GRID-BASED STATISTICAL TIMING ANALYSIS

Luís Guerra e Silva and L. Miguel Silveira
IST / INESC-ID
Lisboa, Portugal
{*lgs,lms*}*@algos.inesc-id.pt*

ABSTRACT

Timing analysis is concerned with modeling and computing the delay of digital integrated circuits (IC), prior to fabrication. The delay will determine the maximum operating frequency of an IC, and its computation is of paramount importance in electronic system design. As feature sizes decrease in IC technologies, the number of components that can be integrated on a single IC is very large. However, their precise behavior, including their delay, becomes increasingly sensitive to fabrication and operating variations. Standard deterministic techniques for delay analysis are quickly becoming inadequate, because they produce extremely conservative worst-case delay estimates. In this context, several statistical timing analysis algorithms have recently been proposed, based on analytical models. However, neither of them is able to correctly capture and model all the correlations and variations that influence the delays. Up to date, the only method that can produce accurate, yet approximate, delay estimates, taking into account fabrication and operational variations, is based on Monte Carlo simulation. In this paper we study the parallelization of such method, on a distributed infrastructure for parallel computation, based on the grid computing paradigm. Since this is currently the only accurate method available to perform statistical timing analysis of ICs, and given that ICs carry millions of logic blocks, the possibility of dealing with such large designs by using parallel computing is of tremendous practical interest.

KEYWORDS

Statistical timing analysis, Monte Carlo techniques, parallelization, grid computing

1. INTRODUCTION

Most complex integrated circuits (ICs), like microprocessors, microcontrollers, etc, are sequential circuits, consisting of registers interleaved by blocks of combinational logic, as illustrated in Figure 1. Each block of combinational logic implements a specific logic function, and itself consists of smaller blocks. At each clock cycle, new logic values are presented at the block's inputs and, after a delay Δ, the corresponding results are available at the block's outputs, to be latched on the next clock cycle. When the clock signal is activated, the registers read their input's logic values and set their outputs accordingly. If the period of the clock signal were smaller than the delay of any block of combinational logic, the values latched by the registers could be incorrect, and proper functioning would be in jeopardy. It is therefore of paramount importance that the largest delay Δ of any block be known since that is the deciding factor in determining the maximum operating clock frequency.

Obviously, the most accurate way of verifying the maximum clock frequency of a given circuit is to build it and test it in its target environment. However, this option is almost never used due to the time and expense it involves. One therefore resorts to methods that can determine as accurately as possible, before fabrication, whether the delay requirements are satisfied. Detailed transistor level simulation while able to generate highly accurate estimates, is beyond feasibility due to the problem size (Intel's Itanium 2 processor has 410 million transistors). To overcome the computational problem, researchers have come up with less accurate, but much more efficient algorithms.

As we have mentioned, each logic block is made of smaller simpler logic blocks that implement basic logic functionality (AND, OR, NAND, NOR, NOT, etc). The delays of these simple blocks (known as logic gates) are tabulated, and provided by IC foundries, for a given IC technology. By adding the delay values over all the paths, from a block's inputs to its outputs, it is possible to compute the delay of a block, in a linear-time

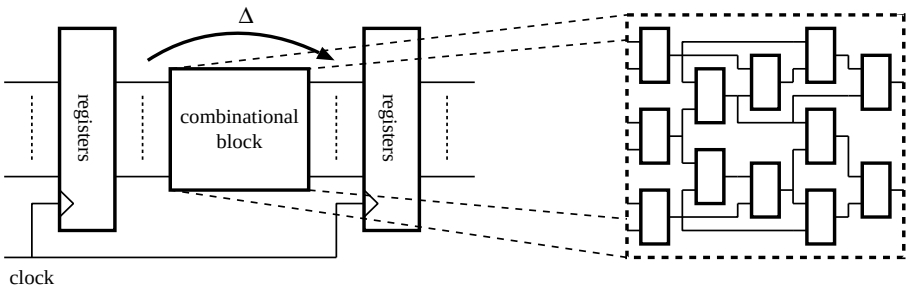

Figure 1: The architecture of a sequential circuit.

procedure. This group of algorithms, known as timing analysis algorithms, is particularly efficient, producing reasonably accurate delay estimates.

