High-Throughput Logic Timing Simulation on GPGPUs

STEFAN HOLST, Kyushu Institute of Technology
MICHAEL E. IMHOF and HANS-JOACHIM WUNDERLICH, University of Stuttgart

Many EDA tasks such as test set characterization or the precise estimation of power consumption, power droop and temperature development, require a very large number of time-aware gate-level logic simulations. Until now, such characterizations have been feasible only for rather small designs or with reduced precision due to the high computational demands.

The new simulation system presented here is able to accelerate such tasks by more than two orders of magnitude and provides for the first time fast and comprehensive timing simulations for industrial-sized designs. Hazards, pulse-filtering, and pin-to-pin delay are supported for the first time in a GPGPU accelerated simulator, and the system can easily be extended to even more realistic delay models and further applications.

A sophisticated mapping with efficient memory utilization and access patterns as well as minimal synchronizations and control flow divergence is able to use the full potential of GPGPU architectures. To provide such a mapping, we combine for the first time the versatility of event-based timing simulation and multi-dimensional parallelism used in GPU-based gate-level simulators. The result is a throughput-optimized timing simulation algorithm, which runs many simulation instances in parallel and at the same time fully exploits gate-parallelism within the circuit.

Categories and Subject Descriptors: B.6.3 [**Logic Design**]: Design Aids—*Simulation*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Parallel processors*

General Terms: Verification, Performance

Additional Key Words and Phrases: Gate-level simulation, general purpose computing on graphics processing unit (GP-GPU), hazards, parallel CAD, pin-to-pin delay, pulse-filtering, timing simulation

ACM Reference Format:
Stefan Holst, Michael E. Imhof, and Hans-Joachim Wunderlich. 2015. High-throughput logic timing simulation on GPGPUs. ACM Trans. Des. Autom. Electron. Syst. 20, 3, Article 37 (June 2015), 22 pages.
DOI: http://dx.doi.org/10.1145/2714564

1. INTRODUCTION

Many EDA tasks involve a massive amount of independent simulations to collect various statistics on the behavior of the circuit. For instance, an efficient way to determine the data dependent power consumption of large synchronous sequential circuits is to perform a complete time simulation and compute the weighted switching activity (WSA) [Najm 1994; Macii et al. 1997; Wang and Roy 1998] based on all the events observed on internal signals. Tasks like fault simulation, variation analysis, or aging analysis have very similar requirements and can be greatly accelerated by a high-throughput simulator. Other design verification tasks may require simulation of a

This work has been supported in part by the DFG under contract WU 245/16-1.
A preliminary version of this work has been published at the Asian Test Symposium (ATS) 2012.
Authors' addresses: S. Holst, Department of Creative Informatics, Kyushu Institute of Technology 680-4 Kawazu, Iizuka 820-8502, Japan; email: holst@ci.kyutech.ac.jp; M. E. Imhof and H.-J. Wunderlich, University of Stuttgart, Pfaffenwaldring 47, 70569 Stuttgart, Germany; email: {imhof, wu}@iti.uni-stuttgart.de.

DOI: http://dx.doi.org/10.1145/2714564

single state for a large number of consecutive clock cycles. Such tasks clearly require low-latency simulators and are not the target applications of this work.

Tremendous speedups are gained by using data-parallel architectures like *general purpose computing on graphics processing units* (GPGPU) [Owens et al. 2008; Nvidia 2013] for problems in electronic design automation [Catanzaro et al. 2008; Croix and Khatri 2009]. Especially, simulation tasks are of particular interest because of their high computational demand and GPGPU implementations were proposed for gate-level logic simulation [Chatterjee et al. 2011], fault simulation [Kochte et al. 2010; Li and Hsiao 2010; Li et al. 2010; Gulati and Khatri 2010], or Monte-Carlo simulations for statistical static timing analysis [Gulati and Khatri 2009]. All these GPGPU gate level simulators either do not consider timing at all (zero delay model) or only calculate the latest transition at each gate [Gulati and Khatri 2009]. This is not sufficient for the tasks mentioned earlier, as hazards may account for up to 70% of dynamic power [Shen et al. 1992] or may affect the fault coverage of a test set [Reddy et al. 1984; Lin and Reddy 1987].

The canonical way to analyze a circuit with all its hazards is to employ event-based simulation. To increase simulation performance, event-based simulators exploit structural parallelism by distributing circuit partitions to multiple processors within a machine or even among multiple machines within a network [Chandy and Misra 1981; Soule and Gupta 1989]. Frequent synchronizations are necessary between the partitions, and these are implemented via shared memory or message passing [Mueller-Thuns et al. 1993; Bailey et al. 1994]. To partly circumvent the latency introduced by the synchronizations, lookahead-rollback mechanisms [Jefferson 1985; Meraji and Tropper 2012] are used, which speculatively simulate individual partitions ahead of time and then rollback if unexpected events are arriving. The message passing parallelization style is very ineffective on GPGPUs and their *single instruction, multiple data* (SIMD) paradigm for parallelism. Moreover, to the best of our knowledge, all parallelization strategies and optimizations proposed for event-based simulation aim to increase the simulation performance for individual simulation instances (i.e., optimization for low latency). Our goal is to optimize simulation throughput. Of course, this could be achieved by running many independent event-based simulators at the same time on a large computing cluster, but our experiments will show, that many hundreds of computing nodes would be necessary to reach the same performance a single GPGPU can deliver.

We present a novel gate-level time simulator for GPGPUs, which takes a combinational time-annotated circuit and calculates for each set of input transitions all events on the internal signals and outputs. The simulator combines for the first time the versatility of event-based timing simulation and multidimensional parallelism for maximum speedup. Specialized versions of this simulator have already been used successfully for variation analysis [Czutro et al. 2012; Sauer et al. 2014] and power simulation [Holst et al. 2012]. Multidimensional parallelism has been used very successfully to speed up logic simulation. This new simulator applies this parallelism concept for the first time to logic timing simulation on GPGPUs. It does so by packaging complete transition histories into *waveforms* for very efficient data-parallel processing on GPGPUs. Similar data structures have been used to speed up serial timing simulation [Min et al. 1996; Li et al. 2000] or small-delay fault simulation [Czutro et al. 2008]. This article presents a very efficient data-parallel waveform processing algorithm along with efficient methods to represent and organize large sets of waveforms in limited memory space. By using waveforms as the basic representation of signal values in time, the simulator avoids dealing with single events on a signal individually. The output waveform of a gate is calculated from its input waveforms in a single processing step without any further data dependencies or synchronization overhead. This allows the computation

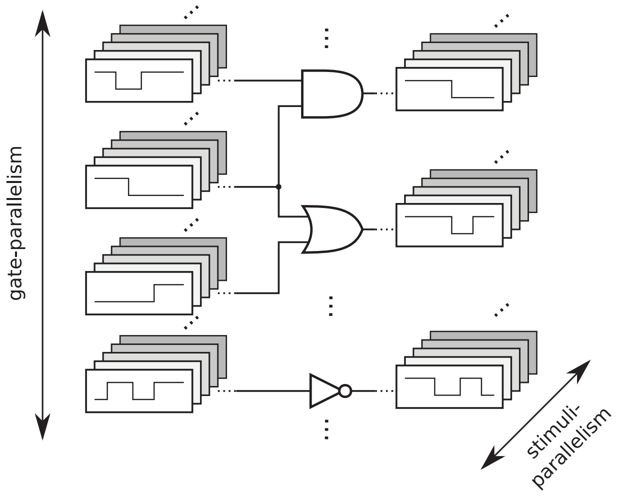

Fig. 1. The waveform principle enabling two dimensions of parallelism.

of all events on all signals with only one single pass over the circuit and enables the exploitation of two dimensions of parallelism, *stimuli-parallelism*, and *gate-parallelism* (see Figure 1). Even more dimensions can be added depending on the applications (e.g., faults for fault simulation, Monte-Carlo samples for variation analysis).

Stimuli-parallelism is exploited by processing s independent input stimuli at the same time. In this simulator, an input stimulus is a waveform at every primary and pseudoprimary input of the circuit. The waveforms at the (pseudo-)primary inputs usually contain only single transitions to model the switch between two input patterns or a state transition in the circuit. In combinational fault simulation or many other problems of structural test, all input stimuli are already known in advance and can be given directly to the timing simulator for parallel processing.

Gate-parallelism is the parallel evaluation of g independent gates within a single simulation instance [Chatterjee et al. 2011]. In its simplest form, the gates within the circuit are ordered topologically, and the resulting levels define the sets of pairwise independent gates. Figure 1 shows one level in such a topological ordering. The gates on a level may be of various types and exhibit different timing behavior. This information is stored as parameters for each individual instance and the simulator will process all gates in a data-parallel fashion regardless of their differences. Therefore, the amount of gates being processed in parallel only depends on the circuit topology and is not constrained by the number of cell types and sizes in the technology library.

