2013 IEEE Congress on Evolutionary Computation
June 20-23, Cancún, México

A Fast Modular Simulator for Combinational Logic Circuits generated by Genetic Algorithm

Daniel M. Cabrita
Ministerio Publico do
Estado do Parana (MP-PR)
Curitiba, Brazil
Email: dancab@gmx.net

Walter Godoy Jr.
Federal University of
Technology - Parana (UTFPR)
Curitiba, Brazil
Email: godoy@utfpr.edu.br

Heitor S. Lopes
Federal University of
Technology - Parana (UTFPR)
Curitiba, Brazil
Email: hslopes@utfpr.edu.br

***Abstract*—This paper presents a structure for a fast generic simulator for combinational logic circuits, intended for use with genetic algorithm (GA) multi-objective optimization. A modular structure is proposed containing a chromosome translator layer, so to isolate the simulator from the idiosyncrasies of the chosen genetic codification. The simulator archieved high performance, considering its software-only implementation, and it was successfully used to generate valid circuits while integrated into a GA.**

I. Introduction

A multi-objective combinational circuit optimization using genetic algorithms (GA) requires a way for simulating candidate circuits, in order to determine the fitness of each individual.

The conventional, and widely used approach, consists of the simulation of the circuit for each possible input state, what increases the number of required simulations exponentially. For example: for a 16 bit input, $2^{16} = 65536$ simulations are required for a single individual, from a population which can be between 100-2000 individuals, what would translate into millions of circuit simulations for each generation. Slowik[1] wrote about this growth of combinations and the corresponding processing time concerns, while citing alternative approaches.

In order to address simulation performance concerns, a simulator specific to the particular GA approach could be designed. Such solution carries the inconvenience of loss of chromosome flexibility. That limitation is detrimental for research purposes, where structural chromosome variations is a possibility, since it makes necessary the reimplementation of the simulator itself and the accompanying GA interface to it.

An alternative is to design a complete generic simulator for logic circuits. While that would satisfy the concerns on simulator reusability, that would suffer from performance penalty and increased GA interfacing complexity.

In this work we present a structure for a combinational logic circuit simulator which addresses both concerns of performance and generalism. The generic nature of this simulator is within the scope of combinational circuits to be generated by GA. The simulator is as generic as necessary for current GA works for generation of combinational circuits, while providing a convenient interface to the GA, and able to provide fast performance.

During research, it was identified a basic common gate array structure presented by a number of papers, with two main general groups: the first uses a matrix structure[2] and the other is vector-based[3].

A vector-based structure was chosen after considering that:

- The matrix approach is convenient for simulators implemented using field-programmable gate array (FPGA), but imposes structural limits which are detrimental to the purposes of a generic simulator. Also, it provides no benefit for simulations performed by general-purpose microprocessors, which natively address data as vectors.
- A matrix structure circuit can be easily translated into a vector prior the simulation. The statistical concerns related to GA convergence, pointed by Vassilev[4], related to the vector addressing structure do not apply, since the GA itself may still be based on a matrix structure regardless the internal vector representation used by the simulator.
- A vector-based circuit structure cannot be trivially translated into a matrix for simulation purposes.

The continuing use of matrix structure topology[5] variants suggests the adequacy of the generalization level chosen for our simulator.

In order to isolate the simulator from the specificities of the GA itself, rigid separation layers were defined: fitness function, chromossome structure and other. As part of this structure, a chromosome translation layer was defined, in order to provide isolation between the simulator internal circuit representation and the circuit representation used by the GA.

II. The Modular Structure

In order to provide a generic, reusable, environment for the circuit simulator, a general modular structure is defined where the GA is divided into separate functional blocks. The modular structure is shown in Fig. 1.

The genetic algorithm section contains the data and software specific to the optimization problem, including GA operators, the population and the fitness function (FF).

The circuit simulation section contains the generic circuit simulator (GCS) itself, and a chromosome translation layer

