GPU-Accelerated Small Delay Fault Simulation

Eric Schneider*, Stefan Holst†, Michael A. Kochte*, Xiaoqing Wen†, Hans-Joachim Wunderlich*
*University of Stuttgart, Pfaffenwaldring 47, 70569 Stuttgart, Germany
{schneiec,kochte,wu}@iti.uni-stuttgart.de
†Kyushu Institute of Technology, 680-4 Kawazu, Iizuka 820-8502, Japan
{holst,wen}@ci.kyutech.ac.jp

***Abstract*—The simulation of delay faults is an essential task in design validation and reliability assessment of circuits. Due to the high sensitivity of current nano-scale designs against smallest delay deviations, small delay faults recently became the focus of test research. Because of the subtle delay impact, traditional fault simulation approaches based on abstract timing models are not sufficient for representing small delay faults. Hence, timing accurate simulation approaches have to be utilized, which quickly become inapplicable for larger designs due to high computational requirements. In this work we present a waveform-accurate approach for fast high-throughput small delay fault simulation on Graphics Processing Units (GPUs). By exploiting parallelism from gates, faults and patterns, the proposed approach enables accurate exhaustive small delay fault simulation even for multi-million gate designs without fault dropping for the first time.**

I. Introduction

Today's nano-scale circuit manufacturing processes undergo high amounts of random and systematic variation during production [1]. With the high performance demands and strict low power requirements near threshold, current designs are run close to their operational limit and thus are highly sensitive to delay faults [2]. Slightest deviations in the physical layout of gates are sufficient to cause so called *small delay faults*, that relate to resistive open defects [3] or varying transistor threshold voltages [4]. A *small delay fault* is considered as a manifestation at an input or output pin of a gate that slows down signal transitions by a small additional amount of time. The faults may cause signal propagation to exceed the nominal clock period before eventually attaining a stable output value (Fig. 1). In contrast to traditional gross delay fault models (such as transition faults [5]), the delay amount introduced is much smaller than the clock period, yet able to cause the device to fail under operating conditions or indicate initial signs of *early life failures* (ELF) [6]. Since small delays are hard to detect and the testing of these faults is quite complex, they have become the focus of recent test research [7–9].

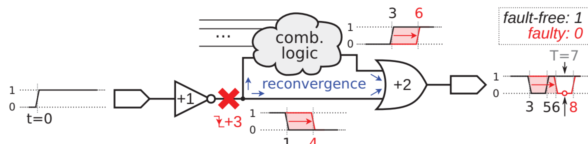

Fig. 1. Example of a *slow-to-fall* small delay fault of *size* 3 at a reconvergent fanout gate.

Well-known fault models, like stuck-at or transition faults, are not sufficient to represent the subtle delay effects of small delay faults [10, 11]. In case of the reconvergent fanout in the example of Fig. 1, a *slow-to-fall* transition fault at the fault location would cause a constant high (fault-free) output signal in zero delay simulation and hence go undetected. On the other hand, the detection probability of a transition fault can be much larger, since all propagation paths in its output cone are affected regardless of the actual path delays, resulting in rather optimistic fault coverages for small delays. The efficient and accurate simulation of small delays is a non-trivial task and involves the computation of the circuit timing, which has higher computational demands than plain logic simulation. Previous algorithms to evaluate circuit timing typically utilize a static analysis with bounded gate delays [12–14] or probabilistic approaches [15] to model the delay propagation in the circuit, which are either too pessimistic or assume robust fault propagation during simulation. Since not all structures can be tested robustly, the presence of hazards or reconvergences may cause the fault detection of tests to be invalidated [11, 16, 17], which renders these approaches inaccurate. For detailed analysis of small delays, full waveform evaluation in the time domain is required to ensure accurate signal timing and to track all hazards. The authors of [18] proposed a waveform-based simulator that evaluates the coverage of resistive open defects by detection intervals in the time domain. Yet, these algorithms show fairly large runtimes even for small circuits, since they have been designed for use on regular CPUs.

With the introduction of the general purpose computing on *Graphics Processing Units* (GPU), the paradigm of the *many-core processing* emerged. GPUs contain many small processing elements and are able to execute thousands to millions of threads concurrently in a *single-instruction-multiple-data* (SIMD) fashion, making it able to achieve enormous speedups for compute intensive problems [19]. This has been pursued in *electronic design automation* (EDA) applications for accelerating circuit simulation [20, 21] and parallel fault simulation [22–27]. However, all these algorithms focus on acceleration of plain logic simulation and do not provide the accuracy required for considering small delay faults. In [28] a first GPU-accelerated logic timing simulator was presented, capable of processing industrial-sized circuits in the time-domain at full waveform granularity.

