Scan Test Power Simulation on GPGPUs

Holst, Stefan; Schneider, Eric; Wunderlich, Hans-Joachim

Proceedings of the 21st IEEE Asian Test Symposium (ATS'12) Niigata, Japan, 19-22 November 2012

doi: http://dx.doi.org/10.1109/ATS.2012.23

Abstract: The precise estimation of dynamic power consumption, power droop and temperature development during scan test 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. We propose a new, throughput–optimized timing simulator on running on GPGPUs to accelerate these tasks by more than two orders of magnitude and thus providing for the first time precise and comprehensive toggle data for industrial–sized designs and over long scan test operations. Hazards and pulse–filtering are supported for the first time in a GPGPU accelerated simulator, and the system can easily be extended to even more sophisticated delay and power models.

Preprint

Scan Test Power Simulation on GPGPUs

Stefan Holst, Eric Schneider and Hans-Joachim Wunderlich
University of Stuttgart
Pfaffenwaldring 47
70569 Stuttgart, Germany
Email: {holst,schneiec,wu}@iti.uni-stuttgart.de

***Abstract*—The precise estimation of dynamic power consumption, power droop and temperature development during scan test 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.**

We propose a new, throughput–optimized timing simulator on running on GPGPUs to accelerate these tasks by more than two orders of magnitude and thus providing for the first time precise and comprehensive toggle data for industrial–sized designs and over long scan test operations. Hazards and pulse–filtering are supported for the first time in a GPGPU accelerated simulator, and the system can easily be extended to even more sophisticated delay and power models.

***Index Terms*—GPGPU, Data–Parallelism, Scan–Test, Power, Time–Simulation, Hazards, Pulse–Filtering**

I. Introduction

Power estimation is one of the most crucial steps in physical design [1–3]. Compared to the functional operation of a design, scan testing generates much higher switching activity in a module [4] and great effort is put into optimizing test patterns [5], scan clock schemes [6], and test schedules [7–10] to keep test power within affordable limits [11–13].

Tremendous speedups are gained by using data–parallel architectures like *general purpose graphics processing units* (GPGPU) [14, 15] for problems in electronic design automation [16] such as electrical simulation [17], circuit optimization [18] or power grid analysis [19]. Gate level simulations have been accelerated using GPGPUs by parallel computation of independent gates [20], faults [21–24], or Monte–Carlo samples for statistical static timing analysis [25].

The most precise way to determine the data dependent power of synchronous sequential circuits is to perform a complete time simulation and compute the weighted switching activity (WSA) [1] based on all the events observed on internal signals. However, all proposed GPGPU gate level simulators either do not consider timing at all (zero delay model) or only calculate the latest transition at each gate [25]. This is not sufficient for power estimation as hazards may account for up to 70% of dynamic power [26].

We propose a novel time simulator for GPGPUs, which takes a combinational time–annotated gate–level circuit and calculates for each set of input transitions all events on the internal signals and outputs. In scan test, all input assignments to the circuit are known in advance and can be evaluated in parallel by the time simulator. Consequently, the simulator is designed for high throughput of many independent simulation runs and taps into the full potential of throughput–optimized data–parallel architectures. An efficient linear–time pre–processing and low memory requirements let the simulator handle even industrial–sized designs. The support for arbitrary cells, individual timing per cell instance and a large class of timing models makes this simulation system an ideal tool for generating high–quality switching activity data for further analysis.

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 III presents the overall time simulation system, and section IV describes the data–parallel simulation core in detail. The series of experiments reported in section V shows the performance benefit of the new simulator.

II. 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 a data–parallel 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 paths. If the control flow of two threads diverges as a result of a data dependent conditional branch, however, some execution units may become idle and the performance degrades until the control flow 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 w bytes wide bus between the on–board memory and the GPGPU. To use all w bytes in a transaction, threads of the same batch must access data in the same region at the same time.

III. Time Simulation System

Figure 1 shows the overall time simulation system with control flow in the vertical direction and data flowing from left to right. The complete time simulation system consists of both sequential and data–parallel tasks. The sequential tasks (white boxes in fig. 1), which involve scheduling and allocations during initialization and calibration, are performed on a latency–optimized CPU. The data–parallel tasks (shaded boxes in fig. 1), which form the inner simulation loop, are performed on the throughput–optimized GPGPU.

