

# Accelerating Monte Carlo based SSTA Using FPGA

Jason Cong, Karthik Gururaj, Wei Jiang, Bin Liu, Kirill Minkovich, Bo Yuan and Yi Zou

Computer Science Department, University of California, Los Angeles

Los Angeles, CA 90095, USA

{cong, karthikg, wjiang, bliu, cory\_m, boyuan, zouyi}@cs.ucla.edu

## ABSTRACT

Monte Carlo based SSTA serves as the golden standard against alternative SSTA algorithms, but it is seldom used in practice due to its high computation time. In this paper, we accelerate Monte Carlo based SSTA using the FPGA platform. A simple dataflow pipeline technique will not work well due to the excessive usage of FPGA logic slices. We leverage the recently proposed pattern matching method to identify common circuit structures, and further use a mathematical programming based formulation to explore the trade-off between performance and logic slices consumption. The proposed design provides two orders of magnitude speedup compared to the CPU-based implementation.

## Categories and Subject Descriptors

B.7.1 [Integrated Circuits]: Types and Design Styles – Algorithms implemented in hardware.

## General Terms

Algorithms, Design, Performance.

## Keywords

FPGA, Monte Carlo, SSTA.

## 1. INTRODUCTION

As the IC technology scales down to nanometer range, the manufacturing variations make Statistical Static Timing Analysis (SSTA) become an essential step in timing sign-off for nanoscale circuit designs. Monte Carlo based SSTA generates a large number of samples, and then uses traditional STA to evaluate each sample. This method avoids the path selection problem in path-based method and the inaccuracy of statistical max operator in block-based method. It provides the most accurate predictions of circuit delay distributions, and often serves as the golden standard to validate the accuracy of all the other methods. But due to the high computation time, it is seldom used in practice. The work in [5] tries to accelerate conventional Monte Carlo based SSTA using graphics processing units.

In this paper, we implement the Monte Carlo based SSTA using FPGA hardware. In the SSTA formulation, the output arrival time of each gate is computed as the maximum value among the sums

of each input pin's arrival time and the corresponding pin-to-output delay. This allows us to model the entire circuit as a data flow graph for static timing analysis, which consists of a large number of "max" and "sum" operations. The data flow graph can be configured in a pipelined fashion—we can ideally accept one STA sample evaluation in each clock cycle. However, this solution consumes a considerable amount of FPGA logic slices, and may not fit into a given FPGA. The challenge of our design is to fit the implementation into the limited FPGA area through extensive use of resource sharing.

In this paper, we leverage the recently proposed pattern matching technique [3] to identify the patterns in the mapped circuit netlist, and then use this information to share hardware resources among the same pattern instances. We further formulate the problem of allocating hardware resources for each pattern within the limited FPGA area bound as an optimization problem. By solving this optimization problem, we can obtain the maximum speedup and execution throughput (with the minimum pipeline initiation interval) on the limited FPGA area.

Our Monte Carlo based SSTA is currently implemented on one FPGA of the BEE3 hardware platform and it shows two orders of magnitude speedup compared to the CPU-based solutions. Note we can easily instantiate four SSTA instances on four FPGAs of the BEE3 hardware platform to obtain an additional 4X speedup.

## 2. MONTE CARLO BASED SSTA

### 2.1 Delay Model

Our delay model is simply the separate rise-fall delay model [7]. Each pin has its own pin phase, which is INV, NONINV, or UNKNOWN depending on whether the logic function is negative unate, positive unate, or binate in each input variable respectively [7]. If the phase of an input pin is INV, the output rising (falling) arrival time of the signal from the input pin is equal to the sum of the input falling (rising) arrival time and the corresponding rising (falling) pin-to-output delay. Similarly, if the phase of the input pin is NONINV, the output rising (falling) arrival time of the signal from the input pin is equal to the sum of the input rising (falling) arrival time and the corresponding rising (falling) pin-to-output delay. If the phase of the input pin is UNKNOWN, the output rising (falling) arrival time of the signal from the input pin is equal to the maximum value of the output rising (falling) arrival times between the INV case and the NONINV case.