In this work we utilize the computing power of GPUs for the purpose of accelerating accurate and exhaustive small delay fault simulation. We propose an efficient approach for timing-accurate simulation of smallest delay faults that adopts the fundamental waveform concept of [28]. By exploiting available dimensions of parallelism and efficient organization, we enable for the first time exhaustive small delay fault simulation even without fault dropping for multi-million gate designs to accurately determine the small delay fault coverage even in presence of hazards and reconvergent fanouts.

II. Fundamental Requirements

With the data-parallel programming paradigm, GPU devices are capable of achieving massive computational throughput for acceleration of high-performance computing applications [19, 29]. However, this ability comes with certain

978-3-9815370-4-8/DATE15/©2015 EDAA

restrictions that often pose major problems when mapping algorithms into code for parallel execution (*kernels*) with many threads. First of all, all threads share the same global device memory of the GPU, which is limited (typically 4–6GB). The accesses are slow compared to the execution of bare arithmetic instructions and in addition, the amount of fast local memory that can be occupied by a single thread is scarce and has to be used efficiently in order to avoid memory spilling. It is important that each thread can run independently on its own working set, as information exchange between different threads can in general only be achieved through expensive global memory accesses and thread synchronization barriers. Furthermore, in accordance to the SIMD paradigm, threads are executed in batches by the processing elements, where each thread has to follow the same control flow by executing the same instruction at any time. The divergence of a thread from the batch causes an execution branch, which is handled serially by the thread scheduler and further reduces the execution speed until the control flow converges again. Also, data transfers and communication between host and device are performance bottlenecks and have to be minimized or avoided at all cost. Thus, efficient memory accesses and uniformity of the execution are of utmost importance for efficient parallelization [19].

Recent GPU-accelerated approaches for simulation of stuck-at faults [22–27] evaluate independent structures in the netlist (*structural-parallelism*) for multiple inputs (*data-parallelism*), such as patterns and faults, at once. As a general principle, each execution thread is assigned a certain structure (a single gate or a fanout-free region as a whole) and some input data to operate on independently. The algorithms utilize efficient gate evaluation based on look-up-tables and further exploit bit-level parallelism within single threads in order to increase data-parallelism. In [20] a circuit simulation approach is presented where the circuit netlist is partitioned into clusters for independent evaluation of output signals. Each cluster is simulated in parallel by a separate block of threads that process levelized gates in parallel. An event-driven solution based on a different circuit partitioning approach is presented in [21]. However, the independent simulation of the individual circuit partitions requires the duplication of gates and reduces the effective global memory of the GPU.

So far, the above mentioned algorithms follow similar concepts of exploiting parallelism in circuit simulation, but without the consideration of circuit timing. In [30] the authors propose a GPU-accelerated *statistical static timing analysis* (SSTA) that processes Monte-Carlo instances simultaneously. The approach accelerates delay computation by parallelized generation and statistical evaluation of random numbers. However, SSTA only performs probabilistic analysis of the circuit timing and does not consider the actual propagation of signal transitions, which is not suitable for determining the detection of a given small delay fault. The accurate investigation of small delay faults with circuit delays requires *waveform-accurate* evaluation methods, that are able to cope with the additional complexity of time. The authors in [28] proposed a GPU-accelerated time simulator for power estimation with a two-dimensional execution scheme that maximizes simulation throughput by exploiting structural gate- and data-parallelism (Fig. 2). The threads each process different gates and input waveforms with floating-point timing accuracy. The algorithm utilizes an efficient data encoding and storage management to compute full switching histories with low memory-footprint and synchronization overhead.

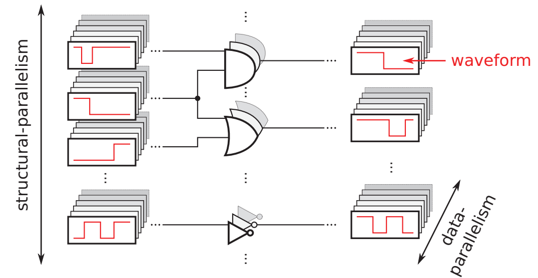

Fig. 2. Waveform principle with two dimensions of parallelism.

The small delay fault simulation approach proposed in this work simultaneously exploits (a) *gate-parallelism*, (b) *fault-parallelism* and (c) *pattern-parallelism* as illustrated in Fig. 3. We adopt the two-dimensional scheme of combining the gate- and pattern-parallelism from [28], which provides the vehicle for fast waveform-accurate time simulation. Fault-parallelism is exploited by evaluating groups of structurally independent small delay faults in the same simulation instance [31].

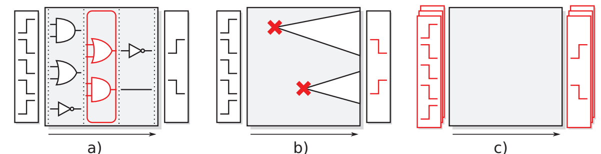

Fig. 3. Dimensions exploited for maximum throughput parallelization: a) gate-parallelism, b) fault-parallelism and c) pattern-parallelism.

