# Multi-Level Timing and Fault Simulation on GPUs

Schneider, Eric; Wunderlich, Hans-Joachim

Elsevier INTEGRATION, the VLSI Journal – Special Issue of ASP-DAC 2018

doi: https://doi.org/10.1016/j.vlsi.2018.08.005

**Abstract:** In CMOS technology first-order parametric faults during manufacturing can exhibit severe changes in the timing as well as in the functional behavior of cells. Since these faults are hard to detect by conventional tests, the accurate simulation of these low-level faults plays an important role for test validation. However, pure low-level fault simulation approaches impose a high computational complexity that can quickly become inapplicable to larger simulation problems due to limitations in scalability. In this paper, the first parallel multi-level fault simulation approach on graphics processing units (GPUs) is presented. The approach utilizes both logic level and switch level descriptions concurrently in a mixed-abstraction timing simulation. The abstraction is lowered in user-defined so-called regions of interest that locally increase the modeling accuracy enabling low-level first-order parametric fault injection. Resulting signal waveforms are transformed between the different abstractions transparently. This way a fast, versatile and efficient multi-level fault simulation approach on GPUs is created that scales for designs with millions of cells while achieving high simulation throughput with runtime savings of up to 84% compared to full switch level simulations.

Preprint

**General Copyright Notice**

This article may be used for research, teaching and private study purposes. Any substantial or systematic reproduction, re-distribution, re-selling, loan or sub-licensing, systematic supply or distribution in any form to anyone is expressly forbidden.

This is the author's "personal copy" of the final, accepted version of the paper published by *Elsevier B. V.*.

©2018 **Elsevier B. V.**

# Multi-Level Timing and Fault Simulation on GPUs

Eric Schneider*, Hans-Joachim Wunderlich

*Institute of Computer Architecture and Computer Engineering, University of Stuttgart, Pfaffenwaldring 47, 70569 Stuttgart, Germany*

**Abstract**

In CMOS technology first-order parametric faults during manufacturing can exhibit severe changes in the timing as well as in the functional behavior of cells. Since these faults are hard to detect by conventional tests, the accurate simulation of these low-level faults plays an important role for test validation. However, pure low-level fault simulation approaches impose a high computational complexity that can quickly become inapplicable to larger simulation problems due to limitations in scalability.

In this paper, the first parallel multi-level fault simulation approach on graphics processing units (GPUs) is presented. The approach utilizes both logic level and switch level descriptions concurrently in a mixed-abstraction timing simulation. The abstraction is lowered in user-defined so-called regions of interest that locally increase the modeling accuracy enabling low-level first-order parametric fault injection. Resulting signal waveforms are transformed between the different abstractions transparently. This way a fast, versatile and efficient multi-level fault simulation approach on GPUs is created that scales for designs with millions of cells while achieving high simulation throughput with runtime savings of up to 84% compared to full switch level simulations.

*Keywords:* parallel fault simulation, multi-level, transistor faults, waveform accurate, GPUs

## 1. Introduction

Simulation plays an integral part for design and test validation as well as diagnosis of current nano-meter technology integrated CMOS circuits. With the strict performance requirements and narrow timing budgets of today's designs, timing-accurate and glitch-aware simulation approaches are demanded that also consider information down to layout [1] to assess the timing and non-functional properties for EDA applications, such as low-power testing [2] and IR-drop estimation [3, 4, 5]. Test methods need to be *cell-aware* [6, 7] by considering lower levels in order to model and test for parametric faults such as *resistive opens* and *bridges*, as well as *cross-wire opens* and *shorts* or *parasitic capacitances* [8, 9, 10]. For these faults, both *smaller* and *larger* fault magnitudes are often hard to detect and not screened properly during testing, mostly because of their complex activation and propagation conditions [11] and their detection can further be tampered and invalidated due to hazards [12, 13, 14, 15] as well as pessimistic and insufficient timing models [16, 17]. Therefore, the consideration of accurate circuit timing and fault modeling has become subject of recent test and diagnosis research [3, 18, 19, 7, 8].

The holistic validation of timing and the simulation of faults in today's CMOS circuits requires a reasonable level of accuracy and magnitude, which has always posed a big complexity problem [20] for test and diagnosis of multi-million gate designs and thereby has been subject of parallelization ever since [21, 22, 23]. Parts of the complexity of these simulations can nowadays be tackled by computing on recent data-parallel general-purpose *graphics processing unit* (GPU) accelerators, that are able to process thousands to millions of light-weight threads concurrently on a single die [24]. Many approaches have been published that rely on the help of GPUs to enable faster circuit- [25, 26, 27, 28] and fault simulation [29, 30], i.e., for computation of fault dictionaries during diagnosis [31, 32] and even allow to significantly speedup timing-accurate simulations at both logic [33] and switch level [34]. Yet, accurate simulations still cannot be applied exhaustively at large scale, due to the runtime complexity, and further speedup of the evaluations is typically achieved again by providing more and faster computing resources or by complete abstraction of the simulation models [35, 36] to logic level [37].

However, simulation approaches can take advantage of both higher- and lower-level abstraction at the same time through the use of mixed abstractions. These *multi-level* approaches form a combined simulation across multiple abstraction levels by sacrificing accuracy for speed and thereby provide flexible trade-off to increase simulation efficiency [38, 39, 40, 41, 42, 5, 43]. For the realization on GPUs, a multi-level solution is, however, not trivial due to strict constraints given by the underlying architecture and parallel programming paradigm [24].

The paper at hand extends the work of [44] and presents a novel approach for efficient and scalable parallel multi-level fault simulation on data-parallel GPU-accelerators. It provides

- an efficient and highly parallel waveform-accurate multi-level fault simulation with mixed abstractions combining both logic and switch level,
- detailed first-order parametric fault modeling with transistor granularity and efficient fault injection,

*Corresponding author.
*Email addresses:* `schneiec@iti.uni-stuttgart.de` (Eric Schneider), `wu@informatik.uni-stuttgart.de` (Hans-Joachim Wunderlich)

- comprehensive output syndrome analysis for detailed evaluation of multi-level faults,
- flexible trade-off between speed and accuracy in order to be as fast as possible, while as accurate as necessary,
- simple interface with transparent transition between higher and lower abstraction levels.

The remainder of this paper is organized as follows: Section 2 summarizes and explains backgrounds of GPU-accelerated simulation approaches. Section 3 provides an overview of the implemented multi-level fault simulator. In Section 4 the basic simulation model is explained in detail. The fault modeling is briefly discussed in Section 6 with the output analysis explained in Section 7. Finally, experimental results are presented and discussed in Section 8 that prove the efficiency of the presented approach.

## 2. Background

The large number of parallel computing resources and the high computational throughput of *graphics processing units* (GPU) enables the massively parallel execution and tremendous acceleration of scientific applications in the many-core era [24, 45]. Parallelized programs for GPUs (called *kernels*) invoke an array of concurrent threads (*thread grid*). For the execution the threads of a thread grid are partitioned into smaller batches (*thread groups*) each of which is distributed and scheduled for processing on one of the available streaming multi-processors on the GPU. The execution of a thread group on a streaming multi-processor is organized as *single instruction multiple data* (SIMD) parallel processing, which heavily relies on a simple and uniform control flow that is followed by all parallel threads. Diverging execution flows (e.g., due to branching) will result in serialization of the thread execution, thereby obstructing the parallelization. Synchronization between threads is limited and often involves expensive memory transactions that cause a loss in performance and should be minimized as much as possible. A GPU device typically has access to a limited amount of global memory only (4–16 GB), that is shared among all the threads invoked by the kernel. In addition, all local registers and resources of a multi-processor are divided among all of its scheduled threads and typically pose a limit to the number of active threads running.

### *2.1. GPU-accelerated Circuit and Fault Simulation*

With the introduction of GPUs, several parallel approaches for circuit simulation [46, 26, 47] and fault simulation [25, 30, 29] have been developed that exploit different dimensions of available parallelism. In general, acceleration is achieved from structural parallelism provided by mutually data-independent cells in the design as well as faults. For this, the cells of the circuit netlist are divided into partitions each of which is independently evaluated by separate individual or groups of threads. Furthermore, through simultaneous evaluation of independent input stimuli [29, 30], the acceleration can be further enhanced by exploitation of data-parallelism. Especially in untimed (*zero-delay*) logic simulation data-parallelism is commonly exploited at word-level using bit-wise logic operations which is also applicable even to serial compute architectures [20].

The simulation of timing is an essential part of timing- and test validation of circuits. Timing-accurate simulations are typically performed at logic-level using event-driven approaches based on the *time-wheel* as proposed in [48]. The time-wheel approach schedules signal switches (*events*) in a circuit globally at discrete points in time into dynamic lists and allows to compute complete switching histories of signals over time as so-called *waveforms*. Since the event-driven simulation only needs to evaluate circuit structures that show events at their inputs, simulation overhead can be efficiently reduced. Still, compared to untimed (*zero-delay*) simulation, the event-driven timing-accurate evaluation of a circuit is compute intensive. The parallelization is complex and requires thorough management as well as synchronization of lists and data structures at the cost of additional memory and computing performance [23].

Instead, [33] proposed a plain time simulation approach at logic level, that utilizes local event scheduling of input waveform events at each cell in combination with an efficient merge-sort evaluation. The approach utilizes both structural parallelism from cells and data-parallelism from waveforms during simulation, and attains speed-ups of up to three orders of magnitude with a simulation throughput of up to several hundred million gate-evaluations per second compared to a conventional event-driven time simulation approach. Although the logic level simulation is timing-accurate and can consider individual pin-to-pin delays as well as transition polarities and glitch filtering, its accuracy is insufficient to reflect actual CMOS switching behavior and low-level parametric faults accurately.

Although GPU-accelerated solutions for accurate, low-level analog SPICE simulations exist [25, 47, 27, 28] the scalability of these approaches is limited. The work of [25] off-loads parts of the SPICE calculations for parallel evaluation to the GPU device. While the overall simulation approach showed a speedup of up to 10×, it is not applicable to multi-million cell designs due to the huge memory footprint. In [47], a complete approach for performing SPICE simulations on GPU is presented, that achieves up to two orders of magnitude in speedup. Yet, the speedup quickly diminishes and the simulator can solely be applied to very small netlists composed of a few transistors only (up to 30). In general, sparse matrix solving poses a huge bottleneck for SPICE simulations. GPU-accelerated solvers based on LU-factorization that evolve around right-looking factorization were proposed in [27] and [28]. The authors showed that the right-looking approach allows to exploit a higher level of concurrency through simultaneous parallelism from columns, submatrices and vector operations, compared to left-looking approaches. Compared to conventional solvers, these approaches achieved speedups of one to two orders of magnitude.

Switch level simulation [36, 35, 49] provides a trade-off in speed and accuracy, due to simplified electrical models based on discrete binary switches and simple electrical components, such as resistors and capacitors. In a recently presented approach [34] fast switch level timing simulation on GPU was presented that reflects functional and timing behavior of CMOS cells using first-order electrical parameters. The simulator implements a simplified electrical model as illustrated in Fig. 1 where the output behavior of the cell is described as a time- and value-continuous function based on the internal RC-properties. Compared to logic level, the switch level model can cover many important timing effects related to CMOS technology, such as glitch-filtering, transition ramps and pattern-dependent delays due to multiple-input switching [16, 17] in a very efficient manner. Despite the more complex modeling, the GPU-accelerated switch level simulator outperforms traditional timing simulation approaches at logic level by over two orders of magnitude. An extension of the simulator [50] features modeling and injection of low-level parametric faults for fast and accurate fault simulation with transistor granularity.

