UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Universitat Politècnica de Catalunya
Facultat d'Informàtica de Barcelona

Master in Innovation and Research in Informatics
Advanced Computing

Master's thesis

AIG transformations to improve LUT mapping for FPGAs

Pol Barrachina

Supervised by Jordi Cortadella Department of Computer Science

June, 2022

In memory of Dani Guerrero-Belfiore

I would like to thank my supervisor, Prof. Jordi Cortadella, for his guidance, support, providing invaluable help and inspiring my interest in logic synthesis.

Abstract

A Field-Programmable Gate Array (FPGA) is a general re-configurable device for implementing logic circuits. This technology is extensively used for prototyping circuits due to its cost and speed. The underlying implementation consists of Lookup Tables (k-LUT), logic functions that can implement any function up to k variables. And-Inverter graphs (AIG) are multi-level networks composed of two input `AND`s and inverters and are the standard format for describing Boolean functions in practical applications of logic synthesis.

In this thesis, we present an orthogonal technique that, interleaved with already known high-effort area mapping, outperforms previous best work on technology mapping. This technique, named AIGROT, explores several ways to exploit the commutativity and associativity of the `AND` operation, thereby reducing the number of LUTs needed to represent the Boolean functions. Experimental results show a substantial circuit minimization on several large public benchmarks without practically increasing the *runtime* or memory requirements. The proposed scheme is a blend of alternating techniques throughout the logic synthesis flow that provides tangible results at least as good as previous ones. In particular, using our technique, we are able to improve the best known area of four circuits from the EPFL benchmark library, which is considered the most comprehensive and diverse set of benchmarks, where all recently developed logic synthesis algorithms are tested.

Keywords

Logic Synthesis, Technology Mapping, FPGA

Contents

1. Introduction

Integrated circuits (IC) are present in every electronic device, from personal computers to light bulbs. From a theoretical point of view, integrated circuits consist of a network of logic gates that compute a Boolean function. From a practical perspective, the pieces that assemble integrated circuits and logic gates are transistors (metal–oxide–semiconductor field-effect transistor or MOSFET) a revolutionary replacement for vacuum tubes in 1960s. These transistors are made of silicon, a semiconductor material that grants them the physical property of acting as either an electrical conductor or an insulator, depending on another electrical signal. Gordon Moore predicted in 1975 that the number of transistors in an IC would double every year[1], this hypothesis known as Moore's law has been proved right in the last few years. Nowadays, the number of these tiny devices is immense, for instance, in a consumer class 16GB DRAM there are over 144 billion transistors. However, Moore's law is expected to break sooner than later, due to physical limitations, including the size of silicon atoms and electronic instability. For this reason, efforts to reduce energy consumption and increase computing power are aimed at optimizing the Boolean functions in order to reduce the size of electronic circuits. Unfortunately, no efficient algorithms are known for circuit minimization. This problem is usually referred as Minimum Circuit Size Problem (MCSP) that asks whether there exists a circuit of size k that computes a particular truth table. MCSP complexity is still unknown and any progress in this direction would have deep consequences in computer science [2].

First designs of IC were done exclusively by hand, quickly becoming a problem for large circuits. The number of transistors and manual mistakes conducted the development and usage of automated methods. Electronic Design Automation (EDA) is a research area that was introduced in 1980 and added an abstraction layer between circuit design and implementation. The first revolutionary change of IC was the introduction of Very High-Speed Integrated Circuit Hardware Description Language (VHDL) whose purpose is to automate the construction of large and complex integrated circuits. Years of research in EDA lead to the development of several intermediary steps to tackle down the task of compiling code into hardware. Logic synthesis is the step that translates low level hardware definitions into logic gates for Integrated circuits or FPGAs. This scope, logic synthesis, is where the research of this project belongs.

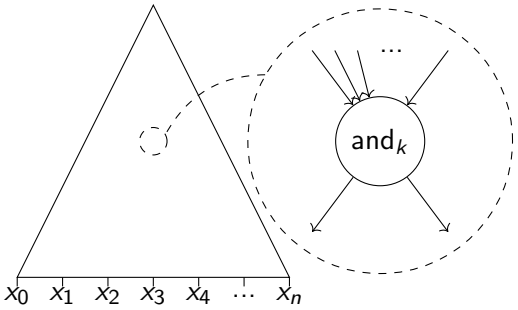


Figure 1: General And-Inverter graphs

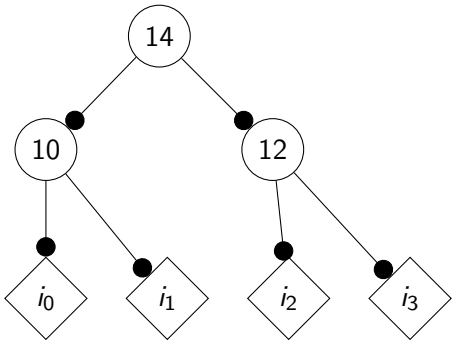


Figure 2: AIG representing the function $f = (i_0 + i_1)(i_2 + i_3)$

An important aspect in logic synthesis is the representation of the Boolean function. Desired properties of such a representation include that the size of the structure does not grow uncontrollably, that any Boolean function can be represented, and that there exist efficient algorithms to access and modify it. Along the history of EDA, several structures have been used, including Truth Tables (TT) Figure 4,

Product of Sums (PoS), Sum of Products (SoP) and Binary Decision Diagrams (BDD) Figure 3. Each structure has its advantages and disadvantages, for instance, truth tables work really good but only for a small number of variables, SoP is a better choice than truth tables as there exist fast algorithms like EXPRESSO for minimizing the representations. BDD and Reduced Ordered Binary Decision Diagram (ROBDD) are also good for representing Boolean functions as they are canonical, which is a desired property for several algorithms, for instance, equivalence checking, but both structures grow exponentially with common functions. In the last few years And-Inverter graphs[3] have been the preferred choice, they scale properly because structural hashing (*strashing* Figure 6) removes redundant nodes and produces a sort of "canonical" representation for each function, however they are not unique. In Figure 1 and Figure 2 we have examples of And-Inverter graphs. Another interesting property of AIG is combinatorial equivalence checking with SAT solvers, which is explained in subsection 2.1. Additionally, the implementation of algorithms over AIG structures is easier and faster. Nowadays, nanotechnology enables the manufacturing of devices with physical properties that revolutionise previous computing paradigms such as quantum cellular automatons. To take advantage of these technologies more powerful representations are needed, e.g. Majority-Inverter-Graphs (MIG)[4] or Biconditional Binary Decision Diagrams (BBDD)[5], these representations are out of the scope of this thesis.

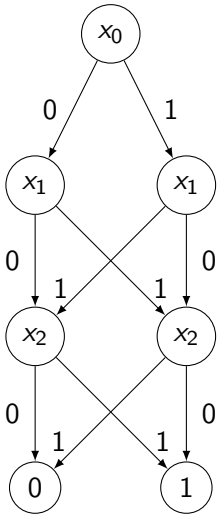


Figure 3: BDD representation: XOR of x_0, x_1, x_2

x_0	x_1	x_2	$x_0 \oplus x_1 \oplus x_2$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Figure 4: Truth Tables: XOR of x_0, x_1, x_2

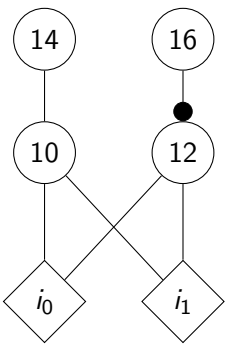


Figure 5: AIG without strashing

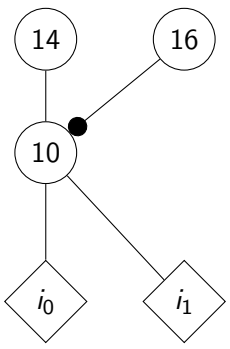


Figure 6: strashed AIG

An And-Inverter graphs is a Directed Acyclic Graph (DAG) that can represent any multi-output Boolean function using only 2-input ANDs and inverters. AIGER[3] is a library that defines the AIG file format and allows to store and retrieve AIGs represented in binary and ASCII. For large circuits the binary format is preferred because the file is orders of magnitude smaller and it is much faster to read and write. AIG is the Boolean function representation used in this project, in section 2 there is an in-depth analysis.