Fig. 4 gives an overview of the proposed simulation approach. The upper part consists of necessary pre-processing steps for reading in the netlist, initializing the simulator and finding fault groups suitable for simulation. The bottom part comprises the actual simulation process composed of parallel fault injection and waveform-accurate time simulation followed by fault detection to capture the output responses at given sample times. The shaded boxes denote parallel actions.

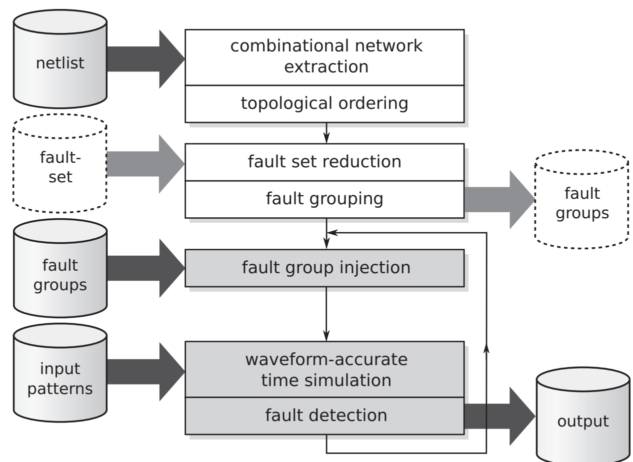

Fig. 4. Flow-chart of the overall simulation algorithm.

This work uses a pin-to-pin delay model and considers individual delay annotations for each input of a cell and transition polarity. The timing data is obtained from *Standard Delay Format* (SDF) files, which are input for the simulator. In the following, the applied parallelization methods of the small delay fault simulation will be explained in more detail.

III. Gate-Parallel Simulation

If two gates are neither in the input- nor in the output-cone of each other, they are mutually *data-independent* as the inputs of any of the gates do not depend on the output of the other and vice versa. Hence, the order of evaluation is not subject of matter, in contrast to data-dependent gates, where a partially ordered evaluation sequence has to be defined in order provide the necessary input signals for succeeding gates. With the parallel architectures, the evaluation of data-independent gates will be performed concurrently.

Fig. 5 depicts the scheme for parallel evaluation of a topologically ordered netlist. The topological ordering partitions the gates of a netlist into *levels* depending on their maximum topological distance to either circuit inputs or outputs. Typically *as-soon-as-possible* (ASAP) schedules are used for ordering. All gates in a partition are mutually data-independent. For every level, the gate evaluation kernel from [28] is invoked, which spawns a thread for each gate on the current level. The threads simultaneously compute the output of their corresponding gates as a list of temporally ordered signal switches (*waveform*) by processing toggle events of input waveforms in a merge-sort fashion from earliest to latest and store the result in the global waveform memory. The amount of parallelism that is exploited per evaluation is restricted by the number of gates residing on each level and typically mitigates towards the outputs in an ASAP-scheduled netlist. Furthermore, the runtime of the simulation depends on the circuit depth, since the levels have to be evaluated in sequential order by individual evaluation kernels. If at some point during the simulation a signal is not input of any of the remaining gates scheduled for evaluation, the associated waveform is deallocated to free memory [28]. This way the required waveform memory for a simulation instance is bound by the maximum number of waveforms of *alive* signals that have to be stored during the simulation run.

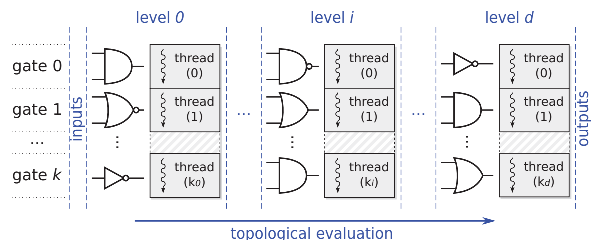

Fig. 5. Parallel evaluation sequence of data-independent gates in a topologically ordered netlist.

As the amount of waveform storage per signal is constrained, overflows might occur during evaluation. If an overflow is detected, the simulation run is repeated with intermediate checks after each level to identify the problem waveforms. The storage of the overfilled waveforms is then increased and reallocated before proceeding with the evaluation [28].

IV. Fault-Parallel Simulation

In order to reduce the amount of fault locations in advance, structural collapsing of the fault list is performed by arranging the faults into *equivalence classes*. All faults of an equivalence class show an identical behavior and thus require only a single *representative* fault for simulation. For small delay faults that affect both rising and falling transitions with the same delay amount, the equivalence rules of *transition faults* [5] are applied:

- If a gate has a single input, then the fault locations at input and output pin of the gate are *equivalent*.
- If a gate has a single fanout, then the location at the output pin and the corresponding input pin of the succeeding gate are *equivalent*.

These rules can be extended to support collapsing for faults with different rising and falling delays.