### 2.2. Multi-Level Simulation

During the design phase, usually not all parts of a design need to be simulated with the highest level of accuracy at the same time for evaluation. It can also be the case that lower level descriptions are not even readily available at all for the whole system. Multi-level simulation approaches [38, 39, 40, 41, 42, 5, 43] have been proposed that evaluate the parts of a design with different abstraction simultaneously. Typically, lower-level electrical models and properties of cells or faults are abstracted in pre-characterization processes [51] and handed to simulations at higher levels for faster processing [5, 6]. While the abstracted data can be stored offline for reuse, the characterization process needs to be repeated for every new process corner, fault models or patterns under consideration, which can also cause high memory costs. Hierarchical multi-level approaches partition the circuit into regions for separate evaluation at higher and lower levels [38, 39, 41, 42]. Parts of particular interest are simulated at more accurate and compute-intensive lower levels, while the remainder of the circuit is evaluated using faster higher level simulation. In-between the evaluation, simulation data is exchanged at the abstraction boundaries. For example, a multi-level solution for efficient fault simulation was proposed in [41], which combines transaction level and logic level simulations achieving speed-ups of up to four orders of magnitude.

In [44] a first multi-level solution for fast and efficient parallel timing simulation on GPUs was presented combining both logic and switch level simulation in a mixed-abstraction fashion. Although the concept of multi-level simulation generally conflicts with parallelization on GPUs and the underlying many-core programming paradigm [24], due to different algorithms, data structures and working sets, the simulator utilizes thoughtful unification and careful organization, allowing for efficient and scalable simulations with a flexible trade-off in speed and accuracy. This work extends the multi-level solution of [44] by fault modeling and simulation [52, 50] enabling fast, efficient and comprehensive simulation of both high and low level faults on GPUs.

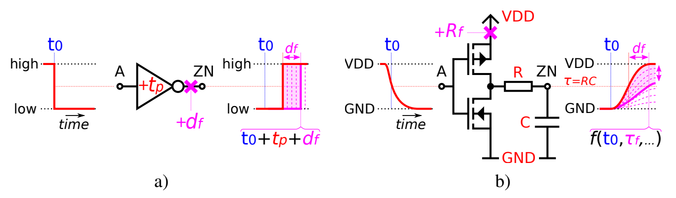


Figure 1: Signal and timing abstraction of a faulty inverter cell (*slow-to-rise*) in a) logic level and b) switch level simulation.

## 3. Parallel Multi-Level Fault Simulation

The multi-level fault simulation technique presented in this paper follows the parallel mixed-abstraction simulation approach of [44] that adopts the concepts of parallel timing-accurate evaluation at logic level [33] with the flexibility to locally increase the simulation accuracy to switch level. The adopted switch level model [34] evaluates CMOS functional and timing behavior based on first-order electrical parameters and enables low-level parametric fault injection at transistor granularity [50]. On the other hand, the logic level abstraction provides a timing-accurate evaluation of the circuit with high-simulation throughput for fast and efficient fault propagation [52]. During simulation, fault-parallelism is exploited by prior partitioning of the fault sets into *fault groups*, each of which consists of mutually output-independent faults which can be simultaneously injected and simulated [21].

In the implemented multi-level simulator, the abstraction of the design under evaluation is lowered in so-called *regions of interest* (ROI), that locally increase the accuracy from logic to switch level upon activation. Each ROI can represent an arbitrary set of nodes of the circuit graph, such as a single isolated nodes (input, output, cell), input-cones, output-cones as well as whole parts of a design (e.g., memory interfaces). When ROIs are *active* in a design, the simulator will use descriptions of both logic and switch level simultaneously in a mixed-abstraction fashion. The logic-level timing annotations are based on *standard delay format* (SDF) [53] that describe propagation delays on a pin-to-pin basis. For the switch level modeling in active ROIs, low-level information extracted from *detailed standard parasitics format* (DSPF) [54] and characterization of standard library cells with SPICE simulation of the associated transistor models are considered.

### 3.1. Overview

The overall flow of the implemented multi-level fault simulation is outlined in Fig. 2. First, a serial pre-processing is performed comprised of reading the synthesized circuit design and timing annotations (step 1). This work assumes full-scan designs and focuses on the evaluation of the combinational logic part. After levelization of the combinational logic netlist (step 2), the fault-list is read and all faults contained are then partitioned into *fault groups* (step 3) for later parallel injection. The remaining steps form the main parallel simulation loop.

In each iteration of the main loop the fault simulator processes an individual fault group. First, the fault locations of all low-level parametric and parasitic faults in the active fault group are marked as ROI to lower the abstraction level (step 4). Additional ROIs can be specified by the user, if necessary. Then the faults of the current fault group are injected into the corresponding node descriptions in the netlist (step 5). The netlist is simulated for the provided stimuli using the multi-level timing simulation kernel (step 6) in order to compute all internal signal histories (so-called *waveforms*) as well as outputs. Note that during the simulation process all active ROIs and all faults remain injected as no fault-collapsing is applied. In a last step, the output waveforms are analyzed and the respective syndromes are computed to determine the fault detection for all stimuli (step 7). After the simulation, all modified node descriptions are restored for the evaluation of the next fault group.

### *3.2. Simulation Parallelism*

This paper adopts a multi-dimensional parallelization scheme [33, 50] to speed-up the simulations on *graphics processing unit* (GPU) accelerators. The implemented simulation kernels exploit up to *three* dimensions of parallelism as shown in Fig. 3.

The first dimension of parallelism (a) considers structural parallelism from nodes that are scheduled on the same level after levelization of the circuit netlist. As during levelization all nodes are topologically ordered according to their data dependencies, all nodes of a level are mutually independent of their inputs and outputs and can therefore be processed concurrently by individual threads [33]. The second dimension (b) exploits data-parallelism by taking advantage from the evaluation of independent stimuli applied to a circuit. Since a stimuli applied to a circuit produces new waveforms at all nodes during simulation, the circuit and hence its nodes can therefore be evaluated for multiple stimuli in parallel by different threads. The third dimension of parallelism (c) exploited is similar to the aforementioned structural parallelism and comprises the partitioning of a given fault set under investigation into *fault groups* [21]. Each *fault group* is a set of mutually output-independent faults. The faults of a fault group do not influence each other by masking or adding and can therefore be simultaneously injected into the circuit for simulation in parallel.

The *grid of threads* for the parallel execution of the multi-level simulation kernels on the GPU are organized as illustrated in Fig. 4. The grids form a two-dimensional array of execution threads with each thread in the grid being responsible for the calculation of the output waveform of one node on a level for one particular stimuli. Threads shown in the *vertical* direction compute the functions of data-independent nodes on a level concurrently for the same stimuli applied to the circuit. Threads shown in the *horizontal* direction evaluate the same node in parallel for different input stimuli. When executing the thread grid of a kernel, thread indices are utilized to align data and navigate memory accesses to circuit information as well as waveform data, which heavily exploits coalescing of memory accesses at subsequent addresses for efficient memory transactions [33]. Since the evaluation kernels are called level by level, implicit synchronization barriers are found between consecutive calls

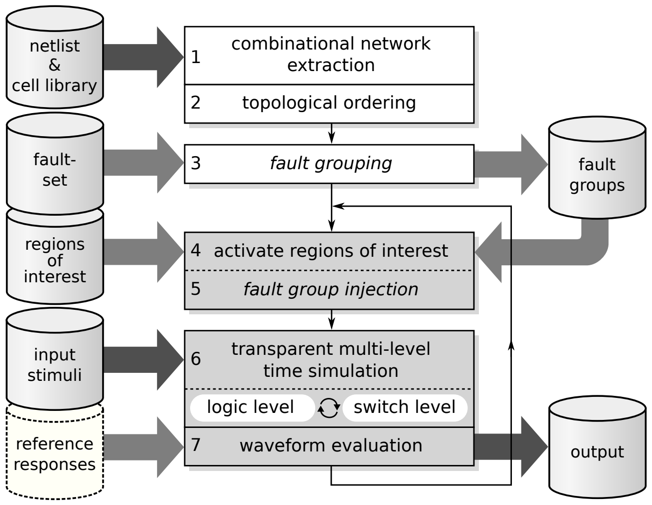


Figure 2: Flow-chart of the implemented multi-level fault simulation.

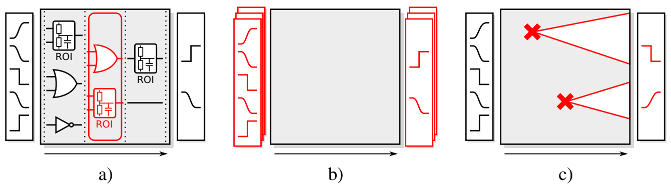


Figure 3: Dimensions of parallelism exploited during simulation: a) node-parallelism, b) stimuli-parallelism, c) fault-parallelism.

only and no additional explicit barriers are required during the execution of the threads. For all nodes marked as ROI, the respective execution threads (denoted by '$*$') run the switch level algorithms for all stimuli. Note that, although the abstractions are mixed throughout the circuit, no additional control-flow divergence is caused during execution of the kernels, since all threads of a *thread group* evaluate the same node with the same abstraction [44].

## 4. Multi-Level Time Simulation Model

This section briefly explains the underlying switch level circuit model of [34] and mixed-abstraction data structures which are utilized by the implemented multi-level fault simulation.

### *4.1. Circuit Model*

The circuit model of the multi-level simulator uses a mixed-abstraction approach where both logic level and switch level descriptions of nodes in the circuit are used simultaneously. The description of each node is composed of two parts: The *abstraction-independent* graph structure with access to its direct predecessors for accessing the input waveforms during evaluation, and the *abstraction-dependent* functional description of the node including the timing annotations. Whenever an ROI is (de-)*activated*, the latter part is swapped out accordingly.

For the logic level modeling, the functional description of the node is determined by annotation of a node *type*, that refers to a Boolean formula (e.g., NAND, XOR). The timing annotations

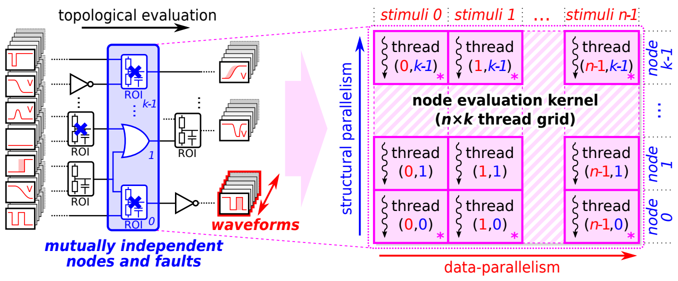


Figure 4: Multi-dimensional parallel evaluation of a levelized netlist with multiple data-independent nodes and waveforms of different abstractions.

follow the modeling of [33] and contain the pin-to-pin propagation delays that also distinguish between *rising* and *falling* transition polarities. These timing annotations are annotated for each input pin of a cell directly.

For the switch level modeling, the transistor netlist of the circuit is partitioned into *channel-connected components* [55, 7] as shown in Fig. 5, which can be derived from most primitive and complex CMOS cells found in standard-cell libraries. A *channel-connected component* (CCC) is composed of pull-up (PMOS) and pull-down (NMOS) transistor sub-networks both of which drive a specific signal line in the circuit. The transistors in each of the sub-networks are connected via their drain/source pins forming a mesh where current can freely flow within. Each transistor mesh connects a power-supply (i.e., VDD for pull-up or GND for pull-down networks) to the signal driven by the CCC, which is controllable via the gate terminal voltages at all of its transistors in the pull-up and pull-down meshes. Since, ideally current does not flow over gate terminals, charging the output of the CCC only draws current from its associated power supply.

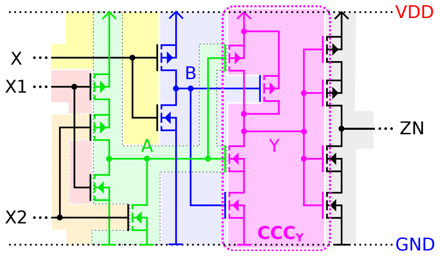


Figure 5: Transistor netlist of a small circuit with highlighted *channel-connected components* (CCCs).