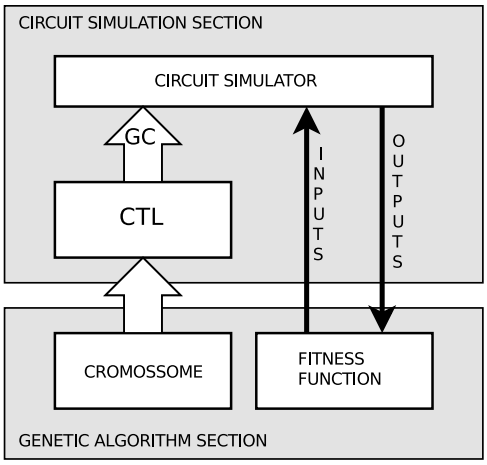

Fig. 1. The Modular Structure where the Circuit Simulator is located

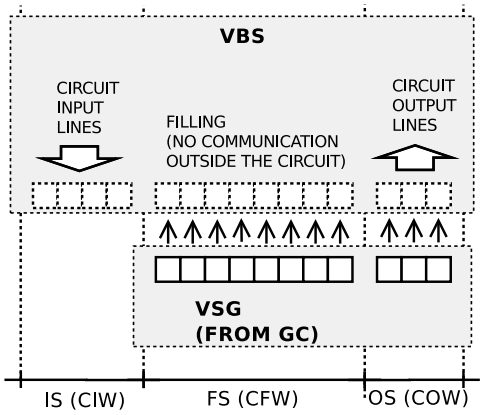

Fig. 2. Simulator - Genes and States

(CTL). The CTL function is to convert an individual's chromosome into the generic chromosome (GC) used by the GCS.

The CTL needs to be invoked each time the simulated circuit changes, typically once for each new fitness evaluation. After this procedure, the circuit may be simulated as many times as needed. Since the CTL needs to be called only once per new simulated circuit, or once per new FF instance, the performance penalty it causes is minimal.

During simulation runs, the FF invokes the GCS, providing the input bits, and collecting the returned output bits. The way the FF processes or stores the collected outputs does not affect the GCS, since the FF acts as merely a client to the GCS. While the circuit is being simulated, the GCS is a black box to the FF.

III. The Generic Circuit Simulator

The GCS requires certain fixed parameters defined prior to the GA run, which are the circuit input width (CIW), circuit filling width (CFW) and circuit output width (COW). Each of those reflect the width of both its corresponding bit states and generic chromosome widths, shown in Fig. 2.

The CIW is the total number of bits in the circuit inputs, while the COW is the analogue to the circuit output bits. The CFW is manually chosen and defines the number of extra gates present in the chromosome, so to allow the number of gates required for the GA solution for a given combinational logic. The CFW may be 0.

A. The Basic Structure of the Generic Chromosome

The GC is provided by the CTL to the circuit simulator, and it is a superset of the chromosome used by the GA.

The GC is a vector of genes, each gene correponding to a single gate simulated, which contains the gate type and the mappings of each of its input ports. The input/ouput source/destination of each gate is explained later.

Considering the value of CFW as N_{CFW} and of COW as N_{COW}, the constant number of gates simulated by the GCW, N_{total}, is expressed as $N_{total} = N_{CFW} + N_{COW}$, where $N_{CFW} \geq 0$ and $N_{COW} \geq 1$.

The value of CIW, N_{CIW}, must also obey a constraint for a valid circuit: $N_{CIW} \geq 1$.

B. Genes and Bit States

The GCS operates with two groups of data: the vector of simulated gates (VSG), received as a translated chromosome from the CTL, and its internal vector of bit states (VBS), as shown in Fig. 2. Both groups of data are divided into three sections: Input Section (IS), Filling Section (FS) and Output Section (OS).

The IS does not contain any simulated gate, it is composed of a section of the VBS representing the circuit inputs provided. These input states are loaded on each circuit simulator invocation.

Both FS and the OS hold gates and the bit states, each of those states correspond to the output of one corresponding gate. The FS and the OS differ from each other in the way the bits states are treated once a single simulation is completed: the OS states are mapped into the circuit outputs, while the FS ones are discarded after the simulation ends.

C. Per-Gate Simulation

Each gene from the translated chromosone maps directly to a single simulated gate. As shown in Fig. 3, each gate carries two groups of information: 1. gate input mappings and 2. gate type, later explained.

The circuit simulation is done by simulating one single gate at a time, in a sequential fashion from the first to the last one. Each gate collects its input states from the a single bit from the VBS, according to its input mappings to the VBS.

The gate is then simulated according to its type, in a process explained later, and its output is sent to VBS bit corresponding to that specific gate. This process is repeated until the last gate is simulated.

For each simulated gate in the n^{th} position there is a limited set of total valid bit inputs T it may safely use: $T = n - 1$. The remaining part of the VBS is located in the region of pending simulation, with its bits holding no meaningful information. For performance reasons, the simulator itself will not perform any type of validity checking, leaving the GA itself to provide valid input ports' mappings for each gate.