For simulating small delay faults we employ a parallelization scheme based on groups of independent faults [31]. If the output cones of two faults share no common output logic, they are referred to as *output-independent* as they propagate to different parts in the circuit. Since these faults have no mutual influence, they can be injected in the same simulation instance for parallel evaluation. In the following, such sets of output-independent faults will be referred to as *fault groups*. In order to find suitable fault groups for a given fault set, the mutual output-independence of each fault pair has to be mapped to an *output-independency graph*. Nodes in this graph represent faults and each edge connects two nodes iff the associated faults do not share any output logic. Hence, an edge indicates the eligibility of faults for parallel simulation. Each clique in the graph represents a fault group, since all pairs of nodes of the clique are connected and hence mutually output-independent. After obtaining a fault group, the faults can be injected for simulation and all nodes of the clique can be removed from the graph. For maximum simulation speedup it is favorable to keep the fault groups large and the group count low by processing as many faults as possible in parallel in the least amount of simulation instances. An optimal schedule of a given fault set requires to repeatedly find a *maximum clique* and remove the associated nodes until the remaining graph is completely empty. However, finding a maximum clique is an NP-complete problem which is not applicable for multi-million gate designs.

To allow efficient computation of fault groups for multi-million gate designs, we follow a heuristic approach as outlined in Fig. 6. The algorithm takes as input a set of fault locations, such as a pin of a certain gate. These fault locations are sorted in topological order from outputs towards inputs, which are then processed in an *as-late-as-possible* (ALAP) manner. Starting from a fault location f, the list of reachable outputs is first determined by traversing the netlist towards the circuit outputs. This list is then compared with the reachable outputs of a group in order to determine any shared outputs. A map is stored for each group G that holds the reachable outputs of all faults contained. Initially, the map of a group is empty. If the outputs of a group G and a fault f are disjoint, f is inserted into G and the reachable outputs of f are added to the output map of the group. If a common output has been detected, the comparison process is repeated for the next group in the list. In case that none of the currently existing groups is disjoint with f, a new group will be created.

Once a fault has been inserted into a group, the index of the group is assigned to the fault location and propagated towards the inputs by annotating the nodes in the input cone. This annotation is part of the heuristic and will be used as a starting group index when trying to find output-independent groups for further faults. This avoids unnecessary group comparisons when processing these faults, due to the transitive structural dependency of succeeding gates, since a fault f cannot be scheduled at the same time as the faults in its output cone. Initially, the starting group of every node is initialized with 0.

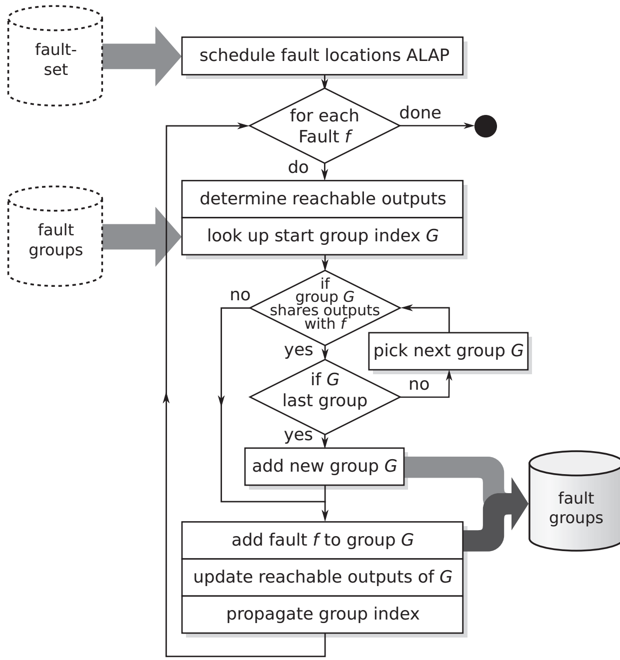

Fig. 6. Flow-chart of the fault grouping heuristic.

For a particular fault location f, the starting group $start(f)$ is then computed as:

$$start(f) := \max\{start(g) \mid g \in \{f \cup fanout(f)\}\} + 1$$

whose value is propagated throughout the *support-cone* of the fault location, by forward-propagation towards the primary outputs followed by a back-propagation to the primary inputs. The propagation is terminated at nodes that were already assigned higher group index. Since all faults are processed in ALAP-order, the starting group of each fault allows to quickly skip all comparisons of nodes in its output cone.