The switch level simulator of [34] uses so-called *resistor-resistor-capacitor* (RRC) cells as basic simulation primitives, that provide a unidirectional model for the simulation of a channel-connected component in a circuit. A CCC is mapped to a single RRC-cell as shown in Fig. 6. All transistor devices $D$ in the CCC (either PMOS or NMOS) are modeled as voltage-controlled (binary) variable resistors $R^D(v)$ each of which is represented as a 3-tuple $D = (V_{th}, \{R_{off}, R_{on}\})$, such that

$$R^D(v) := \begin{cases} R_{off} & \text{if } (v < V_{th}), \\ R_{on} & \text{else.} \end{cases} \tag{1}$$

The $R_{off}$ and $R_{on}$ parameters of the tuple description of $D$ correspond to the drain-source resistances of the device for *blocking* and *conducting* state and $V_{th}$ is the threshold voltage that divides the state based on the applied input voltage $v$ at the gate terminal of $D$. For each network of transistors (pull-up and pull-down) the respective equivalent network-resistances ($R_u$ and $R_d$) are derived using simple nodal analyses based on Kirchhoff's laws. The equations of the transistor networks are hardcoded into the simulation kernels for compilation. Both $R_u$ and $R_d$ are utilized to form a voltage divider $(R_u, R_d)$ that drives the lumped output capacitance $C_{load}$ with voltage $\overline{v}$ associated to the output signal $Y$ of the channel-connected component $\text{CCC}_Y$. The *stationary voltage* $\overline{v}$ of the voltage divider is computed as $\overline{v} = (S \cdot V) + \text{GND}$, where $S = R_d/(R_u + R_d)$ is the ratio of the divider and $V = (\text{VDD} - \text{GND})$. The pull-up and pull-down resistances $R_u$ and $R_d$ as well as the stationary voltage $\overline{v}$ need to be computed whenever a transistor changes its state. A lumped wire resistances $R_w$ associated to the interconnects in the CCCs can also be incorporated during the calculations as shown in [34].

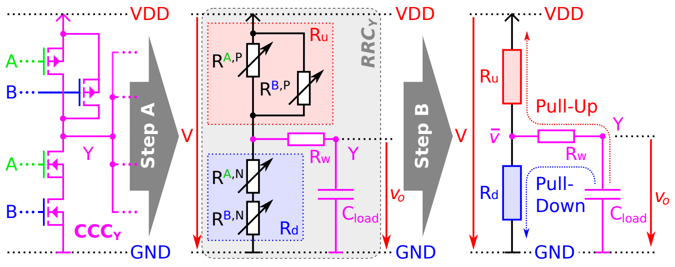


Figure 6: Mapping of a CCC (signal Y) to a corresponding switch level RRC-cell (step A) and electrical equivalence model (step B).

## 4.2. Mixed-Abstraction Time-Behavioral Modeling

The presented multi-level simulator uses a generic data structure in order to model time-continuous signal histories or *waveforms*. A *waveforms* $w = \{e_0, e_1, ...\}$ is a sequence of temporally ordered events $e_i$, each of which reflects a value change at a certain point in time. Each *event* $e_i = (t_i, p_0, p_1, ...)$ is modeled as a tuple composed of an *event time* $t_i \in \mathbb{R}$ and a set of parameters $\{p_0, p_1, ...\}$ that describe the change in value. The value of a waveform $w$ at a given point in time $t$ is denoted as $w(t)$.

The parameters of an event depend on the type of abstraction of the associated waveform. While throughout the simulation the implemented multi-level simulator uses both logic and switch level descriptions at the same time, the abstraction of each waveform is determined by the generating node (i.e., switch level when the node is marked as ROI or logic level otherwise). Header information is added to each waveform to identify the abstraction by the threads, such that during simulation all events contained are read from memory accesses and processed accordingly. In the following, the waveform structures encountered during logic and switch level simulation will be explained.

### 4.2.1. Ternary Logic Level Waveforms

To express the time-continuous signal history at logic level, the implemented multi-level simulator utilizes the efficient modeling of [33] originally proposed for the use in Boolean logic domain $\mathbf{B}_2 = \{0, 1\}$. The implemented multi-level simulation extends the waveform structure for the use with *ternary* (three-valued) logic $\mathbf{E}_3 = \{0, 1, X\}$ [37], allowing to consider *uncertainties* ($X$) in an efficient manner. At logic level, the *waveform* $w = \{e_0, e_1, ...\}$ of a signal is modeled as sequence of temporally ordered logic-level *events* $e_i = (t_i)$, each of which expresses a discrete value change at an event time $t_i \in \mathbb{R}$.

While transitions at logic level with multiple logic values can in general be modeled as time-value pairs [56], this paper extends the waveform structure of [33] for an efficient modeling of *undefined* values without introducing additional memory overhead. The extension comprises the use of negative numbers and the sign bit information of the event times $t_i \in \mathbb{R}$, which are implemented in the IEEE floating point standard [57] to allow the transition to two different signal states from any value as shown in Fig. 7. The *sign function* $sgn(x)$ of a floating point number $x \in \mathbb{R}$ delivers $sgn(x) = 1$ iff. the sign bit is set in its respective floating point representation, or otherwise $sgn(x) = 0$. If a signal waveform is in a *defined* state (logic-0 or logic-1), and an event $e_i = (t_i)$ with $sgn(t_i) = 1$ occurs, the waveform enters the *undefined* state $X$ at time $|t_i|$. From this state $X$, it can switch to either '0' or '1' depending on the sign of the succeeding event $e_{i+1}$ in the waveform.

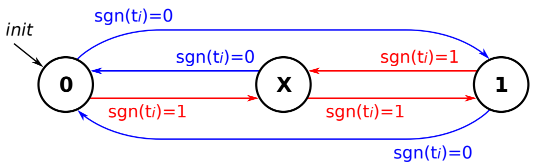


Figure 7: State transitions of waveform signals in *ternary* logic $\mathbf{E}_3 = \{0, 1, X\}$.

In order to sustain the temporal order of the events in the logic level waveforms and to allow efficient mergesort processing, all events $e_i = (t_i), e_j = (t_j) \in w$ are ordered at indices $i$, $j$ in the waveform $w$ according to increasing absolute values of the event times, such that

$$\forall e_i, e_j \in w, t_i \neq -\infty, t_j \neq -\infty : (i < j) \Rightarrow (|t_i| < |t_j|). \qquad (2)$$

Initially, for time $t = -\infty$ each waveform is assumed to have a value of $w(t) = 0$. The last event in a logic level waveform always has an event time $\infty$, which is used simply for termination as no event can occur after infinity. All events with signal switches in a waveform are assumed to happen at time $t$ after $-\infty$ and before $\infty$. To determine the signal value $w(t)$ at a given time $t \geq 0$, the events $e_i \in w$ in the corresponding waveform are read in temporal order of the absolute event times $|t_i|$ from earliest to latest. During the process the signal value is switched according to each event $e_i$ encountered until time $t$ is eventually reached. The value of $w(t)$ then corresponds to the signal value $w(t_i)$ after the latest event $e_i$ with $\max_i\{e_i = (t_i) \in w : |t_i| < t\}$. Note that, although negative event times $t_i$ are used, the order of the events $e_i$ follows the natural concept of time through $|t_i| \geq 0$. Similar to [33], initializing events $(-\infty)$ are utilized to set the initial signal values '1' and '$X$' .

For example: A constant-0 waveform is expressed as $w_0 = \{(\infty)\}$, a constant-1 waveform as $w_1 = \{(-\infty), (-\infty), (\infty)\}$, and a constant-$X$ waveform as $w_X = \{(-\infty), (\infty)\}$. Since the IEEE floating point standard also distinguishes negative ($-0$) and positive zeroes ($+0$) [57], the modeling also allows for arbitrary state changes at the transition launch time $t = 0$.

### 4.2.2. Switch Level Waveforms

In a similar manner, the signal switching histories of the switch level simulation [34] are modeled as sequence $w = \{e_0, e_1, ...\}$ of switch level *events* $e_i$ that provides a piece-wise approximation of the time- and value-continuous switching signal. Whenever a transistor switches its state at a given *event time* $t_i$, the output of an RRC-cell $v_o(t)$ will follow an exponential behavior for $t \geq t_i$ with *time constant* $\tau = S \cdot R_u \cdot C_{load}$ due to the RC-properties of the circuit. The time-continuous change in the output voltage is then expressed by a switch level event in the waveform. An event is modeled as tuple $e_i = (t_i, \overline{v}_i, \tau_i)$ that represents a continuous exponential curve segment in-between the interval $[t_i, t_{i+1}]$, where $t_{i+1}$ corresponds to the event time of the next event $e_{i+1}$ in order. The three parameters of an event tuple allow to closely describe the charging or discharging process of an RC-sub-circuit as shown in Fig. 8. These parameters are:

- $t_i$: The *event time* at which the curve segment is initiated.
- $\overline{v}_i$: The *stationary voltage* the curve is heading for.
- $\tau_i$: The *slope* of the exponential curve or *time constant*.

The associated curve segment of a switch level transient event $e_i = (t_i, \overline{v_i}, \tau_i)$ initiated at time $t_i$ is expressed as [34]:

$$w(t) := (w(t_i) - \overline{v}_i) \cdot e^{-\frac{\Delta t}{\tau_i}} + \overline{v}_i, \quad t_i \leq t \leq t_{i+1} \qquad (3)$$

in the interval $[t_i, t_{i+1}]$ between two consecutive events $e_i$ and $e_{i+1}$ in the waveform, with $w(t_i)$ being the value of the waveform at the start of the interval and $\Delta t = (t - t_i)$. The initial value $w(-\infty)$ of a waveform is set by an initialization event $e_{init} = (-\infty, V_{init}, -)$ at time $t = -\infty$, where $V_{init}$ corresponds to the initial voltage value.

Similarly to the logic level waveforms, the switch level waveform value $w(t)$ at a given time $t$ is determined by computing the curve intervals from earliest to latest [34]. Starting from the initial value $w(-\infty)$ at time $-\infty$, the algorithm iterates over all events $e_i \in w$ of the waveform in temporal and computes the voltage change at $w(t_i)$ in each interval $[t_i, t_{i+1}]$ based on

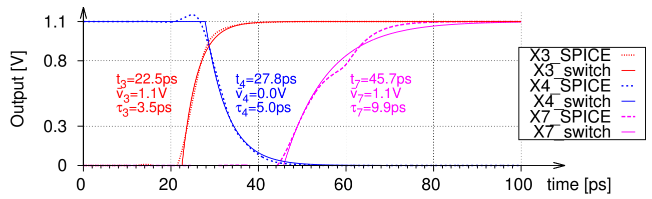


Figure 8: Signals of nodes with varying fanout from SPICE transient analysis (dotted) and switch level event representation (bold).

Eq. (3). Note that between event times $t_i$ and $t_{i+1}$ of consecutive events all resistances and supply voltages in the RRC-cell remain constant. This evaluation process is repeated until the latest event $e_i$ is found with $\max_i\{e_i \in w : t_i < t\}$, that describes the curve segment in which time $t$ is located. As a final step, the voltage change of the event $e_i$ is computed for the remaining time $\Delta t = (t - t_i)$ using Eq. (3).

## 5. Multi-Level Simulation Algorithm

In Algorithm 1 the implemented multi-level simulation of a circuit is outlined [44]. As input of the evaluation, the levelized circuit netlist $G$ with both logic and switch level descriptions for each node is provided. Since certain logic level nodes need to be represented by multiple RRC-cells at switch level [34], corresponding placeholder nodes were added prior to the levelization of the netlist. This way, the levelized graph structure does not need to be altered and the topological order is sustained. Input stimuli for all circuit primary and pseudo-primary inputs are provided for a set $P$ of different delay tests in the global waveform memory $W$ on the GPU device. The circuit is then processed level by level from inputs towards outputs. For each node $N$ on a level its respective input waveforms $I$ of the predecessors nodes are evaluated and a corresponding output waveform $w_N$ is computed (lines 8–31). These evaluations are performed concurrently by individual threads for all nodes on a level and all assigned delay tests $P$ following the parallelization scheme presented in [33] and [50]. The general evaluation algorithm executed for each node uses a mergesort waveform processing, which sorts the local switching events of the respective input waveforms in a local schedule that is processed in temporal order such that the output waveform can be generated in a single pass. The set of waveforms generated during processing of a circuit for a delay test $p \in P$ will be referred to as $W_p$. If multiple delay tests $p \in P$ are provided at a time, each corresponding $W_p$ represents a distinct partition of the waveform memory $W$.

