Comparative Evaluation of Algorithms for Identifying Delay Arcs in Logic Circuits

Rita Louro Barbosa, Bernardo Borges Sandoval, Pedro Henrique Aquino Silva and Cristina Meinhardt
Department of Informatics and Statistics, Federal University of Santa Catarina - UFSC, Florianópolis, Brazil
rita.louro.b@grad.ufsc.br, bernardoborgessandoval@gmail.com, pedro.aquino@posgrad.ufsc.br, cristina.meinhardt@ufsc.br

***Abstract*—The identification of delay arcs of a circuit is an essential step in the simulation and design analysis of logic circuits. This identification allows the measurement of delay times in output state transitions given a change in a specific input. The automation of this step is relevant for its potential to facilitate the study of circuits with truth tables with a large number of inputs and to speed up electrical characterization work in adjacent research. This work provides a comparative evaluation of two algorithmic alternatives for recognizing these arcs. The evaluation metrics are: Complexity analyses, Runtime, Memory and processor usage. The evaluation shows the advantage on complexity for the second version of the algorithm, defined by $O(k \log k)$, and the implications of this on runtime, memory accesses and number of instruction for a set of benchmarks.**

***Index Terms*—Cell characterization, Delay Arcs, Algorithm Evaluation**

I. Introduction

Cell characterization is a common task in the definition of a Standard Cell library. With the advance of technology nodes, more complex conditions must be considered when defining the cells electrical behaviors. Traditional approaches for cell characterization are based on electrical simulations that represents a series of evaluation corners With the technology evolution, more evaluation corners need to be considering during the cell characterization step, mainly in nanometer technologies [1]. Tools to execute the cell characterization are included in commercial and some open source EDA flows [2] [3] [4] [5]. However, the learning of how to execute a cell characterization is briefly introduced in most computer science or electronic engineering graduation courses. Knowing how to identify the characteristics that will be abstract in the RTL level as the delay of a cell is a relevant topic to students, mainly to instigate the electrical-level evaluation or for the elaboration of standard cell libraries.

There are few educational tools for microelectronics in the literature [6] [7]. None of these tools, to the best knowledge of the authors, explore the electrical characterization and help young students acquire research interest in following these integrated circuit design areas. Thus, this work is an initial part of an educational project that aims to provide open-source tools to help students understand the electrical cell characterization steps and easily transpose the unknown subject barrier. This initial work focuses on the algorithms to find and identify the delay arcs from a Truth Table.

In this work, we present two algorithms to identify the delay arcs from a logic function. These algorithms are evaluated in terms of computational complexity and execution traces when applied to a set of benchmarks composed of functions with a growing number of inputs. This initial step of the educational project is relevant to help students to understand the delay arcs characterization and recognizing how the number of inputs in a function impacts algorithms This knowledge is valuable even for designers working at advanced abstraction levels of integrated circuit design.

II. Methodology

The measurement of a circuit's critical timing (delay) depends on identifying all delay arcs of the function. Delay arcs are defined by the transitions in the input vector (a transition from input vector 'a' to input vector 'b') in which only one entry changes and that change reflects a transition in the output value. In the case of circuits with multiple inputs and outputs, characterization and analysis of delays becomes a task with scalar complexity proportional to the number of entries and rows in the circuit's Truth Table.

Given a circuit with n inputs, its complete Truth Table will have 2^n rows. Considering only Truth Tables without "don't care" representation, we consider four possible situations: 1) complete and ordered; 2) complete and unordered; 3) incomplete and ordered; and 4) incomplete and unordered. A complete Truth Table contains all 2^n input vectors (rows). It is considered ordered when the input vectors are positioned following the binary sequence.

Now, thinking about the possibilities of arcs, the relationships are bidirectional. That is, when a transition satisfies the necessary conditions to be considered a delay arc, the opposite transition will also satisfy the conditions, i.e., if (a, b) matches the conditions, so will also arc (b, a). In this work, we evaluate and compare two algorithms for finding the delay arcs. The conditions for a pair of input vectors to be considered a delay transition arcs are: 1) the input vectors in the pair present different output values, and 2) the input vectors in the pair has only one bit difference.