For example, consider a NOR2 gate. Let  $AT_i^{rise}$  denote the arrival time of a rising signal at pin  $i$ ,  $AT_i^{fall}$  denote the arrival time of a falling signal at pin  $i$ ,  $Delay_i^{rise}$  is the rising delay from the input pin  $i$  to the output pin,  $Delay_i^{fall}$  is the falling delay from the input pin  $i$  to the output pin. Let the two input pins of the NOR2 gate be

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FPGA'10, February 21–23, 2010, Monterey, California, USA.  
Copyright 2010 ACM 978-1-60558-911-4/10/02...\$10.00.

$a$  and  $b$ , and the output pin be  $c$ . Since the logic function of the NOR2 gate is negative unate in the input pin  $a$  and  $b$ , the phases of input pin  $a$  and  $b$  are INV. Then, the rising (falling) arrival time at output pin  $c$  are:

$$AT_c^{rise} = MAX(AT_a^{fall} + Delay_a^{rise}, AT_b^{fall} + Delay_b^{rise}) \quad (1)$$

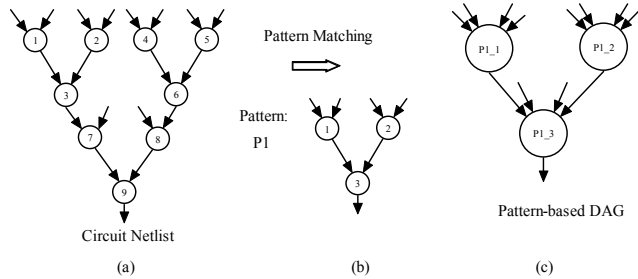
$$AT_c^{fall} = MAX(AT_a^{rise} + Delay_a^{fall}, AT_b^{rise} + Delay_b^{fall}) \quad (2)$$

## 2.2 Modeling of Delay Variations

In this paper, we assume the pin-to-output delay satisfies the Gaussian distribution. Each pin-to-output delay has its own variation. We implement the Gaussian random number generator based on the piecewise linear approximation method [8]. Currently we assume the Gaussian distributions of these pin-to-output delays are independent. In practice, these delays may have spatial correlations. We can modify the random number generator accordingly to accommodate the correlations.

## 3. OVERVIEW OF DESIGN FLOW

The input of the SSTA is a mapped circuit netlist, which is represented using a directed acyclic graph (DAG), where each node represents a logic cell. For example, a nine-node circuit is shown in Figure 1(a).

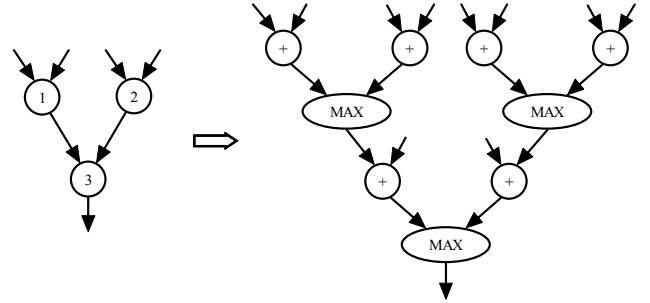


**Figure 1. Pattern matching: transform the circuit netlist into pattern-based DAG**

If the mapped circuit netlist is directly transformed into a data-flow graph, it will have excessive area consumption. To reduce the area, we use the technique of pattern matching to identify the patterns in the mapped circuit netlist. During the pattern matching process, we find out the patterns which can cover the entire circuit without overlapping. These patterns form a pattern-based DAG. It is important that the patterns do not create cycles since that would cause problems for delay propagation. An example of the pattern conversion is shown in Figure 1. Through pattern matching, we find a three-node pattern  $P1$  as shown in Figure 1(b), which has three pattern instances:  $P1\_1$ ,  $P1\_2$ ,  $P1\_3$ . These pattern instances form a pattern-based DAG as shown in Figure 1(c). Details about pattern matching will be discussed in Section 4.

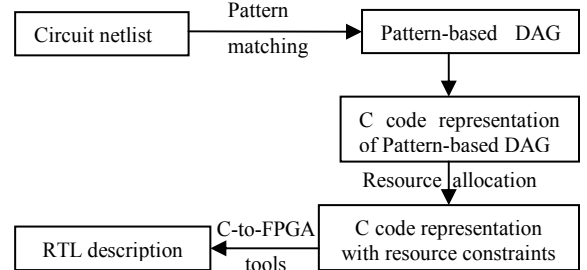
For the above pattern we find, it is a three-node circuit shown in Figure 2. From the delay model, we can find out that the operations used in STA are “max” and “sum”. This allows us to model the three-node circuit as a data flow graph. In the three-node circuit, each node has two inputs. Because the output arrival time of the signal from each input is the sum of the input arrival time and the corresponding pin-to-output delay, for the 2-input node, each pin corresponds to one “sum” operation in the data flow graph. The “max” operator is used to choose the larger output arrival time of the signals from the two input pins. In our example,

we assume that the pin phase is INV or NONINV. If the pin phase is UNKNOWN, then each pin corresponds to two “sum” operations and one “max” operation. Based on the method above, we can apply our delay model to the mapped circuit netlist and transform the mapped circuit netlist to data flow graph.