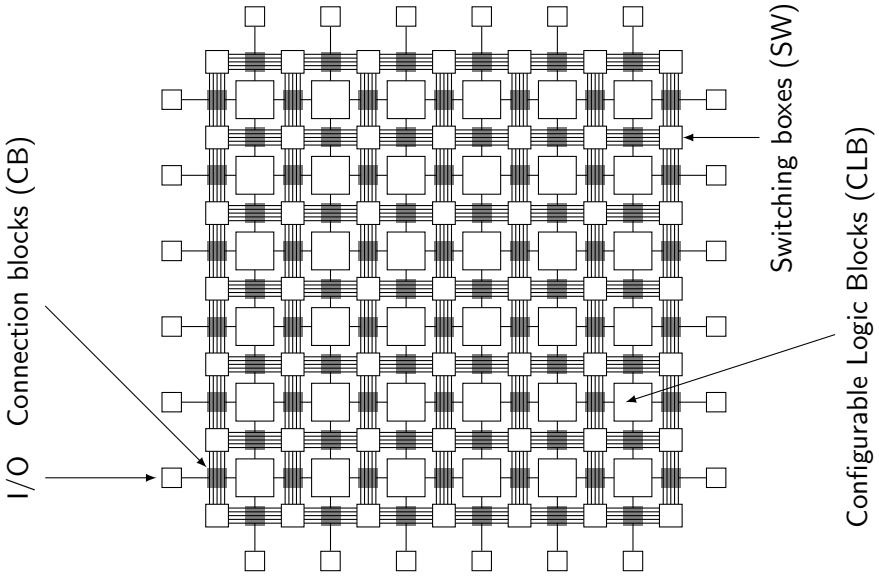


Figure 7: FPGA Island-style Architecture

A Field-Programmable Gate Array (FPGA) is a general purpose electronic device that can be programmed to implement any circuit where the limiting constraints depend on the size and characteristics of that particular FPGA model. Informally, a FPGA is a device that allows the emulation of an integrated circuit. In general, most commercial FPGAs can be reprogrammed as many times as needed, allowing the user to try several implementations or completely repurpose the device, for instance, a device used to train machine learning neural networks can be reprogrammed to perform image processing tasks. Usually the time needed to reprogram a FPGA, when it is possible, is between hundreds of milliseconds and a couple of seconds. A quick comparison between integrated circuits and FPGAs is that producing integrated circuits is really expensive but on a large scale those costs are offset by their speed and power efficiency, on the other hand, FPGAs are cheaper but slower and more power demanding. FPGAs are designed to perform specific repetitive tasks faster than a general purpose CPU, in the same way that GPUs are used to perform parallel graphics computing, FPGA can also be used to offload the CPU and speed up the computing. The underlying technology in FPGAs is a network of programmable lookup tables, that can compute any function up to k variables, the connections between LUT can also be rewritten, meaning that in general the output of one LUT can be used in any other LUT as an input. Notice that an AIG is unable to uniquely represent this type of network, and as we will see later other formats are used.

On Figure 1 we observe the island-style architecture of FPGAs that consists in a matrix of connected Configurable Logic Blocks (CLB). As we can see, there are two types of controllers for signal routing, Connection blocks, that manage the I/O between the FPGA and other devices and the connections between CLBs, and that are able to connect any pair of tracks, including those that are in the same channel. In this architecture, the lookup tables are placed inside CLBs.

Prior to the existence of FPGAs, the available technologies were Programmable Array Logic (PAL), a device that allows to implement functions using SoP, and Complex Programmable Logic Devices (CPLD) that are able to store more complex functions than SoP using Read-Only Memory (ROM). Both devices are still extensively used in combination of FPGA, however, for complex functions FPGAs are the only choice.

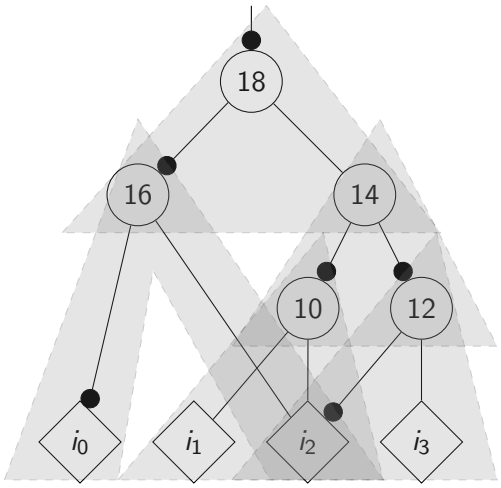


Figure 8: 2-LUT Mapping with area 5 and delay 3.

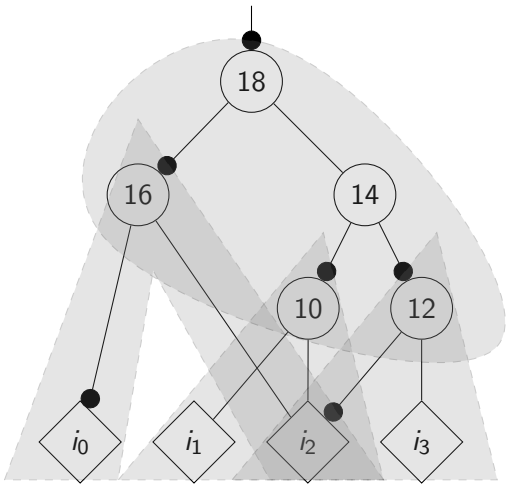


Figure 9: 3-LUT Mapping with area 4 and delay 2.

Technology mapping is the process of implementing a circuit using only a certain type of logic gates. In the scope of this project, the goal of technology mapping is to implement an AIG that uses inverters and 2-AND using k-LUT. In Figure 8 and Figure 9 we observe how the size of LUTs, in terms of number of variables, produces mappings with different area and delay. Notice that Figure 9 can be mapped using just 2 3-LUTs, as it is possible to join the LUTs of nodes 10 and 12, that use the inputs i_1,i_2 and i_3 using one LUT of three variables, this is the optimization problem that we analyze in this project.

1.1 Motivation

In the last few years, computer scientists have been warning that physical properties of transistors, such as the thickness of the insulator gate and the voltage cannot be further reduced and will set a hard lower limit on the size of transistors[6]. These bad news have a major impact in computing, for instance, power consumption is directly correlated with the size of transistors, which in turn limits the computation power and overheats the circuit. Assuming that these limitations do not allow to build smaller transistors, the only possibility left to lower the power consumption and increase the computational power is to further optimize the circuit, reducing as much as possible the implementation of the Boolean function with the minimum number of logic gates.

In FPGAs, the circuit is represented as a set of k-LUTs. Technology mapping is the last step of logic synthesis that performs the conversion from an AIG into k-LUTs. The two most common metrics for evaluating the quality of a mapping are the area, which is the overall number of LUT, and the delay, which is the number of LUT in the critical path. The critical path is the longest path, in terms of nodes, between primary inputs and primary outputs.

In this project, our efforts are towards minimizing the area, but as we will see later, in most cases, reducing the number of LUT also reduces the circuit delay. The problem of deciding if there exists a circuit

mapping of at most m LUTs is known to be NP-Complete for $k \geq 5$, and whether there exists an efficient algorithm for $k = 2, 3, 4$ is still an open question[7]. In general, the size of circuits is large and the problems that appear are complex, the usage of randomized algorithms and heuristics is a must. Exact solutions that rely on SAT solvers are computationally expensive and used only on crucial parts, such as equivalence checking.

1.2 Contributions

Despite the advancements in technology mapping for FPGA in the last few years, there is still a lot of room for improvement. Proof of this is the continuous appearance of new methods that improve particular circuits of the EPFL benchmark[8]. A summary of these techniques is provided in section 2.

The fundamental idea in our contribution is to segregate k-AND inputs, in such a way that the nodes that use similar variables become closer and those which are unrelated become isolated. This idea has a strong relation to k-LUT, where the limiting factor is not the complexity of the function, but rather the number of variables. This process can also be thought of in terms of logic sharing, there exist several techniques to merge and optimize similar nodes, when performing this process we also unlock new opportunities that would pass undetected. We propose and implement the algorithm AIGROT, which takes an AIG as input and outputs an equivalent AIG with reordered AND gates, without changing the number of ANDs, the number of inverters nor the number of connections between nodes. In the following Figure 10 we observe two equivalent AIGs with different structure. The AIG on the left has a shorter critical path than the one on the right, this is possible because AIG are not canonical representations of Boolean functions. This is an example of reordering AND gates, as we can see, the number of AND, inverters and connections are invariants.