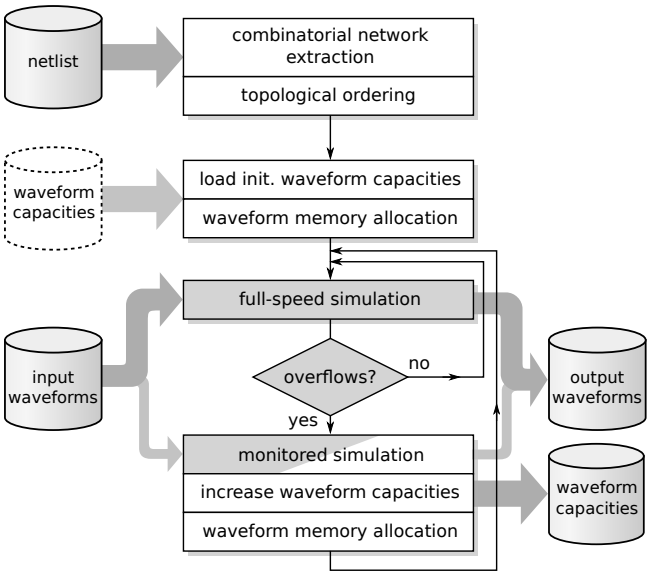

Fig. 1. Overall simulation flow.

The sequential tasks are concerned with maximizing the amount of parallelism for a given circuit to reach maximum performance in the inner simulation loop. In the inner loop, data and control dependencies are reduced to a minimum and maximum simulation throughput is obtained by spawning the maximum number of parallel threads for the simulation task.

A. Design Preprocessing and Topological Ordering

Event–based time simulation approaches are not the optimal choice for data–parallel architectures because of the high number of necessary synchronizations and irregular memory access patterns. Instead, the proposed time simulator follows an oblivious simulation approach that propagates the input data towards the outputs in a single pass. 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.

B. Waveset Capacities and Calibration

The input data will be processed within a single pass over the circuit propagating intermediate results over each level of gates until the outputs are reached. For each signal and input assignment, the complete history of transitions is stored in a data structure called *waveform* [27–29]. Details on the encoding of a waveform will be given in section IV. Multiple independent input assignments are simulated at the same time (pattern–parallelism). Therefore, each internal signal needs storage for a set of waveforms, called a *waveset*. The storage requirement per waveset depends on the number of input assignments s processed in parallel and the number of transitions c to be stored per signal. The number of transitions for a signal is not known in advance and may be different for each signal and input assignment. 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, all waveforms in a waveset have the same *capacity* c.

The number of possible hazards on a signal is bound by the amount of logic between this signal and the driving sequential elements. Signals near the pseudo–primary inputs usually show less hazards within a single clock cycle than signals near the outputs. It is therefore reasonable to associate each internal signal with an expected number of transitions. If this information is available from previous simulation runs, it is loaded in step three (fig. 1) to initialize the capacities accordingly. Otherwise, all capacities are initialized to some initial value c'.

During simulation, the number of transitions on a signal may exceed the capacity of its associated waveset causing an *overflow*. If this situation is detected after processing a set of inputs, the simulator will enter a *calibration loop* (the outer loop in fig. 1). This calibration loop processes the same set of inputs again level by level in a monitored simulation, which checks each generated waveset for overflows. If an overflow is detected in a waveset, its capacity is doubled and the same level is simulated again until all overflows are avoided. 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 (the inner loop in fig. 1), which provides an efficient, global overflow check and avoids all memory re–organizations to reach maximum simulation performance.

C. Waveset Memory Allocation

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 storage spaces for intermediate signal wavesets necessary to complete

one simulation pass over the circuit. The number of input assignments s that can be time simulated in parallel is

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

The waveset capacities are already determined by the number of transitions on the signals. To maximize pattern–parallelism, r has to be reduced to a minimum. Figure 2 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 an allocation with minimum r with a single pass over the circuit, re–using each location as soon as its intermediate result is not anymore required.