Over the last two decades timing analysis has been of great importance, and is at the heart of digital circuit design. Foundries provide average and worst-case nominal delay figures for every logic gate and, using that information, timing analyzers are able to compute estimates of average and worst-case delay for combinational blocks. By comparing the estimated worst-case delay value with the required clock period, the designer can assert if the circuit will function properly, prior to fabrication.

Even though these worst-case timing analysis techniques have been successfully employed for many years, they are quickly becoming inadequate. New generations of deep submicron IC technologies exhibit significant sensitivity to process and environmental variations, introducing a large variability in the timing behavior of logic blocks. As a result, such techniques targeting guaranteed design functionality produce extremely pessimistic delay estimates, and are unable to provide quantitative information, such as yield estimation, necessary to effectively guide design optimization.

To overcome these limitations, several statistical timing analysis (STA) algorithms have recently been proposed [3, 1]. By computing delay distributions instead of single worst-case values, these algorithms are a first attempt to correctly handle the impact of process and environmental uncertainties in newest deep submicron IC technologies, simultaneously providing the necessary quantitative information. In this context, several statistical timing analysis algorithms have recently been proposed, based on analytical models. However, neither of them is able to correctly capture and model all the variations that influence the delays. As an alternative, Monte Carlo-based methods are know to produce accurate, yet approximate, delay estimates, taking into account fabrication and operational variations. In this paper we study the parallelization of such method, on a distributed infrastructure for parallel computation, based on the grid computing paradigm. Since this is currently the only accurate method available to perform statistical timing analysis of ICs, and given that ICs carry millions of logic blocks, the possibility of dealing with such large designs by using parallel computing is of tremendous practical interest.

This paper is organized as follows. In the next section we describe the timing analysis problem, and provide some context and background information necessary to understand the remainder of the paper. In the following section, the statistical approach to the timing analysis problem is introduced. A Monte Carlo-based algorithm is presented, and implementation details are explained. In the next section, we discuss a parallel implementation of the same algorithm. In order to validate the proposed approaches, the next section describes experimental results, for the parallel execution of the statistical timing analysis algorithm, for 16 benchmark circuits. Finally, in the last section we present some conclusions and topics for future research work.

2. TIMING ANALYSIS

In this section we introduce the problem of timing analysis of combinational circuits. We start by defining a combinational circuit and related concepts. Next, we describe the modeling framework used in the characterization of the timing behavior of a combinational block. Finally, we state the timing analysis problem and describe a simple intuitive analysis method.

2.1 Combinational Circuit

A *combinational circuit* or *combinational block* is an aggregate of combinational logic that implements a given logic function. The logic function receives its arguments from the block inputs and produces its results in the block outputs. Internally, a combinational block is made of simpler combinational blocks, each of which implements a given logic functionality. The overall logic function of a combinational block is implemented by appropriately connecting the inputs and outputs of these simple blocks.

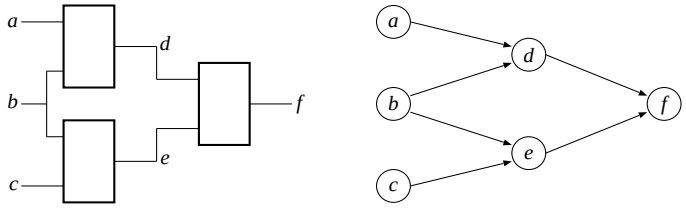

Figure 2: Block and graph representation of a combinational circuit.

In following we will represent a combinational circuit or block as an annotated directed acyclic graph, $C = (V,E)$, referred to as the *circuit graph*, composed of simple combinational blocks and primary inputs, represented by nodes (vertices) in the graph, and *connections*, represented by edges between them. V and E denote respectively, the set of nodes and the set of edges. The *primary inputs* are nodes with no incoming edges. All the nodes with no outgoing edges are *primary outputs* (note that there may be some primary output nodes with outgoing edges). A *path* is an alternating sequence of nodes and edges, connecting any two nodes. A *complete path* connects a primary input to a primary output.

2.2 Timing Characterization

In simple terms, the timing analysis of a combinational block, consists in determining the time instants at which the logic values set at the primary inputs of a combinational block, will propagate to its internal nodes and finally produce the desired result in its primary outputs. In the following, we will describe the framework employed in modeling this process.