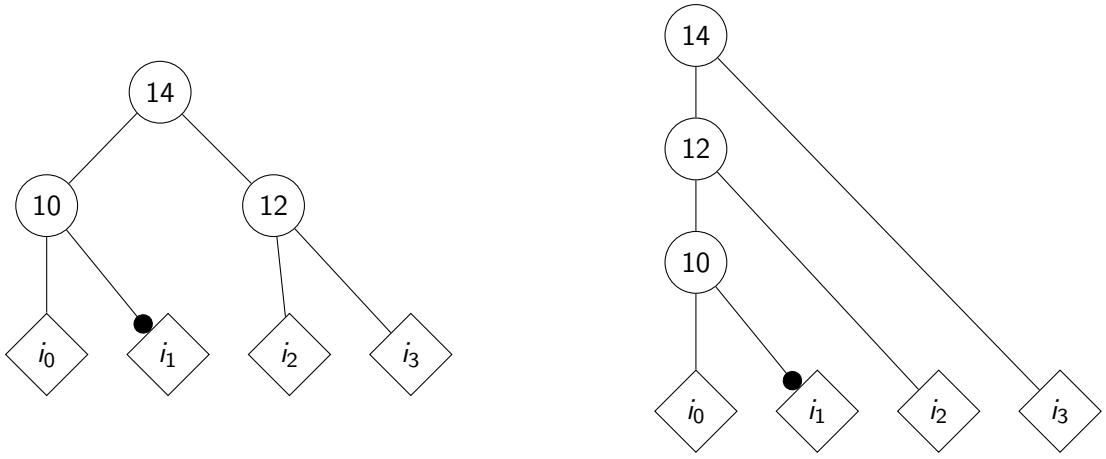


Figure 10: AIGs representing the function $f = AND(i_0, \bar{i_1}, i_2, i_3)$

An AND cluster is basically a collection of nodes which may or may not be inverted, connected through ANDs without any inverter in-between, that act like a k-input AND or k-AND. AIGROT takes advantage of the commutativity and associativity properties of the AND operation and reorders the nodes in the cluster according to different heuristics, with the objective of segregating and ordering the nodes by their similarity, it is a simple but transversal technique that can be applied to any AIG, before and after other techniques as the ones explained in the following section 2. This technique is technology independent, as it does not

restrict or depend on the mapping logic gates, in this case we analyze the effects of using it in k-LUT mapping.

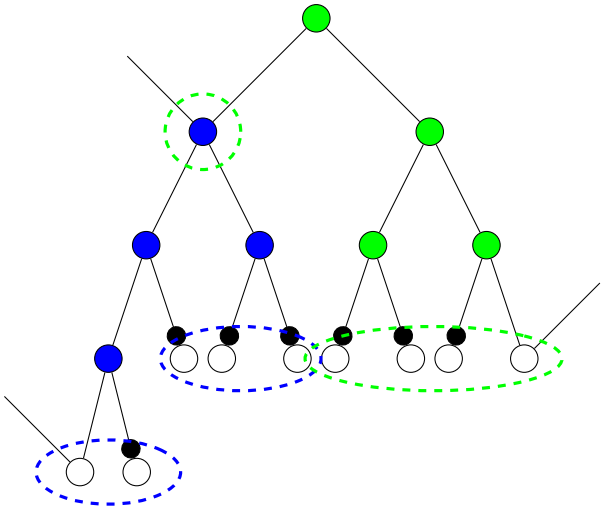


Figure 11: Example two overlapping AND clusters

When dealing with large AIG circuits, clusters can overlap each other. On Figure 11 we observe how two clusters delimited by negated and multiple fan-out nodes overlap. The dashed lines cover the cluster's inputs. Notice that the root of the blue cluster is an input of the green one. It is possible to swap the position of any two nodes inside the dashed region of the same colour, which reorders the inputs of the k-AND, and it is also possible to reorder the colored ANDs without changing the Boolean function of any output. These modifications are completely local and performing them does not have an impact on other parts of the AIG, this is good for speed and memory usage as it only needs a small subgraph of large AIG, however this also means that AIGROT is structurally biased, meaning that the original AIG determines excessively its effectiveness. Structural bias is minimized using an iterative process of a collection of techniques detailed in section 4.

This algorithm is implemented in C++ and uses the AIGER library developed in C for reading and writing the AIG files. We choose a low level language, because its interoperability with C and the performance and memory management in this type of applications are important. Large circuits have millions of nodes and efficiently modifying data structures becomes too expensive on high level languages. In section 3 there is a detailed discussion about the algorithm and design.

ABC is a System for Sequential Synthesis and Verification [9] that contains most state-of-the-art algorithms for logic synthesis. Our results in section 5 show that using AIGROT in combination of already implemented state-of-the-art techniques, we are able to obtain better results with the usage of AIGROT than without it. Our results show an improvement in the minimum area needed for the representation of several circuits over the best known results in the EPFL Benchmark[8].

1.3 Thesis structure

This thesis is organized as follows:

- *Background and Related work* A gentle introduction to state-of-the-art technology mapping and AIG

optimization techniques.

- *AIG Rotation* An in-depth explanation with examples of AIGROT, used heuristics and pseudo-code.
- *Methodology* Integration of AIGROT with other tools and definition of the pipeline of experiments.
- *Results* Evaluation of AIGROT with other techniques against the EPFL Dataset.
- *Conclusions and future work* Final conclusions and proposals for further research.

2. Background and related work

Traditional flow for FPGA programming begins by implementing the description of the circuit using a hardware definition language, typically VHDL or Verilog, which is then translated into logic gates and processed using logic synthesis. Technology mapping is the final step of logic synthesis, logic gates are mapped into target gates, which in FPGA are look-up tables of up to k variables. These steps are usually independent of each other, this is problematic because past decisions may impede to obtain better solutions in future steps, efforts are being made to merge these steps in such a way that it is always possible to reach the best mappings [10]. AIG with choices are a step in this direction, it is possible to define equivalent implementations of a node and let the technology mapping choose the implementation that better fits the goal, this is commonly referred as *loss-less technology mapping*.

Before technology mapping, most efforts are towards AIG optimization, which consists of AIG transformations that preserve functionality but help the technology mapper to produce better quality results with less delay and area. It is difficult to know ahead which AIG modifications will improve or decrease quality of the resulting technology mapping, naive heuristics such as the number of AIG nodes or the depth of the critical path are good estimators, but not perfect. This reinforces the idea of using *loss-less technology mapping*, preserving intermediary optimizations as equivalent nodes can improve the mapping.

This section is organized in five parts, first we introduce SAT solvers, that are a fundamental part of state-of-the-art logic synthesis and the basic concepts of And-Inverter graphs, then we enumerate fast, effective and commonly used algorithms for AIG optimization, then, we describe other techniques that work on already mapped networks. The last two sections include an introduction to ABC along with a summary of the commands that will be used to describe the optimization pipeline in section 4 and a detailed list of the best mappings of the EPFL benchmark library.

2.1 SAT and SAT solvers

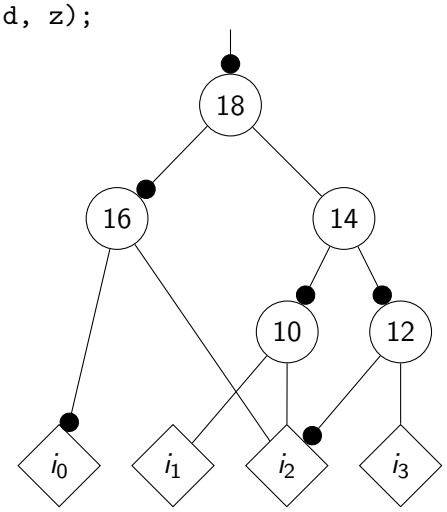
The Boolean Satisfiability Problem or SAT is the decision problem that given a Boolean formula asks whether there exists an assignment of the input variables such that the evaluation of the formula is true. Usually SAT instances are represented in Conjunctive Normal Form (CNF): an `AND` of `OR`s. i.e. $(x_1 \vee x_2) \wedge (x_1 \vee ...) \wedge ...$, each chunk of `OR`s is called a clause. The hardness of an instance is not directly tied to the number of clauses or variables, however, the formulation of a particular problem can help the SAT solver dramatically.