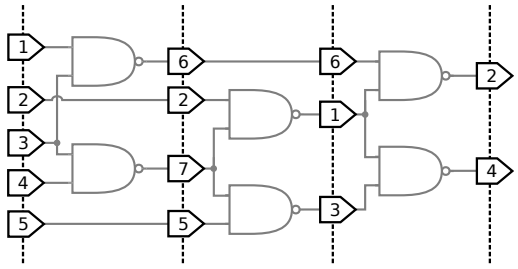

Fig. 2. A possible waveset allocation for the c17 with ASAP–scheduling. Locations 1 and 3 are re–used in the third barrier and locations 2 and 4 are re–used in the last one.

However, wavesets may have different capacities and using just storage locations of one common chunk size would be very inefficient. Moreover, capacities may change during calibration and efficient efficient re–allocation should be possible, too. The sizes of memory chunks to are proportional to $2^i \cdot c'$ with $i \geq 0$ and c' the base capacity before any calibration. One very efficient way to manage memory chunks of these sizes is a buddy system [30, 31]. 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 3 shows an example of such a tree. 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 exist, 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 3. 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 [30], 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.

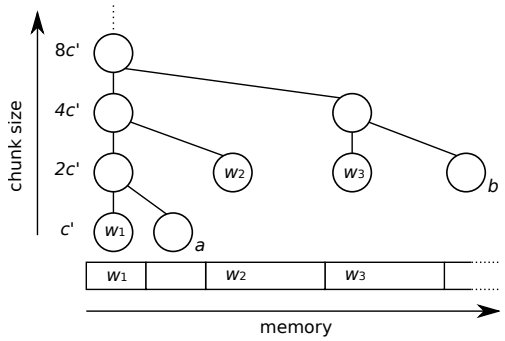

Fig. 3. Example of a binary tree for waveset memory management. The waveset w_1 was allocated with size c', w_2 and w_3 are both of size $2c'$.

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 re–allocating wavesets during calibration. If the monitored simulation discovers the first overflows on level l, the appropriate waveset capacities are doubled and all wavesets for this level are newly allocated using the binary tree of level $l-1$. All remaining levels $> l$ are re–allocated as well during the current calibration run updating the trees for each level accordingly. The re–allocation of later levels could be avoided by using more sophisticated methods, but many wavesets on these levels are likely to be increased as well due to the propagation of hazards, which will lead to re–allocations anyway. More intelligent allocation methods would provide no benefit.

D. Parallelism

Figure 4 shows the two dimensions of parallelism exploited by the simulator. Pattern–parallelism is exploited by processing s independent input assignments at the same time, and gate–parallelism is exploited by evaluating all g gates within a level in parallel. Only the combination of these two dimensions generates enough $(s \cdot g)$ threads to fully occupy typical GPGPUs. Small circuits with low memory requirement r allow for more input assignments s to be processed in parallel. For larger circuits with higher memory requirement r, the number of samples 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 size M.

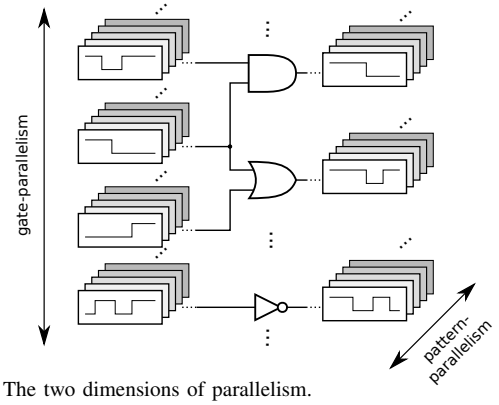

Fig. 4. The two dimensions of parallelism.

IV. 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. 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.

As parallelism is bound by the on–board memory size M, the waveform encoding must be very memory efficient. At the same time, the encoding should allow for fast gate evaluations with a simple control flow for efficient data–parallel execution and the best possible use of memory transactions for optimal on–board memory throughput. The waveform representation and evaluation algorithm presented here is tuned towards 2–valued simulations for maximum efficiency. However, the principle evaluation approach is also applicable to multi–valued simulations and waveform representations like in [29].

A. Waveform Representation

Let v_t be the signal value at time t. A waveform is a description of signal values v_t for all $t > 0$. 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:

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

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. 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.

Figure 5 shows some waveforms and their representations. 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.

