# MAGICAL - Genetic Algorithms for More Efficient In-memory Computation through Applied Graph Analysis

Andey Robins
*Dept. of Electrical and Computer Engineering*
*University of Central Florida*
Orlando, USA
andey.robins@ucf.edu

***Abstract*— Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius.**

***Index terms*—in-memory computation, artificial intelligence, genetic algorithms, computer aided design**

## I. Introduction

Data and the calculations performed on data are physically separated in modern computing devices. Data is moved from a storage media to closer and closer locations before it is finally used for computation before being places back into memory. This paradigm of computing, the Von-Neumann paradigm, has been instrumental in the development of modern computing solutions. As processing has sped up though, this transfer of information from storage to processing has begun to form a bottleneck which limits the computational speeds which can be achieved by state-of-the-art devices. Specialized processing units, such as Graphics Processing Units and other purpose-built hardware, often attempts to side-step the problem by increasing the potential throughput of information; however, the theoretical problem of moving data around is not solved, only mitigated, by these solutions. An alternative paradigm to the von-Neumann architecture could be performing the computation at the same place the data is stored. This computing paradigm is aptly referred to as "in-memory computation."

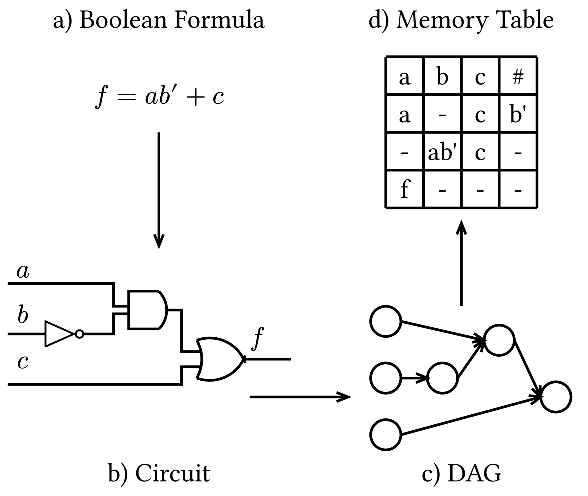
Memristor Aided Logic (MAGIC) is an emerging computing paradigm making use of parallel, write-based systems to perform calculation in-memory [1]. This requires scheduling operations for the computation; however, the scheduling order, upon execution, may have substantially differing memory footprint requirements. State-of-the-art solutions model this dependence as a graph problem and perform scheduling as a graph covering problem. In this work, we characterize a number of properties of these evaluations graphs and apply those observations to the development of a genetic algorithm which produces reductions in the memory footprint of execution between 14% in the worst case and 26% in the best case when compared to standard algorithmic approaches

## II. Prior Works

Within the realm of the MAGIC, application of formal design principles has continued with advances in technology mapping [2] leading to advances in area, energy, and operation counts.

## III. Problem Specification

In-memory computation can be modeled as an execution graph. Translated from traditional combinational logic, each vertex in the graph corresponds with a boolean logic gate. Within the memristor array, the input values to the logic gate can be selected by the write signal before being written to an empty location in the memristor array. An edge exists in the graph from a vertex to another if they have this depednece relation. As an example, for evaluating the boolean function $f = ab' + c$, Figure 1 details the transformation from function to circuit to graph to in-memory computation. Beginning with a boolean function $f$, existing processes are highly capable of mapping this to a minimal circuit. This circuit can then be transformed into the graphs under discussion in this work by assigning each gate a vertex and making each wire an edge in



|  |  |  |  |
|---|---|---|---|
| a | b | c | # |
| a | - | c | b' |
| - | ab' | c | - |
| f | - | - | - |

Figure 1: A diagram which presents the mapping of information through associated domains. A boolean formula is mapped to a boolean logical circuit, to a DAG, and finally to a memory execution sequence.

the graph. The final step illustrated in Figure 1 is the evaluation using in-memory computation and illustrates the requirement of more memory cells than inputs to the function. A '-' indicates that a memory cell is currently free and can have a value allocated. A '#' indicates that the memory cell hasn't been allocated yet, but will be in the future (in the example in the figure this only occurs in the fourth column of the first row, when we need to allocate a new space for the very first operation). A fact worth explicit mention: a value cannot be read and written to the same memory block by the same operation. The objective of this work is to minimize the extra memory needed.

Formally, the interdependence between computational nodes in the execution graph is a directed acyclic graph (DAG) in which the children of a vertex must not be executed until that vertex is executed. Thus, for any vertex, it can be viewed as both the root of

### A. *Cost Metric*

The memory footprint, in other words the cost, of a sequence $S$ derived from an adder DAG is defined as the number of memory cells that are need to evaluate the entire DAG. Determining the cost of a sequence can be done in linear time by simulating the execution sequence as a series of instructions. A memory bank is initialized with the labels of each vertex in the DAG with no parents (the input vertices). Then, the next element in the sequence is put into a free spae in the memory bank, expanding the width if necessary. This vertex is then marked as processed and all vertex labels which have all of their children marked as processed are removed from the memory bank. The maximum width at any point in this process is defined as: $\text{cost}(S)$.