SAT was the first problem to be proven NP-Complete and no polynomial-time algorithms are known for solving it, however, by means of heuristics and efficient implementations such as Minisat[11] or Glucose[12], large instances of more than a million clauses can be solved by commercial CPUs. SAT is closely related to logic synthesis and most optimization problems that appear in logic synthesis can be expressed nicely in terms of SAT. The principal usage is combinatorial equivalence checking, which consists on given two circuits decide if there exists an input that outputs different values, in a similar way, they are used for property checking and circuit minimization.

Reducing the time to solve SAT instances is crucial to achieve better circuits and technology mappings. Research for custom solvers and algorithms for cleverly translating circuit constraints into easier to solve CNF clauses is being done [13]. SAT Modulo Theories (SMT) are a generalization of SAT that asks whether a mathematical formula is satisfiable, these instances are at least as hard as SAT, but this approach is useful for problems such as placement optimization[14] as they are easy to represent as instances of SMT.

2.2 And-Inverter Graphs

```
module example(a, b, c, d, z);
    input a, b, c, d;
    output z;
    wire x, y, w, v;
    assign x = a & b;
    assign y = ~c & d;
    assign v = ~a & c;
    assign w = x | y;
    assign z = w | v;
endmodule
```




```
aag 9 4 0 1 5
2
4
6
8
19
10 4 2
12 8 7
14 13 11
16 6 3
18 17 14
```

Max var
Inputs
Latches
Outputs
Ands

Inputs

Output

ANDs

Figure 12: Example AIG from Verilog. $f = (ab) \vee (\bar{c}d) \vee (\bar{a}c)$.

An AIG is a DAG of `AND` nodes where each node has two childs (fan-in) and an index that uniquely identifies it. There are no restrictions in the number of parents of a particular node (fan-out). Each node has two literals, the positive l and the inverted or negative $\bar{l}$, these literals are represented in binary using the least significant bit as the sign, the whole index is left shifted before adding the sign bit, so no information is lost, if index_i is the i-th bit of index and s is the sign, then $\text{index}_n \ldots \text{index}_0 s$ is the respective literal. For instance, literal 10 represents node 5 with positive sign and literal 11 represents the negation of the output of node 5. As we can see in Figure 12, when the AIG is plotted, the number inside the nodes is the positive literal of the node. This trick for representing inverted literals facilitates the implementation of algorithms as they do not have to deal with inverter gates. Primary Input (PI) and Primary Output (PO) are also treated as nodes, hence they also have an index and two literals. PI/PO are specified as a list of literals apart from the network as we can see in Figure 12.

The human-readable ASCII representation of the AIG (AAG) on the right of Figure 12 is not the standard representation, instead the AIGER format defines a binary representation[3] that uses several tricks to encode the AIG in order to produce smaller files that are easier to read and write. Experimental results show that the improvement is better than just using general compression algorithms.

Dealing with large AIG of millions of nodes and connections can be cumbersome for most algorithms, next we describe the techniques *windowing*, *cuts* and MFFC that allow to divide the circuit using sub-graphs. The downside of these techniques is that they are structurally biased, meaning that it could be the case that not knowing all pieces of information prevent the algorithm from doing further improvements. However, repeating these algorithms in different sub-graphs of the AIG until convergence usually is enough to reduce structural bias.

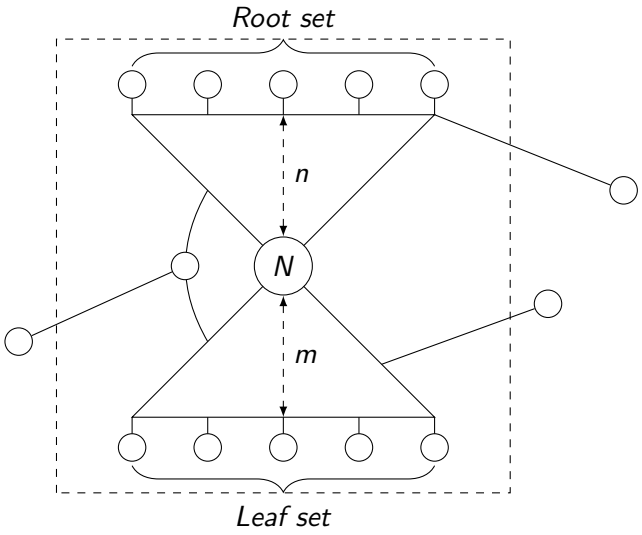


Figure 13: Window $n \times m$ at node N

Windowing [15] is a technique useful when dealing with large circuits, it is based on the idea of using a subgraph of the original AIG. A pair of nodes are at distance k if the number of edges that form the shortest path between them is k. The window $n \times m$ of a node N is the set of nodes that are in a path between the root set and the leaf set. The leaf set is the set of nodes below N at distance m, similarly the root set are the nodes above N at distance n. Typically, algorithms use windows of at most $m, n \leq 10$.

MFFC Maximum Fan-out Free Cone[15] of a node N is the set of nodes that contain N and all the nodes that always pass through N to reach the circuit outputs. This is useful specially when a node N is removed or substituted, because its MFFC can also be removed, this is a common technique in AIG optimization. Informally, the MFFC of a node is the logic that is not shared and is only used to compute the partial Boolean function of such node.

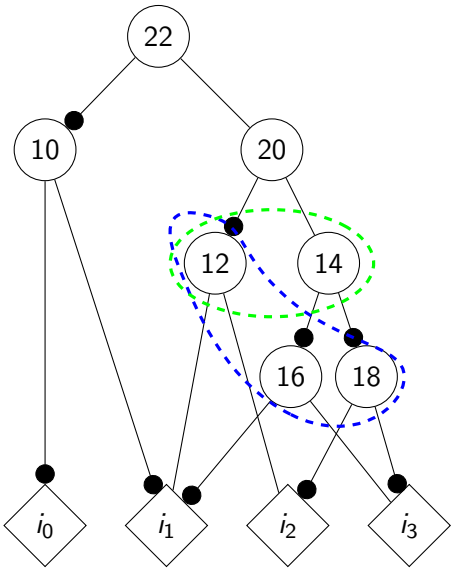


Figure 14: Examples of 3-Feasible and 2-Feasible cuts for node with literal 20 in blue and green respectively.

Cuts and k-Feasible Cuts [16] A cut of a node N is a set of nodes below N such that any path from inputs to N passes at least through one node in the cut. k-Feasible Cuts are cuts of at most k nodes. k-Feasible cuts are important because of their role in k-LUT mapping and AIG rewriting, the main issue is that there are too many, in a network of n nodes there are n^k k-Feasible cuts[16]. Recent techniques can reduce the number of k-Feasible Cuts[16] by keeping only a representative subset.

Functionally Reduced AIG (FRAIGs)[17] are AIG where no two nodes n_1, n_2 compute the same function $f_{n_1}(x) \neq f_{n_2}(x)$ and $f_{n_1}(x) \neq \overline{f_{n_2}(x)}$. The underlying idea behind FRAIGs is to restrict the AIG representation removing functionally equivalent nodes so it is more canonical. FRAIG representations of AIG can also contain *choices*, equivalent representations of nodes, usually coming from intermediate steps of logic synthesis. This information is useful for other steps of logic synthesis, from equivalence checking to technology mapping. The number of nodes and lengths of paths are good estimators to predict the quality of technology mapping, but in some cases reducing the number of nodes increases the area and in the same way, reducing the height of the AIG can increase the delay of the mapping. Using FRAIGs to keep the original representation and adding as choices the functionally equivalent nodes is a procedure that guarantees that technology mapping will not be worse after modifying the AIG.

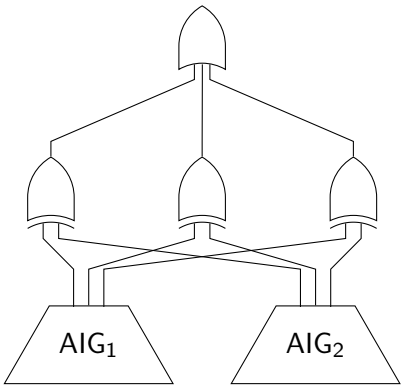


Figure 15: Miter from AIG_1 and AIG_2

Combinatorial Equivalence Checking (CEC)[18] Checking that two circuits are equivalent is a crucial part of EDA, unluckily this problem is intractable, but still possible to solve for circuits of millions of nodes using specific SAT solvers. A randomized algorithm is also used to speed up the detection of non-equivalent circuits, a random input is fed as input to both AIG and the circuit is simulated. In case that both circuits output the same value for all random inputs, a SAT solver is used to check if any other input can produce different output values, this is a resource intensive task and the number of SAT calls should be minimized. By means of a SAT solver it is possible to get the inputs that give different outputs in two supposedly equivalent circuits, to do so a miter Figure 15 is used, a miter consists of an `OR` of the `XOR` of all outputs with the same name in both circuits. Converting this circuit to clauses and feeding them to the SAT solver will output UNSAT if there is no input that produces a different output in both AIG, hence confirming the hypothesis that both are equivalent. The idea behind this procedure is that the `XOR` of two inputs is one if and only if they are different.