B. Waveform Evaluation

The algorithm described below computes the output waveform z for a gate with $n \ge 1$ inputs. Cells with more than one output are supported by performing the evaluation for each output individually. Given are the logic function of the gate $f(v^1, \ldots, v^n)$, the delay for a rising transition at the output δ_r, the delay for a falling transition δ_f, the positive and negative inertial delays δ_{ip}, δ_{in}, and the waveform at each input $w^1, \ldots, w^n$.

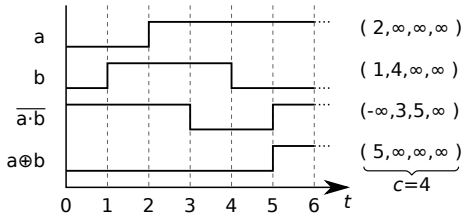

Fig. 5. Waveforms and their representations. Rising, falling and inertial delays are $1t$.

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 does not generate positive pulses smaller than δ_{ip} or negative pulses smaller than δ_{in}. The logic values for the evaluation of f are given by the current positions in the input waveforms. As in a waveform $w = (t_0, t_1, \ldots)$, t_0 is always the time of a rising transition towards 1 and t_1 is always the time of a falling transition towards 0, the index i of a t_i can be used to determine the signal values at the transition. The signal value before a transition at t_i is just $v_{t_i-\varepsilon} = i \bmod 2$ (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 or a collision, 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—or even earlier than—a transition t' saved previously, it is discarded and t' is removed from z by decreasing the output waveform index by one.

Figure 6 shows the computation of a waveform in pseudo–code. In addition to the parameters already introduced, each waveform w contains a special integer field recording overflows w_{ov}. A positive value in such an overflow field indicates, that transitions may be missing in the waveform due to an overflow in the current or previous gate evaluations. Lines 1–7 initialize the input waveform indices $i_1, \ldots, i_n$ and the output waveform index j. The overflow indicator z_{ov} is initialized to the sum of the overflow field values of all inputs. The running sum over the overflow fields allows a very efficient overflow check at the very end of the simulation. If the function f evaluates to one for all zero inputs, the output waveform is initialized with an initial value of one. The following while–loop processes transitions until the terminal symbol ∞ is reached for all the input waveforms. Lines 9–11 select an earliest, unprocessed transition (there may be more than one) and consumes it by advancing the appropriate index i_k. If the logic function of the cell produces a value different from the current one ($j \bmod 2$), a transition may be generated in z at time $t + \delta[j \bmod 2]$. The if–statement at line 14 implements three important tasks. It checks for collisions (with $\delta_r \neq \delta_f$, output transitions may overtake each other and have to be discarded), it handles the pulse–filtering condition (inertial delays), and it prevents the storage of redundant $-\infty$ symbols in the output waveform. If the output waveform is empty ($j = 0$), the transition is always stored. Otherwise, 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. 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 if–statement at line 15 checks for the overflow condition. If an overflow occurs, the overflow indicator z_{ov} is increased and the while–loop is terminated at this point. After the loop, the symbol ∞ is added to z.

parameters:
function of the cell: $f(v^1, \ldots, v^n)$
array containing rising and falling delays: $\delta = [\delta_r, \delta_f]$
array containing inertial delays: $\delta_i = [\delta_{in}, \delta_{ip}]$
waveforms at the cell's inputs: $w^1, \ldots, w^n$
overflow indicators: $w^1_{ov}, \ldots, w^n_{ov}$

results:
waveform at the cell's output: z
overflow indicator: z_{ov}

```
1:  i_1, ..., i_n ← 0
2:  j ← 0
3:  z_ov ← Σ_{x=1}^{n} w^x_ov
4:  if f(0, ..., 0) = 1 then
5:     z_0 ← −∞
6:     j ← 1
7:  end if
8:  while min{w^1_{i_1}, ..., w^n_{i_n}} < ∞ do
9:     t ← min{w^1_{i_1}, ..., w^n_{i_n}}
10:    choose k with w^k_{i_k} = t
11:    i_k ← i_k + 1
12:    if (j mod 2) ≠ f(i_1 mod 2, ..., i_n mod 2) then
13:       t ← t + δ[j mod 2]
14:       if j = 0 or t − z_{j−1} > δ_i[j mod 2] then
15:          if j = |z| − 1 then
16:             z_ov ← z_ov + 1
17:             break
18:          end if
19:          z_j ← t
20:          j ← j + 1
21:       else
22:          j ← j − 1
23:       end if
24:    end if
25: end while
26: z_j ← ∞
```

Fig. 6. Waveform evaluation in pseudo–code.