The evaluation kernel itself first fetches all input waveforms $I$ of the current node $N$ (line 8) and determines the type of abstraction from the waveform header. Then the data structures for storing the current input event are set up and the initial value of each input waveform $w_i \in I$ in $W_p$ is computed (line 9–13). After putting the first events of each waveform in the (thread-)local schedule $E$, the output state of the node $N$ is computed and the output waveform $w_N$ is initialized with the corresponding signal value (line 14). In the main simulation loop (lines 16–29), the events of the local event schedule $E$ are processed in temporal order from earliest to latest. In each iteration, the earliest next input event is *consumed*, which indicates a change in the associated input signal. The input event is transformed to the target abstraction level of the node $N$. Depending on whether $N$ is marked as ROI, the implications of the value change are incorporated into the state of $N$ by selecting either the *low-level* switch level (line 20) or *high-level* logic level kernel (line 23 for the evaluation. In case the consumed event causes a change in the output signal of $N$, an output event is generated and appended to the output waveform $w_N$ (line 26). After the event has been processed, the algorithm determines the next event of the corresponding input waveform and schedules it for evaluation in the next iteration of the main loop (line 28). In case all input events have been processed, the main simulation loop is terminated and the output waveform $w_N$ of the node is stored in the waveform memory partition $W_p$ of $W$ (line 31).

Note that during evaluation of a node, each thread in the implemented multi-level simulation computes only a single output waveform for its applied input stimuli. In case multi-output cells need to be modeled, a duplication of the node can still be used where each output of the cell is handled by an individual duplicate. This way, the working set (e.g., number of local registers required) and memory access patterns of the threads are not affected.

### *5.1. Multi-Level Waveform Transformation*

Since the implemented multi-level fault simulator uses waveforms of mixed abstraction, bi-directional transformations between the different waveform types need to be provided. This work utilizes two mappings for the transformations between the higher logic and lower switch level abstraction [44]. The transformations are applied to the input waveforms by the evaluation kernels on a per-event basis, i.e., an event of a *source* abstraction has a corresponding set of events at the other *targeted* abstraction level. All transformed events are identified and inserted in the event scheduler for processing by the kernel. The *source* abstraction level of each input waveform is acquired once from the respective waveform header information in the beginning of the node evaluation. The *target* abstraction of the transformation is determined by the node under evaluation. In case *source* and *target* abstraction are equal, the input waveform events can be directly processed.

#### *5.1.1. Logic to Switch Level Transformation*

The signal transitions expressed by waveform events at logic level (e.g., *rising* or *falling*) are considered to be rectangular in shape. The *instantaneous* transitions expressed by these events can be modeled by infinitely small time constants $\tau_\varepsilon > 0$ at switch level. Given the power supply VDD and ground GND voltage levels of the targeted technology, all transition events in the original logic level waveform source $w$ are transformed to switch level events $e_i$ by substitution. Depending on the targeted logic value of a transition (e.g., '1' for *rising* and '0' for *falling*) the transformation of a logic level event at time $t_i$ is performed as follows [44]:

$$t_i \mapsto e_i := \begin{cases} (t_i', \mathrm{VDD}, \tau_i) & \text{if } (t_i \textit{ rising}), \\ (t_i', \mathrm{GND}, \tau_i) & \text{elif } (t_i \textit{ falling}), \\ (t_i', \frac{\mathrm{VDD+GND}}{2}, \tau_i) & \text{else}, \end{cases} \tag{4}$$

where $t_i'$ is the time of the new switch level event $e_i$, the stationary voltage represents the electrical interpretation of the logic value (VDD for '1', GND for '0' and $\frac{1}{2} \cdot (\mathrm{VDD} + \mathrm{GND})$ for '$X$') and $\tau_i$ is the corresponding time constant.

**Algorithm 1:** Transparent multi-level simulation (serial).

```
Input: netlist $G$, input stimuli $P$ in waveform memory $W = \cup_{p \in P}\{W_p\}$
Output: output responses (stored in $W$)
foreach level $L$ in the netlist $G$ do
    /* GPU: The following loops over $L$ and $P$ are executed as
       two-dimensional grid of parallel threads (cf. Fig.4): */
    foreach node $N$ on level $L$ do
        /* .................. node-parallelism .................. */
        foreach stimuli $p$ in $P$ do
            /* ............... stimuli-parallelism ............... */
            // A. initialization
            Load input waveforms $I \subseteq W_p$ for node $N$.
            foreach waveform $w_i$ in $I$ do
                Look-up abstraction level of $w_i$.
                Set-up data structures and initial state.
                Get first event $e$ of $w_i$ and put into schedule $E$.
            end
            Initialize output waveform $w_N$.
            // B. event processing
            while Events to process in schedule $E$ do
                Remove earliest event $e$ from $E$.
                if node $N$ is ROI then
                    Transform $e$ to switch level event.
                    Compute new switch level state of $N$.
                else
                    Transform $e$ to logic level event.
                    Compute new logic level state of $N$.
                end
                if new state of $N$ causes output change then
                    Compute output event and add to $w_N$.
                end
                Get next event $e$ of $w_i \in I$ and put into $E$.
            end
            // C. finalization
            Store $w_N$ to waveforms $W_p := W_p \cup \{w_N\}$.
        end
    end
end
```

In general, the resulting switch level events can be fitted in time and slope to match the RC-characteristics of the driving signal. According to the standard definition of the signal propagation delay, it is assumed for logic level transitions that the corresponding electrical signal passes the 50% voltage threshold $V_{th} = \frac{\text{VDD+GND}}{2}$ at the exact time. Suppose a signal transition of a logic level waveform occurs at time $t_i$. The following equation allows to fit the time $t_i'$ for a given time constant $\tau_i$ of the RC-properties of a driving cell [44]:

$$t_i \overset{!}{=} t_i' - \tau_i \cdot \log\left(\frac{V_{th} - \overline{v}_i'}{w(t_i) - \overline{v}_i'}\right). \tag{5}$$

The fitted time parameter of the switch level curve segment is then obtained by transformation of the above equation and solving for $t_i'$:

$$t_i' := t_i + \tau_i \cdot \log(0.5), \tag{6}$$

which corresponds to a shifted starting point of the new event $e_i$. By considering the RC-properties in the transients, more realistic representations of the input signals for the targeted node are obtained. An example is shown in Fig. 9, which illustrates both transformations with an infinitely small time constant $\tau_\varepsilon \approx 0$ ("$\tau_\varepsilon$ trans.") as well as with consideration of RC-characteristics in the time constant ("*RC* 50%") to match $V_{th}$ as explained above. Note that according to Eq. (6), the resulting error $\epsilon = |t_i' - t_i|$ of the $\tau_\varepsilon$-transformation is always smaller than $\tau_\varepsilon$, since $|\log(0.5)| < 1$. Thus, for $\tau_\varepsilon \to 0$, it closely resembles the logic level representation ("*source*") with a negligible error of $\epsilon = |t_i' - t_i| < \tau_\varepsilon$.

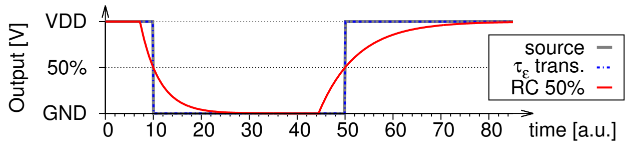


Figure 9: Transformation of waveform (*source*) from logic to switch level with different RC-characteristics for *rising* and *falling* transitions.

### 5.1.2. *Switch to Logic Level Transformation*

If during simulation, a logic level node $n$ has a switch level waveform as input, lower level signal information needs to be transformed to the higher level accordingly. At switch level all signals are continuous in value and do not always correspond to *well-defined* logic *high* or *low* symbols. Since intermediate signal values can be interpreted differently by succeeding stages an *undefined* value ($X$) is assumed. In this work, an open interval $(V_{thL}, V_{thH}) \subset [\text{GND}, \text{VDD}]$ bounded by a *low* ($V_{thL}$) and a *high* ($V_{thH}$) signal threshold is utilized to map from the continuous voltage domain to discrete logic symbols of a *ternary* (three-valued) logic $\mathbf{E}_3 = \{0, 1, X\}$ with *undefined* values [37]. The mapping is defined as follows [50]:

$$val : \mathbb{R} \to \{0, 1, X\}, val(v) := \begin{cases} 0 & \text{if } (v \leq V_{thL}), \\ 1 & \text{elif } (v \geq V_{thH}), \\ X & \text{else.} \end{cases} \tag{7}$$

Hence, for the voltage $v$ of a signal with $v \geq V_{thH}$ ($v \leq V_{thL}$) the signal value is interpreted as defined logic-1 (logic-0). In case $v \in (V_{thL}, V_{thH})$, then $v$ is assumed to be *undefined* ($X$).

The transformation of a waveform is performed event after event in temporal order from earliest to latest. For all curve interval $[t_i, t_{i+1}]$ of consecutive events $e_i$ and $e_{i+1}$, the intersection points of the corresponding curve segment with the low ($V_{thL}$) and high ($V_{thH}$) thresholds are identified. At each intersection point, the interpreted logic value of the input waveform is assumed to change at the given time which is added to the event schedule for processing the current node $n$.

The threshold-based switch to logic level transformation is illustrated in Fig. 10 on the example of an arbitrary continuous waveform signal. For each intersection point with a threshold $V_{thL}$ or $V_{thH}$, the corresponding logic value is determined and added to the resulting output waveform. Note that during this transformation pulse filtering can be applied to either logic-1/0 or $X$-pulses to remove glitches that are physically impossible to be generated.

At logic level, the implemented simulation algorithm performs a pessimistic propagation of $X$es in the circuit. In case a node enters an undefined state ($X$) due to an input transition, always the minimum propagation delay at the arriving input pin is applied. In case the node transitions from $X$ to a defined

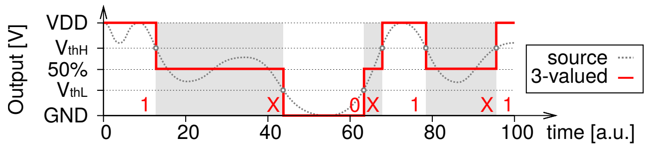


Figure 10: Transformation of an arbitrary continuous signal to discrete ternary logic symbols (3-valued) based on thresholds $V_{thL} < 50\%$ and $V_{thH} > 50\%$.

state, the maximum propagation delay at the pin is considered. Unknown signals at node inputs subject to controlling off-path signal values and can get masked during propagation.

## 6. Fault Modeling

The data structures of the presented multi-level timing simulation are used to model and inject timing-related faults modeled at higher abstraction (i.e., *small delay faults*) as well as parametric and parasitic faults at switch level (e.g., resistive opens and shorts). In this section, the fault modeling at both higher and lower abstraction levels is presented. Faults of both abstraction types can be used simultaneously using the presented mixed-abstraction simulation. In general, all faults $f$ of any of the implemented fault models are represented as tuple

$$f = (loc, \{\delta_0, \delta_1, ...\}), \tag{8}$$

independent of its abstraction, where *loc* pin-points to a specific fault location of the targeted abstraction (i.e., a node pin) and the set $\{\delta_0, \delta_1, ...\}$ corresponds to the parameter deviation introduced by $f$ at *loc*.

The remainder of this section briefly describes the modeling and mechanics of the supported fault models as well as the injection scheme.

### *6.1. Logic Level Faults*

At logic level, the underlying simulation model of the timing simulator is utilized to model *small (gate) delay faults*, that slow down the propagation of signal transitions through an affected fault site by adding an additional small amount of delay [58]. The core model of [33] annotates each node with pin-to-pin delays at their inputs and also distinguishes delays for *rising* and *falling* polarities. Following Eq. (8), each *small (gate) delay fault* $f$ is therefore modeled by a tuple $f = (loc, \{\delta_{rise}, \delta_{fall}\})$ [52], where the parameter deviations $\delta_{rise}$ and $\delta_{fall}$ of the fault describe the additional propagation delays for the *rising* and *falling* transition polarities at the node pin associated with *loc*, respectively. Using this model, a small delay fault can affect a *rising* transitions ($\delta_{rise} > 0$ and $\delta_{fall} = 0$), *falling* transitions ($\delta_{rise} = 0$ and $\delta_{fall} > 0$) or both polarities simultaneously ($\delta_{rise} > 0$ and $\delta_{fall} > 0$). Since the underlying simulation model annotates timing with respect to input pins only, small delay faults located at the output pin of a node are mapped to a sets of corresponding small delay faults at the inputs [52].