The time-simulator processes fault groups as a whole by injecting all faults of a group into a simulation instance. For each gate in a circuit, the data structure of its timing annotation is organized as a set D of tuples with pin delay values, with each tuple representing the delays for rising and falling transitions at a certain input-pin of the associated gate:

$$D = \{\{d^0_{rise}, d^0_{fall}\}, \{d^1_{rise}, d^1_{fall}\}, \ldots\}.$$

The underlying simulation algorithm (cf. Section III) processes transitions with respect to the polarity at the gate inputs and also annotates pin-to-pin delays at the gate inputs. Each small delay fault f is represented by a tuple $f = (loc, \{\delta_{rise}, \delta_{fall}\})$ consisting of a particular gate pin as *fault location* and a set of delay values for the rising and falling transition polarity as *fault size*. The injection of a fault into the circuit is done by modifying the gate timing descriptions prior to the simulation as follows:

- For a fault at an input the delay size values are added to the rising and falling delay values of the associated pin timing description of the affected gate.
- Faults at a gate output are modeled by injecting the delay into the delay descriptions of *every* input pin of the affected gate.

For the evaluation kernel, the presence of injected faults is completely transparent and thus causes no additional control-flow divergence during thread execution. After simulation of a fault group, all injected small delay faults are removed by restoring the nominal delay specification of all cells in the circuit marked as faulty. This way, the injection scheme causes only few small memory operations and thus keeps data communication and synchronization overhead at a minimum.

V. Pattern-Parallel Simulation

The time simulation algorithm (Section III) adopts a two-dimensional parallelization concept in which multiple gates are processed for different stimuli concurrently at a time. For the two-dimensional simulation, the evaluation kernels invoke a two-dimensional *grid* of execution threads as shown in Fig. 7. Threads in the vertical direction process the different gates on one topological level in parallel. In the horizontal direction, the threads each evaluate the same gate, yet operate on different input stimuli. The threads are scheduled in *batches* of 32 by the thread scheduler for simultaneous execution in the multi-processing cores. Each batch evaluates the exact same gate for multiple stimuli. In global memory, the necessary waveform toggle data is aligned in such a way that the memory accesses of the threads within a batch create fully utilized 128-byte memory transactions. This efficient coalescing of the waveform accesses and caching within thread blocks reduce the overall amount of global memory transactions and maximizes computational throughput.

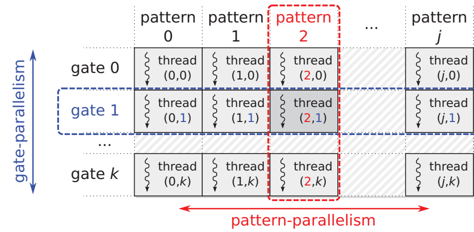

Fig. 7. Two-dimensional organization of concurrently executed threads.

The number of patterns that can be processed in parallel depends on the available memory and the memory required for a single simulation instance (Sec. III). In contrast to the level-dependent gate-parallelism, the available parallelism obtained from patterns remains constant throughout the simulation. If more patterns are provided than the memory can store at the same time, the simulation is split in a sequence of executions, each processing a different bunch of patterns. Therefore, larger memories allow for a higher degree of parallelization as more stimuli evaluations can be performed concurrently.

For the evaluation of the circuit responses, the signal values of all output waveforms are captured at a given sample time T. Again, a two-dimensional kernel grid is used with each thread traversing the toggle list in its associated output waveform $wave(t)$ until time T is reached. The waveform value $wave(T)$ then represents a captured value. In order to obtain the information about fault detection, the syndrome $syn(T)$ of the waveform is computed during the evaluation process. Since the simulated small delay faults are of finite size, the stabilized *good* value at each output can be acquired directly from the waveforms at $t = \infty$ without the need of an explicit *good value* simulation. The syndrome waveform $syn(t)$ of a signal $wave(t)$ is '1' iff the value $wave(t)$ is the opposite of its final stable value $wave(\infty)$:

$$syn(t) := wave(t) \oplus wave(\infty).$$

Therefore, if a fault has been detected at an output, the syndrome is '1', otherwise it is '0'. The syndrome values of the batches are encoded as 32-bit strings, which contain the compressed detection information, and the detection of each fault is determined by looking up all computed syndromes of the respective reachable outputs. The output sampling is not limited to a single capture time. As the output waveforms remain untouched during the capture process, multiple captures at different sample times can be evaluated quickly in succession. Furthermore, individual capture times can be provided for each output to model skew in the clock distribution tree.

VI. Experimental results

We compare the runtime of the proposed simulation approach, against a state-of-the-art commercial event-based logic-level time-simulator. Our set of benchmark circuits contains the largest designs from ISCAS'89 and ITC'99, as well as industrial designs provided by NXP. All designs have been mapped to the NanGate 45nm Open Cell Library [32]. During this process, the state elements were removed, thus leaving only the combinational circuit structure. For the evaluation of each benchmark circuit 10,240 random input stimuli (pairs) have been generated to be applied in succession. As a fault set we consider one small delay fault affecting both rising and falling transitions for each pin (inputs or outputs) of a gate. Given a specific clock frequency, the size of each small delay fault has been set halfway between the slack of the longest and shortest path through each location as obtained from static timing analysis. All experiments were executed on a NVIDIA® GeForce® GTX™ Titan Black consumer GPU card with 2880 cores, 6GB of global memory and a 900MHz clock. The host system contains 16 Intel® Xeon® processors clocked at 3.4GHz and 256GB of RAM. However, the peak memory consumption on the host system never exceeded 10GB.