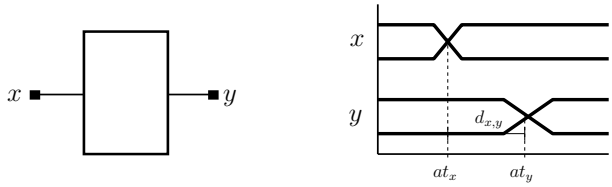

Figure 3: Timing characterization of a combinational block.

Let us consider a simple block, with an input node *x* and an output node *y*, illustrated in Figure 3. The most important property that characterizes the temporal behavior of this block is its *delay*, represented by $d_{x,y}$ (meaning the delay between nodes *x* and *y*). This measures the length of the time interval that elapses since a logic value is set at input node *x*, until the corresponding block response is observed at the output node *y*. The instants at which results arrive to a certain node are designated by *arrival times*. The arrival times for *x* and *y* are represented by at_x and at_y, respectively.

2.3 Problem Statement

Let us consider the combinational block presented in Figure 4, with primary inputs *a*, *b*, *c*, *d*, *e* and *f*, and primary outputs *y* and *w*. Assuming that new logic values are set at the primary inputs at time instant 0, the timing analysis problem is concerned with computing the time instants (arrival times) at which the corresponding logic results are produced at either primary output.

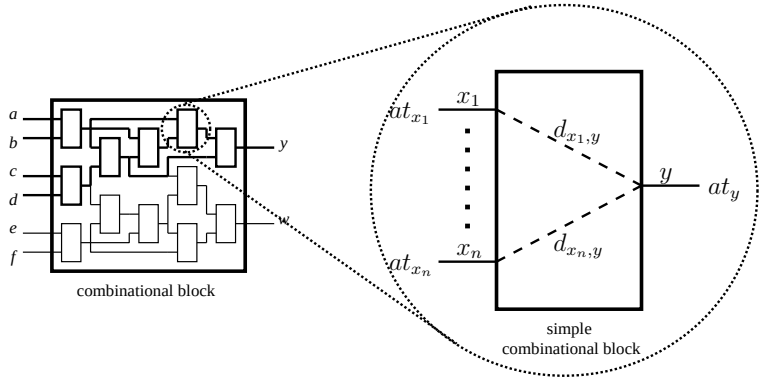

Figure 4: Arrival time computation at the outputs of a combinational block and simple block.

As mentioned before, combinational blocks are themselves made of simpler combinational blocks that implement basic logic functions. In Figure 4 all the simple blocks that belong to any path that ends in primary output *y*, and their connections, are represented in strong line. In order to compute the arrival time at node *y* we must progressively compute the arrival times at the output nodes of every simple block, starting from the primary inputs. Therefore, for every *n*-input simple combinational block, as the one represented in Figure 4, with inputs $x_1, x_2, ..., x_n$ and one output node *y*, given the values of the arrival times at each of these nodes $at_{x_1}, at_{x_2}, ..., at_{x_n}$ and at_y, and the values of the simple block delays between nodes $x_1, x_2, ..., x_n$ and *y*, we will be concerned with computing, for each simple block the arrival time at its output, given by,

$$at_y = max(sum(at_{x_1}, d_{x_1,y}), sum(at_{x_2}, d_{x_2,y}), ..., sum(at_{x_n}, d_{x_n,y})) \quad (1)$$

By analyzing this expression becomes immediately clear that two operations must be performed: *sum* and *max*. Clearly, if the delays and arrival times are real numbers this timing analysis procedure can be performed in linear time, by computing arrival times at simple block outpus in a levelized breadth-first traversal of the circuit graph.

3. STATISTICAL ANALYSIS

In this section we present a statistical version of the deterministic timing analysis problem presented in the previous section. We start by shortly describing the sources of variation in the performance of newest IC technologies, that motivate the use of a statistical approach to the timing analysis problem. Further, we explain how statistical correlations are induced on problem variables (delays and arrival times). Next, we state the statistical timing analysis problem. Further, we propose a Monte Carlo-based statistical timing analysis algorithm, subsequently describing some of its implementation issues.

3.1 Sources of Variation and Correlation

No two ICs are equal. The fabrication of an IC is a sequence of hundreds of operations, each of which will be performed a little differently on each IC. *Manufacturing process variations* are one of the main sources of variability in the performance of ICs. An example is the fact that frequently one IC is significantly faster than another. Another source of variation are the *environmental operating conditions* of the IC, such as changes in power supply voltage or operating temperature. Finally, a more subtle source of variation are *device fatigue phenomena*. Examples are electromigration, hot electron effects and NBTI (negative bias temperature instability).