Only the combination of these two dimensions generates enough $(s \cdot g)$ threads to fully occupy typical GPGPUs for a wide range of design sizes. Small circuits with low memory requirement per instance allow for more input stimuli s to be processed in parallel. For larger circuits with higher memory requirement, the number of simulations s is reduced, but each level contains more gates and gate-parallelism dominates. Overall, a large number of threads can be created over a wide range of circuit sizes, and the number of threads is only bound by the memory available for waveforms.

The next section briefly describes the execution model of typical GPGPU architectures in order to provide the necessary background for understanding the design decisions made for the presented algorithm. Section 3 describes the waveform data structure, its representation in memory, and the supported delay model along with its challenges for an efficient GPGPU implementation. The GPGPU time simulation core, which processes a set of independent gates with many stimuli in a data-parallel fashion, is described in Section 4. For the sake of maximum performance, the GPGPU simulation core itself provides only very rudimentary controls. Design preprocessing, memory management as well as the proper control for simulating complete combinational networks is done by the system CPU. This part is described in Section 5 and completes the time simulation system for combinational circuits. The series of experiments reported in Section 6 shows the performance benefit of the new simulator.

2. GPGPU EXECUTION MODEL

GPGPUs are throughput oriented architectures. Instead of reducing latencies with techniques like out-of-order execution, speculative computing and complex control hardware, GPGPU architectures use a massive amount of lightweight threads to hide latencies caused by data dependencies and memory accesses. Thousands of these threads are necessary to fully occupy such an architecture. Each thread executes the same code, but operates on different data. Threads are scheduled in batches causing multiple units to execute the same code in a lock-step fashion. This is most efficient if many threads follow exactly the same execution path. If the control flows of two threads diverge as a result of a data dependent conditional branch, however, some execution units may become idle and the performance degrades until the control flows of the threads merge again. Only threads of the same batch can share data during execution over fast but small local memories. Information exchange between threads from different batches is only possible with very expensive global synchronizations, which should be avoided as much as possible.

The memory hierarchy of data-parallel architectures is kept very flat and the amount of cache available per thread is very limited. Besides the high latencies for memory reads partly hidden by the thread scheduler, this also exposes physical properties of the connection between the GPGPU and the on-board memory. Every memory access results in a transaction on a several bytes wide bus between the on-board memory and the GPGPU. To use all bytes in a transaction, threads of the same batch must access data in the same region at the same time.

The key challenges in designing a simulation system on GPGPUs are the following.

—*Control flow divergences* lead to idle computing resources and should be minimized as much as possible by avoiding branches or at least keep the computations within branches as simple as possible.
—*Global synchronizations* involve very large overheads and must be reduced to a minimum.
—*Memory efficiency*. The less memory is needed per simulation instance, the more instances can be handled in parallel. To use all available computing power on a GPGPU, thousands of instances must be available at any given time.
—*Memory access patterns* should be as regular as possible to fully utilize the available memory bandwidth. Compared to the available computing power, memory bandwidth of GPGPUs is already quite limited and a proper memory organization must be used to reduce idle cycles to a minimum.

The time simulation system presented here addresses all these challenges.

3. MODELING OF WAVEFORMS AND DELAYS

As parallelism is bound by the on-board memory size M, the waveform representation must be very memory efficient. At the same time, the representation should allow for fast processing with a simple control flow for efficient data-parallel execution. The waveform representation and evaluation algorithm presented here is tuned towards 2-valued simulations for maximum efficiency. However, the principle evaluation

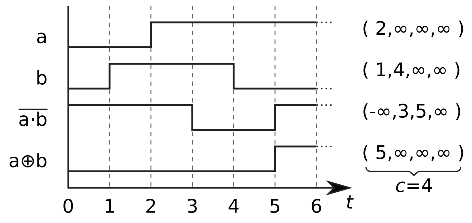

Fig. 2. Waveforms and their representations. Rising, falling and inertial delays are 1t.

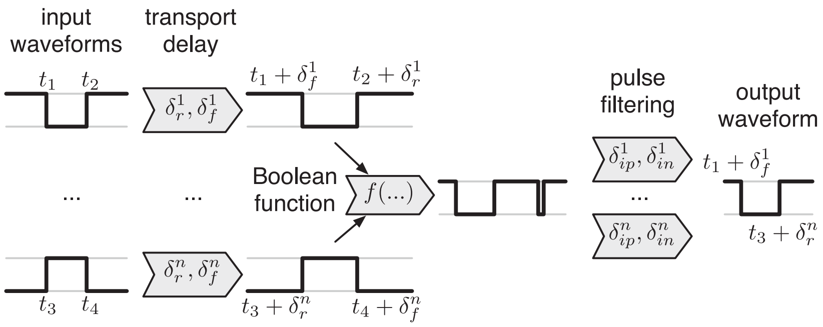

Fig. 3. Elemental delay processing.

approach is also applicable to multivalued simulations and waveform representations as in Czutro et al. [2008].

Let v_t be the signal value at time t. A waveform is a description of signal values v_t for all $t \geq 0$. In 2-valued simulation, transitions are always alternating between rising and falling on a single signal. Therefore, any signal value $v_{t'}$ is determined by a known value v_t and the number of transitions between t and t'. In the representation used here, the initial signal value is always zero ($v_{-\infty} = 0$) by default. With this signal value given, only the time points t_i need to be stored:

$$w = (t_0, t_1, t_2, \ldots, t_{c-1}) \qquad \text{with } t_0 \leq t_1 \leq \cdots \leq t_{c-1}.$$

Figure 2 shows some waveforms and their representations. The time of the first transition (which is always rising) is t_0, the time of the second transition is t_1, and so on. In other words, the initial value of a transition t_i is just the parity $\pi(i) = i \bmod 2$ of its index i. To encode an initial value of 1 on a signal, t_0 is set to a large negative value, denoted by the symbol $-\infty$. A waveform is terminated by a large value (symbol ∞) after the last valid transition time. The number of transitions that fit into a waveform is bound by a waveform *capacity* c. If the initial value of a signal is 0, at most $c - 1$ transitions can be stored, if the initial value of a signal is 1, at most $c - 2$ transitions can be stored.

With all waveforms available at the inputs of a cell, the output waveform of the respective element is calculated based on a two-valued pin-to-pin delay model. Figure 3 shows the elemental delay processing necessary for supporting this delay model for cells and wire delays. The transitions in the n input waveforms are first shifted by the delay values δ_r^i (rising), δ_f^i (falling) assigned to each input i. With the newly generated waveforms as parameters, the Boolean function is evaluated at each point in time generating an intermediate output waveform.

Generally, short pulses or hazards can be filtered at two places, the inputs or the output of a cell. Pulse filtering at the inputs provides more flexibility such as distinct

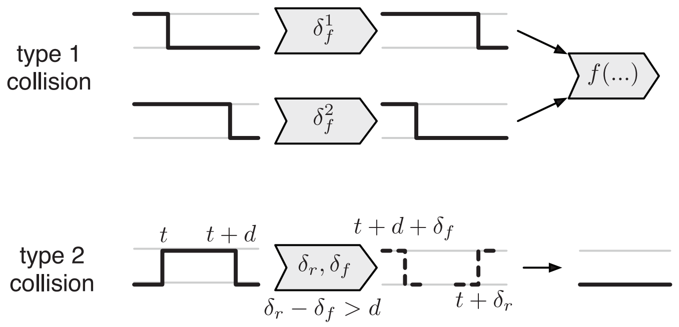

Fig. 4. The two possible collision types.

thresholds for each input, pulse filtering at the output requires less computing power because it has to be performed only once per cell. This delay model combines the advantages by implementing output pulse filtering for high performance and allowing at the same time distinct thresholds for each input. Positive pulses of width less than δ_{ip}^i and negative pulses less than δ_{in}^i (with i being the input causing the later transition of the pulse) are filtered from the intermediate waveform to obtain the final output waveform. This model matches the behavior of pure input pulse filtering very well in all important aspects. If the exact behavior of input pulse filtering must be implemented, buffers can be added with the appropriate threshold values at each input of the cell.

In addition to pulse filtering, there are more situations, which require the removal of transitions from the output waveform and further complicate the evaluation process. These situations are known as collisions and classified into two types (see Figure 4). In a type-1 collision, the relative order of transitions changes due to different delays at each input. The upper part of Figure 4 shows input waveforms each containing a single falling transition at inputs 1 and 2 of a cell. The falling transition at input 2 arrives later than the falling transition at input 1. If input 1 has a larger delay than input 2 ($\delta_f^1 > \delta_f^2$), the delayed transitions may have a different order in time. In this example, the transition at input 2 must be processed before the transition at input 1 to obtain the correct intermediate output waveform.