2.3 Common optimization techniques

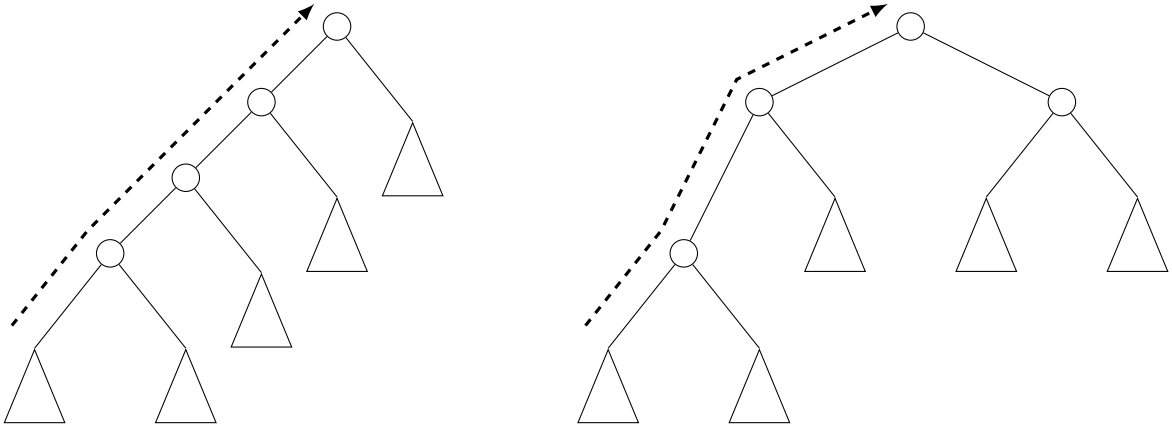


Figure 16: Equivalent unbalanced AIG with depth 4 and balanced AIG with depth 3, from left to right.

Balancing [19] is technique used to reduce the circuit delay reorganizing nodes of the AIG in such a way that the distance from any node to the output is minimized. As we will explain in subsection 3.3, balancing is a technique similar to the opposite of AIGROT as one technique cancels out the other. Balance uses the AND commutativity to reduce all paths from inputs to outputs, this procedure, in some cases prevents the LUT mapping from reaching an optimal solution in terms of area, as we will show in section 3. On Figure 16 we observe how AIG balancing can reduce the delay of the critical path.

Rewriting is a greedy algorithm that runs in linear time to optimize AIGs. The goal of rewriting is to reduce the number of nodes while preserving the depth of the circuit. It works by replacing cuts of 4 variables with optimal precomputed sub-graphs [20][21] using their NPN equivalence class[22]. Two functions belong to the same NPN class if one can be obtained from the other negating or permuting the inputs or negating the output. Using SAT solvers it is possible to quickly check NPN equivalence of functions with hundreds of inputs[23].

Refactor is a technique similar to rewrite, conceptually and in terms of time, but performs deeper and less structurally biased adjustments[21]. Refactor heuristically selects cuts of at least 10 nodes, then increases the window size and after some heuristics to select only windows that can be optimized, it is transformed to SoP and factored, after that, the sub-graph is processed using rewriting and if the resulting AIG is better in terms of nodes and depth, the cut is substituted with the new representation[22].

Resubstitution implements the function of a particular node using other available nodes in the circuit [21]. When a node is replaced, the whole MFFC of that particular node can be removed, in most cases this is not entirely possible, however using other nodes and a bit of logic, it is often possible to reduce the number of nodes in the MFFC, this process is called k-resubstitution, a generalization of resubstitution that partially replaces the MFFC.

2.4 Technology mapping optimization

All techniques discussed in previous sections are technology independent and can be applied to any AIG regardless of the technology library. In this section, we present just one technique that we experimentally got good results with, that is employed on already mapped circuits.

SAT LUT[24] is a SAT based technique that takes a circuit already mapped as k-LUT and tries to reduce its area without increasing the delay. This procedure uses windows of LUTs, starting with only one LUT and at each step adding the adjacent LUT that adds the least possible number of nodes. When the maximum window size is reached, then it defines a set of constraints in CNF restricting the number of LUTs to be one less, if the SAT solver is able to find a solution, the restriction is lowered again until the minimum number of LUTs is reached. This is done for all windows of LUTs in the mapped circuit. This process seems very slow, specially considering the fact that is calling a SAT solver multiple times for each window, however, for small windows of at most 128 nodes it is reasonable and the results are usually good.

2.5 ABC

ABC[9] is an open source software for logic synthesis and formal verification widely used in academic research. ABC incorporates state-of-the-art algorithms for AIG optimization and technology mapping, including every technique presented in this section. In practice, most algorithms are not independently used, and an iterative process repeatedly applies them until no improvements are made. The alias *resyn3* (resynthesis) command performs area optimization without increasing delay, its defined as `b; rw; rf; b; rw; rwz; b; rfz; rwz; b` which is a combination of the following commands `b`: *Balance*, `rw`: *Rewrite* and `rf`: *Refactor*, the `z` is a parameter that allows zero-replacements for further optimization [15].

The package ABC9 is the new version of ABC that it is available from within ABC, it is possible to share AIGs and mappings between both versions without exiting the program. ABC9 has optimizations that run faster and new methods for logic synthesis, typically ABC9 commands start with & for instance `&if, &read, &satlut`. The *SAT LUT* command is only available in ABC9. Another improvement of this version of ABC is allowing technology dependent mapping using custom properties for each type of LUT, but in the scope of this work, all LUTs are assumed to be equal.

2.6 EPFL Benchmark results

In this section, we show the best currently known mapping for all circuits in the EPFL benchmark library, along with the technique used to obtain it. As we can see, each technique is capable to improve only a few circuits. This is to be expected as they use heuristics that might perform well in a particular case, but produce poor results on others.

Filename	Area	Delay	Tool
Divisor	3248	1194	HIMap
Sine	1205	61	LUT-optimization
Decoder	264	2	Support-Reducing Decomp
int to float converter	24	4	Resubstitution
Barrel shifter	512	4	ABC Extreme Mapper
Priority encoder	100	26	HIMap
Lookahead XY router	50	5	Support-Reducing Decomp
Square-root	3027	1096	HIMap
Alu control unit	27	2	PIMap
Square	3232	76	LUT-optimization
Memory controller	2019	21	LUT-optimization
Adder	185	119	scaleSyn mapper
i2c controller	200	10	Support-Reducing Decomp
Hypotenuse	39826	4492	LUT-optimization
Voter	1279	19	scaleSyn mapper
Multiplier	4792	114	scaleSyn mapper
Max	522	189	Boolean Methods
Coding-cavlc	68	3	Support-Reducing Decomp
Log2	6513	132	LUT-optimization
Round-robin arbiter	304	80	scaleSyn mapper

Figure 17: Best currently known 6-LUT mappings for EPFL benchmark library. Source[25]

Several methods mentioned in Figure 17 combine different techniques in an iterative process, in a similar manner, our solution AIGROT also exploits the benefits of performing multiple optimization rounds. On the next Figure 18, we include a list of these techniques and their description.