The overflow fields for all waveforms at the inputs of the circuit are initialized to 0. If at the end of a simulation run any output waveform shows a positive overflow field, a calibration loop is performed on the input data. This overflow check is very efficient and does not add any additional data dependencies. During monitored simulation, the overflow fields are checked after gate evaluation of each level.

The weighted switching activity for each input assignment is accumulated in much the same way as overflows are recorded and propagated. The only difference is, that a field is increased by the weighted number of transitions in each waveform evaluation and special care is taken not to add activities twice at re–convergences. After simulation, a simple sum over all circuit outputs gives the total weighted switching activity for each assignment.

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 the most expensive operations (summing, min–operator and gate function evaluation) are unconditioned. The conditioned operations include only quite inexpensive floating–point additions, 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. During this operation, each thread will access their input waveforms only once and in the order of increasing transition times. I.e. in the first iteration, all threads access t_0 in their input waveforms. The memory layout of a waveset is organized in a way, such that the t_0 fields of all the waveforms are stored in the beginning, followed by all t_1 fields, and so on. This allows a batch to fill the memory transactions as much as possible during the loop iterations.

V. Experimental Results

The simulation algorithm has been implemented on CUDA™–enabled hardware from NVIDIA® [15]. This hardware provides high performance for single precision floating point operations, therefore 32–bit floating point numbers are chosen for the transition times in the waveforms. The host system for the simulation experiments contains Intel® Xeon® processors with 3.33GHz and 48GB RAM. The CUDA™–device is a Fermi GPU clocked with 1.1GHz and is attached to 6GB of on–board memory.

The experiments were performed on the largest ITC'99 circuits and industrial designs provided by NXP. All benchmarks were mapped to primitive gates with at most two inputs and the delay of all gates is set to the same constant value. The scan–chain configuration for the NXP benchmarks were known, for the ITC'99 circuits, configurations with a maximum chain length of 100 were generated.

A stuck–at test set was generated for each circuit and expanded beforehand according to the scan–chain configuration into the set of s input stimuli for the combinational circuit. This data was given to a state–of–the–art commercial event–based simulator running on a single CPU of the host system and to our GPGPU–based simulation system to measure the runtimes. In the commercial simulator, actual transition counting was not performed as this would have led to costly file operations and a biased comparison.

Table I compares the simulation performance of the commercial event–based simulator to our approach. The first two columns show design names and their size g in gates. The next columns show the number of stuck–at fault test patterns p and the number s of input stimuli after expansion. The column e shows the simulation result, the number of transitions caused on all internal signals by shifting all p test patterns through the scan–chains. The transition counts range from 568 Million for p78k up to many Billion for the larger circuits. Column T_{CS} shows, that the commercial event–based simulator takes hours for the smaller circuits and even days for the larger designs to complete. In the first run on the GPGPU based simulator, no waveform capacity data is available and all capacities are initialized to $c' = 16$. The column T_{cold} already shows a significant runtime improvement over the commercial simulator despite the fact, that some calibration loops are needed to adjust the waveform capacities. Calibrations take time as they involve overflow checking for each level and additional communication between CPU and GPGPU. However, only very few calibration loops are needed and the performance is still faster by factors of 24X–188X

compared to the commercial simulator. The GPGPU based simulator was run again with the waveform capacities obtained from the first run and the time is reported in column T_{fs}. This represents the best case, as only full–speed simulations are performed without any calibrations. Now, speedups of at least two orders of magnitude are obtained in every case, reaching up to 731X for p78k. Simulation effort is reduced from days to a few minutes for larger designs.