A type-2 collision arises, when the width of an incoming pulse is less than the difference between rising and falling delays at an input. The second example in Figure 4 shows a positive pulse of width d arriving at time t. The rising transition gets moved to time $t + \delta_r$, and the falling transition gets moved to time $t + d + \delta_f$ by the respective delays at this input. If the difference between the rising and falling delays is larger than the pulse width ($\delta_r - \delta_f > d$), the order of these pulses change ($t + \delta_r > t + d + \delta_f$) rendering both transitions invalid. From an electrical viewpoint, the input pulse is too short to load the gate capacities and cause a switch of the gate. This situation has to be detected as well to compute the correct result, which is constant 0 in this example.

A GPGPU implementation demands for a rather simple control flow in order to reduce diverging threads to a minimum and simple memory access patterns for using the available bandwidth efficiently, and the waveform data structure is crucial for meeting these demands. Instead of processing each input transition sequentially, having access to all transitions at all the inputs during the evaluation is very beneficial for resolving the mentioned collisions and implementing efficient pulse filtering without any additional synchronization or rollback overhead. Implementing this delay model and processing is still very challenging, because of the numerous situations that require data-dependent branches. Moreover, different waveforms may contain different numbers of transitions and therefore require different amount of storage. Keeping the

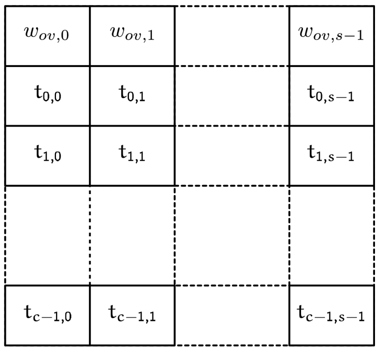

Fig. 5. Memory organization of a waveset with capacity c and s samples.

memory access patterns regular even in this situation is also a key feature of the method described further on.

4. GPGPU TIME SIMULATION CORE

The principle operation of a single thread on the GPGPU is to compute the output waveform at a single gate with given input waveforms. A single invocation of the simulation core will generate a two-dimensional array of threads which compute $s \cdot g$ waveforms for g mutually independent gates and s different input stimuli (Figure 1). Each thread is spawned with different parameters for the gate and locations of its input and output waveforms in memory for fully data-parallel operation. All s waveforms for each signal form a *waveset* data structure described in the next section. After this, the core evaluation algorithm is presented that efficiently operates on these wavesets.

4.1. Wavesets and Overflows

Multiple independent simulations with different input stimuli are performed at the same time (stimuli-parallelism). Therefore, each internal signal needs storage for a set of waveforms, called a *waveset*. To make the best possible use of memory transactions for optimal on-board memory throughput, the s individual waveforms in a waveset are stored in an interleaved manner. Each thread will read its waveform in the order of increasing transition times $(t_0, t_1, \dots)$. Therefore, all t_0 of the waveforms are stored in direct succession in the waveset and the same holds for all the other transitions t_i $(0 \leq i < c)$. The resulting memory layout of a waveset is shown in Figure 5.

All waveforms in a waveset have the same capacity c. The actual number of transitions for a signal is not known in advance and may exceed the waveform capacity for certain signals and input stimuli. However, using variable-sized or dynamically growing data structures for waveforms would again lead to irregular memory access patterns and a severe performance impact on the data-parallel code. Instead, the simulation core will simply truncate the output waveform and flag the incomplete waveform by increasing an overflow counter w_{ov} stored along the waveform itself. This overflow indication enables the overall simulation system to increase the appropriate waveform capacities and reallocate the wavesets while fully data-parallel operation is maintained in the inner simulation loops.

4.2. Waveform Evaluation

This algorithm computes the output for a gate with n inputs. It is designed to operate in a data-parallel fashion on wavesets where each individual thread computes one of the s waveforms in the output waveset. For the sake of easier discussion, first, the operation of a single thread computing a single output waveform z from n input waveforms is presented.

Given are the logic function of the gate $f(v^1, \dots, v^n)$, the delays for a rising transition at each input $\delta_r^1, \dots, \delta_r^n$, the delays for falling transitions $\delta_f^1, \dots, \delta_f^n$, the positive and negative inertial delays $\delta_{ip}^1, \dots, \delta_{ip}^n, \delta_{in}^1, \dots, \delta_{in}^n$, and the waveform at each input $w^1, \dots, w^n$. The input waveforms are processed in the order of their representation with a merge-sort approach. A transition is added to the output waveform, if (1) the logic value of f changes with the currently processed input transition, and (2) the transition originating from an input k does not generate a positive pulse smaller than δ_{ip}^k or a negative pulse smaller than δ_{in}^k. The logic values for the evaluation of f are given by the parities of the indices of the currently processed transitions. As already mentioned previously, the signal value before a transition at t_i in a waveform $w = (t_0, t_1, \dots)$ is just $v_{t_i - \varepsilon} = \pi(i)$ for a sufficiently small $\varepsilon > 0$. If f changes its value indeed, two cases are possible. In the case, that the new transition does not generate a short pulse, it is saved in z and the index for the output waveform is advanced. In the case, that the new transition is too close to the previously saved transition t', it is discarded and t' is removed from z by decreasing the output waveform index by one. During this processing it has to be ensured, that collisions of type 1 and type 2 are handled appropriately.

Algorithm 1 implements the computation of an output waveform directly from the input waveforms with very low intermediate memory requirements and simple control flow. The lines starting at 1 initialize the input waveform indices $i_1, \dots, i_n$ and the output waveform index j. If at least one of the input waveforms $w^1, \dots, w^n$ is incomplete due to an overflow happened previously, the output waveform z might be incomplete as well. Therefore, the output overflow indicator z_{ov} is initialized to the sum of the overflow field values of all inputs. If the function f evaluates to one for all zero inputs, the output waveform is initialized with an initial value of one. The variables $t^1, \dots, t^n$ are initialized to the first transition in each input waveform plus the corresponding delays. As in the waveform representation, the first entry is always a rising transition, δ_r^k is added for each input k.

The part beginning at 2 contains a while-loop, that processes transitions until the terminal symbol ∞ is reached for all the input waveforms or the output waveform reached its capacity. The input transitions are processed in the order of their delayed point in time and t stores this time for the current loop iteration. This resolves all type-1 collisions as the order of the transitions is evaluated after the addition of the input delays. However, type-2 collisions are not yet resolved at this point and cause t to decrease at a certain point as the min-operator first selects the later of the two invalid transitions and then the earlier one. It is important to observe, that both invalid transitions will always be processed in direct succession. If the min-operator selects an invalid transition at time t^k, by definition, there is no earlier transition currently under consideration. The loop-iteration will consume the transition at t^k and put the next invalid transition at $t'^k < t^k$ under consideration. For the following iteration, the min-operator will always select t'^k. This observation allows the filtering of the two collided transitions along with the pulse-filtering described here without any additional branches.

The body of the while-loop first selects an earliest, unprocessed transition (there may be more than one) and consumes it by advancing the appropriate index i_k. The transition time t^k is updated to the value, that will be selected in a later loop iteration. If the logic function of the cell produces a value different from the current one $\pi(j)$, a transition may be generated in z at time t. The if-statement at 3 implements three important tasks. It handles the pulse-filtering condition, it prevents the storage of redundant $-\infty$ symbols in the output waveform, and it removes invalid transitions

ALGORITHM 1: Waveform evaluation in pseudo-code.

```
   Data: Function of the cell: $f(v^1, \dots, v^n)$, arrays containing rising and falling delays for
         each input: $\delta^1 = [\delta_r^1, \delta_f^1], \dots, \delta^n = [\delta_r^n, \delta_f^n]$, arrays containing inertial delays:
         $\delta_i^1 = [\delta_{in}^1, \delta_{ip}^1], \dots, \delta_i^n = [\delta_{in}^n, \delta_{ip}^n]$, waveforms at the cell's inputs: $w^1, \dots, w^n$, overflow
         indicators: $w_{ov}^1, \dots, w_{ov}^n$
   Result: Waveform at the cell's output: $z$, overflow indicator: $z_{ov}$
1  $i_1, \dots, i_n \leftarrow 0$
   $j \leftarrow 0$
   $z_{ov} \leftarrow \sum_{x=1}^{n} w_{ov}^x$
   if $f(0, \dots, 0) = 1$ then
       $z_0 \leftarrow -\infty$
       $j \leftarrow 1$
   end
   $t^k \leftarrow w_0^k + \delta_r^k \;\; \forall\, 1 \le k \le n$
2  $t \leftarrow \min\{t^1, \dots, t^n\}$
   while $t < \infty$ and $j < |z|$ do
       choose $k$ with $t^k = t$
       $i_k \leftarrow i_k + 1$
       $t^k \leftarrow w_{i_k}^k + \delta^k[\pi(i_k)]$
       if $f(\pi(i_1), \dots, \pi(i_n)) \ne \pi(j)$ then
3          if $j = 0$ or $t^k < t$ or $t - z_{j-1} > \delta_i^k[\pi(j)]$ then
               $z_j \leftarrow t$
               $j \leftarrow j + 1$
           else
               $j \leftarrow j - 1$
           end
       end
       $t \leftarrow \min\{t^1, \dots, t^n\}$
   end
4  if $t < \infty$ then
       $z_{ov} \leftarrow z_{ov} + 1$
   end
   $z_j \leftarrow \infty$
```