Tool	Authors	Description
HIMap	Xing Li et al.	Based on heuristic and iterative mapping combined with improved logic optimization and post-mapping methods in ABC.
LUT-optimization	L. Amaru et al.	Based on LUT optimization[26]
Support-Reducing Decomp	L. Machado and J. Cortadella	Based on the algorithm [27]
Resubstitution	I. Lemberski et al.	Based on the resubstitution method presented in [28]
ABC Extreme Mapper	Robert K. Brayton & Alan Mishchenko	Interactive optimization using a variaty of optimization scripts in ABC
PIMap	Gai Liu & Zhiru Zhang	Based on parallelized iterative improvement mapping presented in[29]
scaleSyn mapper	Longfei Fan and Chang Wu	Area-oriented technology mapping combined with pre- and post-mapping logic optimization methods in ABC
Boolean Methods	E. Testa et al.	Algorithm presented in[30]

Figure 18: Authors and summary of techniques that improve current best known mappings of the EPFL Benchmark library. Source[25]

3. AIG Rotation

AIGROT is a technology-independent clustering algorithm that rewrites internal connections of k-AND that are present in AIGs aiming to improve the quality of technology mapping in terms of number of LUTs. AIGROT is not the opposite of balancing, they share one fundamental idea that is gate reordering, that takes advantage of the associative and commutative properties $(x_1x_2)x_3 \rightarrow x_1(x_2x_3)$, but with a crucial difference which is that there is only one way place the internal AND gates in a balanced AIG and multiple ways of placing them in an unbalanced one. Gate reordering in AIGs is fast, and in case of balancing the computation is minimal, on the other hand the number of possible binary trees of size n is large as we can see in Figure 19. Enumerating all possible binary trees of a particular cluster it is not feasible even for small circuits, especially considering the fact that AIGROT is meant to be used multiple times in the optimization process.

AIG has an ordering property that establishes that input literals of ANDs are in ascending ordering. This ordering removes redundancy and does not have an impact in technology mapping, but it does reduce the search space as we can see on the following Figure 19. As we will see in the details of the algorithm, for internal ANDs, this property is ignored and literals are recomputed at the end.

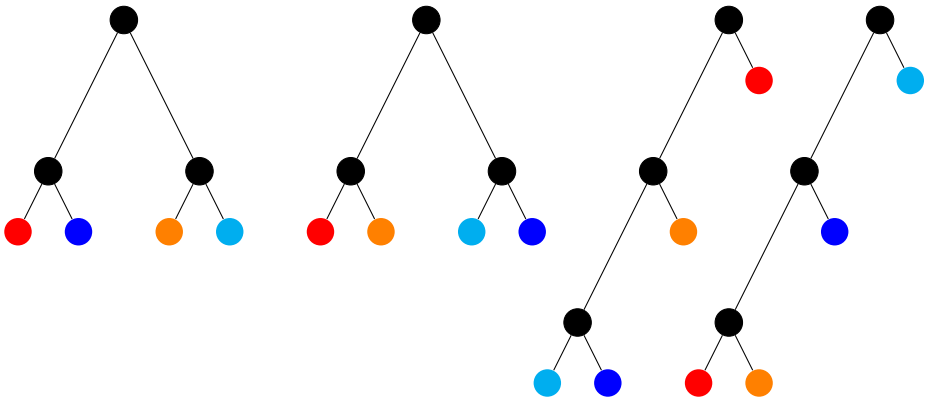


Figure 19: Examples of reordering of ANDs of a 3-size cluster. Black nodes are the internal ANDs, coloured nodes represent the distinct inputs of the 4-AND. Notice that for any pair of inputs of the cluster in the same AND the order does not change. In this case, the ordering is red $<$ orange $<$ cyan $<$ blue.

In logic synthesis there is always a trade-off between area and delay, the technique presented in this thesis aims only at reducing the area of the resulting mapping. However, experimental results show that when the size is reduced the depth of the circuit also decreases. In any case, the delay will be much worse than the minimum depth equivalent circuit which usually has a critical path orders of magnitude smaller, at the cost of a much larger area.

For purposes of evaluating the time performance of AIGROT, clustering detection and AIG processing can be ignored as it is negligible and unavoidable, the principal variable is the selected heuristic. Our proposal is really fast, it runs in a couple seconds even on largest networks from the EPFL benchmark. This property makes it suitable to be used with other logic synthesis algorithms, in our experiments we used an iterative process until convergence detailed in section 4.

3.1 Motivating example

In this section we present one example of a situation where current technology mapping with 3-LUT does not produce the optimal result with 2 LUTs and instead uses 3 LUTs. Common techniques for logic synthesis such as rewriting, resubstitution or refactoring are not able to produce an AIG that can be mapped using only 2 LUT, which is the optimum. In Figure 22 and Figure 21 we represent the function $f(i_0, i_1, i_2, i_3, i_4) = (i_4 + i_0)(\bar{i_1} + \bar{i_2})(\bar{i_3} + i_1)(i_2 + i_3)$ using two equivalent AIGs. Notice that nodes 22, 14 and 20 behave like a 4-input `AND`, and the only difference between them is that the inputs are reordered.

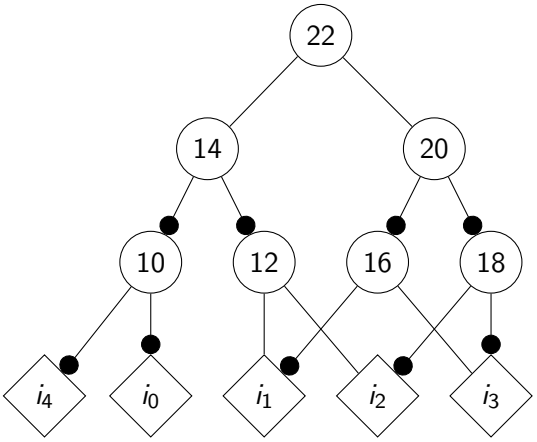


Figure 20: Balanced AIG

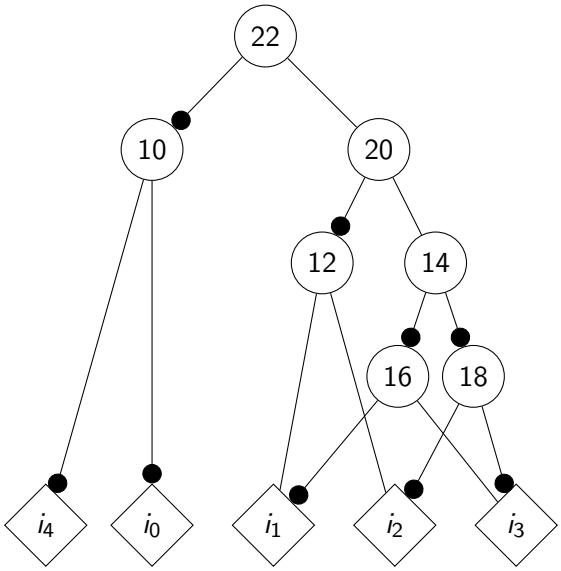


Figure 21: Rotated AIG

Technology mapping using 3-LUT produces the following results:

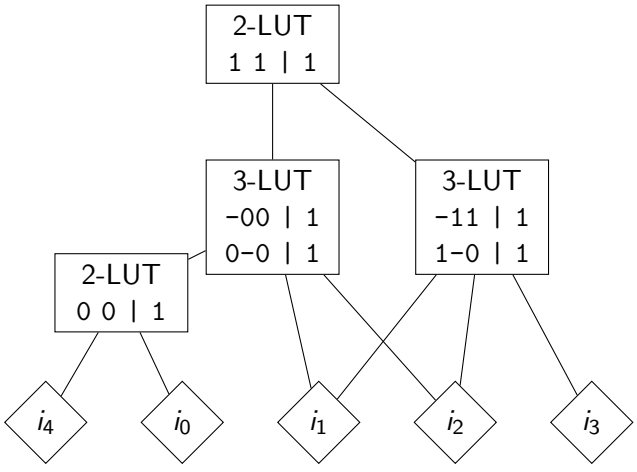


Figure 22: Balanced AIG with area-driven technology mapping *if -a -K 3*

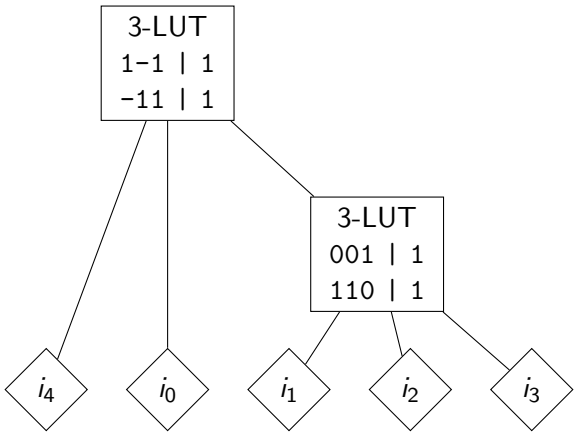


Figure 23: Rotated AIG mapping with *if -a -K 3*

As we can see in Figure 23 and Figure 22 the ordering of the 3-AND clustering has an impact in the number of LUT needed in technology mapping, the balanced AIG on the left has a mapping with 4 LUT, on the other hand the unbalanced AIG has a better mapping with size 2. This result can be accomplished using our implementation of AIGROT. There is an step-by-step explanation of AIGROT over this example in the following subsection 3.3.