The algorithms were implemented using the Rust programming language, version 1.72.0. The Truth Table structure was created to store the data in a Truth Table. It contains the following attributes: 1) A vector of output vectors to store each row of the Truth Table corresponding to the outputs;

2) A vector of input vectors that stores each Truth Table column corresponding to an input; and 3) the information on the number of inputs, outputs, and Truth Table rows.

Both algorithms begin with the parsing of a Truth Table. A Truth Table instance can be created through a function called `process_pla` that extracts all required information represented in a PLA file format.

A. *Algorithm Find Arcs V1*

The Algorithm Find Arcs V1 can be used in the four possible situations presented before, i.e., with complete and incomplete tables, which may or may not be ordered with increasing sequences of input vectors. In this algorithm, the truth table is interpreted in the sense of its input columns. The value of each input must be stored individually in a vector and will also be accessed separately inside the algorithm.

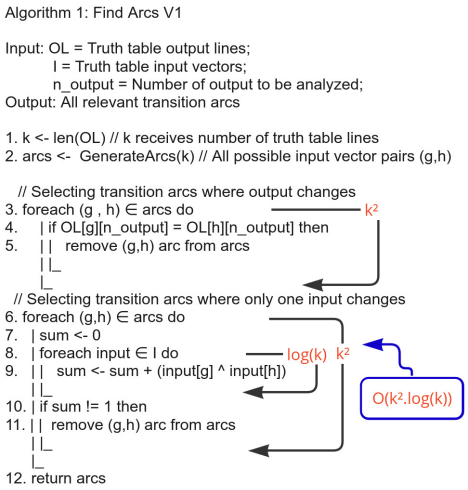

Fig. 1. Algorithm Find Arcs V1

The algorithm is presented in Fig. 1. The input is a vector containing the output lines of the Truth Table (the structure OL), a vector containing the input vectors (I) , and the output number to be considered (n_output). The algorithm returns the set of transition arcs relevant to analyzing a circuit's delays in the format of a vector of tuples corresponding to the rows of delay arcs on the truth table. Initially, the algorithm call the auxiliary algorithm "Generate Arcs" (line 2), presented in Fig. 2. This algorithm generates all bidirectional arcs $(a, b) = (b, a)$ with possible input vector pairs that will be after selected in the algorithm Find Arcs V1.

The next step in Algorithm Find Arcs V1 is a loop from lines 3-5 that will select the transition arcs from the generated arcs that obey the first arc condition, i.e., presents a change in the output. Then, from the remaining arcs, the loop form lines 6 to 11 removes the pairs that does not obey the second arc condition, i.e, change more than one bit in the input vectors. This selection is made by summing the results of the application of the XOR operation on the value of the arc's input vectors and subsequently checking this sum. If the result

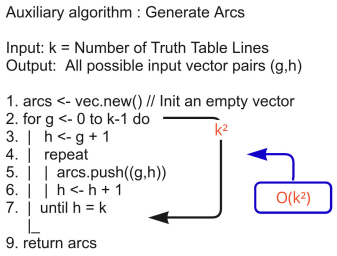

Fig. 2. Auxiliary Algorithm Generate Arcs

is greater than 1, it means that more than one input changes in this arc.

B. *Algorithm Find Arcs V2*

In this algorithm, the Truth Table is interpreted from the perspective of the value of each row, taking advantage of the fact that each input row (input vector) corresponds to the value of an integer $i \in [0, k-1], k = 2^n$.

The V2 is presented in Fig. 3. This version receives as input the same structures OL and n_output used in the V1 Algorithm. In addition, it receives the number of Truth Table inputs (n_output). It returns the set of transition arcs relevant to analyzing a circuit's delays, in the format of a vector of vectors. However, differently from the V1 algorithm, the V2 returns the numbers inside each vector corresponding to the input vectors that form the transition arc when paired with the number of the vector's position. Initially, the algorithm calculates the total number of rows in the Truth Table. Differently from the V1 that calls the auxiliary algorithm Generate arcs, on the V2, the valid arcs are generate internally (lines 6-9), checking the transition arcs conditions.