### *6.2. Switch Level Faults*

The switch level simulation simulation allows to model low-level parametric and parasitic faults based on first-order electrical parameters in CMOS. The identification of manufacturing defects and their relevant faults at electrical level is neither trivial nor intuitive and typically requires a lot of manual work. However, *cell-aware test* approaches typically rely on the identification and extraction of potentially relevant faults using *user-defined fault models* [6] as well as layout-aware extraction of defects by inductive fault analysis such as proposed in [59, 51] or [60]. This work utilizes the switch level algorithm and data-structures of [50] for modeling and injection of low-level parametric faults without higher level abstraction as follows:

#### *6.2.1. Resistive Transistor Shorts and Opens*

Faults related to the resistive properties of a transistor device are mapped to the corresponding device description $D = (V_{th}, R_{off}, R_{on})$ of the RRC-cells. A *resistive transistor fault* $f$ of a cell is described as a tuple $f = (loc, \{\Delta R_f\})$, according to the aforementioned representation. It comprises the fault location *loc* referring to either blocking $R_{off}$ or conducting $R_{on}$ property of the targeted transistor device $D$, and a fault parameter $\Delta R_f \in \mathbb{R}$ that describes the deviation in Ohms ($\Omega$) of the resistive parameter $R^{loc}$ selected by *loc*. The resulting device parameter is then modified accordingly, such that $\tilde{R}^{loc} := R^{loc} + \Delta R_f$ [61]. In the simplest case, a *transistor open* fault is modeled by increasing the conducting resistance $R_{on}$ of a device $D$ ($\Delta R_f > 0$). Analogously, a simple *transistor short* is modeled by lowering ($\Delta R_f < 0$) the blocking resistance $R_{off}$ of $D$.

More complex examples of fault injections are shown in Fig. 11, which illustrates the modeling of a resistive open fault (a) and a resistive short forming a bypass (b) at an NMOS transistor of an RRC-cell. For the injection of the depicted resistive open fault, the NMOS transistor device resistances are modified to $\tilde{R}_{on} := R_{on} + R_{open}$ and $\tilde{R}_{off} := R_{off} + R_{open}$ with $\Delta R_f = R_{open} > 0$ by changing the two device parameters. The resistive short fault as shown in Fig. 11-b), on the other hand, is a more complex example and acts as a bypass to the transistor thereby lowering the effective resistance of the transistor. For injecting the fault, the device resistances are set to $\tilde{R}_{on} := R_{on} + \Delta R_{on}$ with $\Delta R_{on} = -R_{on}^2/(R_{on} + R_{short}) < 0$ and $\tilde{R}_{off} := R_{off} + \Delta R_{off}$ with $\Delta R_{off} = -R_{off}^2/(R_{off} + R_{short}) < 0$, respectively.

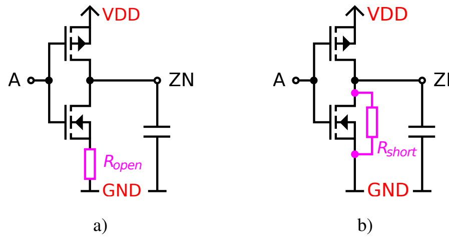


Figure 11: Example of a faulty RRC inverter-cell with (a) *resistive open* fault and (b) *resistive short* fault at the NMOS transistor.

### 6.2.2. *Capacitive Faults*

The modeling of capacitive faults related to the circuit interconnects and fanout descriptions [54] assumes a lumped capacitance model [62] which is incorporated into the RRC-cell model [61]. A *capacitive fault* is described as $f = (loc, \{\Delta C_f\})$ using the presented tuple-description. The fault location pinpoints to the output load capacitor $C_{load}$ of a targeted RRC-cell, where $\Delta C_f \in \mathbb{R}$ introduces an additional capacitance, such that $\tilde{C}_{load} := C_{load} + \Delta C_f$.

### 6.2.3. *Voltage-Related Faults*

Voltage-related faults target the voltage parameters of RRC-cells, such as power supply VDD and ground GND of the cells as well as the threshold voltages $V_{th}$ of the transistor devices [50]. A *voltage fault* $f = (loc, \{\Delta V_f\})$ models a particular shift $\Delta V_f \in \mathbb{R}$ in volts in a voltage parameter $V^{loc}$ of a targeted RRC-cell or device indicated by *loc*, such that $\tilde{V}^{loc} := V^{loc} \mp \Delta V_f$. By altering the VDD and GND voltages of the RRC-cells, adaptive voltage scaling and power-related issues like fluctuations in the power-grid due to *IR-drop* and *ground bounce* [63] can be reflected. Yet, modeling of dynamic fluctuations requires additional implementation effort and is not focus of this work. Voltage faults affecting the threshold voltages $V_{th}$ of transistor devices allow to model degradation phenomena, e.g., caused by aging effects, such as *negative-bias temperature instability* (NBTI) or *hot-carrier injection* (HCI). Both NBTI and HCI can increase the threshold voltage of a targeted NMOS (or PMOS) device over time and thereby delaying the transistor switching processes [64, 65].

### 6.3. *Fault Injection*

All faults of the presented fault models are injected into the circuit description prior to the simulation run. The injection of a fault $f = (loc, \{\delta_0, \delta_1, ...\})$ is performed by manipulation of the parameters expressed by *loc* at the associated node based on the fault description $\{\delta_0, \delta_1, ...\}$ and marking of the node as *faulty*. Without loss of generality, the fault locations of injected switch level faults are turned to ROIs automatically in the mixed-abstraction environment of the implemented multi-level simulator. In case a small delay fault is injected, any ROI flag at the fault location is removed, such that the corresponding node is simulated with logic level abstraction. Multiple faults can be injected into the circuit for simultaneous evaluation, which do not only allow to exploit *fault-parallelism*, but also allows for modeling of multi-faults across the circuit. After the simulation of the faulty circuit is finished, the node descriptions of all fault sites marked as *faulty* are restored to their original specifications. Since both the node descriptions and the descriptions of faults are quite compact (tens to hundreds of bytes each), the ROI activation and fault injection process comprises only a few small memory transactions.

### 6.4. *Fault Grouping*

The implemented fault simulation approach exploits *fault parallelism* from simultaneous injection and evaluation of mutually output-independent faults in a circuit [21]. For this, the fault set under investigation is partitioned into *fault groups*, each of which contains mutually output-independent faults whose fault effects cannot add or mask each other during simulation. The paper at hand utilizes a fast heuristic [52, 50] for identification of the *fault groups* of a given set of faults. The applied heuristic considers the fault location during grouping and therefore can be applied to arbitrary sets of faults containing both logic and switch level faults for mixed-abstraction fault simulation. The heuristic processes the initial fault set in reverse topological order one by one from inputs to outputs.

First, a fault is assigned an initial fault group, as determined by its fault location. For each fault location, a group index is kept, which is used as initial index for associated faults. If a fault location has not been assigned an index yet, it is scheduled in the first group by default. The grouping algorithm then compares the reachable outputs of the fault with the reachable outputs of the group for the following groups until an output-disjoint group is found. The fault and the reachable outputs are then added to the group and the index of the group is assigned to the fault site. The distribution of the grouping information to input predecessors allows to reduce the number of comparisons and hence the required grouping attempts in order to quickly find a suitable fault group for each fault [52].

## 7. Syndrome Evaluation

Once the circuit has been fully simulated, the waveforms of all output signals are present in the GPU device memory. For each fault, the waveform of all output pins in the output-cone of the respective fault site are captured at a given sample time $t_{samp}$ according to [52] for logic and [50] for switch level waveforms. The sampled values are then applied to Eq. (7) if necessary to obtain a corresponding logical interpretation of the signals, which is stored in a dedicated syndrome memory [61]. The syndromes are then used to distinguish *faulty* from *fault-free* output signals and therefore allow to determine whether a fault was detected or not.

### 7.1. *Computation of Discrete Syndromes*

To compute the syndrome at an output the value $w(t)$ sampled from the corresponding output waveform $w$ at time $t$ is compared against the *fault-free* response denoted as $w(\infty)$. This paper assumes that all *fault-free* values of signals are stabilized and correspond to defined *high* or *low* signals. The syndrome information of an output is obtained by evaluation of a so-called *syndrome waveform*, which is derived from the output waveforms and the reference responses. For the sake of simplicity, values of logic level output waveforms are mapped to their corresponding voltage values at switch level (cf. Sec. 5.1.1). The values of the *syndrome waveform* $syn_w(t)$ of an output waveform $w$ are then generated similar to [50]:

$$syn_w(t) := \begin{cases} val(w(t)) & \text{if } (w(\infty) \leq \frac{\text{VDD+GND}}{2}), \\ val(\overline{w}(t)) & \text{else,} \end{cases} \tag{9}$$

where $\overline{w}$ is the *inverted waveform* of $w$ that is generated by mirroring the values $w(t)$ at $\frac{\text{VDD+GND}}{2}$. This *inverted waveform* is computed as

$$\overline{w}(t) = (\text{VDD} - w(t) + \text{GND}), \forall t : \overline{w}(t) \in [\text{GND}, \text{VDD}].$$

After computing the syndrome the original output value $w(t)$ is recognized as *faulty* if the extracted syndrome is $syn_w(t) = 1$ or *fault-free* if $syn_w(t) = 0$. In case the interpreted value of $w(t)$ is *undefined*, the corresponding syndrome is *unknown* ($syn_w(t) = X$). The output signal is then pessimistically considered as *possibly erroneous*.

### 7.2. Multi-Level Fault Detection

The detection of a fault $f$ is determined by looking up the computed syndromes $syn_o(t_{samp})$ at a given capture time $t_{samp}$ for all of the respective outputs $o \in O_f$ in the output cone $O_f$ of $f$. Given the captured syndromes, fault $f$ is classified either as *detected*, as *undetected* or pessimistically as *possibly detected* as follows [50]:

- DT (*detected*) iff $\exists o \in O_f : syn_o(t_{samp}) = 1$, hence *at least one* output signal in the output-cone of the fault shows a *faulty* syndrome.
- UD (*undetected*) iff $\forall o \in O_f : syn_o(t_{samp}) = 0$ with *all* output signals in the output-cone of the fault being *fault-free*.
- PD (*possibly detected*) iff $(\forall o \in O_f : syn_o(t_{samp}) \neq 1) \wedge (\exists o \in O_f : syn_o(t_{samp}) = X)$, such that *no* output in the output cone is *faulty*, but a *non-empty* sub-set is *unknown*.

Since all computed waveforms remain untouched in the GPU memory during the computation of the syndromes, the outputs can be evaluated for multiple capture times quickly in succession. Furthermore, the evaluation also allows the modeling of clock skew in the clock distribution tree [66] by introducing individual capture times $t^o_{samp}$ for each output pin $o \in O$ of the circuit. In addition, both *space-* and *time-compaction* of the test responses can be applied during computation of the syndromes, as all output responses are available for all the simulated input stimuli.

## 8. Experimental Results

For the experiments, benchmark circuits from ISCAS'89, ITC'99 and industrial designs kindly provided by NXP were synthesized using a 45nm academic standard-cell library in a commercial tool-chain. Full-scan designs were assumed with all sequential elements having been removed and only the combinational logic remaining. For each circuit, a 10-detect transition delay fault test set with an average test coverage of over 98.7% was generated by a commercial ATPG-tool to be applied during the simulation. All experiments were conducted on an NVIDIA® Tesla$^{\text{TM}}$ P100 GPU device with 3584 cores and 16GB of global device memory. The host system was composed of two Intel® Xeon® E5-2687W v2 processors clocked at 3.4GHz with access to 256GB of main memory.

### 8.1. Fault Behavior

The modeling capability and detail of the multi-level fault simulation is briefly discussed using Fig. 12, which shows the waveform behavior of resistive open transistor faults in a two-input NOR-cell with an input hazard [50]. Both visualizations of the waveforms from SPICE simulation and the implemented switch level model are compared for varying fault sizes.