from type-2 collisions. If the output waveform is empty ($j = 0$), the transition is always stored. If the next transition of input k is earlier than the current one (t), there is a type-2 collision at input k and the current transition is invalid. However, this transition is also always stored in the output waveform, which seems unreasonable at the first glance. But the earlier one of the collided transitions (currently saved in t^k) will be processed in the next loop iteration and both transitions will then be filtered because $t - z_{j-1}$ will be negative. Moreover, as the current transition is invalid, it must not cause a filtering of a transition from a different input. If there are transitions in the output waveform and the current transition is not the later one involved in a type-2 collision, the difference between the current transition t and the last one in z (z_{j-1}) must be larger than the appropriate value in δ_i for the current transition to be valid. When the output waveform contains a $-\infty$ symbol for an initial value of 1 ($z_0 = -\infty$) and a $-\infty$ symbol from an input changes the value of the gate, both symbols are discarded as well. The algorithm exploits here the limited precision of floating point values in order to ensure this filtering for any reasonable delay values δ and δ_i. The symbol $-\infty$ is encoded with such a large negative value, that any addition with a relatively small positive delay δ results again in the same value $-\infty$. Consequently, $t - z_{j-1}$ will always be 0 in the situation described previously and both symbols are discarded even if pulse-filtering is not used in this evaluation ($\delta_i = 0$). New transitions are stored by updating z and increasing j. If a transition is discarded, the previously stored transition is also removed from z by decreasing j. The while-iteration ends with the selection of the next earliest transition time from the input waveforms. Again, if the current iteration involved the later one of two collided transitions ($t^k < t$ is true), the time t^k will be selected next and the appropriate transitions are filtered in the next iteration.

The if-statement at 4 after the while-loop checks for the overflow condition and increases the overflow indicator z_{ov} if necessary. At the end, the terminal symbol ∞ is added to z.

This algorithm is well suited for data-parallel execution in a lock-step fashion. Control flow divergences (if-statements) are reduced to the absolutely necessary and all expensive operations (summing, min-operator and gate function evaluation) are unconditioned. The conditioned operations include only quite inexpensive floating-point comparisons and index manipulations. In a batch of m threads, m loops are executed in parallel and the batch is active until the while-loop in every thread is finished. Within a batch, each thread will evaluate an individual gate instance (a gate type with its unique timing) but with different stimuli. The other gate instances on the current level are evaluated by separate batches. All batches are again pairwise independent and are in turn executed in parallel.

Each thread y of an m-sized batch first reads the overflow fields $w^x_{ov,y}$ $(0 \leq y < m)$ of the input waveforms. Due to the memory organization of wavesets (Figure 5), all threads access similar memory locations concurrently filling the generated transactions perfectly. The same holds for the access to $t^x_{0,y}$, but during the loop iterations, the individual threads may be forced to consume input transitions on different inputs and the indices i_k diverge in their values. Therefore, each thread may consume the transitions in a waveset at a different pace. Transactions are filled as much as possible, but not perfectly anymore. However, the cache available in recent GPGPU architectures compensates nicely by holding the data of the last few transactions for the threads with slower pace to read.

5. TIME SIMULATION SYSTEM

The time simulation system invokes the GPGPU simulation core repeatedly to implement high-throughput timing simulation for combinational logic circuits and ensures correct results in the cases where the simulation core reports overflows. Figure 6 shows the overall time simulation system with the control flow depicted in the vertical direction and data flowing from left to right. The white tasks in this figure are performed entirely on the latency-optimized CPU and involve mostly design preprocessing and memory management. The shaded tasks are are performed with the help of the data-parallel simulation core described in the previous section and form the inner simulation loops.

For maximum possible performance, the system has to ensure that:

—Each call to the simulation core contains the maximum amount of parallelism for optimal GPGPU workload and minimum number of calls.
—The communication between GPGPU and CPU is minimized.

To maximize parallelism per invocation of the simulation core, the system needs to determine large sets of independent gates and at the same time minimize the memory footprint per stimulus. Reducing this memory footprint is equivalent to minimizing the

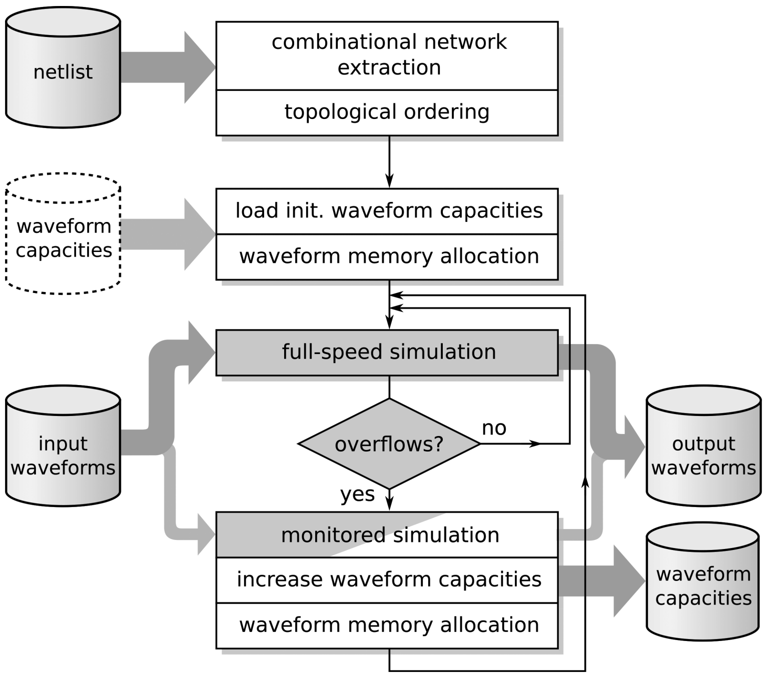

Fig. 6. Overall combinational simulation flow.

number of wavesets necessary for storing all intermediate results during a single pass over the circuit and finding their optimal capacities.

5.1. Preprocessing and Topological Ordering

In a given gate-level design, first, all the state elements like flip-flops and latches are replaced by pairs of pseudo-primary inputs and outputs. To reduce the number of synchronizations to the minimum, the resulting cycle-free, purely combinational network is ordered topologically using an as-soon-as-possible (ASAP) scheduling. The first level contains all primary and pseudo-primary inputs, and the gates on the following levels only depend on gates and inputs from previous levels. All gates in a single level are pairwise independent and will be evaluated in parallel during simulation. As the number of necessary synchronizations equals the length of the longest structural path in the circuit, the ASAP-schedule guarantees the maximum amount of gate-parallelism for the given design.

This simple preprocessing is done in linear time and results in fixed design data transferred to the GPGPU just once. The design data contains the types and delay parameters for each gate and the gates in the circuit are grouped by their topological level. It is shared among all simulation instances (each processing their own input stimuli) and therefore takes only very little memory compared to the wavesets. The following invocations of the simulation core just takes the level number as parameter and processing will start on the GPGPU using the available design data and without any further data transfer.

5.2. Waveset Memory Management

The amount of data-parallelism is bound by the size of the on-board memory M. Let us consider for now, that all wavesets have the same capacity c, and let r be the number of

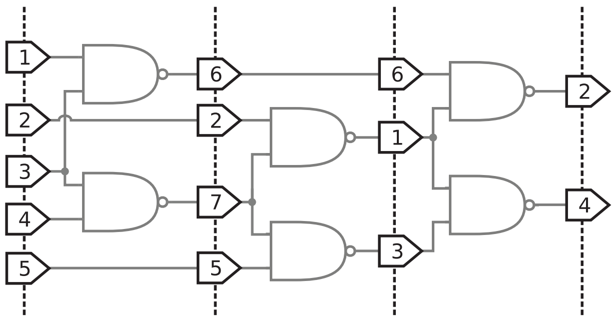

Fig. 7. A possible waveset allocation for circuit c17 with ASAP-scheduling. Locations 1 and 3 are reused in the third barrier and locations 2 and 4 are reused in the last one.

storage spaces for intermediate signal wavesets necessary to complete one simulation pass over the circuit. The number of independent simulations s that can be performed in parallel is

$$s = \frac{M}{c \cdot r}.$$

The waveset capacities are already determined by the number of transitions on the signals. Therefore, r has to be reduced to a minimum in order to maximize stimuli-parallelism. Figure 7 shows an example of an allocation with $r = 7$. Wavesets that pass a barrier indicated by a dashed line need to be stored in the on-board memory, and the numbers denote their storage locations. Simple reference counting can be used to find a minimum cost allocation with a single pass over the circuit, reusing each location as soon as its intermediate result is no longer required.