The complexity of the formulation of the STA problem is mainly due to the statistical correlations between problem variables. Both block delays and arrival times can exhibit correlation. Block delay are correlated because, in general, they depend on the same sources of variation. Arrival times are also correlated, because

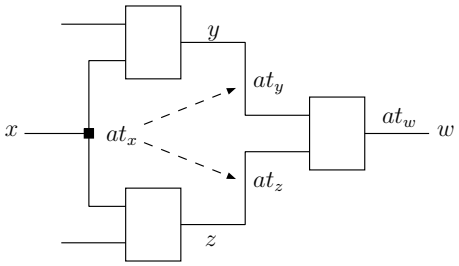

Figure 5: Arrival time correlations due to reconvergent fanouts.

they are computed from block delays which, as we explained, are themselves correlated. Additionally, arrival times can exhibit correlation due to the existence of reconvergent fanouts, which are common in any optimized circuit. As illustrated in Figure 5, the existence of reconvergent fanouts, like from node *x* to *w*, induces correlation between arrival times in the corresponding fanout cone, as happens with at_y and at_z. Since they are correlated and therefore not statistically independent, that increases the complexity of combining at_y and at_z to compute at_w. An ideal STA algorithm should be able to efficiently and accurately handle all the types of correlation mentioned.

3.2 Problem Statement

The problem of STA is very similar to the problem of simple, corner-based (average or worst-case), deterministic timing analysis. In both cases, it consists in computing a sequence of *sum* and *max* operations, in a traversal over the circuit graph. However, their implementation is quite different, because the data they handle is also different. A deterministic timing analyzer manipulates nominal values, while a statistical timing analyzer manipulates distributions.

Given an *n*-input block, with input nodes $x_1, x_2, ..., x_n$, and one output node *y*, given the random variables that represent the arrival times at each of these nodes $\mathbf{at}_{x_1}, \mathbf{at}_{x_2}, ..., \mathbf{at}_{x_n}$ and $\mathbf{at}_y$, and the random variables that represent the block delays between them $\mathbf{d}_{x_1,y}, \mathbf{d}_{x_2,y}, ..., \mathbf{d}_{x_n,y}$, we will be concerned in computing, for every such circuit block

$$\mathbf{at}_y = max(sum(\mathbf{at}_{x_1}, \mathbf{d}_{x_1,y}), sum(\mathbf{at}_{x_2}, \mathbf{d}_{x_2,y}), ..., sum(\mathbf{at}_{x_n}, \mathbf{d}_{x_n,y})) \quad (2)$$

If we consider that all the random variables are statistically independent, these operations are simple. However, in the real world this is not true, and therefore the accurate implementation of this operations, accounting for all variable correlations, is quite complex, and still an open topic of intense research.

3.3 Monte Carlo Stochastic Approach

The Monte Carlo method, described in [2], is used in many science and engineering fields for approximately solving problems by the simulation of random quantities. In a Monte Carlo analysis, a series of deterministic simulation runs (trials) are performed, replacing in each run the value of every stochastic input variable by a randomly generated number, according to its predefined distribution. The final result will be the set of samples generated in each trial, for each problem output variable, from which the desired problem solution can be estimated. This method is computationally extremely expensive and only provides an approximate result, even though it can be very accurate, given a large enough number of samples (trials). Therefore, it is mostly used when no analytical solution is easy or even possible to obtain or implement. The error associated with the estimated quantity is, as a rule, inversely proportional to the square root of the number of trials, meaning that a large number of iterations is usually necessary in order to obtain statistically significante results.

In the context of STA, in each trial of the Monte Carlo simulation we will perform a deterministic timing analysis run, as described ealier, but where block delays are replaced by randomly generated values, that obey a known block delay distribution. As a result, for each circuit node, we will obtain a set of arrival time samples, each of which was generated in a different trial. Given a sequence of *n* arrival time samples for a given circuit

node, $\xi_1, \xi_2, ..., \xi_n$, computed on n trials of a Monte Carlo simulation, then its *sample mean*, $\overline{\mu}$, and *sample variance*, $\overline{\sigma}^2$, is given by,

$$\overline{\mu} = \frac{1}{n} \sum_{i=1}^{n} \xi_i \tag{3}$$

$$\overline{\sigma}^2 = \frac{1}{n} \sum_{i=1}^{n} \xi_i^2 - \overline{\mu}^2 \tag{4}$$

3.4 Implementation Issues

The description of the circuit conectivity (e.g. the way the various simple blocks are connected within a combinational block – the circuit) is stored in an OPENACCESS database. This open source framework provides a database format for storing IC designs, which is acessable through a C++ API. The mean and variance of simple block delay distributions is also stored in the database, and acessable through the same API.