In the first case (a), the resistive open was injected into an NMOS transistor of the parallel pull-down net. As expected, the fault delays the falling transition significantly with increasing ohmic resistances. For 10MΩ, the resulting conductance of the pull-down net is already too small to effectively discharge the node output and the voltage level at the time of the fault activation sustains for the whole time-frame. Similarly in the second case (b), the resistive open fault was injected into a PMOS transistor in the serial pull-up net. The fault now affects the rising transition of the output signal. Since the conductance of the pull-up net is lowered, the charging process is delayed and cannot be completed before the second input arrives, which eventually pulls the output signal to ground completely unaffected by the fault. In both of the cases, the waveforms obtained from the switch level simulation showed a fairly high similarity of the modeled fault effect compared to the SPICE reference simulation.

### 8.2. Runtime Performance

Table 1 reports the runtimes of the multi-level simulator for varying ROI scenarios. In column 2 and 3 the size of each circuit in number of nodes (inputs, outputs and cells) and the delay test pairs in the ATPG-generated test pattern set are stated for each circuit. Column 4 states the runtime of processing the test set for each circuit using a commercial event-driven timing simulator. The remaining columns then report the runtimes using the multi-level simulator for different simulation scenarios with varying amount of active ROIs from lowest to highest. A logic level simulation using the multi-level simulator with zero active ROIs is stated in column 5, while the runtime of a full simulation at switch level with all nodes marked as active is presented in the last column. In-between a mixed-abstraction simulation is performed with increasing amount of ROIs: First in absolute amount (Col. 6–7) followed by relative amount with respect to all circuit nodes in percent (Col. 8–13). The ROIs of the mixed abstraction scenarios were chosen randomly. All runtimes shown were averaged from three simulation runs with unique seeds for drawing the random numbers. Each row of a circuit also reports the runtime savings $S(x)$ of the multi-level simulator compared to a full simulation at switch level computed as

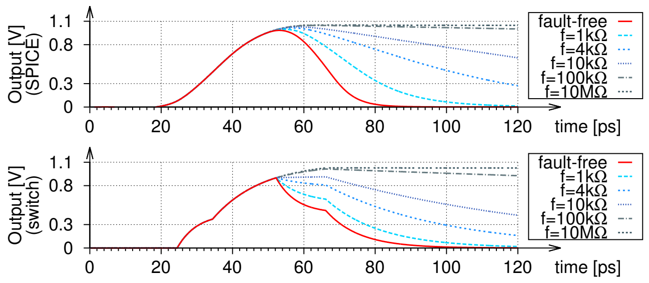
$$S(x) := 100\% \cdot \left(1 - \frac{T_{ML}(x)}{T_{swl}}\right), \tag{10}$$

where $T_{ML}(x)$ is the runtime of the multi-level simulator for a given ROI scenario $x$, and $T_{swl}$ is the reference runtime of a full switch level simulation.

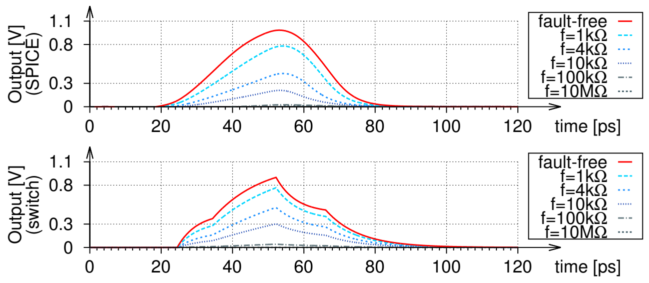
As shown, for all simulation scenarios the runtimes were in the order of seconds for smaller circuits and a few minutes for

Table 1: Multi-level simulation runtime (fault-free) for the evaluation of simulation scenarios with different amounts of active regions of interest (ROIs) from lowest to highest. All ROI locations were chosen randomly.

| Circuit[1] | Nodes[2] | Pattern-Pairs[2] | Comm. Event-Driven[4] | | Full Logic-Level[5] | Mixed Abstraction (active ROIs) | | | | | | | | Full Switch-Level[14] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | #Nodes | | % of total Nodes | | | | | | |
| | | | | | | 1[6] | 100[7] | 1%[8] | 5%[9] | 10%[10] | 25%[11] | 50%[12] | 75%[13] | |
| s38417 | 19.0k | 348 | 3.11s | *runtime:*<br>*saving:* | 20ms<br>78.7% | 51ms<br>45.7% | 48ms<br>48.9% | 55ms<br>41.5% | 53ms<br>43.6% | 56ms<br>40.4% | 67ms<br>28.7% | 75ms<br>20.2% | 82ms<br>12.8% | 94ms<br>– |
| s38584 | 23.1k | 563 | 5.29s | *runtime:*<br>*saving:* | 28ms<br>79.7% | 66ms<br>52.2% | 74ms<br>46.4% | 55ms<br>60.1% | 75ms<br>45.7% | 67ms<br>51.4% | 85ms<br>38.4% | 104ms<br>24.6% | 112ms<br>18.8% | 138ms<br>– |
| b17 | 42.8k | 2135 | 35.13s | *runtime:*<br>*saving:* | 107ms<br>83.6% | 155ms<br>76.3% | 147ms<br>77.5% | 179ms<br>72.6% | 207ms<br>68.3% | 247ms<br>62.2% | 315ms<br>51.8% | 423ms<br>35.3% | 549ms<br>16.1% | 654ms<br>– |
| b18 | 125.3k | 3174 | 6:42m | *runtime:*<br>*saving:* | 436ms<br>86.4% | 568ms<br>82.3% | 580ms<br>81.9% | 650ms<br>79.8% | 884ms<br>72.5% | 939ms<br>70.7% | 1.18s<br>63.2% | 1.84s<br>42.8% | 2.46s<br>23.4% | 3.21s<br>– |
| b19 | 250.2k | 4651 | 0:17h | *runtime:*<br>*saving:* | 1.14s<br>87.2% | 1.34s<br>84.9% | 1.36s<br>84.7% | 1.84s<br>79.3% | 2.30s<br>74.1% | 2.67s<br>70.0% | 3.57s<br>59.9% | 5.09s<br>42.8% | 6.80s<br>23.5% | 8.89s<br>– |
| p500k | 527.0k | 5012 | 0:49h | *runtime:*<br>*saving:* | 4.05s<br>78.7% | 4.55s<br>76.0% | 4.09s<br>78.5% | 5.67s<br>70.1% | 6.41s<br>66.2% | 6.01s<br>68.3% | 8.64s<br>54.4% | 11.61s<br>38.8% | 15.21s<br>19.8% | 18.97s<br>– |
| p533k | 676.6k | 3417 | 1:07h | *runtime:*<br>*saving:* | 2.75s<br>84.3% | 3.79s<br>78.4% | 3.83s<br>78.2% | 3.66s<br>79.2% | 5.92s<br>66.3% | 5.54s<br>68.5% | 8.19s<br>53.4% | 10.50s<br>40.3% | 13.94s<br>20.7% | 17.58s<br>– |
| p951k | 1.09M | 7063 | 3:00h | *runtime:*<br>*saving:* | 16.99s<br>64.6% | 18.72s<br>61.0% | 15.27s<br>68.2% | 19.46s<br>59.5% | 21.13s<br>56.0% | 20.28s<br>57.8% | 26.87s<br>44.1% | 33.85s<br>29.5% | 40.20s<br>16.3% | 48.03s<br>– |
| p1522k | 1.09M | 17980 | 8:21h | *runtime:*<br>*saving:* | 29.87s<br>75.4% | 24.79s<br>79.6% | 31.67s<br>73.9% | 36.46s<br>70.0% | 43.58s<br>64.1% | 49.32s<br>59.4% | 59.23s<br>51.2% | 1:18m<br>34.9% | 1:42m<br>15.4% | 2:01m<br>– |
| p2927k | 1.67M | 22107 | 18:17h | *runtime:*<br>*saving:* | 52.79s<br>77.3% | 54.63s<br>76.5% | 55.06s<br>76.3% | 1:07m<br>71.0% | 1:30m<br>61.0% | 1:44m<br>55.1% | 1:47m<br>53.5% | 2:24m<br>37.7% | 3:08m<br>18.8% | 3:52m<br>– |
| p3188k | 2.85M | 26502 | 42:02h | *runtime:*<br>*saving:* | 2:23m<br>73.2% | 2:22m<br>73.5% | 2:00m<br>77.4% | 2:39m<br>70.2% | 3:05m<br>65.4% | 3:31m<br>60.5% | 3:56m<br>55.9% | 5:23m<br>39.6% | 7:08m<br>20.1% | 8:55m<br>– |
| p3726k | 3.56M | 15512 | 39:24h | *runtime:*<br>*saving:* | 1:31m<br>79.8% | 1:27m<br>80.7% | 1:25m<br>81.2% | 3:29m<br>53.6% | 6:36m<br>12.1% | 4:30m<br>40.1% | 3:33m<br>52.6% | 4:56m<br>34.5% | 6:24m<br>15.0% | 7:31m<br>– |
| p3847k | 2.96M | 31653 | 40:12h | *runtime:*<br>*saving:* | 2:40m<br>75.3% | 3:00m<br>72.3% | 3:31m<br>67.5% | 3:39m<br>66.2% | 3:39m<br>66.4% | 3:49m<br>64.8% | 4:31m<br>58.4% | 7:03m<br>35.1% | 8:06m<br>25.3% | 0:10h<br>– |
| p3881k | 3.69M | 12092 | 23:53h | *runtime:*<br>*saving:* | 1:13m<br>76.1% | 1:43m<br>66.3% | 1:19m<br>74.4% | 1:27m<br>71.8% | 1:38m<br>68.1% | 2:20m<br>54.4% | 2:30m<br>51.2% | 3:07m<br>39.2% | 3:54m<br>24.2% | 5:08m<br>– |



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



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

Figure 12: Resistive-open fault in a) NMOS- and b) PMOS-transistors [67, 68] of a two-input NOR-cell [69] in presence of an input hazard.

the larger designs all outperforming the simulation of the commercial event-driven approach. In case of the mixed-abstraction cases, the runtimes were observed to scale linearly with the amount ROIs active from lowest to highest. However, for scenarios with small ROI count (1 and 100), the runtimes strongly fluctuate due to more complex waveforms generated at ROIs and varying propagation conditions of the ROI-dependent location and fanout. When comparing the multi-level simulator to a native logic level simulation using [33, 52], the full logic simulation in the mixed-abstraction simulation was observed 40% slower in average for the smaller circuits and only 2% slower for the million-node designs. The difference in runtime is due to the increased number of processing resources required by all the kernel threads. The multi-level evaluation kernel requires roughly double the amount of local processor registers compared to the native logic level evaluation kernel. However, the amount of registers is the same as in the native switch level kernel. Consequently, with the higher amount of resources required per thread, the number of concurrent threads running per multi-processor need to be reduced which results in less freedom for the thread schedulers and less effective utilization of the multi-processors during execution. Compared to the native switch level simulation, the processing of the full-switch level scenario multi-level simulator was observed to be 5% slower in average than its native counterpart, but for the million-node designs also 2% faster. Runtimes deviations in this magnitude will therefore be considered as negligible random fluctuations.

Fig. 13 investigates the speedup of the multi-level simulator (left y-axis) compared to the commercial unparallelized timing simulator at logic level for all circuits and mixed-abstraction scenarios in more detail. The speedups of the logic level simulation range from 155× to more than 1600×, while the switch level speedups range between 33× (s38417) and 314× (p3726k). In general, the speedups tend to be higher for the larger designs due to a better utilization of the GPU computing resources. The red line in the figure (right y-axis) indicates the ratio of the speedups from full switch level and full logic level simulation, which shows a runtime difference ranging from 3× up to 8× (average 6×).