However, wavesets may have different capacities and using only storage locations of one common chunk size would be very inefficient. Moreover, capacities may change to avoid overflows detected during simulation and efficient reallocation should also be possible. One way to tackle this issue is to allocate an additional chunk of memory (page) whenever an overflows occurs [Zhu et al. 2011]. Because the memory for a single waveset may get fragmented over several pages, additional bookkeeping and indirect addressing is necessary. This overhead slows down execution even in cases when no overflow takes place during simulations. To ensure maximum performance, the following approach ensures that the memory of a single waveset is always a continuous chunk of memory and avoids additional bookkeeping and indirect addressing.

As described in the next section, the simulation system will double the waveset capacities if an overflow has occurred. The sizes of memory chunks are therefore proportional to $2^i \cdot c'$ with $i \geq 0$ and c' the base capacity before any simulation run. One very efficient way to manage memory chunks of these sizes is a buddy system [Knowlton 1965; Knuth 1969]. The allocated memory chunks are organized in a binary tree, and each node in the binary tree corresponds to a specific location in the memory. The lowest child nodes represent memory chunks of size c', their parent nodes represent chunks of size $2c'$ and so on up to the root node, which represents all available memory. Figure 8 shows an example of such a tree.

The nodes in the tree may have either exactly two children or no children at all. Nodes without children are called *leafs*. A leaf node may have a label indicating the piece of data stored in the associated memory chunk. Leaf nodes without label indicate free memory and can get assigned a piece of data by assigning the label, or get split into smaller memory chunks by adding two children to this node.

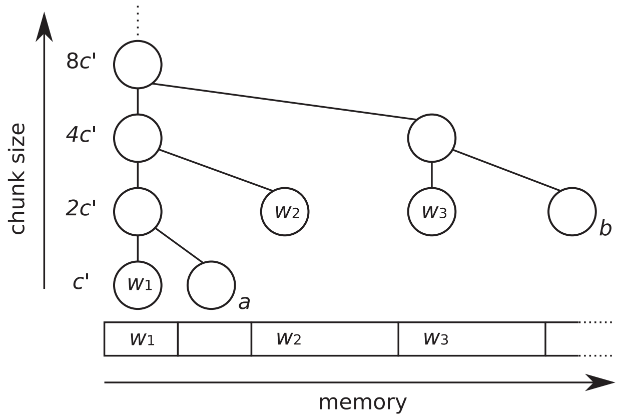

Fig. 8. Example of a binary tree for waveset memory management. The waveset w_1 was allocated with size c', w_2 and w_3 are both of size $2c'$.

A new memory chunk is allocated by looking up the next free leaf node on the level corresponding to the desired chunk size. If no such node exists, new child nodes are generated for a free leaf node of a bigger chunk size. For instance, a new waveset w_4 of size c' would be associated with node a in Figure 8. For the next waveset w_5 of the same size, two children would be generated for leaf node b and the waveset would be associated with the first child. Deallocation is just the reverse operation. If w_1 is not needed anymore, for instance, its storage space is merged with its buddy (sibling node a) by just removing these two nodes from the tree. With the proper management of available leaf nodes in free-lists [Knowlton 1965], these operations can be implemented in near-constant time (logarithmic time in the worst-case for the split and merge operations) to allocate all wavesets for the circuit in near-linear time.

All wavesets are allocated level-by-level in a single pass over the circuit. For each level, the binary tree is stored in order to serve as a basis for reallocating wavesets after a change in their capacities. This allocation method determines the amount of memory m needed to perform simulation with a single set of input waveforms (i.e., the memory requirement for $s = 1$). The amount of stimuli-parallelism is now determined by choosing the largest possible s with $M \geq s \cdot m$. While the memory requirement m is different from design to design, the simulation system always uses all the available memory resources to provide as much throughput as possible.

5.3. The Main Simulation Loops

Typically, more than s simulations have to be performed and multiple passes over the circuit are required. One iteration of a simulation loop will first transfer s new waveforms for each input to the GPGPU, propagate the waveforms with a single pass over the circuit by invoking the simulation core for each topological level, and then collect the results.

The simulation system uses two simulation loops that are different in their overflow detection and location granularity. In *full-speed simulation* (the inner loop in Figure 6), overflows are just recorded in each waveform by the simulation core and propagated together with the waveform information towards the outputs at negligible additional cost. At the end of a simulation pass over the complete circuit, reduction is performed over all output waveforms resulting in the sum of overflows happened during simulation. Zero overflows indicates a completely valid simulation run, consistent output waveforms and statistics. A positive number of overflows indicates, that some transitions were skipped somewhere in the circuit. If this situation is detected after processing a set of inputs, the simulator will enter a *calibration loop* (the outer loop in Figure 6). This calibration loop processes the same set of inputs again level by level in a monitored simulation. After the evaluation of all gates in a level, all newly generated wavesets are checked for overflows individually. If an overflow is detected in a waveset on level l, its capacity is doubled, the wavesets of this level are newly allocated on the basis of the buddy tree of level $l-1$, and then the same level is simulated again. This procedure is repeated until all overflows on level l are resolved. The current calibration run then continues with the remaining levels $> l$, which are reallocated as well resolving all overflows resulting from the additional transitions propagated through the circuit.

After a calibration loop, some waveset capacities have been adjusted to guarantee correct and complete results at all internal signals and outputs. The next set of inputs is again simulated with the full-speed simulation loop, which avoids all fine-grained overflow monitoring and memory reorganizations to reach maximum simulation performance. In typical designs, only very few signals show large numbers of hazards. After the first few calibration loops, overflows become relatively rare and the simulation system is able to stay in the high performance full-speed simulation loop for most of the passes. This is even true in variation analysis where gate timing is slightly altered between simulations [Czutro et al. 2012; Sauer et al. 2014], because the efficient collision detection and glitch filtering still limits the number of hazards on a signal. The waveform capacities are stored to disk and reloaded in subsequent simulations of the same design (Step three in Figure 6) to initialize the capacities and avoid most calibrations altogether.

6. EXPERIMENTAL RESULTS

The simulation algorithm has been implemented on CUDA™-enabled hardware from NVIDIA® [Nvidia 2013]. This hardware provides high performance for single precision floating point operations, therefore 32-bit floating point numbers are used for the transition times in the waveforms. The host system for the simulation experiments contains Intel® Xeon® processors with 2.8 GHz and 256 GB RAM. The CUDA™-device is a Kepler GPU clocked with 980 MHz and 6 GB of on-board memory.

6.1. Benchmark Circuits and Delay Annotation

The experiments were performed on the largest ITC'99 benchmark circuits and industrial designs provided by NXP. All benchmarks were mapped to the NanGate Open Cell Library [OCL 2011] using primitive gates with at most two inputs by a commercial synthesis tool. This tool provides also timing estimates for all gates and wires in the Standard Delay Format (SDF). The delay model presented in Section 3 is sufficient for the most important DELAY constructs defined in SDF. More specifically, ABSOLUTE and INCREMENT delays are supported with IOPATH, PORT, INTERCONNECT, and DEVICE entries. The optional inertial delay specifications PATHPULSE and PATHPULSEPERCENT are supported as well. Conditional delays (COND and CONDELSE) are not supported and since only two-valued logic is used, RETAIN times are ignored as well. Each primitive logic gate corresponds to one instance of the elemental model shown in Figure 3 with the δ values set to the appropriate numbers from IOPATH, DEVICE, PATHPULSE and PATHPULSEPERCENT entries. PORT and INTERCONNECT delays are mapped to buffer elements inserted between the gates. These buffer elements have one input and the Boolean function is the identity. The four delays of the buffer δ_r^1, δ_f^1, δ_{ip}^1, δ_{in}^1 are set to the appropriate values to model the delay of the wire or port it is located on.

Figure 9 shows a small, delay annotated circuit (a) and its canonical mapping to delay processing elements (b). Table I reports the number of gates in column g and the

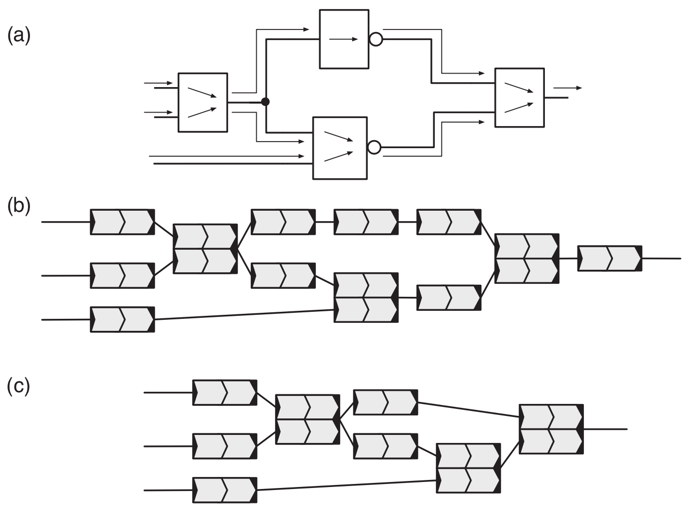

Fig. 9. (a) A combinational circuit with delay annotations for interconnects and IO-paths symbolized by arrows. (b) A canonical mapping to delay elements. (c) An optimized timing model obtained by merging delay elements.

number of delay elements resulting from this canonical mapping in column d_{can} for all considered benchmark circuits. The data shows, that the number of delay elements is almost three times as large as the number of gates in the circuit. This is no surprise and also state-of-the-art timing simulators treat wires as separate delay entities internally in order to support wire delays.