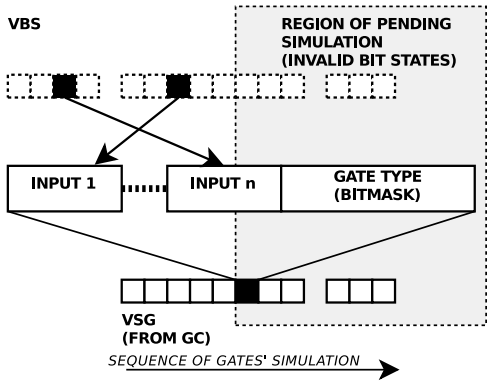

Fig. 3. Simulator - Per-gate simulation

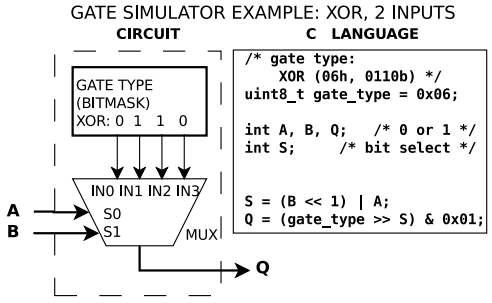

Fig. 4. Simulator - Simulation of a XOR gate, 2 inputs

D. Gate Type Simulation

The gate simulator mimics the behavior of a given logic operation, such as AND, OR, XOR or other. Each gate has a fixed number of input ports, and all gates within the simulated circuit must have this same number of inputs. Considering the number of per-gate input ports p, the port type bitmask (Fig. 4) requires w bits and the total number of gate types are s, defined as: $w = 2^p$ and $s = 2^w$, thus $s = 2^{2^p}$. Among those s gate types there are unstandard - yet functional - ones, including gates that ignore one or more of its inputs.

This general form of individual gate simulation, though with different approaches, was proposed before with a FPGA look-up table (LUT)[6], and even earlier with a multiplexer-based scheme[7].

If a pre-translation circuit to be simulated has ports with variable number of inputs, such as 2-input AND and 1-input NOT, the number of ports to be simulated p must be according to the gate with the highest input port count. In such cases the CTL must map gate types into ones that properly simulate gates with less inputs than p, rendering the excess ports inactive.

Coello et al[8] pointed out no good results of using a binary mask representation for gate types, what it used by this simulator, instead of an alphabet-based one. Regardless the merits of either encoding variant, such concerns do not apply to this simulator, since the CTL allows mapping from alphabet-based gate types to binary masks.

The gate simulator consists of a multiplexer (MUX) and uses no lookup tables nor has any knowledge of logic operations. Its behavior is exemplified in Fig. 4, where the simulation of a 2-input XOR port is shown, in both circuit form and C language notation.

E. Circuit Inputs and Outputs

As previously explained, the circuit inputs are mapped directly into the first corresponding bits of the VBS, in its IS.

Once all gates have been simulated, the circuit output is already present in the last COW bits of the VBS, and reflect the bit states located in the OS (Fig. 2).

IV. Simulator Performance Tests

A. Implementation Details

A circuit simulator based on the system described above was implemented in C programming language, and its correctness was verified prior to the performance tests.

A batch of performance tests was performed with varying circuit sizes and parameters. The processing done in the tests did not include GA processing, since it would interfere in the performance statistics.

The performance metrics was defined as the number of complete circuit simulations for a single input value, per second. Assuming a GA with no fitness/chromosome caching, where all possible binary input combinations need to be evaluated, considering the number of CIW as N_{CIW} and the value of single simulations per second as S_{perf}; the simulation performance for a single individual I_{perf} and for a single generation G_{perf} with a population of P_{total} can be calculated as shown in Eq. 1 and Eq. 2.

$$I_{perf} = \frac{S_{perf}}{(2)^{N_{CIW}}} \tag{1}$$

$$G_{perf} = \frac{I_{perf}}{P_{total}} \tag{2}$$

The tests shown in Fig. 5 and Fig. 6 include the overhead of the CTL, which is a requirement of this proposed system and not a part of the GA processing itself. The test shown in Fig. 7 do not include a CTL as the previous two - the chromosomes were directly mapped into the simulator instead - since our CTL implementation supports post-translation gates with 2 inputs only.

The simulated circuits were randomly generated in a single run, being the same for all genetic individuals. A total of 4 distinct circuits' results are shown in each plot. Although all individuals were the same, for a given run, the CTL was invoked once per individual in order to simulate its processing overhead, as it would happen if the simulator was integrated into a GA.

The simulation was performed on a per-individual basis, each one covering all possible circuit input combinations.