In a preliminary stage, the analyzer generates a C source file with a function named `circuit()`, for the circuit under analysis. This function encodes the various tasks that are necessary to perform on each STA Monte Carlo trial. First, for every primary input node, it calls a `setup_input()` function that receives as argument the input node index, and sets the corresponding arrival time sample to 0. Afterwards, it calls a `process_block()` function, for each simple block found, which computes the arrival time at the block's output node, given the arrival times at its input nodes and a randomly generated block delay value which is computed on-the-fly. The `process_block()` function calls, for the various blocks, are inserted in the C file in a order that corresponds to a forward levelized breadth-first traversal of the circuit graph. This specific order assures that when the function is called for a given simple block, all the necessary arrival times of the block's input nodes have already been computed. In run-time, the `setup_input()` and `process_block()` functions will store and retrieve the computed arrival times from the `at` array. This array consists in a sequence of `double` values, that are the arrival times for every circuit node.

A second stage follows, when the generated C source file is compiled and linked with an object file that contains the implementation of all the previously mentioned functions, including the execution entry point function `main()`. Finally, the generated file is executed. During the execution, the `main()` function iteratively calls the generated `circuit()` function, for the specified number of Monte Carlo trials. Finally, the sample mean and variance of the delay distributions at every node are computed and presented to the user.

Instead of generating a program that performs the Monte Carlo simulations of each circuit, we could have simply performed them by continously traversing the circuit graph obtained from the design database. Even though this option has a smaller overhead, the run time of each trial is much larger. Since in general many trials need to be performed in order to obtain statistically significant results, the initial overhead is widely compensated by the better performance obtained in each trial.

At first glance, it may seem that in order to compute the sample mean and variance of the arrival time of every node it would be necessary to store all the samples computed in each trial. However, that is not true. After the execution of a single trial, all the samples can be destroyed if, for every node, we maintain two parameters, between Monte Carlo trials. For a given node x, we will designate them by ω_x and λ_x. They will both be initialized at 0, before any Monte Carlo trials are performed. When, on a given trial i, the value of the arrival time sample at node x, ξ_i, is computed, then ω_x and λ_x should be updated as follows,

$$\omega_x = \omega_x + \xi_i \tag{5}$$

$$\lambda_x = \lambda_x + \xi_i^2 \tag{6}$$

After n Monte Carlo trials, that produced a sequence of arrival time samples for a given node x, $\xi_1, \xi_2, ..., \xi_n$, the parameters ω_x and λ_x are given by,

$$\omega_x = \sum_{i=1}^{n} \xi_i \tag{7}$$

$$\lambda_x = \sum_{i=1}^{n} \xi_i^2 \tag{8}$$

From this two parameters, the sample mean and variance at a given node x can easily be computed at the end, as can be observed by the formulae below.

$$\overline{\mu}_x = \frac{1}{n}\sum_{i=1}^{n}\xi_i = \frac{\omega_x}{n} \tag{9}$$

$$\overline{\sigma}_x^2 = \frac{1}{n}\sum_{i=1}^{n}\xi_i^2 - \overline{\mu}_x^2 = \frac{\lambda_x}{n} - \left(\frac{\omega_x}{n}\right)^2 \tag{10}$$

By storing between trials only two arrays, containing ω and λ parameters for every node, instead of all the arrival time samples generated at each trial, an enormous memory optimization is achieved.

4. EXPLOITING PARALLELISM

In this section we discuss the parallelization of the Monte Carlo-based statistical timing analysis algorithm. We start by presenting the computing network which we used to conduct our experiments. Precise definitions of what is meant by speedup and efficiency of a parallel computing environment are given. Finally we discuss how to break up the problem into the various processing units.