**Figure 2. Transform the circuit netlist to data flow graph**

In the pattern-based DAG, there is a set of patterns, and each pattern has a number of instances. Pattern provides the coarse-grain sharing opportunity to facilitate circuit reuse. Let us denote each pattern as a unique type of functional unit. We may allocate multiple instances of one type of functional unit. The optimization step tries to determine the exact number of instances of different functional units that we should allocate for all patterns. For the simple example shown in Figure 1, we can allocate one, two or three instances of the functional unit for pattern  $P1$ , and each of which will have different area/performance trade-offs. A mathematical programming formulation is used in Section 5 to address the tradeoff.



**Figure 3. Key steps in the design flow**

Figure 3 describes the key steps in the design flow. We leverage state-of-the-art C-to-FPGA tools to generate the RTL design. The input C code to C-to-FPGA tools is generated based on the pattern-based DAG and the data flow graph of each pattern. Note the resource allocation part generates some resource constraints for the C code, so that C-to-FPGA tools can take the input C code as well as the resource constraints to generate an efficient RTL implementation.

## 4. PATTERN MATCHING

As we said, a flat data flow graph representation of the netlist cannot fit easily. Due to resource limits of FPGA, the final performance of our approach is determined by the area optimization algorithm. We apply the recent pattern matching technique in [3] to find patterns in the circuit netlist. These patterns provide coarse-grain sharing opportunities for design-reuse and area reduction.

The pattern matching algorithm in [3] has two phases: pattern enumeration and pattern selection. Pattern enumeration process discovers a set of patterns based on the structure information of the circuit netlist. Pattern selection algorithm attempts to find a set of pattern instances, which can cover the entire pattern-based DAG and maximize the area reduction. A greedy algorithm is used for pattern selection. At each step, the best pattern is chosen based on an area-reduction metric, and all its pattern instances are removed from the pattern-based DAG. The whole process is repeated until the available patterns are empty. Note any available pattern that shares common nodes with patterns that have already been selected will be discarded, as we do not allow pattern overlap.

The area-reduction metric is shown below, which represents the reduced area by allocating one functional unit instance of pattern  $P$ :

$$gain(P) = area(P) * (\#inst(P) - 1) - area_{overhead}, \quad (3)$$

where  $gain(P)$  is the reduced area,  $\#inst(P)$  is the number of pattern instances, and  $area_{overhead}$  is the area overhead brought by newly introduced multiplexers when functional units are shared among pattern instances. The area overhead has a linear relation with the number of inputs of pattern  $P$ .

$$area_{overhead} = port(P) * \#inst(P) * mux_{1-input}, \quad (4)$$

where  $port(P)$  is the number of inputs of pattern  $P$ ,  $mux_{1-input}$  is the estimated area of 1-input multiplexer.

The greedy pattern selection process is efficient enough to find good pattern candidates.

## 5. RESOURCE SHARING & ALLOCATION

The allocation of the number of instances of all functional units plays a critical role in the tradeoff between performance and resource consumption. In this section, we discuss the problem of functional unit allocation in the context of pipelining with patterns. The goal is to obtain the highest possible throughput under a limited resource budget. The problem we discuss in this section can be described as follows.

Given: a target circuit which is covered by patterns  $\{P_1, P_2, \dots, P_N\}$ . For each pattern  $P_i$ , the amount of resource consumed by a corresponding functional unit is denoted as  $r_i$ . The number of instances for each pattern  $P_i$  is denoted as  $Inst_i$ . The total resource budget is denoted as  $Rbudget$ .

Output: the number of allocated functional unit instances for each pattern  $s_i$ , so that the total resource consumed by these patterns does not exceed  $Rbudget$ .