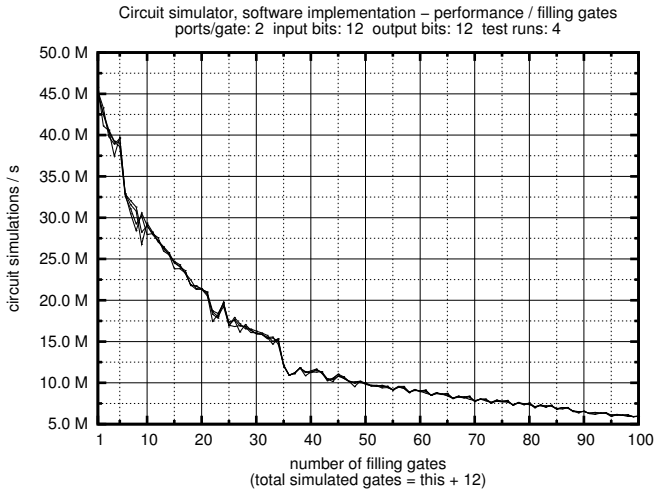

Fig. 5. software simulation performance vs filling gates

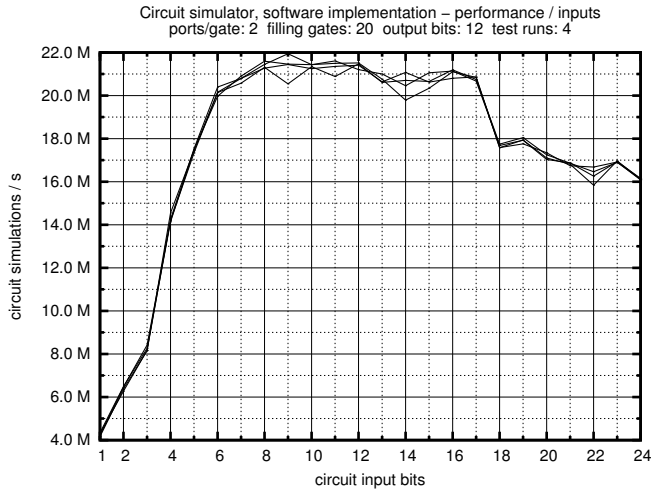

Fig. 6. software simulation performance vs circuit input bits

B. Testing Hardware and Software Environment

The hardware in which the tests were performed was a desktop computer with an Intel Core i7 2600K at 3.4 GHz. The host operating system was Debian Linux 6.0.6.

This software implementation does not support multi-threading. As consequence, only one core was used, out a total of 4 present in the processor.

C. Performance Results

The performance curve for varying CFW, with fixed values for circuit input/output bits (each set to 12) and 2 inputs/gate, is shown in Fig. 5. The curve starts with a CFW of 1, resulting in 45M simulations/s, and ends with over 5M simulations/s for a CFW of 100.

The impact of an increasing number of circuit input bits is shown in Fig. 6. A lower number of circuit inputs means fewer simulations performed per individual, increasing the proportion of the CTL processing overhead, what explains the reduced simulation performance for such circuits. After a certain number of input bits the curve starts to decrease, at least in part, due to processor cache pressure of write operations for a larger array of stored results.

The simulator supports a variable number of inputs/gate defined equally for all ports of a simulated circuit, the performance degradation scales well, as shown in Fig. 7.

The number of circuit output bits does not affect the number of simulated ports, nor the number of simulations performed per individual, reflecting in a minimal performance impact. For that reason, the corresponding graph was not included here.

D. Performance Comparison to Other Works

Benchmarks of circuit simulation execution time are not common in works concerning combinational logic circuit generation with GA, and the performance data that was found had to be extracted indirectly from the reported statistics.