4.1 Distributed Computing Environment

In this work, the distributed computing environment was a Grid of computers consisting of five machines interconnected in a grid-like fashion (`www.gridforum.org`). The Grid refers to an infrastructure that enables the integrated and collaborative use of high-end computers, networks, databases, and scientific instruments owned and managed by multiple organizations.

In essence, this is a loosely-coupled distributed memory machine, which uses standard Ethernet protocols for communication. Grid computing has emerged recently as an interesting new paradigms for large scale independent computations. The five machines on our Grid consist on two Intel Pentium IV at 2.60 GHz, two Intel Pentium IV at 2.40 GHz, and a dual-processor Intel Xeon at 2.40 GHz. The first four have 1GB of RAM and the last has 4GB of RAM. All the machines are running GNU/Linux. The Grid is setup using the Globus toolkit from the Globus Alliance (`www.globus.org`)

The underlying communication interface was MPI, the Message Passing Interface standard. MPI standard is the *de facto* industry standard for parallel applications. It was designed by leading industry and academic researchers, and builds upon two decades of parallel programming experience.

For the computational Grid, we chose an implementation of MPI named MPICH-G2 which is a Grid-enabled implementation of the MPI v1.1 standard. That is, using services from the Globus Toolkit (e.g., job startup and security), MPICH-G2 allows one to couple multiple machines, potentially of different architectures, to run MPI applications. MPICH-G2 automatically converts data in messages sent between machines of different architectures and supports multi-protocol communication by automatically selecting TCP for inter-machine messaging and (where available) vendor-supplied MPI for intra-machine messaging.

4.2 Parallel Computing Performance Metrics

When considering the performance of a parallelized algorithm, one must compare it with a serial, single-processor implementation. To that end, if we denote by $T(n,p)$ the time to solve a problem with n unknowns on a parallel environment using p processors, then we can define the parallelization *speedup* as

$$S(n,p) = \frac{T(n,1)}{T(n,p)} \tag{11}$$

Perfect utilization of resources is obtained when $S(n,p) = p$. Another relevant definition, containing similar information, is the measure of *efficiency*, that can be computed as,

$$E(n,p) = \frac{T(n,1)}{p \times T(n,p)} \tag{12}$$

Perfect utilization of resources is obtained when $E(n,p) = 1$. Clearly, for effective utilization of resources, the original code must be scalable.

4.3 Parallelizing Statistical Timing Analysis

The Grid infrastructure supplies each instance with two parameters: the total number of processors p and the specific instance processor identifier id. In a first step, each instance, knowing if it is the master ($id = 0$) or one of the slaves ($id \neq 0$), computes the number of iterations to perform. Given the total number of required iterations, N, and given the number of processors available p, the number of iterations performed by the master and slave machines is given by,

$$n_{master} = N/p + N\%p \tag{13}$$

$$n_{slave} = N/p \tag{14}$$

where $/$ is the integer division and % is the modulus. Notice that if the total number of iterations is not multiple of the number of processors, then the master machine will perform the extra $N\%p$ iterations necessary to achieve the total number required. Also using the instance processor identifier, id, each instance generates a different seed for the random number generator, in order to prevent undesired correlations between samples produces in different instances.

Next, each instance performs the number of Monte Carlo iterations previously computed. As in the non-parallel version, the values of ω and λ are stored in each instance, for each circuit node. After completing the simulation each slave instance sends their ω and λ arrays to the master, which receives them and adds the value of each array position to the values in the corresponding position of its own ω and λ arrays. When all slaves have sent their ω and λ arrays, the master computes the sample mean and variance for every node, printing them on the output.

In this implementation, the communication cost between each slave and the master is the cost of sending the ω and λ arrays, which corresponds to the cost of sending one `double` multiplied by twice the number of nodes.

5. EXPERIMENTAL RESULTS

In this section we present experimental results for the standalone and parallel grid-based versions of the statistical timing analysis algorithm previously described, performing 20000 Monte Carlo trials. In the parallel version, all the five available processor were used. The first 10 circuits belong to the well know ISCAS'85 benchmark suite. The remaining 6 circuit are carry-skip adders of various sizes. The first columns of the table exhibit relevant information about each circuit, such the number of simple block it contains, the number of nodes (internal + primary inputs + primary outputs), the number of primary inputs and the number of primary outputs. The following columns present CPU time in seconds and memory in MB used in the standalone and parallel grid-based versions of the statistical timing analyzer. For the grid version, the CPU time and memory reported are the worst values among all the machine that integrate the grid. The last column shows the speedup achieved with the parallelization.