The presented delay model allows for a simple optimization to reduce the number of delay elements and improve simulation performance. Multiple single-input delay elements on a signal can be merged into a single delay element with equivalent behavior by combining the delay values appropriately. Also, a multi-input delay element can be merged with a single-input delay element at its output given that there are no branches in between. Figure 9(c) shows the optimized timing model for the example circuit after merging four delay elements. Buffers for PORT and INTERCONNECT delays remain only at circuit inputs and fanout branches. Column d_{opt} in Table I shows that the number of delay elements remaining after this simple optimization is significantly reduced compared to the canonical mapping. For all the following simulation experiments, this optimized mapping is used.

6.2. Case Study: Scan Test Power Simulation

The sample application used here to illustrate the performance benefits of the presented simulation method is scan test power estimation [Holst et al. 2012]. The simulation-based power estimation is fault-model independent and can be performed for any pattern set. Here scan-based testing using the stuck-at fault model is considered. During scan test, test patterns are shifted into the circuit via scan chains causing a lot of switching activity within the circuit under test. For the sake of accurate power estimation, all internal signal transitions caused by applying a complete test pattern set are going to be simulated. The trace data generated during simulation can then be used to calculate the weighted switching activity.

The test pattern sets used for each circuit are standard stuck-at test sets generated by a commercial test generation tool. The number of patterns in each test set is reported in column p of Table I. These test sets are expanded according to the scan-chain configuration into a set of input stimuli for the combinational circuit. For the NXP designs

Table I. Circuit and Test Data Characteristics

	Gates	Delay Elements		Stuck-at Test Set	
Circuit	Fig. 9-a g	Fig. 9-b d_{can}	Fig. 9-c d_{opt}	Patterns p	Stimuli s
b21_1	7.9k	23k	18k	319	32k
b20_1	8.0k	23k	18k	347	35k
b20	9.3k	27k	21k	451	46k
b21	9.3k	27k	21k	454	46k
b22_1	12k	35k	28k	363	37k
b22	14k	41k	32k	439	44k
p35k	21k	62k	46k	1.2k	277k
p45k	22k	64k	48k	2.1k	139k
b17_1	22k	65k	49k	574	58k
b17	24k	70k	53k	545	55k
p469k	32k	94k	74k	315	328k
p77k	34k	98k	75k	515	274k
p100k	50k	145k	115k	2.1k	1.3M
p89k	53k	154k	116k	630	323k
p78k	58k	169k	136k	82	8.6k
b18_1	61k	179k	139k	584	59k
b18	63k	184k	143k	582	59k
p81k	71k	205k	171k	325	326k
p141k	96k	280k	212k	622	566k
b19_1	122k	357k	278k	644	65k
p269k	122k	361k	271k	748	566k
p267k	122k	361k	271k	728	551k
b19	125k	367k	285k	644	65k
p239k	145k	418k	329k	490	457k
p279k	148k	433k	330k	744	486k
p295k	149k	441k	326k	1.6k	5.3M
p330k	159k	466k	360k	2.3k	1.3M
p259k	181k	527k	415k	612	571k
p286k	187k	549k	423k	1.0k	684k
p418k	221k	643k	493k	841	793k
p500k	258k	750k	587k	595	484k
p388k	264k	772k	610k	453	446k
p378k	290k	843k	680k	80	8.4k
p483k	290k	842k	656k	346	322k
p874k	301k	884k	672k	1.2k	1.7M
p533k	363k	1.1M	827k	473	441k
p951k	478k	1.4M	1.1M	1.6k	4.0M
p1522k	550k	1.2M	1.2M	4.2k	12M

the scan-chain configurations were known while for the ITC benchmarks, configurations with a maximum scan chain length of 100 flip-flops were generated randomly. Figure 10 illustrates this step by a simple full-scan circuit with two scan chains of three flip-flops each. The application of a single test will generate three sets of input stimuli for loading the pattern, one set of stimuli for the capture cycle and another three stimuli for shifting out the response. These seven sets of input stimuli are easily obtained from the test pattern set in preprocessing and then simulated in parallel with the presented approach. The number of input stimuli s for each circuit is shown in the last column of Table I.

6.3. Simulation Runtime Performance

Since there are no standard implementations of GPGPU-based timing simulators with comparable capabilities available, the runtimes of a state-of-the-art event-based commercial simulator are used here as baseline. The event-based simulator ran on a single CPU of the host system. Although commercial simulators that support multithreading

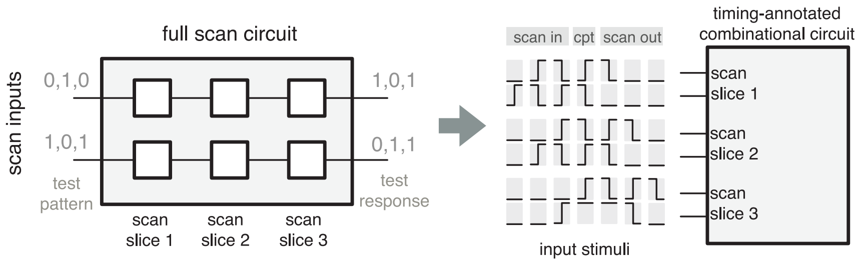

Fig. 10. Expanding a scan test pattern to input stimuli for parallel timing simulation.

are available, their scalability varies greatly with design data characteristics like hierarchy or modularity as well as used application features like parallel logging or reporting. Using a multithreaded event-based simulation tool as baseline would therefore introduce many uncertainties and additional variables which would render the reported speedup numbers less useful. By using single-threaded event-based simulation as baseline, the benefit of the presented approach over any particular multithreaded simulation environment can be easily calculated based on the speedup delivered by this environment and the numbers reported here.

We validated our simulator by running both simulators with the same input data and confirmed that they indeed produced exactly the same trace data on all signals within the design. Thus, the actual power estimation results are the same for both the event-based commercial simulator and the presented approach. However, as recording tracing data largely dominates the overall runtime in both simulators, tracing was disabled during performance measurements.

The runtimes reported for both the single-threaded event-based simulator and the presented approach only includes the elapsed time (wall-clock time) spent on actual simulation. For the event-based simulator, the timer is started just before the first stimulus is applied by the testbench and it is stopped with the successful simulation of the last stimulus. For the presented approach, the timer starts with the first communication to the GPU (transferring design data and first stimuli) and it is stopped as soon as the last level of the last simulation pass has been processed. This way, runtimes for the proposed approach include all sequential parts such as the time spent for transferring the design, test data and timing annotation to the GPU, as well as all simulation overheads for controlling and calibrating the simulation including reallocating waveform memory on the CPU while the GPU is idle.

Setup time spent on design loading, preprocessing and compilation was excluded for both simulators. While compilation time for the commercial event-based simulator was quite long for larger designs, the presented high-throughput simulator loads and prepares all designs quickly due to its efficient linear-time preprocessing. The largest reported circuit, p1522k, took less than 6 minutes from starting the simulation system until the first communication with the GPU and the start of the simulation performance timer.

The runtimes for state-of-the-art event-based timing simulation are shown in column t_{ebs} of Table II. Timing simulations even for the rather compact stuck-at test sets take very long, because every shift cycle is simulated separately. For example, b21_1 needs only 319 test patterns, but shifting causes 32k distinct input stimuli and event-based simulation takes 5.7 minutes. For larger designs, simulation times of several days were observed. Simulations of larger pattern sets as used in delay testing or logic built-in self testing are just impractical.