## IV. Genetic Algorithm

Genetic algorithms seek to emulate the processes of evolutionary biology observed in the physical world. They do this by modeling populations, natural selection, and reproduction over a series of "generations." An optimal solution, whether globally optimal or locally optimal, is found over time through selecting only the most viable samples from the population for inclusion in subsequent generations.

Four common phases make up genetic algorithms: encoding, selection, crossover, and mutation [3]. This work uses a topological sorting of the DAG for the encoding. It uses a simple rank selection mechanism based on the score metric defined in Section III.A. For crossover, single point crossover based on analysis of the graph is employed, and swapping is used to mutate the sorting when mutation is applied. Each element of the algorithm is detailed in this section.

### A. *Encoding and Selection*

Encoding and selection are both informed by the underlying DAG, but make less explicit use of graph analysis than the other phases of the algorithm. The encoding of a sequence is a straightforward topological sort of the vertex labels in the graph. Trivially, a valid execution sequence can be attained using breadth first search. Sequences are therefore sorted according to the cost of the sequence for selection. During the selection phase, only the best performing sequences (those with minimal score) are selected to be present in the next evolutionary generation.

### B. *Crossover*

An observation of the DAGs within this problem domain will quickly illustrate that there is a "cost-maximizing" step from which the cost of the remainder of execution is monotonically decreasing. This implies that for a block in the DAG which occurs after the cost-maximizing step occurs other values could remain in memory without increasing the cost of the sequence. Therefore, modifying the execution order of the block can be done without necessarily incurring a cost increase. However, the larger this block is, the more likely it is to include this cost-maximizing step (or to become this step by violating the dependence requirements of the DAG).

With these two ideas in mind, a single-point crossover mechanism is a natural fit to this problem. While it will occasionally break a block during the crossover, due to the nature of the DAG and this algorithm's place in the larger genetic algorithm this crossover mechanism could behave in a manner characterized as a search for an optimal order of executing blocks in the post-cost-maximizing phase of execution. As long as their order doesn't impact the overal cost of the sequence, the problem imposes no specific constraints on its performance. Therefore a crossover mechanism such as this

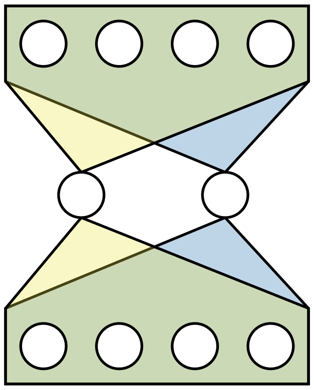


Figure 2: The overlapping nature of children and parents illustrated visually. Both central nodes share ancestors and decendants; however, there exists no direct dependence relationship between each other. We describe these nodes as "peers."

will also fit naturally with the mutation mechanism employed for cost minimization.

### C. Mutation

A naive approach to mutation for this problem would be to randomly select two vertices in the sequence and swap their positions. This is a straightforward solution to implement, and it leads to sequence synthesis which performs similarly in regards to the sequence cost as the greedy algorithm presented in the literature and previously referred to in this work. However, in the context of the DAG modeling the problem, this strategy becomes more evidently sub-optimal. Two random vertices are likely to have an ancestor-descendant relationship of some sort, which implies that their swapping would create an invalid execution sequence.

Instead of swapping vertices randomly in the execution sequence, we instead compute all of the peers of each vertex where a pair of vertices are peers if they have intersecting parents and children but are not directly dependant on one another. See Figure 2 for a visual depiction of this relationship. Extending this logic, a swapping of these two elements in the sequence can be seen as determining which of the blocks of the graph should be prepared for evaluation earlier in the sequence.

When paired with the block-targeting crossover mechanism, this leads to an algorithm which is responsive to the underlying data modeled by the DAG. This leads to both faster convergence on a valid execution sequence than random mutation, but also improves cost in comparison with the greedy scheduling algorithm.

## V. Evaluation

### A. Data Set

| Adder Width | Mem. Footprint |
|---|---|
| 1 | 6 |
| 2 | 9 |
| 4 | 14 |
| 8 | 26 |
| 16 | 50 |
| 32 | 91 |

Figure 3: The memory footprints of adders with varying width discovered through the greedy algorithm [4].

The dataset evaluated in this work is synthesized from n-bit adder specifications using NOR and NAND gates. The netlist for the adder is modeled as a DAG and described using a proprietary format. This is parsed into a general DAG for evaluation. Six different adder widths were analyzed representing all powers of two less than 64 (i.e. [1, 2, 4, 8, 16, 32]). Execution sequences for each DAG were also provided. The memory footprints of these sequences are evaluated in the same way as candidate solutions will be evaluated in this work to determine a baseline memory footprint to compare future solutions against. Figure 3 lists each adder's size and the best provided sequence's memory footprint. Complete BLIF specification adders come from the work of Rashed et al. [4].