Table I contains circuit and fault set information of the designs. The circuit names, their sizes, as well as the logic depths in gates are given in the first three columns. Columns 4 and 5 show the total number of faults in the universe before and after removing equivalent faults. The group pre-processing was able to partition the fault set into groups with *average* sizes ranging from 1.2 to 773.2 faults, as shown in column 6 and 7. This factor directly impacts the overall simulation runtime (inversely proportional), since the total number of simulation runs to process all faults equals the amount of fault groups. For circuit p469k, the grouping is less effective, due to a large number reconvergent fanouts in the circuit. On the other hand, p378k has less reconvergent fanouts and more data independent nodes, which allows to reduce the number of simulation instances by a factor of over 100×. For the circuits investigated, the runtime of the grouping heuristic, shown in the last column, ranges from seconds to roughly over an hour. Compared to the full serial simulation, this is a negligible amount of time spent for pre-processing, since the time for simulating the faults is still dominated by the number of instances. Even in the case of p469k, the small grouping factor of 1.2 saves almost 15 percent (≈ 22,000) of the total simulation instances.

Table II compares the runtime results of the presented simulation approach to the commercial event-based time-simulator. The first two columns list the circuit names and the average runtime of the event-based solution for evaluating a single circuit instance with all input patterns. Similar to [28] we split the runtimes of the proposed approach in *worst-case* and *best-case*. In a first run, the accelerated time simulator showed to be 18–832 times faster than the event-based solution (Col. 4). If the simulation is repeated, the overhead for the memory management of the waveform reallocation is reduced and eventually the simulator is able to run at full speed, showing an increase in speedup. This yields an additional improvement by factors of up to 38× as shown in columns five and six. Note that the speedup of the simulation run is independent of the fault grouping as the presence of faults is completely transparent to the evaluation kernel. The rightmost column contains the totaled runtime of the proposed approach for the exhaustive fault simulation of each fault and every pattern without *fault dropping*. Since the fault simulation is the repeated execution of simulation runs with the same pattern set applied every time, the memory calibration quickly converges, which results almost exclusively in full-speed runs. Therefore, the proposed simulator is able to effectively reduce the runtime of exhaustive small delay fault simulation.

TABLE I. Circuit and fault grouping statistics.

Circuit	Gates	Depth	Faults	Reduced	Groups	*avg.*	Runtime
s38417	27.5k	51	58.0k	25.4k	896	28.3	782ms
s38584	25.3k	61	57.7k	31.3k	1120	27.9	707ms
b17	38.5k	104	102.1k	70.3k	8299	8.5	23.33s
b18	131.7k	175	354.5k	240.2k	17.8k	13.5	51.32s
b19	265.1k	180	714.0k	484.3k	18.7k	25.9	3:33m
p35k	46.6k	72	107.8k	57.1k	29.9k	1.9	5:51m
p45k	45.1k	59	103.5k	59.6k	5220	11.4	8.61s
p77k	71.9k	555	174.9k	102.4k	32.2k	3.2	31.49s
p78k	74.9k	45	196.2k	138.3k	894	154.6	3.30s
p89k	90.2k	110	220.8k	132.8k	7000	19.0	19.18s
p100k	96.1k	103	227.9k	137.2k	5220	26.3	23.22s
p267k	272.6k	73	608.1k	303.3k	4947	61.3	1:15m
p330k	348.1k	70	825.3k	458.1k	32.2k	14.2	65:36m
p378k	374.5k	45	981.2k	691.3k	894	773.2	19.52s
p388k	482.4k	222	1.18M	714.1k	11.0k	64.8	4:10m
p418k	442.9k	206	1.01M	561.2k	9291	60.4	1:03m
p469k	97.4k	221	262.8k	154.3k	131.9k	1.2	27:24m
p533k	652.8k	114	1.61M	1.01M	4458	227.5	1:21m
p951k	1.01M	140	2.15M	1.25M	6733	186.2	3:26m
p1522k	1.09M	504	2.57M	1.46M	31.5k	46.4	42:12m

In Table III we compare the fault coverage of both transition and small delay faults for the applied random patterns. Column two shows the exact number of fault locations considered. Columns three and four contain the number of detected transition faults and the portion of small delay faults that have not been detected at the same location ("$\supseteq$ *SD und.*"). As shown, a fair amount of small delays could not be detected although the transition fault was detectable, confirming the well-known fact that transition faults overestimate the small delay fault coverage.