The current implementation of Algorithm V2 can only be used for complete ordered truth tables. To solve this limitation, an optimization of Algorithm 2 could consider include a Hashmap to represent the output lines.

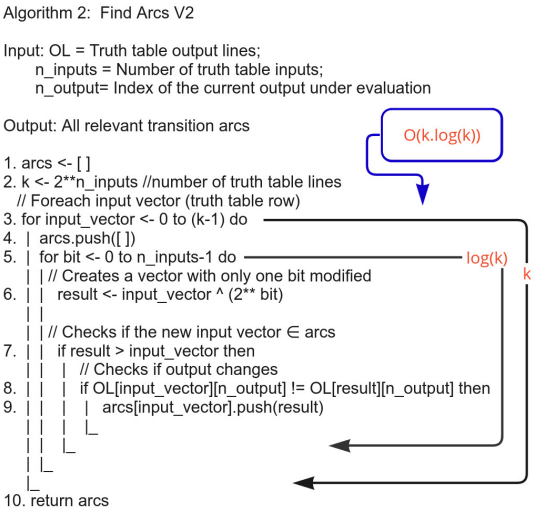

Fig. 3. Algorithm Find Arcs V2

III. Comparative evaluation of the Algorithms

We begin discussing the algorithms through the code's computational complexity in Big O notation. This describes the upper bound in computation time required to run an algorithm. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform. After that, we also evaluate the algorithms execution for a set of benchmarks, presenting execution time, memory and processor usage. The experiments are executed in a 64-bit Intel® Core™ i5-8265U CPU @ 1.60GHz with Cache L1: 64 KB (per core), Cache L2: 256 KB (per core), Cache L3 : 6 MB (shared) and 8 GB of RAM running Ubuntu 22.04.4 LTS.

A. Computational Complexity

We start the evaluation with the Auxiliary algorithm for Generate Arcs, presented in Fig. 2. Eq. 1 shows the complexity, considering that all assignments and vector insertions have time complexity $O(1)$. Thus, as the number of iteration is equivalent to the number of total possible arcs in a Truth Table with k lines. The total number of arcs can be found by using the general formula for the sum of the terms of an AP, where f is the total number of terms. In our case, k is the number of rows in the Truth Table, a_1 is the initial term (0), and a_f corresponds to the last term , which is equal to $k-1$. Thus, the complexity of this auxiliary algorithm is defined as $O(k^2)$.

$$Total = \frac{(a_1 + a_f)f}{2} = \frac{((k-1)+0)k}{2} = \frac{k^2-k}{2} = O(k^2) \quad (1)$$

The computational complexity of Algorithm Find Arcs V1 is dependent on the complexity of the auxiliary Algorithm Generate Arcs. So, as shown in Fig. 1, lines 3 to 5 are $O(k^2)$. The loop between lines 6-11 is repeated for the same maximum number of arcs. However, the internal loop defined in the lines 8 and 9 runs in $O(\log k)$ over the number of inputs of the Truth Table. Thus, this algorithm executes in $O(k^2.\log k)$.

Differently, the Algorithm Find Arcs V2 has the computational complexity dependent on the number of Truth Table lines (k) and the number inputs of the Truth Table. The loop between lines 3-9 interacts over the input vectors (k), thus the complexity is $O(k)$. The internal loop in lines 5-9 creates all possible input vectors, corresponding to the number of inputs of the Truth Table, with complexity $O(\log k)$. Thus, the complexity of V2 is reduced to $O(k.\log k)$.

B. Performance evaluation

To evaluate the performance of the two algorithms, we select a set of benchmark cases including functions with 3 to 10 inputs. The evaluations adopt open source profiling tools available for Rust language integration, that allow running the algorithms and collecting specific data about the execution. All experiments considers executables compiled in release mode, with the same optimization options set on the compiler.