## VI. Results

Execution sequences synthesized with the genetic algorithm were smaller than those found by the greedy algorithm presented in the literature in all tested cases. The improvements ranged from a 14% smaller memory footprint in the worst case (adder width = 4) to a 26% smaller memory footprint in the best case (adder width = 16). The average improvement was a 19.8% smaller memory footprint with the genetic evolutionary algorithm when compared to the greedy algorithm.

### A. Synthesis Stopping Conditions

| Adder Width | New Footprint | Memory Savings |
|---|---|---|
| 1 | 5 | 1 |
| 2 | 7 | 2 |
| 4 | 12 | 2 |
| 8 | 20 | 6 |
| 16 | 38 | 12 |
| 32 | 74 | 17 |

Figure 4: The memory footprints of adders with varying width discovered through the genetic evolution algorithm.

The stopping conditions for synthesis were that a configurable number of generations passed without improvement to the current best solution. This number of generations ranged from 50 to 1000 with larger values being used for adders with larger width. This was done to counteract the exponential growth in the search space for each increase in adder width. Bayesian optimization was employed to find an optimal configuration balance between population size and generations before stopping.

Large values for both parameters were found to be the most effective. Intuitively, this makes sense, as we care more about the final synthesized solution than we do about the time it takes to get there. On the machine used to synthize solutions (a desktop running Arch Linux (EndeavourOS) with kernel version 6.6.2, an AMD Ryzen 9 7950X, an NVIDIA Geforce RTX 4090 GPU, and 64 GB RAM), synthesis of the width 32 adder took 844 seconds (slightly over 14 minutes). A significant portion of this time was spent with the minimal sized solution already found, so if bounds were able to be tightened on the stopping conditions, synthesis time would similarly begin to shrink. Further analysis of the DAG underlying this problem may yield such findings.

## VII. Discussion

Traditionally, this problem is analysed in a manner similar to the synthesis of traditional hardware circuits. It is viewed as a graph covering problem where the goal is to map circuit elements form a template library onto the netlist of the circuit to be realized. Recontextualizing it as a topological sorting problem rather than a covering problem allowed for a different form of analysis than has historically been employed in solving this problem.

The genetic algorithm performed well and was reasonably adapted from the underlying graph problem. As an algorithm for artificial intelligence, it was well suited for adaptation into the synthesis process. This raises significant questions about the role that artificial intelligence can, and more importantly should, play in hardware synthesis. The final memory footprint values attained from the sequences generated by the genetic algorithm are more efficient than the sequences generated by the greedy algorithm, but there are no guarantees about the optimality of these solutions. Running the algorithm for only a single more generation could potentially have yielded an even more effective solution. With clear evidence that artificial intelligence algorithms can be effectively employed to improve state-of-the-art synthesis tasks in computer engineering domains, the question may now turn to determining the engagement paradigms with such systems in order to ascertain the best means of interaction with artificially intelligent design systems.

## VIII. Conclusion

This work contributes two specifics to the problem of scheduling in-memory, MAGIC operations. First, it highlights characteristics and properties of the graph problem underlying the sequencing problem and identifies ways to exploit these capabilities to generate more effective sequences. Second, it adapts these observations and intuition to a genetic algorithm which is able to produce more efficient sequences than the currently reported greedy algorithms present in the literature.

One primary limitation of this work is that it is based upon the adders synthesized using traditional CAD workflows. Exploring genetic algorithm based artificial intelligence and its applications within the synthesis of hardware components is one future direction for research. Additionally, other works often compare implementations of their synthesis operations to the energy and footprint requirements of other solutions. While this is future work which could be done within the scope of these findings as well, the scope of this work is in exploring the internal organization of individual circuit elements such as adders and not the role they play as elements within a larger computation environment. Decreasing the memory requirements of each component will in turn decrease the overall memory utilization.

Another important limitation still remaining is that synthesizing a solution with this genetic algorithm has no obvious stopping condition. Solutions regularly converge in times which are reasonable comparable to the input size, but there are no guarantees that running the process for longer would not yield a better result. As results currently converge significantly below the solutions found by traditional algorithms, this isn't an immediate concern; however, before the adaptation of genetic synthesis algorithms into production level workflows can be completed, more in-depth analysis of appropriate stopping and convergence conditions is necessary.

Finally, future work will explore the ability of these algorithms, and therefore the generalizability of these analyses of the dependence graphs, to other circuit structures. While adders are able to be meaningfully improved in their execution sequence with these techniques, other circuits must also be evaluated to determine the transferability of these findings.

## References

[1] S. Kvatinsky *et al.*, "MAGIC—Memristor-aided logic", *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.

[2] M. R. H. Rashed, S. Thijssen, S. K. Jha, and R. Ewetz, "Automated Synthesis for In-Memory Computing".

[3] S. Mirjalili and S. Mirjalili, "Genetic algorithm", *Evolutionary Algorithms and Neural Networks: Theory and Applications*, pp. 43–55, 2019.

[4] M. R. H. Rashed, S. K. Jha, and R. Ewetz, "Logic Synthesis for Digital In-Memory Computing", in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022, pp. 1–9.