The last two columns show the detection numbers for small delay faults respectively. As expected, the small delay fault coverage is generally lower than the transition fault coverage. However, there are also numerable cases of small delay faults, which are detected despite their corresponding transition faults being not detected. Here, the small delay faults were propagated along reconvergent fanouts and caused hazards to appear at the circuit outputs while the output signals were being captured as depicted previously in Fig. 1. These faults showed to be detectable only for smaller (*finite*) fault sizes that cannot not be represented by transition faults. Although these cases seem rare, they are especially important for diagnosis and failure analysis, thus emphasizing the *necessity of fast and accurate small delay fault simulation* for determining the detection of small delays.

TABLE II. RUNTIME COMPARISON FOR 10,240 PATTERNS.

Circuit	Fault-Free Simulation					Exhaustive
	Event-	Cold-Run (GPU)		Re-Run (GPU)		GPU Fault
	Based	Time	X	Time	X	Simulation
s38417	2:04m	308ms	402	304ms	407	3:25m
s38584	1:35m	375ms	252	375ms	252	5:47m
b17	3:59m	1.23s	194	422ms	564	0:54h
b18	0:20h	8.84s	130	1.66s	693	8:55h
b19	1:10h	31.23s	132	3.22s	1285	21:60h
p35k	4:06m	1.59s	154	529ms	464	3:45h
p45k	2:54m	1.37s	127	569ms	305	0:44h
p77k	0:22h	13.29s	95	1.36s	930	13:16h
p78k	0:25h	3.58s	409	1.07s	1364	0:28h
p89k	7:58m	1.69s	282	941ms	507	1:36h
p100k	0:11h	4.73s	135	1.07s	600	1:52h
p267k	0:25h	16.41s	89	2.45s	596	3:13h
p330k	1:04h	28.33s	134	3.39s	1126	32:40h
p378k	4:12h	31.19s	483	4.65s	3243	4:08h
p388k	1:37h	1:47m	54	5.36s	1081	29:46h
p418k	0:52h	1:49m	28	4.45s	698	16:13h
p469k	4:09h	17.92s	832	5.49s	2716	154:02h
p533k	3:17h	2:06m	93	7.82s	1507	41:13h
p951k	2:49h	9:04m	18	34.02s	297	135:11h
p1522k	3:34h	0:12h	18	18.74s	683	315:54h

TABLE III. FAULT DETECTION OF TRANSITION FAULTS (TF) AND SMALL DELAYS (SD) AT SAME LOCATIONS.

Circuit	Faults	Transition (TF)		Small Delay (SD)	
		TF *det.*	$\supseteq$ SD *und.*	SD *det.*	$\supseteq$ TF *und.*
s38417	25384	22971	5726	17258	13
s38584	31303	29630	7735	21903	8
b17	70257	44118	29042	15120	44
b18	240221	159529	68725	91330	526
b19	484292	323084	154521	169240	677
p35k	57076	31810	14453	17447	90
p45k	59619	55040	15194	39876	30
p77k	102407	57887	18143	45473	5729
p78k	138256	138256	12910	125346	0
p89k	132812	95896	35512	60426	42
p100k	137197	128802	35849	93768	815

VII. CONCLUSION

This work presents an approach for enabling fast and accurate simulation of small delay faults on data-parallel GPU architectures. The fault simulation is waveform-accurate and supports individual rising and falling pin-to-pin delay annotations as well as glitch filtering. It maintains full information about hazards and glitches, and allows to determine the coverage of small delays in the presence of hazards and reconvergences. Rather than focusing on latency-optimized evaluation, the proposed method utilizes the many dimensions of parallelism found in circuit simulation (gates, faults and patterns) and careful memory management to attain maximum simulation throughput and speedup. Runtime results of the approach have shown speedups of up to three orders of magnitude compared to conventional logic-level timing simulators. With this significant simulation speedup, the proposed approach enables for the first time waveform-accurate and exhaustive small delay fault simulation even without fault dropping for large industrial designs with more than a million gates.

ACKNOWLEDGMENT

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

REFERENCES

[1] A. Srivastava, D. Sylvester, and D. Blaauw. *Statistical Analysis and Optimization for VLSI: Timing and Power*. Springer, 2005.

[2] The International Technology Roadmap for Semiconductors: 2013. http://www.itrs.net/Links/2013ITRS/Home2013.htm, 2014.

[3] R. R. Montanes, J. P. de Gyvez, and P. Volf. Resistance characterization for weak open defects. *IEEE Design Test of Computers*, 19(5):18–26, Sep. 2002.

[4] Y. Taur, D. A. Buchanan, W. Chen, et al. CMOS scaling into the nanometer regime. *Proceedings of the IEEE*, 85(4):486–504, Apr. 1997.

[5] J.A Waicukauski, E. Lindbloom, B. K. Rosen, et al. Transition Fault Simulation. *IEEE Design Test of Computers*, 4(2):32–38, Apr. 1987.

[6] Y. M. Kim, Y. Kameda, H. Kim, et al. Low-cost gate-oxide early-life failure detection in robust systems. In *Proc. IEEE Symp. on VLSI Circuits (VLSIC)*, pp. 125–126, 2010.

