and the used register sizes. If the wavecaps are reduced, the available space can be used for additional patterns. Also, circuits that require less registers can simulate more patterns at once and vice versa.

List of Symbols

$\delta(n)$	Propagation delay of a node
$\Delta^T(p)$	Slack of path p in a circuit with sampling time T
δ_f	Defect size for fault f
δ_P	Defect size regarding a set of paths P
$\Delta_{max}(n)$	Maximum slack of the paths $p \in P_n$
$\Delta_{min}(n)$	Minimum slack of the paths $p \in P_n$
ϕ_Z	Gate function, function for signal Z
Π	Pattern set
π	Pattern
$\sigma, \iota, \tau, \gamma$	POINTER evidence values
A, B, Z	Gate related signals (inputs A, B – output Z)
$a[i]$	Arrival time
$D^T_\Delta(n)$	List of defect sizes at a node n for all paths P_n
D_n	Defect range of a node n
e	Evidence
f	Fault
G	Levelized circuit
i, j, k	Common indices
L, L_i	Level in circuit G
N	Number of gates in circuit
n, g	A node or gate
n_Z	Node corresponding to signal Z
P	Set of paths
p, q	Path from circuit input to output
$p[i]$	Propagation time
P_n	Set of all paths going through node n
S	Signal
$S(t)$	Value of signal at time t

T	Circuit sampling time
t_i	Point of time (transition)
t_p	Propagation delay of path p
$t_{(i,j)}$	Combined path delay of all nodes on segments i and j
W_S	Waveset of signal S
w_S	Waveform representation
$w_S(i)$	Time of the i-th transition stored in w_S

Bibliography

[BA02] M. L. Bushnell, V. D. Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Kluwer Academic Publishers, 2002. ISBN:0-7923-7991-8.

[CDB09a] D. Chatterjee, A. DeOrio, V. Bertacco. Event-driven gate-level simulation with GP-GPUs. In *Proc. ACM/IEEE Design Automation Conf. (DAC '09)*, pp. 557–562. 2009. doi:10.1145/1629911.1630056.

[CDB09b] D. Chatterjee, A. DeOrio, V. Bertacco. GCS: High-performance gate-level simulation with GPGPUs. In *Proc. IEEE Design, Automation & Test in Europe Conf. & Exhibition (DATE '09)*, pp. 1332–1337. 2009.

[CIR87] J. L. Carter, V. S. Iyengar, B. K. Rosen. Efficient test coverage determination for delay faults. In *Proc. IEEE Int. Test Conf. (ITC '87)*, pp. 418–427. 1987.

[GK08] K. Gulati, S. P. Khatri. Towards acceleration of fault simulation using Graphics Processing Units. In *Proc. ACM/IEEE Design Automation Conf. (DAC '08)*, pp. 822–827. 2008. doi:10.1145/1391469.1391679.

[GK09] K. Gulati, S. P. Khatri. Accelerating statistical static timing analysis using graphics processing units. In *Proc. Asia and South Pacific Design Automation Conf. (ASP-DAC '09)*, pp. 260–265. 2009. doi:10.1109/ASPDAC.2009.4796490.

[HPA96] K. Heragu, J. H. Patel, V. D. Agrawal. Segment Delay Faults: A New Fault Model. In *Proc. IEEE VLSI Test Symp. (VTS '96)*, pp. 32–39. 1996. doi:10.1109/VTEST.1996.510832.

[HW09] S. Holst, H.-J. Wunderlich. Adaptive Debug and Diagnosis Without Fault Dictionaries. *Journal of Electronic Testing*, 25:259–268, 2009. doi:10.1007/s10836-009-5109-3.

[IRW90] V. S. Iyengar, B. K. Rosen, J. A. Waicukauski. On Computing the Sizes of Detected Delay Faults. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(3):299–312, 1990. doi:10.1109/43.46805.

[KC98] A. Krstić, K.-T. T. Cheng. *Delay Fault Testing for VLSI Circuits*. Kluwer Academic Publishers, 1998. ISBN:0-7923-8295-1.

[MA98] A. K. Majhi, V. D. Agrawal. Tutorial: Delay fault models and Coverage. In *Proc. IEEE Int. VLSI Design Conf. (VLSID '98)*, pp. 364–369. 1998. doi:10.1109/ICVD.1998.646634.

[MAJP00] A. K. Majhi, V. D. Agrawal, J. Jacob, L. M. Patnaik. Line Coverage of Path Delay Faults. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(5):610–614, 2000. doi:10.1109/92.894166.