TABLE I
PERFORMANCE COMPARISON

circuit	g	p	s	e	T_{CS}	T_{cold}	T_{fs}
b22	32k	964	97k	2.5G	0:56h	24s (142X)	7.3s (463X)
b17	36k	1.3k	129k	2.0G	0:33h	32s (62X)	10s (197X)
p77k	72k	588	180k	18G	3:56h	4.9m (49X)	1.0m (234X)
p78k	74k	81	5.3k	568M	0:15h	4.9s (188X)	1.3s (731X)
p100k	97k	2.1k	1.6M	57G	13:25h	0:20h (39X)	4.5m (181X)
p81k	109k	1.3k	687k	35G	9:43h	9.7m (60X)	2.2m (266X)
b18	125k	1.5k	148k	15G	4:37h	4.6m (60X)	46s (363X)
p141k	173k	1.6k	784k	79G	1d 8h	0:33h (58X)	4.5m (430X)
b19	252k	1.7k	170k	35G	12:26h	0:15h (47X)	1.7m (438X)
p239k	259k	1.1k	586k	106G	1d 6h	1:13h (25X)	5.4m (344X)
p267k	272k	1.1k	562k	61G	19:50h	8.1m (148X)	4.3m (279X)
p279k	288k	1.3k	528k	54G	19:32h	0:51h (23X)	4.3m (270X)
p259k	335k	1.2k	677k	135G	2d 9h	1:36h (36X)	7.5m (462X)
p500k	496k	2.4k	1.1M	370G	7d 1h	5:28h (31X)	0:25h (394X)
p874k	717k	1.8k	1.4M	527G	8d 7h	8:23h (24X)	0:31h (381X)
p951k	1.0M	2.3k	3.1M	2.0T	37d10h	1d 9h (27X)	1:43h (519X)

VI. CONCLUSIONS

The presented throughput–optimized timing simulator computes all signal transitions and supports for the first time hazards and pulse–filtering in a GPGPU implementation. It generates high–quality switching activity data more than two orders of magnitude faster than commercial event–based simulators and thus enables exact scan test power estimation for industrial–sized designs and large test sets.

The partitioning of the simulation system in sequential and data–parallel tasks, the consequent reduction of data dependencies and synchronizations in the data–parallel part and the exploitation of the two dimensions of parallelism tap into the full potential of GPGPUs. The memory efficient encoding and fast evaluation of waveforms with careful consideration of control flow and memory access patterns allows to exploit this potential.

VII. ACKNOWLEDGEMENT

We would like to thank Michael Imhof and Claus Braun for their support.

REFERENCES

[1] F. Najm, "A survey of power estimation techniques in VLSI circuits," *IEEE Trans. on VLSI Systems*, vol. 2, no. 4, pp. 446–455, Dec 1994.

[2] E. Macii, M. Pedram, and F. Somenzi, "High-level power modeling, estimation, and optimization," in *Proc. 34th Design Automation Conference (DAC)*. ACM, 1997, pp. 504–511.

[3] C. Wang and K. Roy, "Maximum power estimation for CMOS circuits using deterministic and statistical approaches," *IEEE Trans. on VLSI Systems*, vol. 6, no. 1, pp. 134–140, 1998.

[4] P. Girard, "Survey of low-power testing of VLSI circuits," *IEEE Design & Test*, pp. 82–92, 2002.

[5] K. Butler, J. Saxena, T. Fryars, G. Hetherington, A. Jain, and J. Lewis, "Minimizing power consumption in scan testing: Pattern generation and DFT techniques," in *Proc. IEEE International Test Conference (ITC)*, 2004, pp. 355–364.

[6] P. Girard, L. Guiller, C. Landrault, S. Pravossoudovitch, and H.-J. Wunderlich, "A modified clock scheme for a low power BIST test pattern generator," in *Proc. 19th IEEE VLSI Test Symposium (VTS)*, 2001, pp. 306–311.

[7] Z. He, Z. Peng, P. Eles, P. Rosinger, and B. Al-Hashimi, "Thermal-aware soc test scheduling with test set partitioning and interleaving," *Journal of Electronic Testing: Theory and Applications*, vol. 24, no. 1-3, pp. 247–257, 2008.

[8] M. E. Imhof, C. G. Zoellin, H.-J. Wunderlich, N. Maeding, and J. Leenstra, "Scan test planning for power reduction," in *Proc. 44th Design Automation Conference (DAC)*. ACM, June 2007, pp. 521–526.