3.2 Statistics of AND clusters on the EPFL benchmark

We have shown that this technique can improve technology mapping, in this section we evaluate the possible impact of using it in real circuits. To do so, we use the circuits of the EPFL Benchmark[8], and analyze the quantity and size of AND clusters of at least 3 inputs that are present in each one.

Filename	No. k-ANDs	No. of internal ANDs (mean)
Divisor	300	2.16
Sine	663	3.67
Decoder	0	0
int to float converter	52	2.21
Barrel shifter	384	2.84
Priority encoder	26	2.23
Lookahead XY router	19	10.37
Square-root	3083	2.08
Alu control unit	12	2.08
Square	334	2.0
Memory controller	9466	2.06
Adder	0	0
i2c controller	246	2.24
Hypotenuse	8135	2.0
Voter	457	2.0
Multiplier	1462	2.85
Max	294	2.42
Coding-cavlc	89	2.15
Log2	3902	3.83
Round-robin arbiter	129	2.47

Figure 24: Number of k-ANDs of $k \geq 2$ in EPFL Benchmark circuits

We observe that multiple fan-in ANDs are common in most circuits, and that the number of possibilities of rearranging their inputs is large, hence trying all combinations would be too time consuming. In cases *adder* and *dec* the number of multiple fan-in ANDs is zero, this does not mean that this technique can not do anything to improve this particular circuit mapping. Our proposal in section 4 consists in applying AIGROT in an iterative sequence of algorithms such as rewriting, refactoring and it is highly probable that the number of clusters increases or at least is more than zero in an intermediary step.

3.3 Algorithm and Implementation

In this section we define AIGROT and the data structures used to compute the proposed heuristic. This section is divided in two parts, first we introduce definitions and perform overview of the algorithm applied to the motivating example in subsection 3.1. Afterwards, we formally define the algorithm and detail the data structures used.

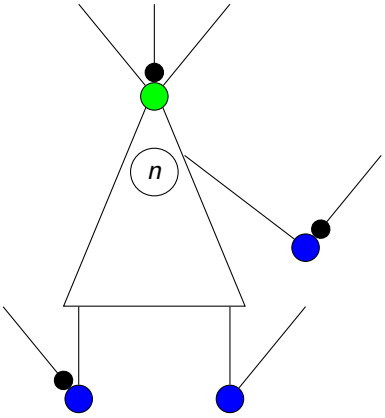


Figure 25: AND cluster of node n. Notice that node n is not necessarily the root, which is represented as the green node. Blue nodes are not part of the AND cluster, they have multiple outputs and can not be reordered freely, instead they are part of the input.

We define the input_set of node n as the subset of primary inputs that are in the cut of node n, in the previous Figure 25 we want to know the primary inputs that are in the cuts of blue nodes. Similarly, we are interested in the other_set of node n, which is the set of primary inputs that are still used if the the cut of node n is removed from the circuit, in other words, the other_set of a node n is the union of the input_set of all other nodes in the circuit that are not in the cut of n. Technically, for each AND cluster we only consider the subset of nodes and primary inputs that are present in the cut of the root node. If there is not any path between a primary input and a particular k-AND, that input is not considered as it does not provide any information for reordering the gates and reorganizing the gates does not have any impact on that primary input. This means that the other_set of a node n in a particular k-AND cluster can be computed as the union of all the input_set of the rest of the k inputs.

The algorithm starts by finding all k-ANDs that are present the circuit. The first step to process a cluster is to remove internal ANDs, these nodes have exactly one fan-out and are not outputs of the circuit, with the exception of the root node which is a special case that can break both rules. These two mentioned properties are what allows to freely move and interchange the nodes without fear of modifying the function of any primary output of the circuit.

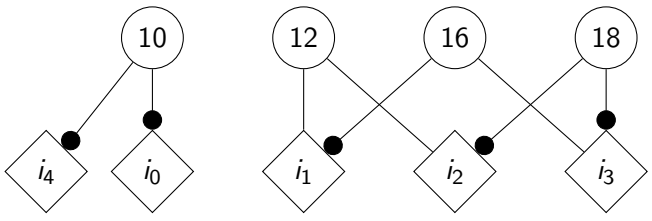
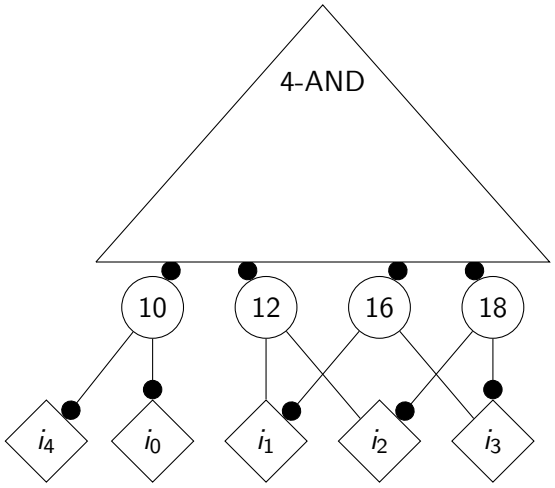


Figure 26: Before clustering

Essentially, AIGROT is a hierarchical agglomerate clustering (HAC) algorithm, hence we must define a node merging operation and a selection criterion, also known as distance between pairs of nodes. When two nodes are merged, an `AND` is placed in the circuit with those nodes as children, then `input_set` and `other_set` are computed for the new `AND` node.

When a pair of nodes p, q are merged, the `input_set` of the union is the union of their `input_set`. The `other_set` is not that trivial to compute, as it depends on the `input_set` of other nodes, there are more efficient ways to do this, but we do so applying the definition, we compute the union of all `input_set` skipping nodes p and q. For the heuristic criteria, we store how these sets evolve between iterations, we define $\Delta I = |\texttt{input\_set}_{pq}| - \textbf{max}(|\texttt{input\_set}_p|, |\texttt{input\_set}_q|)$ and $\Delta O = |\texttt{other\_set}_p \bigcup \texttt{other\_set}_q| - |\texttt{other\_set}_{pq}|$. ΔI is the increase of the `input_set` and ΔO is the decrease of the `other_set`. Both variables are positive integers. On the following Figure 27 we see an example of these variables in our motivating example.



Node	`input_set`	`other_set`
10	$\{i_0, i_4\}$	$\{i_1, i_2, i_3\}$
12	$\{i_1, i_2\}$	$\{i_0, i_4, i_1, i_2, i_3\}$
16	$\{i_1, i_3\}$	$\{i_0, i_4, i_1, i_2, i_3\}$
18	$\{i_2, i_3\}$	$\{i_0, i_4, i_1, i_2, i_3\}$

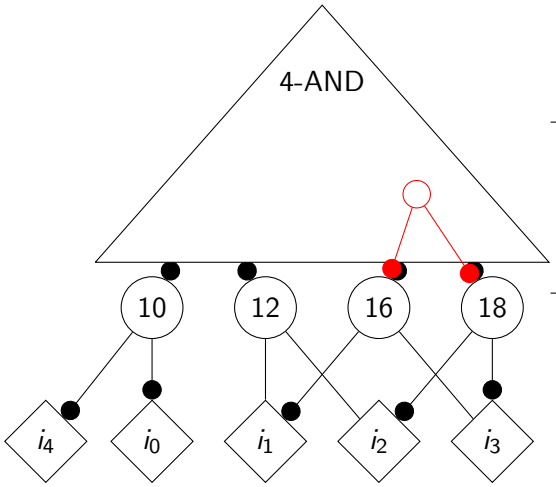
Pair	`input_set`	`other_set`	ΔI	ΔO	β
10,12	$\{i_0, i_4, i_1, i_2\}$	$\{i_1, i_2, i_3\}$	2	2	2
10,16	$\{i_0, i_4, i_1, i_3\}$	$\{i_1, i_2, i_3\}$	2	2	2
10,18	$\{i_0, i_4, i_2, i_3\}$	$\{i_1, i_2, i_3\}$	2	2	2
12,16	$\{i_1, i_2, i_3\}$	$\{i_0, i_4, i_2, i_3\}$	**1**	1	2
12,18	$\{i_1, i_2, i_3\}$	$\{i_0, i_4, i_1, i_3\}$	**1**	1	2
16,18	$\{i_1, i_3, i_2\}$	$\{i_0, i_4, i_1, i_2\}$	**1**	1	2

Figure 27: Computing the sets of each node and the heuristics of all pairs of nodes.