### 5.1 Performance Model

The performance of Monte Carlo based SSTA is mainly decided by initiation interval  $II$  (or equivalently, the throughput). In general,  $II$  is limited by two factors: recurrence and resource. In our implementation, one STA sample is computed in one iteration, and different iterations are completely independent, and thus only resource needs to be considered. In our implementation, each functional unit is pipelined, and the initiation interval of each functional unit is one. Thus, a lower bound of  $II$  considering the  $i^{\text{th}}$  type of functional unit is given as follows:

$$II \geq Inst_i / s_i, \quad (5)$$

### 5.2 Resource Model

The resources required by every allocated functional unit instance and the multiplexers are used to estimate the total resource usage.

$$R = \sum_{i=1}^N (s_i r_i + R_{i,mux}), \quad (6)$$

where  $R_{i,mux}$  is the resource required by the multiplexers for pattern  $P_i$ . On a typical FPGA, the resource required to implement a multiplexer grows at least linearly with total bitwidth of the inputs of the multiplexer. For pattern  $P_i$ ,  $Inst_i$  pattern instances share  $s_i$  corresponding functional unit instances. Since large multiplexers are undesirable, an optimized sharing solution is often balanced, i.e., the sizes of the multiplexers for different functional unit instances of pattern  $P_i$  should be almost equal. Thus, we use  $Inst_i / s_i$  as an estimate of the number of pattern instances of pattern  $P_i$  sharing one functional unit instance. Then, at each port of the functional unit instance, an  $n$ -to-1 multiplexer is needed, where  $n \leq Inst_i / s_i$ . Typically,  $n$  is smaller than  $Inst_i / s_i$ , as some operands from different operations may come from the same data source. Suppose the total bitwidth of the input ports of the functional unit for pattern  $P_i$  is  $b_i$ . We assume that the size of a  $b_i$ -bit  $n$ -to-1 multiplexer is proportional to  $n$  and  $b_i$ . Thus, the resource required by the multiplexers for pattern  $P_i$  is:

$$R_{i,mux} = s_i \times cb_i n, \quad (7)$$

where  $c$  is a constant. To estimate  $n$ , we introduce  $\alpha$ , so that  $n = \alpha Inst_i / s_i$ . Clearly,  $\alpha$  is between 0 and 1, and the value of  $\alpha$  depends on characteristics of the application and the synthesis tool. The updated resource usage is:

$$R = \sum_{i=1}^N (s_i r_i + s_i \times cb_i \alpha Inst_i / s_i) = \sum_{i=1}^N (s_i r_i + cb_i \alpha Inst_i) \quad (8)$$

### 5.3 Optimization

With performance and resource models discussed above, we can formulate the problem as a mathematical programming:

Minimize  $II$

$$s.t. \quad Inst_i / s_i - II \leq 0$$

$$\sum_{i=1}^N (s_i r_i + cb_i \alpha Inst_i) \leq Rbudget \quad (9)$$

$$s_i \geq 1, II \geq 1$$

In the above formulation,  $II$  and  $s_i$  are integer variables and all others are constants. The allocation problem formulated above has been extensively studied. It can be solved by using the incremental algorithm in [4]. For the incremental algorithm, each time we select a pattern and increase the number of allocated functional unit instances for the selected pattern by one so that we have the maximum efficiency (performance gain/area cost). Clearly, this is a polynomial algorithm.

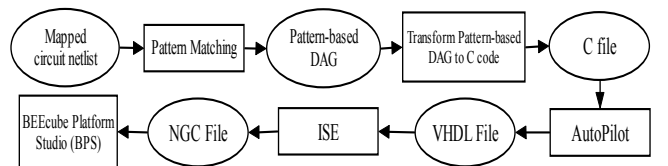


Figure 4. Experimental flow

## 6. EXPERIMENTAL RESULTS

### 6.1 Experiment Setup

Our experimental flow is shown in Figure 4. Before SSTA can be computed on a benchmark, the circuit is first converted into a mapped netlist by using ABC [6] and the standard cell library, mcnlib, from the SIS distribution [7]. We write a standalone program which parses the netlist, extracts the patterns and dumps out the C file description of the pattern-based DAG automatically. Then we use the state-of-the-art C-to-FPGA compilation tool AutoPilot [1], which can take C/C++/SystemC and some design hints/constraints as input, and generate synthesizable and optimized RTL.