[9] P. Rosinger, B. Al-Hashimi, and K. Chakrabarty, "Rapid generation of thermal-safe test schedules," in *Proc. Design, Automation and Test in Europe (DATE)*, 2005, pp. 840–845.

[10] R. Sankaralingam, N. A. Touba, and B. Pouya, "Reducing power dissipation during test using scan chain disable," in *Proc. 19th IEEE VLSI Test Symposium (VTS)*, 2001, pp. 319–325.

[11] P. Girard and H.-J. Wunderlich, "Models for power-aware testing," in *Models in Hardware Testing - Lecture Notes of the Forum in Honor of Christian Landrault*. Springer-Verlag Berlin Heidelberg, 2009.

[12] L. Whetsel, "Adapting scan architectures for low power operation," in *Proc. IEEE International Test Conference (ITC)*, 2000, pp. 863–872.

[13] S. Gerstendörfer and H. Wunderlich, "Minimized power consumption for scan-based BIST," in *Proc. International Test Conference (ITC)*, 1999, pp. 77–84.

[14] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU computing," *Proc. of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[15] "NVIDIA CUDA homepage." [Online]. Available: `http://developer.nvidia.com/object/cuda.html`

[16] J. F. Croix and S. P. Khatri, "Introduction to GPU programming for EDA," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD) - Digest of Technical Papers*, Nov. 2009, pp. 276–280.

[17] K. Gulati, J. F. Croix, S. P. Khatr, and R. Shastry, "Fast circuit simulation on graphics processing units," in *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2009, pp. 403–408.

[18] Y. Liu and J. Hu, "GPU-based parallelization for fast circuit optimization," in *Proc. 46th Design Automation Conference (DAC)*, 2009.

[19] J. Shi, Y. Cai, W. Hou, L. Ma, S. X.-D. Tan, P.-H. Ho, and X. Wang, "GPU friendly fast poisson solver for structured power grid network analysis," in *Proc. 46th Design Automation Conference (DAC)*, 2009.

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

[21] M. A. Kochte, M. Schaal, H.-J. Wunderlich, and C. G. Zoellin, "Efficient fault simulation on many-core processors," in *Proc. 47th IEEE/ACM Design Automation Conference (DAC)*, 2010, pp. 380–385.

[22] M. Li and M. Hsiao, "FSimGP^2: An efficient fault simulator with GPGPU," in *Proc. 19th IEEE Asian Test Symposium (ATS)*, Dec. 2010, pp. 15–20.

[23] H. Li, D. Xu, Y. Han, K.-T. Cheng, and X. Li, "nGFSIM: A GPU-based fault simulator for 1-to-n detection and its applications," in *Proc. IEEE International Test Conference (ITC)*, Nov. 2010, pp. 12.1/1–12.1/10.

[24] K. Gulati and S. P. Khatri, "Fault table computation on GPUs," *Journal of Electronic Testing*, vol. 26, no. 2, pp. 195–209, Apr. 2010.

[25] K. Gulati and S. Khatri, "Accelerating statistical static timing analysis using graphics processing units," in *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2009, pp. 260–265.

[26] A. Shen, A. Ghosh, S. Devadas, and K. Keutzer, "On average power dissipation and random pattern testability of CMOS combinational logic networks," in *Proc. IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, 1992, pp. 402–407.

[27] Y. Min, Z. Zhao, and Z. Li, "An analytical delay model based on boolean process," in *Proc. 9th Conference on VLSI Design*, Jan. 1996, pp. 162–165.

[28] L. Li, X. Yu, C.-W. Wu, and Y. Min, "A waveform simulator based on boolean process," in *Proc. 9th Asian Test Symposium (ATS)*, 2000, pp. 145–150.

[29] A. Czutro, N. Houarche, P. Engelke, I. Polian, M. Comte, M. Renovell, and B. Becker, "A simulator of small-delay faults caused by resistive-open defects," in *Proc. 13th IEEE European Test Symposium (ETS)*, May 2008, pp. 113–118.

[30] K. C. Knowlton, "A fast storage allocator," *Commun. ACM*, vol. 8, no. 10, pp. 623–624, Oct. 1965.

[31] D. E. Knuth, *The Art of Computer Programming Vol. 1 - Fundamental Algorithms*, 2nd ed. Addison-Wesley, 1969.