The execution time is obtained from the Glassbench tool [8]. The time report shows the average time from successive runs for the same function from the benchmarks, to minimize random dispersion. The time required for the initialization of data structures and variables is not taken into account. In addition, the execution of the algorithm has been isolated by a function that prevents compiler optimizations for this part of the code, avoiding interference in the results. Fig. 4 summarizes the runtime report. Algorithm 1 showed accelerated exponential growth, with runtime of 0.313 microseconds and 4.660 seconds, to 3 and 10 inputs, respectively. Algorithm 2, on the other hand, showed more balanced exponential growth, with runtime limits of 0.272 microseconds and 54.3 microseconds for the same test cases. This represents a difference of 5 orders of magnitude between the highest result for Algorithm 1 and the highest result for Algorithm 2. This result reinforces the insight gained from the complexity analysis, that the greater the number of inputs, the greater the time complexity advantage of Algorithm 2 over Algorithm 1.

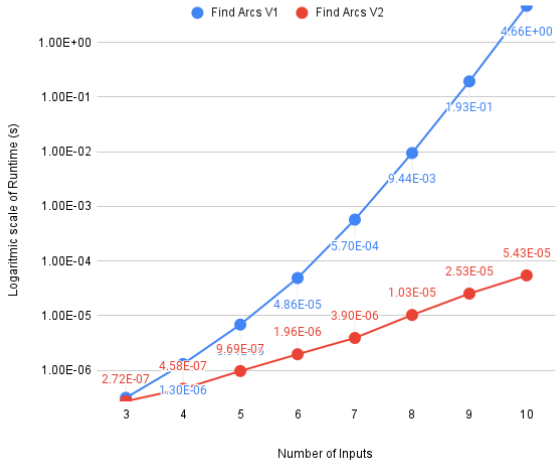

Fig. 4. Mean runtime of Find Arcs V1 and Find Arcs V2.

We adopt the Heaptrack tool [9] for the memory usage evaluation. In this evaluation, we are considering the data structure and variables initialization. It is important to highlight that the same $TruthTable$ data structure is initialized before running both algorithms, so we can assume that the discrepancies between the memory usage results for each algorithm in the same test case (same Truth Table) refer to the difference in memory accessing sequences particularities of each algorithm. Figures 5 and 6 shows the memory consumption for Algorithm 1 and 2, respectively. The memory usage differs between the evaluated algorithms throughout the execution. Algorithm 1 maintained a low and stable growth rate according to the increase on the number of inputs. The memory usage accelerate from the test case with 8 inputs, peaking at 10 inputs, with consumption of 1.2 MB. On the other hand, Algorithm 2 achieved a relatively more balanced growth in

memory consumption, reaching 149.5 kB in its largest test case (function with 10 inputs).