Table II. Performance Comparison and Speedups over State-of-the-Art Event-Based Timing Simulation for an Initial Waveform Capacity c' of 16 and 32

		Init. Waveform Capacity $c' = 16$					Init. Waveform Capacity $c' = 32$				
	Event-Based	Cold Run			Rerun		Cold Run			Rerun	
Circuit	t_{ebs}	cl_{16}	t_{cold16}		t_{full16}		cl_{32}	t_{cold32}		t_{full32}	
b21_1	5.7 m	2	2.3 s	149 X	940 ms	362 X	0	732 ms	465 X	788 ms	432 X
b20_1	5.3 m	2	2.5 s	126 X	949 ms	335 X	0	756 ms	420 X	788 ms	403 X
b20	9.3 m	3	3.6 s	156 X	1.0 s	538 X	0	1.1 s	500 X	1.0 s	543 X
b21	8.4 m	3	3.6 s	139 X	1.1 s	481 X	0	1.1 s	449 X	1.0 s	495 X
b22_1	8.7 m	3	3.7 s	142 X	1.1 s	486 X	0	1.2 s	453 X	1.1 s	475 X
b22	0:12 h	4	4.8 s	159 X	1.4 s	535 X	0	1.5 s	512 X	1.4 s	526 X
p35k	1:08 h	40	31 s	134 X	10 s	405 X	0	11 s	371 X	11 s	372 X
p45k	0:46 h	23	18 s	158 X	5.8 s	485 X	0	5.9 s	474 X	5.7 s	494 X
b17_1	0:13 h	8	6.6 s	119 X	2.4 s	334 X	0	2.2 s	361 X	2.3 s	349 X
b17	0:15 h	8	9.4 s	97 X	2.5 s	367 X	0	2.5 s	369 X	2.5 s	360 X
p469k	31:11 h	34	4.6 m	408 X	1.8 m	1044 X	38	4.4 m	422 X	1.8 m	1042 X
p77k	2:50 h	62	2.0 m	86 X	23 s	448 X	91	2.2 m	77 X	25 s	416 X
p100k	24:54 h	327	8.6 m	173 X	2.1 m	702 X	1	2.2 m	668 X	2.2 m	690 X
p89k	3:24 h	5	33 s	369 X	28 s	440 X	0	28 s	442 X	28 s	431 X
p78k	0:27 h	4	9.5 s	173 X	1.5 s	1089 X	0	1.5 s	1090 X	1.4 s	1155 X
b18_1	1:02 h	19	57 s	65 X	8.2 s	453 X	38	1.4 m	43 X	8.2 s	455 X
b18	1:08 h	20	1.1 m	63 X	8.2 s	499 X	39	1.5 m	46 X	8.5 s	483 X
p81k	6:18 h	110	4.1 m	93 X	42 s	539 X	0	43 s	532 X	43 s	533 X
p141k	19:16 h	265	0:13 h	86 X	1.7 m	693 X	0	1.8 m	656 X	1.7 m	669 X
b19_1	2:32 h	44	3.6 m	42 X	17 s	521 X	88	5.2 m	29 X	18 s	506 X
p269k	20:45 h	45	4.5 m	278 X	2.1 m	590 X	0	2.3 m	544 X	2.3 m	546 X
p267k	20:21 h	46	4.3 m	286 X	2.0 m	596 X	0	2.2 m	547 X	2.2 m	553 X
b19	2:49 h	45	3.6 m	47 X	17 s	580 X	92	5.5 m	30 X	18 s	561 X
p239k	32:05 h	500	1:06 h	28 X	2.5 m	773 X	134	0:16 h	115 X	2.7 m	704 X
p279k	15:47 h	187	0:15 h	60 X	2.2 m	434 X	0	2.4 m	396 X	2.4 m	399 X
p295k	175:42 h	413	0:54 h	192 X	0:22 h	460 X	0	0:25 h	408 X	0:25 h	421 X
p330k	80:37 h	994	1:19 h	61 X	6.6 m	728 X	0	7.5 m	647 X	7.2 m	675 X
p259k	46:56 h	728	1:54 h	24 X	3.8 m	748 X	187	0:28 h	100 X	4.2 m	675 X
p286k	30:03 h	264	0:30 h	59 X	3.8 m	469 X	0	4.2 m	428 X	4.1 m	439 X
p418k	63:58 h	869	1:45 h	36 X	6.2 m	620 X	58	0:14 h	271 X	7.4 m	521 X
p500k	58:06 h	906	2:42 h	21 X	5.0 m	699 X	717	2:06 h	27 X	5.9 m	586 X
p388k	67:05 h	726	2:22 h	28 X	4.4 m	909 X	610	1:55 h	34 X	5.0 m	812 X
p378k	2:11 h	21	3.0 m	43 X	7.2 s	1093 X	0	7.6 s	1043 X	7.1 s	1109 X
p483k	67:46 h	664	1:43 h	39 X	3.8 m	1061 X	542	1:20 h	50 X	4.4 m	917 X
p874k	187:39 h	1.7k	6:13 h	30 X	0:20 h	554 X	592	2:06 h	88 X	0:26 h	428 X
p533k	79:47 h	1.1k	3:27 h	23 X	6.4 m	752 X	646	1:56 h	41 X	7.5 m	641 X
p951k	790:10 h	8.4k	36:33 h	21 X	1:56 h	405 X	371	3:54 h	202 X	2:26 h	323 X
p1522k	1956:26 h	8.1k	61:23 h	31 X	6:28 h	301 X	2.1k	22:19 h	87 X	8:09 h	239 X

The runtime performance of the presented simulator depends on the initial waveform capacity c'. Low initial waveform capacities can cause many overflows and calibration loops at first, which reduces the overall performance significantly. This effect is reduced by choosing a higher initial waveform capacity. If the initial waveform capacity is chosen too high, however, a lot of memory is wasted on signals with a low number of hazards and data parallelism is reduced, which in turn again lowers the overall simulation performance.

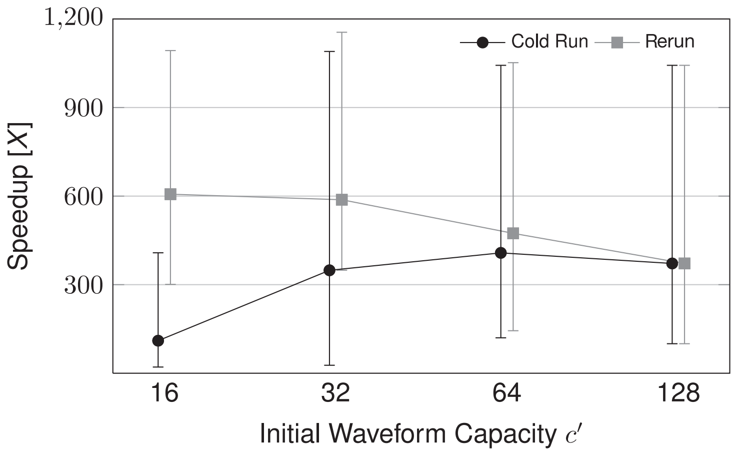

Fig. 11. Minimum, average, and maximum speedup over all benchmark circuits in relation to initial waveform capacity.

To investigate this trade-off, simulation runs with four initial waveform capacities ($c' = 16$, $c' = 32$, $c' = 64$, and $c' = 128$) were performed. For each capacity, two simulation runs are conducted: One *cold run*, which starts with the initial capacity c' for all waveforms and handles all overflows by calibration loops, and one *rerun*, which uses the final, calibrated waveform capacities obtained from the cold run in order to avoid calibration loops altogether. The runtimes obtained from cold runs represent the worst case and the times obtained from the reruns represent the best case for each benchmark design.

Table II shows the results for the two initial waveform capacities $c' = 16$ and $c' = 32$ for each benchmark circuit. A comparison of the number of necessary calibration loops for these initial waveform capacities (columns cl_{16} and cl_{32}) shows, as expected, a significant reduction for higher c'. The increase from $c' = 16$ to $c' = 32$ is already sufficient to avoid all overflows and to obtain the highest simulation performance even in the cold run for many benchmarks. The reduction of calibrations directly translates into improved runtimes for the cold runs (columns t_{cold16} and t_{cold32}). In the case of reruns (previously calibrated simulations) the performance for $c' = 16$ reported in column t_{full16} is better than for an initial capacity of $c' = 32$ (column t_{full32}). This expected effect is due to reduced data-parallelism for $c' = 32$ as in this case even signals with no hazards at all take double the amount of memory compared to a simulation with initial capacity of $c' = 16$.

Figure 11 shows the speedups obtained for $c' = 16$ and $c' = 32$ (data from Table II) as well as $c' = 64$ and $c' = 128$. The black graph shows the speedup on cold runs, and the grey graph shows the speedup on reruns. For each case and each initial capacity, the minimum, the maximum, and the average speedups over all benchmark circuits are shown.

Cold runs are slowest with low initial capacities like $c' = 16$ and fastest at $c' = 64$. After that point, the initial capacity of all signals is so high that no calibration loops are necessary anymore and both cold runs and reruns provide the same speedup. In reruns, the trend of decreasing speedup with higher initial waveform capacities continues for $c' = 64$ and $c' = 128$. However, the impact of reduced data-parallelism due to the higher c' is by far not as pronounced as the impact of calibration loops in the cold runs. Even for the largest benchmarks, the runtimes increase by only a few seconds or a minute at most, which is barely noticeable during real applications. This shows, that the benefit of putting great efforts into carefully adjusting waveform capacities exactly to the specific needs of signals is quite limited. Although a lot of memory remains unused by generously allocating space for signals with only few hazards, the resulting undisrupted data-parallel operation of the GPU provides a higher performance benefit than elaborate memory management and calibration techniques. This fact again underlines the efficiency of the memory allocation approach chosen for this simulator, which uniformly allocates space sufficient for the vast majority of signals and refines this allocation only in rather few cases where overflows occur.