## 8.3. *Throughput Performance*

The simulation performance of the presented approach is summarized in Fig. 14. On the left side, the simulation throughput in *million node evaluations per second* (MEPS) depending on the circuit size is shown, while the right side presents the average simulation time spent per pattern pair in milliseconds. The multi-level simulator was able to achieve more than 1000 MEPS at full logic level (average 740 MEPS) and an average of over 132 MEPS in the full switch level case with a peak performance of 174 MEPS. Again, the switch and logic level simulation performances differ by roughly one order of magnitude. As for the per-pattern runtimes, the average runtimes spent per stimuli ranged from less than 0.1ms to 25ms for the largest circuit investigated. While the effective parallelism from nodes typically diminishes on deeper levels of a design, the available stimuli parallelism compensates for this loss, allowing for a good utilization of the GPU.

As shown, the average runtime time per pattern trend scales with the circuit size and the throughput performance shows a constant trend due to full utilization of the GPU device. This saturation can be overcome by running multiple simulation processes and parallelization across multiple devices on recent architectures [70]. All fault groups and stimuli are independent simulation problems and can be partitioned and distributed for execution on different GPUs or even compute nodes. Especially for the larger circuits, the process initialization time is negligible and the overall execution time is dominated by the actual simulation for processing the faults and stimuli sets. Hence, for a sufficiently large set of simulation problems, the theoretical speedup obtained would be linear in the number of considered GPU devices and allow for further scalability.

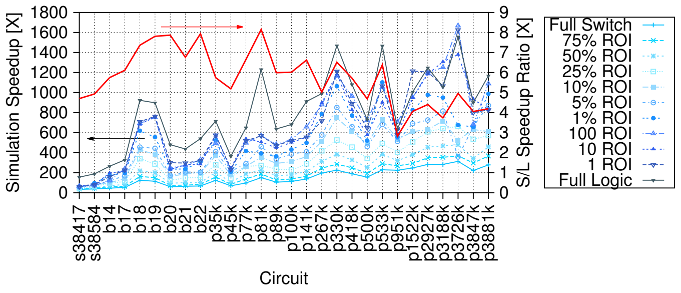


Figure 13: Speedup of the multi-level simulation compared to commercial logic level time simulation and *full-switch*-to-*full-logic* level (S/L) speedup ratio.

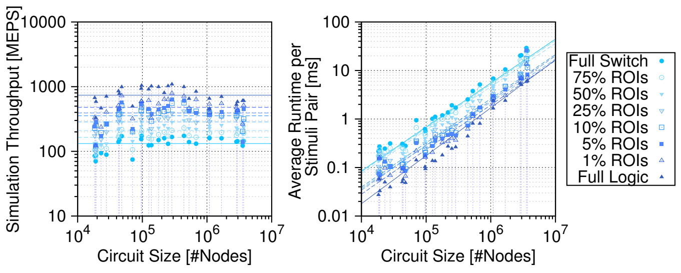


Figure 14: Simulation throughput in MEPS (left) and average simulation time per stimuli pair (right) for different ROI scenarios.

## 8.4. *Simulation Efficiency*

The implemented multi-level fault simulation approach utilizes sparse ROI activation for increasing speedup and simulation efficiency of low level fault simulations. For this, a *single-fault-single-ROI* activation is assumed upon injection of a fault for each fault of a fault group.

In this work, a location-exhaustive low-level parametric fault set is assumed where a single fault is contained for each transistor in the design. The fault set was grouped using the fault grouping heuristic of [50] and no fault dropping is performed. Fig. 15 reports the resulting sizes of the fault groups obtained (solid lines), which were plotted on the x-axis normalized due to different magnitudes in the circuit sizes. Furthermore, the cumulative amount of processed faults with respect to the full fault set is indicated by the dotted lines. The leftmost groups cover the most faults at once, since they contain faults that are closer to the circuit outputs which have a higher probability of being mutually output-independent. After processing 25% of the fault groups already more than 90% of the overall faults were processed in average and more than half of the groups contained in average more than ten faults, thus indicating the effectiveness of the fault-parallel processing. The size of each fault group was less than 10% of the amount of nodes contained in its respective circuit.

The efficiency of the multi-level fault simulation becomes evident from Fig. 16, which illustrates the runtime savings for the different mixed-abstraction scenarios. Since for active ROIs, the node descriptions have to be updated on the GPU anyway, the fault injection scheme itself introduces no additional costs [52, 50]. With the number of ROIs of the injected faults of each fault group being less than 10% of the total circuit nodes

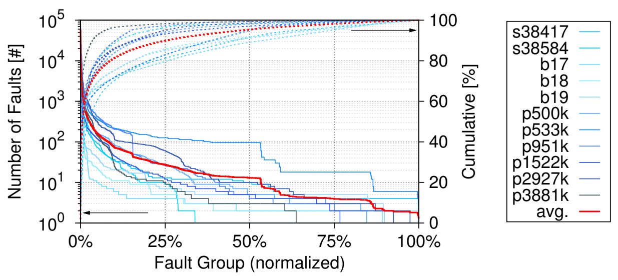


Figure 15: Sizes of obtained fault groups and cumulative amount of faults processed [50].

at all time, the average runtime savings of the simulation of a group will range from 60% for the larger to 70% for the smaller fault groups (minimum 1 fault). This way, the presented multi-level simulation approach allows for fast and efficient realistic fault simulation on GPUs.

## 9. Conclusion

In this paper the first high-throughput multi-level timing simulator for fast and efficient fault simulation on graphics processing units (GPUs) was presented. It utilizes waveform-accurate evaluation in a mixed-abstraction fashion by considering both logic and switch level descriptions simultaneously for a trade-off in simulation speed and modeling accuracy. The presented approach carefully exploits similarities in data-structures and execution patterns of the different abstraction levels and transparently transitions between the abstractions using waveform transformations. The presented fault modeling allows for both higher level (e.g., small delay faults) as well as lower level parametric and parasitic fault models (e.g., resistive open transistor) and highly efficient fault simulation is achieved through combination of fault injection at lower level with fault propagation at higher level and comprehensive syndrome analysis. Experiments have shown that the implemented GPU-accelerated multi-level fault simulator outperforms conventional commercial timing simulation solutions at logic level by speedups up to 1600× with a simulation throughput of more than 1000 million node evaluations per second, and achieves runtime savings of up to 84% compared to a full switch level simulation at GPU while being scalable for designs with millions of cells.

## Acknowledgment

This work has been funded by the German Research Foundation (DFG) under the project PARSIVAL (WU 245/16-1).

## References

[1] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, M. R. Stan, HotSpot: A Compact Thermal Modeling Methodology for Early-Stage VLSI Design, IEEE Trans. on Very Large Scale Integration Systems (TVLSI) 14 (5) (2006) 501–513. doi:10.1109/TVLSI.2006.876103.

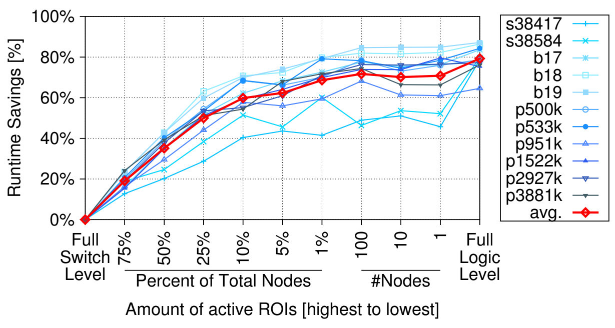


Figure 16: Runtime savings of sparse ROI activation compared to full switch-level simulation.

[2] P. Girard, N. Nicolici, X. Wen (Eds.), Power-Aware Testing and Test Strategies for Low Power Devices, Springer New York, 2010. doi:10.1007/978-1-4419-0928-2.

[3] M. Tehranipoor, K. Peng, K. Chakrabarty, Test and Diagnosis for Small-Delay Defects, Springer New York, 2011. doi:10.1007/978-1-4419-8297-1.

[4] Y. Yamato, T. Yoneda, K. Hatayama, M. Inoue, A Fast and Accurate Per-Cell Dynamic IR-drop Estimation Method for At-Speed Scan Test Pattern Validation, in: Proc. IEEE Int'l Test Conf. (ITC), 2012, pp. 1–8, Paper 6.2. doi:10.1109/TEST.2012.6401549.

[5] J. Jiang, M. Aparicio, M. Comte, F. Aza is, M. Renovell, I. Polian, MIRID: Mixed-Mode IR-Drop Induced Delay Simulator, in: Proc. 22nd Asian Test Symp. (ATS), 2013, pp. 177–182. doi:10.1109/ATS.2013.41.

[6] F. Hapke, W. Redemund, A. Glowatz, J. Rajski, M. Reese, M. Hustava, M. Keim, J. Schloeffel, A. Fast, Cell-Aware Test, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 33 (9) (2014) 1396–1409. doi:10.1109/TCAD.2014.2323216.

[7] H. H. Chen, S. Y. H. Chen, P. Y. Chuang, C. W. Wu, Efficient Cell-Aware Fault Modeling by Switch-Level Test Generation, in: Proc. IEEE 25th Asian Test Symp. (ATS), 2016, pp. 197–202. doi:10.1109/ATS.2016.33.

[8] A. D. Singh, Cell Aware and Stuck-Open Tests, in: Proc. IEEE 21st European Test Symp. (ETS), 2016, pp. 1–6, Paper 15.1. doi:10.1109/ETS.2016.7519316.

[9] R. L. Wadsack, Fault Modeling and Logic Simulation of CMOS and MOS Integrated Circuits, The Bell System Technical Journal 57 (5) (1978) 1449–1474. doi:10.1002/j.1538-7305.1978.tb02106.x.

[10] J. C. Li, C.-W. Tseng, E. J. McCluskey, Testing for Resistive Opens and Stuck Opens, in: Proc. Int'l Test Conf. (ITC), 2001, pp. 1049–1058, Paper 38.2. doi:10.1109/TEST.2001.966731.

[11] C. Han, A. Singh, Testing Cross Wire Opens within Complex Gates, in: Proc. IEEE 33rd VLSI Test Symp. (VTS), 2015, pp. 1–6. doi:10.1109/VTS.2015.7116301.

[12] C. Han, A. D. Singh, Improving CMOS Open Defect Coverage Using Hazard Activated Tests, in: Proc. IEEE 32nd VLSI Test Symp. (VTS), 2014, pp. 1–6. doi:10.1109/VTS.2014.6818740.

[13] S. Eggersglüß, R. Drechsler, As-Robust-As-Possible Test Generation in the Presence of Small Delay Defects using Pseudo-Boolean Optimization, in: Proc. Conf. on Design, Automation & Test in Europe (DATE), 2011, pp. 1–6. doi:10.1109/DATE.2011.5763207.

[14] I. Pomeranz, S. M. Reddy, Hazard-Based Detection Conditions for Improved Transition Fault Coverage of Scan-Based Tests, IEEE Trans. on Very Large Scale Integration (VLSI) Systems (TVLSI) 18 (2) (2010) 333–337. doi:10.1109/TVLSI.2008.2010216.

[15] H. Konuk, On Invalidation Mechanisms for Non-Robust Delay Tests, in: Proc. Int'l Test Conf. (ITC), 2000, pp. 393–399, Paper 14.3. doi:10.1109/TEST.2000.894230.

[16] E. Melcher, W. Röthig, M. Dana, Multiple input transitions in CMOS gates, Microprocessing and Microprogramming 35 (1–5) (1992) 683–690. doi:10.1016/0165-6074(92)90387-M.

[17] L.-C. Chen, S. K. Gupta, M. A. Breuer, A New Gate Delay Model for Simultaneous Switching and Its Applications, in: Proc. 38th Design Automation Conf. (DAC), 2001, pp. 289–294, Paper 19.2. doi:10.1109/DAC.2001.156153.

[18] Y. M. Kim, T. W. Chen, Y. Kameda, M. Mizuno, S. Mitra, Gate-Oxide Early-Life Failure Identification using Delay Shifts, in: Proc. 28th VLSI Test Symp. (VTS), 2010, pp. 69–74. doi:10.1109/VTS.2010.5469615.

[19] S. Hellebrand, T. Indlekofer, M. Kampmann, M. A. Kochte, C. Liu, H.-J. Wunderlich, FAST-BIST: Faster-than-At-Speed BIST Targeting Hidden Delay Defects, in: Proc. IEEE Int'l Test Conf. (ITC), 2014, pp. 1–8, Paper 29.3. doi:10.1109/TEST.2014.7035360.