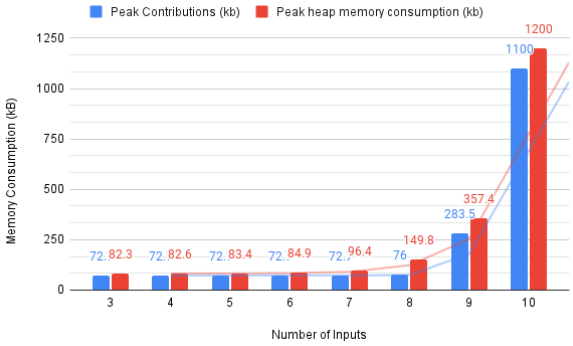

Fig. 5. Memory consumption during the execution of Algorithm 1

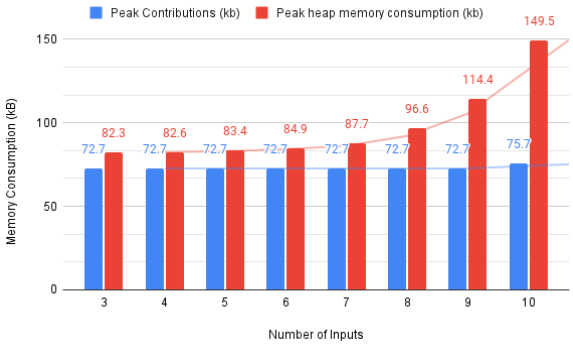

Fig. 6. Memory consumption during the execution of Algorithm 2

Table I presents the number of instructions executed, the number of access to the first level of cache (L1), to second level of cache (L2) and to the RAM. These results are obtained with the *iai* tool [10] that is an experimental benchmarking harness that uses Cache grind to perform extremely precise single-shot measurements of Rust code. We also present the estimated number of cycles. Due to space limitations, in this paper we opt to show the ratio relative comparison, considering the Algorithm 2 as reference. Observing the relative comparison, Algorithm 1 runs less instructions, and has less memory accesses than Algorithm 2 only for functions with 3 inputs. For functions with 4 to 5 inputs, the Algorithm 1 show less L2 and RAM accesses, that could represent that in these cases, the hit rate of L1 cache is higher for the Algorithm 1, increasing the L1 accesses. For functions with more 5 inputs, Algorithm 1 increases significantly the number of instructions executes, cycles required and memory accesses. It is a reflect of the exponential behavior shown in the Algorithm 1 complexity evaluation.

IV. Conclusion

The comparative evaluation performed in this work suggest a notable advantage in the estimated complexity, time efficiency and memory consumption for Algorithm 2 over Algorithm 1. Algorithm 1 is currently an option to unsorted Truth Table, however, with the future Hashmap optimization, Algorithm 2 will bring advantages for all cases. Next steps on this research includes defining the minimal simulation steps required to execute electrical simulations and obtain the delay and power electrical characterization for a given Truth Table. Thus, the develop algorithms will be integrated in an open-source educational tool to help students to understand the cell characterization process.

TABLE I
Relative comparison of processor, cache and memory usage considering Algorithm 2 as the reference

Number of Inputs	Instructions	L1 Accesses	L2 Accesses	RAM Accesses	Estimated Cycles
3	0.99	0.98	0.88	0.98	0.98
4	1.2	1.11	0.90	0.97	1.09
5	1.59	1.59	0.88	0.97	1.53
6	3.22	3.38	1.07	1.11	3.25
7	96.57	198.22	1.71	1.32	190.90
8	954.76	2021.08	15178.37	1.96	1987.50
9	8399.48	17876.78	62492.32	4.78	17832.97
10	53813.13	112432.07	317829.02	9.44	112951.71

Acknowledgment

This work was financed in part by CNPq and the Propesq/UFSC.

References

[1] Tianliang Ma, Zhihui Deng, Xuguang Sun, and Leilai Shao. Fast cell library characterization for design technology co-optimization based on graph neural networks. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 472–477, 2024.

[2] Library Characterization and Optimization - Silvaco — silvaco.com. https://silvaco.com/library-characterization-optimization/ . [Accessed 30-05-2024].

[3] What is Library Characterization? – How it Works & Techniques — Synopsys — synopsys.com. https://www.synopsys.com/glossary/what-is-library-characterization.html. [Accessed 30-05-2024].

[4] Liberate Trio Characterization Suite — cadence.com. https://www.cadence.com/en_US/home/tools/custom-ic-analog-rf-design/library-characterization/liberate-trio-characterization-suite.html. [Accessed 30-05-2024].

[5] Kronos Characterizer resources.sw.siemens.com. https://resources.sw.siemens.com/en-US/fact-sheet-kronos-characterizer. [Accessed 30-05-2024].

[6] G. Mishuris L. Brooks C. McLean V. Chan J. A. del Alamo, J. Hardison and L. Hui. Educational experiments with an online microelectronics characterization laboratory. https://mtlsites.mit.edu/users/alamo/pdf/2002/RC-91 [Accessed 30-05-2024].

[7] Samuel J. Dickerson and Renee M. Clark. A classroom-based simulation-centric approach to microelectronics education. *Computer Applications in Engineering Education*, 26(4):768–781, June 2018.

[8] GitHub - Canop/glassbench: A micro-benchmark framework to use with cargo bench — github.com. https://github.com/Canop/glassbench. [Accessed 31-05-2024].

[9] GitHub - KDE/heaptrack: A heap memory profiler for Linux — github.com. https://github.com/KDE/heaptrack. [Accessed 31-05-2024].

[10] GitHub - bheisler/iai: Experimental one-shot benchmarking/profiling harness for Rust — github.com. https://github.com/bheisler/iai. [Accessed 31-05-2024].