We use the BEE3 hardware platform developed by BEEcube Inc. [2] in our actual implementation. BEE3 hardware platform has 4 Xilinx Virtex5 LX155T FPGAs. Currently, we only use one of the four. Note it is straightforward to run four SSTA instances on four FPGAs to obtain an additional 4X speedup. We use Xilinx ISE tool to generate the NGC file from the RTL. The NGC file is used to generate custom IP for the BEEcube Platform Studio.

### 6.2 Speedup Measurement

In order to evaluate the performance of our FPGA implementation, we implemented the same Monte Carlo based SSTA on a 2GHz Xeon CPU with 4 GB of memory running Linux. This implementation provides a reference point, in terms of performance and correctness, for our FPGA implementation.

The test circuits were chosen from ISCAS85, ISCAS89 and MCNC benchmark suits. Compared to our software implementation, our FPGA implementation provides a speedup ranging from 30X to 52X. The data is shown in Table 1.

Table 1 Speedup measurement

circuit	CPU through-put samples/sec	BEE3 per-chip through-put samples/sec	GPU per-chip through-put samples/sec [5]	BEE3 per-chip speed-up	GPU per-chip speed-up [5]
C432	4525	237918	412903	52.6X	91.2X
C499	5263	232727	184438	44.2X	40.8X
C880	4238	151658	252964	35.8X	58.4X
C1908	3786	142857	143497	37.8X	37.9X
C2670	1980	75188	80301	38.0X	40.6X
s832	4329	178571	214765	41.2X	49.6X
s1238	2457	92592	148148	37.6X	60.3X
s1196	2747	98039	164948	35.7X	60.0X
s1423	2544	111111	122841	43.7X	48.3X
elliptic	1434	43668	NA	30.4X	NA

### 6.3 Comparison with GPGPU implementation

Paper [5] presents an implementation of Monte Carlo based SSTA using Nvidia GPUs. Their implementation is up to 2X faster than our implementation (For circuit C499, FPGA implementation is faster). This is partly due to the fact that the sample-level parallelism is indeed a good fit for SIMD GPU architectures,

while we mainly uses the node-level parallelism of the circuit and also use loop pipelining techniques in our FPGA implementation. Note we used only one Virtex-5 LX155T FPGA. We expect we can have a larger speedup if we use multiple FPGAs or a larger FPGA e.g., LX330T. Moreover, the FPGA implementation is much more power efficient compared to GPGPU implementation.

## 7. CONCLUSIONS AND ONGOING WORK

In this paper, we accelerate the Monte Carlo based SSTA using FPGAs. We consider the entire circuit as a data flow graph. We explore the advantage of FPGA to pipeline the data flow graph. We apply the techniques of pattern matching and pattern recognition to fit the custom design into a given FPGA. We obtained two orders of magnitude speedup using a single FPGA of the BEE3 hardware platform.

Currently, our implementation is circuit-specific. That is, for each input netlist, we need to generate a different FPGA implementation for acceleration. In order to make our implementation independent on the input circuits, we are building a special-purpose processor with some elementary SSTA computing patterns implemented in FPGAs. Using this approach, we expect that we can handle resource sharing in a more unified and flexible way.

## 8. ACKNOWLEDGEMENTS

This work is partially funded by grants from Nvidia Corporation and Mentor Graphics Corporation under the UC Discovery program.

## 9. REFERENCES

- [1] *AutoESL Website*. <http://www.autoesl.com>
- [2] *BEEcube Website*. <http://www.beecube.com>
- [3] J. Cong and W. Jiang. Pattern-based behavior synthesis for FPGA resource reduction. In *Proc. International ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, pages 107-116, 2008.
- [4] D. -Z. Du and P. M. Pardalos, editors. *Handbook of Combinatorial Optimization*, Vol. 2, 1999.
- [5] K. Gulati and S.P. Khatri. Accelerating statistical static timing analysis using graphics processing units. In *Proc. Asia and South Pacific Design Automation Conference*, pages 260-265, 2009.
- [6] A. Mishchenko, S. Chatterjee, M. Ciesielski and R. Brayton. An integrated technology mapping environment. In *Proc. International Workshop on Logic and Synthesis*, pages 383-390, 2005.
- [7] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and S. A. Vincentelli. SIS: A system for sequential circuit synthesis, *UC Berkeley, Tech. Rep.*, 1992.
- [8] D.B. Thomas and W. Luk. Non-uniform random number generation through piecewise linear approximations. *IET Comput. Digit. Tech.* 1(4):312-321, 2007