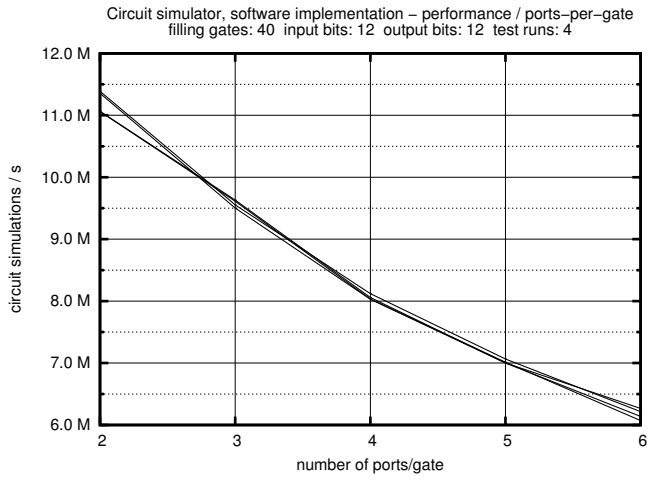

Fig. 7. software simulation performance vs inputs per gate

The performance from Bonilla[9] for his FPGA-based SIE implementation, considering: 12 input bits, 512 individuals, 100 generations results in a total of 209,715,200 circuit simulations. While it is not clear how many gates per circuit were simulated, the performance of that FPGA-based simulator is not sensitive to that parameter. Using 1 SIE nodes the resulting execution time was 5 seconds, thus 41,943,040 simulations/s. That number includes the overhead of the GA itself, which ran for 100 generations.

As shown in Fig. 5, for 12 input bits circuits, our software-only simulator showed performance from the maximum of 45 M simulations/s for $CFW = 1$, over 20 M simulations/s for $CFW = 20$, and 10 M simulations/s for $CFW = 50$. While FPGAs do not suffer from performance decrease with increasing simulated gates, is it clear that software simulation speed remains viable for circuits with total simulated gates $CFW + COW$ beyond 50.

V. Conclusion

A simple structure for a fast combinational logic circuit simulator was presented, with encouraging performance levels, particularly so considering that it is a software-only implementation and not FPGA-based.

Circuits with $CIW \leq 3$ clearly suffer from performance penalty during simulation due to CTL overhead. Those cases are not relevant in practice due to general circuit simplicity, resulting in fast GA convergence. Circuits with $CIW \geq 8$ have no discernible performance impact from the CTL.

Simulation performance is negatively affected by processor cache pressure, what increases with CIW and corresponding larger output tables. Cache pressure is expected to increase in a multithreaded implementation, due to caches shared by the involved cores. This problem can be mitigated with explicit cache management, as a possible extension to the simulator.

The simulator structure can also be extended to provide statistics such as: port count, transistor count, circuit latency and other. Such extensions can be implemented using the GC from the CTL, avoiding chromosome-specific implementations.

For future research, it may be feasible a similar generic simulation structure for real numbers, allowing fast evaluation of GA-generated mathematical expressions.

References

[1] A. Slowik and M. Bialko, "Evolutionary design of combinational digital circuits: State of the art, main problems, and future trends," in *Information Technology, 2008. IT 2008. 1st International Conference on*, may 2008, pp. 1 –6.

[2] C. Coello, E. Alba, and G. Luque, "Comparing different serial and parallel heuristics to design combinational logic circuits," in *Evolvable Hardware, 2003. Proceedings. NASA/DoD Conference on*, july 2003, pp. 3 – 12.

[3] A. Slowik and M. Bialko, "Design and optimization of combinational digital circuits using modified evolutionary algorithm," in *Artificial Intelligence and Soft Computing - ICAISC 2004, 7th International Conference*, vol. 3070, 2004, pp. 468–473, http://www.odysci.com/article/1010112985275770. [Online]. Available: http://link.springer.de/link/service/series/0558/bibs/3070/30700468.htm

[4] V. Vassilev, J. Muller, and T. Fogarty, "Digital circuit evolution and fitness landscapes," in *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, vol. 2, 1999, pp. 3 vol. (xxxvii+2348).

[5] C. Vijayakumari and P. Mythili, "A faster 2d technique for the design of combinational digital circuits using genetic algorithm," in *Power, Signals, Controls and Computation (EPSCICON), 2012 International Conference on*, 2012, pp. 1–5.

[6] S. M. Cheang, K. H. Lee, and K. S. Leung, "Applying genetic parallel programming to synthesize combinational logic circuits," *Evolutionary Computation, IEEE Transactions on*, vol. 11, no. 4, pp. 503 –520, aug. 2007.

[7] A. Hernandez-Aguirre, B. Buckles, and C. Coello-Coello, "Gate-level synthesis of boolean functions using binary multiplexers and genetic programming," in *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on*, vol. 1, 2000, pp. 675 –682 vol.1.

[8] C. A. Coello Coello and A. H. Aguirre, "Design of combinational logic circuits through an evolutionary multiobjective optimization approach," *Artif. Intell. Eng. Des. Anal. Manuf.*, vol. 16, no. 1, pp. 39–53, Jan. 2002. [Online]. Available: http://dx.doi.org/10.1017/S0890060401020054

[9] C. Bonilla and C. Camargo, "Low cost platform for evolvable-based boolean synthesis," in *Circuits and Systems (LASCAS), 2011 IEEE Second Latin American Symposium on*, 2011, pp. 1–4.