[NVI10] NVIDIA. *NVIDIA CUDA C Programming Guide – Version 3.2*, 2010. `http://developer.nvidia.com/`.

[OHL+08] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, J. C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008. doi:10.1109/JPROC.2008.917757.

[PR88] A. K. Pramanick, S. M. Reddy. On the detection of delay faults. In *Proc. New Frontiers in Testing IEEE Int. Test Conf. (ITC '88)*, pp. 845–856. 1988. doi:10.1109/TEST.1988.207872.

[Smi85] G. L. Smith. Model for delay faults based upon paths. In *Proc. IEEE Int. Test Conf. (ITC '85)*, pp. 342–349. 1985.

[WLRI87] J. A. Waicukauski, E. Lindbloom, B. K. Rosen, V. S. Iyengar. Transition Fault Simulation. *IEEE Design & Test of Computers*, 4(2):32–38, 1987. doi:10.1109/MDT.1987.295104.

[WST08] L.-T. Wang, C. E. Stroud, N. A. Touba, editors. *System-on-Chip Test Architectures: Nanometer Design for Testability*. Morgan Kaufmann Publishers, 2008. ISBN:0-12-370597-5.

[Wun09] H.-J. Wunderlich, editor. *Models in Hardware Testing*, volume 43 of *Frontiers in Electronic Testing*. Springer Berlin Heidelberg, 2009. doi:10.1007/978-90-481-3282-9.

[WWW06] L.-T. Wang, C.-W. Wu, X. Wen, editors. *VLSI Test Principles and Architectures*. Morgan Kaufmann Publishers, 2006. ISBN:0-12-370597-5.

All links were last followed on April 22, 2011.

List of Figures

1.1 Fault Simulation Scheme . . . 8
1.2 Example of a Delay Defect . . . 9
1.3 Waveform Summary . . . 11

2.1 Double-Cone of a Fault Site . . . 21
2.2 Computing Path Slacks for c17 . . . 23
2.3 Computed Fault Set of c17 . . . 24
2.4 Defect Range Quantization . . . 27
2.5 SmallDelay Fault Equivalences . . . 28

3.1 Test Pattern Conversion . . . 29
3.2 Fault Simulation Tasks . . . 32
3.3 Simulation Run Outline . . . 33
3.4 Frame-depth of Benchmark Circuits . . . 34
3.5 Hazards and Fault Detection . . . 36

4.1 Evidences in Diagnosis . . . 37
4.2 Delay Fault Diagnosis Outline . . . 41

5.1 ISCAS'85 Circuit Information . . . 44
5.2 DCD Modifications on ISCAS'85 . . . 44
5.3 NXP Circuit Information . . . 45
5.4 ISCAS'85 Fault Simulation Time . . . 47
5.5 NXP Fault Simulation Time . . . 47
5.6 ISCAS'85 Fault Simulation Performance . . . 48
5.7 NXP Fault Simulation Performance . . . 48
5.8 GSI Rank Distribution of Stuck-At Representatives (ISCAS'85) . . . 50
5.9 GSI Rank Distribution after Resimulation (ISCAS'85) . . . 51
5.10 Suspect Evidences with $\gamma \neq 0$ . . . 52
5.11 SIG Rank Distribution after Resimulation (ISCAS'85) . . . 53
5.12 Resimulation of Merged Evidences (ISCAS'85) . . . 56
5.13 Diagnosis Approaches in Juxtaposition . . . 58
5.14 Diagnosing larger Circuits (NXP) . . . 59

C.1 Register Allocation . . . 67

C.2 Frame Information in Memory 68
C.3 Gate Data Structure 69
C.4 Waveforms in Memory 70
C.5 Waveform Examples 71

List of Tables

1.1 Transition Fault Behaviour 10
1.2 Comparison of Delay Fault Models 13

4.1 Middle-Rank Examples 39

5.1 ISCAS'85 Diagnosis Results in Juxtapose 57

C.1 Gate Type Mapping 69
C.2 Kernel Grid Dimensions 72

List of Algorithms

2.1 Computing arrival lists for all nodes (Phase 1) 22
2.2 Computing propagation lists for all nodes (Phase 2) 22
2.3 Determining defect sizes 24
2.4 Modification of DCD for usage as transition-fault model 25

C.1 Gate evaluation using waveforms 71

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)