[20] M. Abramovici, B. Krishnamurthy, R. Mathews, B. Rogers, M. Schulz, S. Seth, J. Waicukauski, What is the Path to Fast Fault Simulation?, in: Proc. Int'l Test Conf. (ITC), 1988, pp. 183–192. doi:10.1109/TEST.1988.207796.

[21] V. S. Iyengar, D. T. Tang, On simulating faults in parallel, in: Proc. 18th Int'l Symp. on Fault-Tolerant Computing (FTCS), 1988, pp. 110–115. doi:10.1109/FTCS.1988.5307.

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

[23] M. L. Bailey, J. V. Briner, Jr., R. D. Chamberlain, Parallel Logic Simula-

tion of VLSI Systems, ACM Computing Surveys 26 (3) (1994) 255–294. doi:10.1145/185403.185424.

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

[25] K. Gulati, S. P. Khatri, Towards Acceleration of Fault Simulation using Graphics Processing Units, in: Proc. ACM/IEEE 45th Design Automation Conf. (DAC), 2008, pp. 822–827, Paper 45.1. doi:10.1145/1391469.1391679.

[26] D. Chatterjee, A. DeOrio, V. Bertacco, Event-Driven Gate-Level Simulation with GP-GPUs, in: Proc. ACM/IEEE 46th Design Automation Conf. (DAC), 2009, pp. 557–562. doi:10.1145/1629911.1630056.

[27] X. Chen, L. Ren, Y. Wang, H. Yang, GPU-Accelerated Sparse LU Factorization for Circuit Simulation with Performance Modeling, IEEE Trans. on Parallel and Distributed Systems (TPDS) 26 (3) (2015) 786–795. doi:10.1109/TPDS.2014.2312199.

[28] K. He, S. X. D. Tan, H. Wang, G. Shi, GPU-Accelerated Parallel Sparse LU Factorization Method for Fast Circuit Analysis, IEEE Trans. on Very Large Scale Integration (VLSI) Systems 24 (3) (2016) 1140–1150. doi:10.1109/TVLSI.2015.2421287.

[29] M. Li, M. S. Hsiao, 3-D Parallel Fault Simulation With GPGPU, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 30 (10) (2011) 1545–1555. doi:10.1109/TCAD.2011.2158432.

[30] M. A. Kochte, M. Schaal, H.-J. Wunderlich, C. G. Zoellin, Efficient Fault Simulation on Many-Core Processors, in: Proc. ACM/IEEE 47th Design Automation Conf. (DAC), 2010, pp. 380–385, Paper 23.4. doi:10.1145/1837274.1837369.

[31] K. Gulati, S. P. Khatri, Fault Table Computation on GPUs, Journal of Electronic Testing 26 (2) (2010) 195–209. doi:10.1007/s10836-010-5147-x.

[32] M. Beckler, R. D. Blanton, Fault Simulation Acceleration for TRAX Dictionary Construction using GPUs, in: Proc. IEEE Int'l Test Conf. (ITC), 2017, pp. 1–9, Paper A.3. doi:10.1109/TEST.2017.8242078.

[33] S. Holst, M. E. Imhof, H.-J. Wunderlich, High-Throughput Logic Timing Simulation on GPGPUs, ACM Trans. on Design Automation of Electronic Systems 20 (3) (2015) 1–22, Article 37. doi:http://dx.doi.org/10.1145/2714564.

[34] E. Schneider, S. Holst, X. Wen, H.-J. Wunderlich, Data-Parallel Simulation for Fast and Accurate Timing Validation of CMOS Circuits, in: Proc. IEEE/ACM 33rd Int'l Conf. on Computer-Aided Design (ICCAD), 2014, pp. 17–23. doi:10.1109/ICCAD.2014.7001324.

[35] R. E. Bryant, A Survey of Switch-Level Algorithms, IEEE Design & Test of Computers 4 (4) (1987) 26–40. doi:10.1109/MDT.1987.295146.

[36] J. P. Hayes, An Introduction to Switch-Level Modeling, IEEE Design & Test of Computers 4 (4) (1987) 18–25. doi:10.1109/MDT.1987.295145.

[37] J. P. Hayes, Digital Simulation with Multiple Logic Values, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 5 (2) (1986) 274–283. doi:10.1109/TCAD.1986.1270196.

[38] S. Gai, P. L. Montessoro, F. Somenzi, MOZART: A Concurrent Multilevel Simulator, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 7 (9) (1988) 1005–1016. doi:10.1109/43.7799.

[39] W. Meyer, R. Camposano, Active Timing Multilevel Fault-Simulation with Switch-Level Accuracy, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 14 (10) (1995) 1241–1256. doi:10.1109/43.466340.

[40] M. Radetzki, R. S. Khaligh, Accuracy-adaptive Simulation of Transaction Level Models, in: Proc. Conf. on Design, Automation and Test in Europe (DATE), 2008, pp. 788–791. doi:10.1145/1403375.1403566.

[41] M. A. Kochte, C. G. Zöllin, R. Baranowski, M. E. Imhof, H.-J. Wunderlich, N. Hatami, S. Di Carlo, P. Prinetto, Efficient Simulation of Structural Faults for the Reliability Evaluation at System-Level, in: Proc. IEEE 19th Asian Test Symp. (ATS), 2010, pp. 3–8. doi:http://dx.doi.org/10.1109/ATS.2010.10.

[42] R. S. Khaligh, M. Radetzki, Modeling Constructs and Kernel for Parallel Simulation of Accuracy Adaptive TLMs, in: Proc. Conf. on Design, Automation Test in Europe (DATE), 2010, pp. 1183–1188. doi:10.1109/DATE.2010.5456987.

[43] N. Hatami, R. Baranowski, P. Prinetto, H.-J. Wunderlich, Multilevel Simulation of Nonfunctional Properties by Piecewise Evaluation, ACM Trans. on Design Automation of Electronic Systems (TODAES) 19 (4) (2014) 37:1–37:21. doi:10.1145/2647955.

[44] E. Schneider, M. A. Kochte, H.-J. Wunderlich, Multi-Level Timing Simulation on GPUs, in: Proc. 23rd Asia and South Pacific Design Automation Conf. (ASP-DAC), 2018, pp. 470–475. doi:10.1109/ASPDAC.2018.8297368.

[45] NVIDIA Corporation, NVIDIA Tesla Supercomputing — NVIDIA (2018).
URL `http://www.nvidia.com/tesla/`

[46] K. Gulati, J. F. Croix, S. P. Khatri, R. Shastry, Fast Circuit Simulation on Graphics Processing Units, in: Proc. 14th Asia and South Pacific Design Automation Conf. (ASP-DAC), 2009, pp. 403–408, Paper 4C–6s. doi:10.1109/ASPDAC.2009.4796514.

[47] L. Han, X. Zhao, Z. Feng, TinySPICE: A Parallel SPICE Simulator on GPU for Massively Repeated Small Circuit Simulations, in: Proc. ACM/EDAC/IEEE 50th Design Automation Conf. (DAC), 2013, pp. 1–8, Article 89.

[48] E. G. Ulrich, Exclusive Simulation of Activity in Digital Networks, Communications of the ACM 12 (2) (1969) 102–110. doi:10.1145/362848.362870.

[49] Z. Barzilai, D. K. Beece, L. M. Huisman, V. S. Iyengar, G. M. Silberman, SLS-a fast switch-level simulator [for MOS], IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 7 (8) (1988) 838–849. doi:10.1109/43.3214.

[50] E. Schneider, H.-J. Wunderlich, SWIFT: Switch Level Fault Simulation on GPUs, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD) (2018) 1–14DOI: 10.1109/TCAD.2018.2802871. doi:10.1109/TCAD.2018.2802871.

[51] F. J. Ferguson, J. P. Shen, Extraction and Simulation of Realistic CMOS Faults using Inductive Fault Analysis, in: Proc. Int'l Test Conf. (ITC), 1988, pp. 475–484. doi:10.1109/TEST.1988.207759.

[52] E. Schneider, M. A. Kochte, S. Holst, X. Wen, H. J. Wunderlich, GPU-Accelerated Simulation of Small Delay Faults, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 36 (5) (2017) 829–841. doi:10.1109/TCAD.2016.2598560.

[53] IEEE Computer Society, IEEE Standard for Standard Delay Format (SDF) for the Electronic Design Process, IEEE Std 1497-2001doi:10.1109/IEEESTD.2001.93359.

[54] IEEE Computer Society, IEEE Standard for Integrated Circuit (IC) Open Library Architecture (OLA), IEEE Std 1481-2009 (2010) c1–658doi:10.1109/IEEESTD.2009.5430852.

[55] A. E. Ruehli, G. S. Ditlow, Circuit Analysis, Logic Simulation, and Design Verification for VLSI, Proceedings of the IEEE 71 (1) (1983) 34–48. doi:10.1109/PROC.1983.12525.

[56] A. Czutro, N. Houarche, P. Engelke, I. Polian, M. Comte, M. Renovell, B. Becker, A Simulator of Small-Delay Faults Caused by Resistive-Open Defects, in: Proc. 13th European Test Symp. (ETS), 2008, pp. 113–118. doi:10.1109/ETS.2008.19.

[57] IEEE Computer Society, IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008 (2008) 1–70doi:10.1109/IEEESTD.2008.4610935.

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

[59] F. J. Ferguson, J. P. Shen, A CMOS Fault Extractor for Inductive Fault Analysis, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 7 (11) (1988) 1181–1194. doi:10.1109/43.9188.

[60] C. Sebeke, J. P. Teixeira, M. J. Ohletz, Automatic Fault Extraction and Simulation of Layout Realistic Faults for Integrated Analogue Circuits, in: Proc. European Conf. on Design and Test (EDTC), 1995, pp. 464–468. doi:10.1109/EDTC.1995.470319.

[61] E. Schneider, H.-J. Wunderlich, High-Throughput Transistor-Level Fault Simulation on GPUs, in: Proc. IEEE 25th Asian Test Symp. (ATS), 2016, pp. 151–156.

[62] N. H. E. Weste, D. M. Harris, CMOS VLSI Design – A Circuits and Systems Perspective, Addison-Wesley, 2011.

[63] M. Shao, Y. Gao, L.-P. Yuan, M. D. R. Wong, IR drop and Ground Bounce Awareness Timing Model, in: Proc. IEEE Computer Society Annual Symposium on VLSI: New Frontiers in VLSI Design (ISVLSI), 2005, pp. 226–231. doi:10.1109/ISVLSI.2005.44.

[64] K. U. Giering, C. Sohrmann, G. Rzepa, L. Heiß, T. Grasser, R. Jancke, NBTI modeling in analog circuits and its application to long-term aging simulations, in: Proc. IEEE Int'l Integrated Reliability Workshop (IIRW), 2014, pp. 29–34. doi:10.1109/IIRW.2014.7049501.

[65] D. Lorenz, G. Georgakos, U. Schlichtmann, Aging Analysis of Circuit Timing Considering NBTI and HCI, in: Proc. 15th IEEE Int'l On-Line Testing Symp. (IOLTS), 2009, pp. 3–8. doi:10.1109/IOLTS.2009.5195975.

[66] M. Shoji, Elimination of Process-Dependent Clock Skew in CMOS VLSI, IEEE Journ. of Solid-State Circuits (JSSC) 21 (5) (1986) 875–880. doi:10.1109/JSSC.1986.1052620.

[67] W. Zhao, Y. Cao, New Generation of Predictive Technology Model for Sub-45 nm Early Design Exploration, IEEE Trans. on Electron Devices 53 (11) (2006) 2816–2823. doi:10.1109/TED.2006.884077.

[68] Nanoscale Integration and Modeling (NIMO) Group, Predictive Technology Model (PTM) (2017).
URL `http://ptm.asu.edu/`

[69] Nangate Inc., NanGate 45nm Open Cell Library (2018).
URL `http://www.nangate.com/`

[70] NVIDIA Corporation, NVIDIA DGX-2 (Apr. 2018).
URL `http://www.nvidia.com/DGX-2`