Depending on the specific application area, either high long term performance (fastest reruns) of high instantaneous performance (fast cold runs) are desired. If maximum performance is desired in the long run, an initial capacity of $c' = 16$ should be chosen. Maximum instantaneous performance is provided with an initial capacity of $c' = 64$.

If no calibration is necessary, the CPU only controls the simulation process and the entire runtime is determined by GPU performance and transfers. With a memory bandwidth of 15.75 GB/s between the CPU and GPU memory, the run-time is dominated by the GPU performance. For example the runtime for circuit p1522k reported in column t_{full32} of table II is 8 hours 9 minutes. It consists of 18 minutes (3%) used for transfers and 7 hours 51 minutes (97%) used in the GPU simulation kernels. In case of calibrations, CPU executes the mentioned memory management tasks, but even then, over 90% of the runtime is spent by the GPU.

Compared to the performance of event-based simulation, the presented simulator is always one or two orders of magnitude faster, with peaks of up to three orders of magnitude for some benchmark circuits. Even with many calibration loops in the cold run with $c' = 16$, column t_{cold16} shows a significant runtime improvement with speedups ranging from 21 X to 408 X across all benchmarks. With an initial waveform capacity of $c' = 32$, the observed speedups for cold runs improve to a range of 27 X to 1090 X. The cold run speedups improve further if an initial waveform capacity of $c' = 64$ is used and range from 120 X to 1043 X. For $c' = 128$, speedups between 100 X and 1043 X are observed. In calibrated simulation runs, speedups of at least 273 X were observed. Calibrated simulations with our simulator reduce an event-based simulation time of 24 hours down to 5.3 minutes in the worst and 1.3 minutes in the best case. Among all GPGPU-based simulators available, the presented simulator is the first that consistently performs at least two orders of magnitude faster than event-based simulation on CPUs even for very large designs. It is also the first that calculates all hazards and handles collisions as well as glitches correctly in the GPGPU simulation kernel in order to produce exactly the same results as commercial simulators on CPUs.

7. CONCLUSIONS

The presented waveform principle, in which complete signal histories are propagated through the combinational circuit, combines the versatility of event-based timing simulation and the computing power of data-parallel architectures in a unique way. For the first time, hazards, pulse-filtering, and pin-to-pin delays are supported in a GPGPU-based gate-level simulator.

The simulator exploits two dimensions of parallelism to generate sufficient GPGPU workload over a wide range of design sizes. The memory efficient encoding and fast evaluation of waveforms with careful consideration of control flow and memory access patterns allows the very efficient use of the computing resources and memory bandwidth available. Together with optimal memory management, minimal data transfer overhead, and efficient calibration, the complete system is able to deliver simulation performance one to more than two orders of magnitude faster than state-of-the-art event-based simulators.

ACKNOWLEDGMENTS

We would like to thank Eric Schneider for his support.

REFERENCES

M. Bailey, J. Briner Jr, and R. Chamberlain. 1994. Parallel logic simulation of VLSI systems. *ACM Comput. Surv.* 26, 3, 255–294.

B. Catanzaro, K. Keutzer, and B. Su. 2008. Parallelizing CAD: A timely research agenda for EDA. In *Proceedings of the 45th ACM/IEEE Design Automation Conference*. IEEE, 12–17.

K. Chandy and J. Misra. 1981. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM* 24, 4, 198–206.

D. Chatterjee, A. Deorio, and V. Bertacco. 2011. Gate-level simulation with GPU computing. *ACM Trans. Des. Autom. Electron. Syst.* 16, 30:1–30:26.

J. F. Croix and S. P. Khatri. 2009. Introduction to GPU programming for EDA. In *IEEE/ACM International Conference on Computer-Aided Design: Digest of Technical Papers*. 276–280.

A. Czutro, N. Houarche, P. Engelke, I. Polian, M. Comte, M. Renovell, and B. Becker. 2008. A simulator of small-delay faults caused by resistive-open defects. In *Proceedings of the 13th IEEE European Test Symposium*. 113–118.

A. Czutro, M. E. Imhof, J. Jiang, A. Mumtaz, M. Sauer, B. Becker, I. Polian, and H.-J. Wunderlich. 2012. Variation-aware fault grading. In *Proceedings of the 21st IEEE Asian Test Symposium*. 344–349.

K. Gulati and S. Khatri. 2009. Accelerating statistical static timing analysis using graphics processing units. In *Proceedings of the Asia and South Pacific Design Automation Conference*. 260–265.

K. Gulati and S. P. Khatri. 2010. Fault table computation on GPUs. *J. Electron. Test.* 26, 2, 195–209.

S. Holst, E. Schneider, and H.-J. Wunderlich. 2012. Scan test power simulation on GPGPUs. In *Proceedings of the 21st IEEE Asian Test Symposium*. 155–160.

D. Jefferson. 1985. Virtual time. *ACM Trans. Program. Lang. Syst.* 7, 3, 404–425.

K. C. Knowlton. 1965. A fast storage allocator. *Commun. ACM* 8, 10, 623–624.

D. E. Knuth. 1969. *The Art of Computer Programming Vol. 1: Fundamental Algorithms* 2nd Ed. Addison-Wesley.

M. A. Kochte, M. Schaal, H.-J. Wunderlich, and C. G. Zoellin. 2010. Efficient fault simulation on many-core processors. In *Proceedings of the 47th ACM/IEEE Design Automation Conference*. 380–385.

H. Li, D. Xu, Y. Han, K.-T. Cheng, and X. Li. 2010. nGFSIM: A GPU-based fault simulator for 1-to-n detection and its applications. In *Proceedings of the IEEE International Test Conference*. 12.1/1–12.1/10.

L. Li, X. Yu, C.-W. Wu, and Y. Min. 2000. A waveform simulator based on Boolean process. In *Proceedings of the 9th Asian Test Symposium*. 145–150.

M. Li and M. Hsiao. 2010. FSimGP2: An efficient fault simulator with GPGPU. In *Proceedings of the 19th IEEE Asian Test Symposium*. 15–20.

C. J. Lin and S. M. Reddy. 1987. On delay fault testing in logic circuits. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 6, 5, 694–703.

E. Macii, M. Pedram, and F. Somenzi. 1997. High-level power modeling, estimation, and optimization. In *Proceedings of the 34th ACM/IEEE Design Automation Conference*. ACM, 504–511.

S. Meraji and C. Tropper. 2012. Optimizing techniques for parallel digital logic simulation. *IEEE Trans. Parallel Distrib. Syst.* 23, 6, 1135–1146.

Y. Min, Z. Zhao, and Z. Li. 1996. An analytical delay model based on Boolean process. In *Proceedings of the 9th Conference on VLSI Design*. 162–165.

R. Mueller-Thuns, D. Saab, R. Damiano, and J. Abraham. 1993. VLSI logic and fault simulation on general-purpose parallel computers. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 12, 3, 446–460.

F. Najm. 1994. A survey of power estimation techniques in VLSI circuits. *IEEE Trans. VLSI Syst.* 2, 4, 446–455.

NVIDIA. 2013. NVIDIA CUDA homepage https://developer.nvidia.com/category/zone/cuda-zone.

OCL. 2011. NanGate Open Cell Library v1.3 http://si2.org/openeda.si2.org/projects/nangatelib.

J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. 2008. GPU computing. *Proc. IEEE* 96, 5, 879–899.

S. M. Reddy, M. K. Reddy, and V. D. Agrawal. 1984. Robust tests for stuck-open faults in CMOS combinational logic circuits. In *Proceedings of the 14th International Fault-Tolerant Computing Symposium*. 4449.

M. Sauer, I. Polian, M. E. Imhof, A. Mumtaz, E. Schneider, A. Czutro, H.-J. Wunderlich, and B. Becker. 2014. Variation-Aware Deterministic ATPG. In *Proceedings of the 19th IEEE European Test Symposium*. 87–92. (Best Paper Award.)

A. Shen, A. Ghosh, S. Devadas, and K. Keutzer. 1992. On average power dissipation and random pattern testability of CMOS combinational logic networks. In *Proceedings of the IEEE/ACM International Conference on Computer-aided Design*. 402–407.

L. Soule and A. Gupta. 1989. Parallel distributed-time logic simulation. *IEEE Des. Test Comput.* 6, 6, 32–48.

C. Wang and K. Roy. 1998. Maximum power estimation for CMOS circuits using deterministic and statistical approaches. *IEEE Trans. VLSI Syst.* 6, 1, 134–140.

Y. Zhu, B. Wang, and Y. Deng. 2011. Massively parallel logic simulation with GPUs. *ACM Trans. Des. Autom. Electron. Syst.* 16, 3, 29.

Received June 2013; revised December 2013, November 2014; accepted January 2015