As expected, both CPU time and memory increase with the number of blocks. Given the amount of information processed, the memory seems to be kept very low. When comparing the memory usage between the standalone and grid versions, it is noticeable that the grid infrastucture introduces an overhead of about 4MB, which is harmless, since it seems to be unrelated with the problem size (number of blocks). In what concerns to CPU time, the grid version effectively parallelizes the workload, achieving in certain cases an almost perfect utilization of resources (speedup near 5).

In Figure 6, we present the variation of the sample mean and variance with the number of iterations, for a node of the c432 benchmark circuit. As can be observed, both values only seem to achieve some convergence after 10000 trials.

Benchmark	*#Blocks*	*#Nodes*	*#PIs*	*#POs*	Standalone		Grid		Speedup
					CPU	MEM	CPU	MEM	
c432	171	207	36	7	1.46	1.52	0.41	4.70	3.56
c499	218	259	41	32	1.82	1.53	0.41	4.71	4.44
c880	383	443	60	26	3.17	1.59	0.72	4.76	4.40
c1355	562	603	41	32	4.33	1.64	1.05	4.81	4.12
c1908	972	1005	33	25	7.15	1.73	1.48	5.02	4.83
c2670	1211	1445	233	140	10.05	2.83	4.11	5.11	3.55
c3540	1705	1755	50	22	13.36	1.95	2.76	5.21	4.84
c5315	2351	2529	178	123	19.61	2.15	4.04	5.44	4.85
c6288	2416	2448	32	32	20.59	2.28	4.33	5.57	4.76
c7552	3624	3832	207	108	30.19	2.52	6.19	5.75	4.88
csa.32.4	200	265	65	33	1.88	1.53	0.41	4.71	4.59
csa.64.4	400	529	129	65	3.75	1.61	0.80	4.78	4.69
csa.128.4	800	1057	257	129	7.54	1.77	1.77	5.04	4.25
csa.256.4	1600	2113	513	257	16.20	2.07	3.51	5.33	4.61
csa.1024.4	6400	8449	2049	1025	68.03	3.88	14.65	7.33	4.64
csa.8192.4	51200	67585	16385	8193	561.40	20.92	124.94	25.09	4.49

Table 1: Table of performance results for ISCAS'85 benchmarks and carry-skip adders.

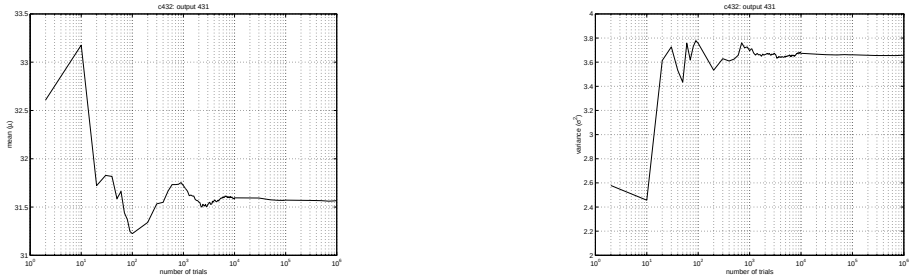

Figure 6: Variation of sample mean and variance with the number of trials.

6. Conclusions

This paper presents a parallel version of a Monte Carlo-based statistical timing analysis algorithm. As expected for Monte Carlo-based methods, the experimental results show that the parallelization is highly effective, leading in certain cases to an almost perfect utilization of the computing resources. Clearly, a much larger grid could be used in performing statistical timing analysis of present microprocessors, that have millions of blocks.

REFERENCES

[1] A. B. Agarwal, D. Blaauw, V. Zolotov, and S. Vrudhula. Computation and Refinement of Statistical Bounds on Circuit Delay. In *Proceedings of the ACM/IEEE Design Automation Conference*, Anaheim, CA, June 2003.

[2] Nicholas Metropolis and Stanislav Ulam. The Monte Carlo Method. *Journal of the American Statistical Association*, 247(44):335–341, September 1949.

[3] C. Visweswariah, K. Ravindran, K. Kalafala, S. G. Walker, and S. Narayan. First-order incremental block-based statistical timing analysis. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 331–336, San Diego, CA, June 2004.