[7] X. Qian and A. D. Singh. Distinguishing Resistive Small Delay Defects from Random Parameter Variations. In *Proc. IEEE 19th Asian Test Symp. (ATS)*, pp. 325–330, 2010.

[8] M. Tehranipoor, K. Peng, and K. Chakrabarty. *Test and Diagnosis for Small-Delay Defects*. Springer, 2011.

[9] M. Sauer, A. Czutro, I. Polian, et al. Small-delay-fault ATPG with waveform accuracy. In *Proc. IEEE/ACM Int'l Conf. on Computer-Aided Design (ICCAD)*, pp. 30–36, 2012.

[10] E. S. Park, M. R. Mercer, and T. W. Williams. Statistical delay fault coverage and defect level for delay faults. In *Proc. Int'l Test Conf. (ITC)*, pp. 492–499, 1988.

[11] H. Konuk. On invalidation mechanisms for non-robust delay tests. In *Proc. Int'l Test Conf. (ITC)*, pp. 393–399, 2000.

[12] V. S. Iyengar, B. K. Rosen, and J. A. Waicukauski. On computing the sizes of detected delay faults. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 9(3):299–312, Mar. 1990.

[13] A. K. Pramanick and S. M. Reddy. On the fault coverage of gate delay fault detecting tests. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 16(1):78–94, Jan. 1997.

[14] S. Bose, H. Grimes, and V. D. Agrawal. Delay fault simulation with bounded gate delay mode. In *Proc. Int'l Test Conf. (ITC)*, pp. 1–10, 2007.

[15] Y. Sato, S. Hamada, T. Maeda, et al. Invisible delay quality - SDQM model lights up what could not be seen. In *Proc. IEEE Int'l Test Conf. (ITC)*, paper 47.1, 2005.

[16] I. Pomeranz and 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*, 18(2):333–337, Feb. 2010.

[17] C. Han and A. D. Singh. Improving CMOS Open Defect Coverage Using Hazard Activated Tests. In *Proc. IEEE 32nd VLSI Test Symp. (VTS)*, pp. 1–6, 2014.

[18] A. Czutro, N. Houarche, P. Engelke, et al. A Simulator of Small-Delay Faults Caused by Resistive-Open Defects. In *Proc. 13th European Test Symp. (ETS)*, pp. 113–118, 2008.

[19] J. D. Owens, M. Houston, D. Luebke, et al. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

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

[21] D. Chatterjee, A. DeOrio, and V. Bertacco. Event-driven gate-level simulation with GP-GPUs. In *Proc. ACM/IEEE 46th Design Automation Conf. (DAC)*, pp. 557–562, 2009.

[22] K. Gulati and S. P. Khatri. Towards acceleration of fault simulation using Graphics Processing Units. In *Proc. ACM/IEEE 45th Design Automation Conf. (DAC)*, pp. 822–827, 2008.

[23] M. A. Kochte, M. Schaal, H. Wunderlich, et al. Efficient fault simulation on many-core processors. In *Proc. ACM/IEEE 47th Design Automation Conf. (DAC)*, pp. 380–385, 2010.

[24] M. Li and M. S. Hsiao. FSimGP^2: An Efficient Fault Simulator with GPGPU. In *Proc. IEEE 19th Asian Test Symp. (ATS)*, pp. 15–20, 2010.

[25] K. Gulati and S. P. Khatri. Fault Table Computation on GPUs. *Journal of Electronic Testing*, 26(2):195–209, Apr. 2010.

[26] H. Li, D. Xu, Y. Han, et al. nGFSIM : A GPU-based fault simulator for 1-to-n detection and its applications. In *Proc. IEEE Int'l Test Conf. (ITC)*, pp. 1–10, 2010.

[27] J. G. Tong, M. Boule, and Z. Zilic. Efficient Data Encoding for Improving Fault Simulation Performance on GPUs. In *Proc. Int'l Symp. on Electronic System Design (ISED)*, pp. 138–142, 2013.

[28] S. Holst, E. Schneider, and H. Wunderlich. Scan Test Power Simulation on GPGPUs. In *Proc. IEEE 21st Asian Test Symp. (ATS)*, pp. 155–160, 2012.

[29] High Performance Computing (HPC) and Supercomputing — NVIDIA Tesla — NVIDIA. http://www.nvidia.com/object/tesla-supercomputing-solutions.html, 2014.

[30] K. Gulati and S. P. Khatri. Accelerating statistical static timing analysis using graphics processing units. In *Proc. Asia and South Pacific Design Automation Conf. (ASP-DAC)*, pp. 260–265, 2009.

[31] V. S. Iyengar and D. T. Tang. On simulating faults in parallel. In *Proc. 18th Int'l Symp. on Fault-Tolerant Computing (FTCS)*, pp. 110–115, 1988.

[32] NanGate 45nm Open Cell Library. http://www.nangate.com, 2014.