On the previous Figure 27 there is a last colum β, this is the last parameter of the heuristic and is defined as $\beta = |\texttt{other\_set}_{pq} \cap \texttt{input\_set}_{pq}|$. Our goal is that the intersection of the `input_set` and the `other_set` is minimum, the underlying idea is to increase the isolation between parts of the circuit that do not share variables.

Finally, having ΔI, ΔO and β the heuristic first selects the pairs that have the minimum ΔI, remember that a k-LUT can compute any circuit of k variables, hence the most prioritary operation is to not increase the number of variables. If there is more than one node that has the minimum ΔI, the heuristic checks ΔO, hence adding the minimum number of variables that decrease the complexity of other nodes. Again, if there is still more than one pair to choose, it uses β to pick the pair that isolates the most the variables between the `input_set` and the `output_set`. If there is still more than one node to choose, it takes the last.

On Figure 27 the first step can pick any of the pairs {12,18}, {16, 18}, {12, 18} as they have the same priority, we pick the last one for the following step, but in terms of technology mapping it would be the same if we choose any other one.

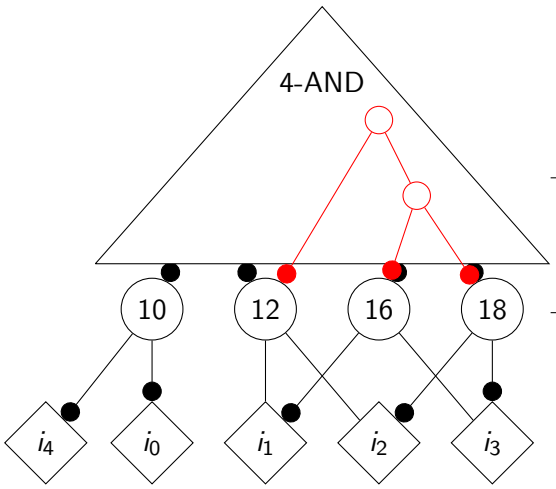


Node	input_set	other_set
10	$\{i_0, i_1\}$	$\{i_1, i_2, i_3\}$
12	$\{i_1, i_2\}$	$\{i_0, i_4, i_1, i_2, i_3\}$
(16,18)	$\{i_1, i_2, i_3\}$	$\{i_0, i_4, i_1, i_2\}$

Pair	input_set	other_set	ΔI	ΔO	β
10,12	$\{i_0, i_4, i_1, i_2\}$	$\{i_1, i_2, i_3\}$	2	2	2
10,(16,18)	$\{i_0, i_4, i_1, i_2, i_3\}$	$\{i_1, i_2\}$	2	3	2
12,(16,18)	$\{i_1, i_2, i_3\}$	$\{i_0, i_4\}$	0	3	0

Figure 28: Second step of the clustering with computed heuristics

As we can see on Figure 28, the heuristic for the last pair $12, (16, 18)$ is perfect, ΔI is zero, meaning that we do not add new variables, we decrease the `other_set` as much as possible and the intersection between the `input_set` and the `other_set` is zero, hence this cluster is fully isolated from the rest.



Node	input_set	other_set
10	$\{i_0, i_4\}$	$\{i_3, i_1, i_2\}$
(12,16,18)	$\{i_1, i_2, i_3\}$	$\{i_0, i_4\}$

Pair	input_set	other_set	ΔI	ΔO	β
(12,10,16,18)	*All*	$\{\emptyset\}$	2	5	0

Figure 29: Third step of the clustering with computed heuristics

On Figure 29 the last step is computed, there is only one choice that is to merge the nodes 10 and $(12, 16, 18)$. Notice that on the last step the `other_set` is empty and β is zero. The result is presented on the following Figure 30.

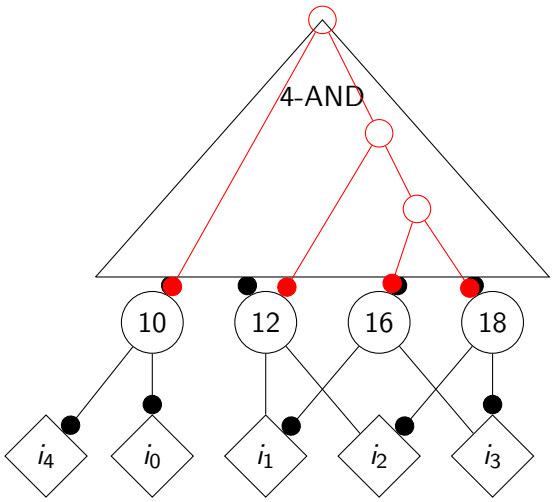


Figure 30: Final result

Now that we have seen how does it work on an example and how accomplishes its task, we formally describe how this is computed and implemented.

First we need the `input_set` of all inputs of the *k*-`AND`, A naive approach that works well is to store in all nodes a vector of bits, called the `input_set` of length equal to the number of inputs of the circuit. Bit *k* of node *n* is 1 if and only if there is path from input *k* to node *n* or in others words, the primary inputs of cut *n*.

Algorithm 1 Compute input set of all nodes

```
1:  function ComputeInputSet(node)
2:      if node.type = input then                          ▷ Recursion base case
3:          node.inputSet = node.input
4:          return
5:      if node.inputSet.size > 0 then                     ▷ Avoid recomputing all childs if called again
6:          return
7:      l ← node.leftChild
8:      r ← node.rightChild
9:      ComputeInputSet(l)
10:     ComputeInputSet(r)
11:     node.inputSet ← l.inputSet ∨ r.inputSet
12:     return
```

The previous algorithm is called for all nodes in the AIG, in some cases starting from all outputs is not enough as there are nodes without parents that should also be computed, for example, if using *choices*. In fact, we do not need the input set of all nodes, only for the inputs of super `AND`s, but in terms of time and space it is not a big deal, and in any case, more than enough for the EPFL benchmark circuits.

Finding all `AND` clusters can also be done in linear time, similarly to finding the primary inputs of nodes, we visit each node that does not already belong to a cluster and start searching a cluster from there. This is done using the function `Expand`.

The function `Expand` obtains the largest cluster of nodes containing *n*. First, it traverses up the AIG

Algorithm 2 Find all AND clusters

```
1: function FindAndClusters(node)
2:     ands ← Set of all ANDs
3:     clusters ← ∅
4:     while ands.size > 0 do
5:         a ← ands.first
6:         c ← Expand(a)
7:         if c.size > 1 then                ▷ If the cluster only contains a it is not a super AND
8:             clusters.append(c)
9:         ands.remove(c)
```

until an inverter, an output or multiple parents are found. It then traverses down adding all nodes that are not outputs, do not have multiple parents and are connected without inverters in-between. The function returns all the nodes that form the *k*-AND including *n*. Notice that the top-most node in the cluster is the only one that can have multiple outputs and that the inputs of the *k*-AND are not included in the cluster.

The following algorithm Expand receives an AND node and traverses the AIG up until it reaches the top node in the cluster, this node is important because it is unique and contains the output of the *k*-AND (See the green node in Figure 25). Then it goes back, from the top node to the leafs of the cluster.

Algorithm 3 Expanding the cluster that contains an AND

```
1: function Expand(node, direction = up)
2:     if direction is up then
3:         parents ← getParents(node)   ▷ The sign function is 1 if the literal is inverted 0 otherwise
4:         while parents.size = 1 and !sign(edge between node and parent) do
5:             node ← node.parents[0]
6:             parents ← getParents(node)
7:             if node is output then break
           return Expand(node, direction = down)
8:     else
9:         cluster ← Set(node)
10:        if !(node.left is output) and getParents(node.left).size = 1 and !SIGN(node.left) then
11:            leftNodes ← Expand(node.left, direction=down)
12:            cluster ← merge(cluster, leftNodes)
13:        if !(node.right is output) and getParents(node.right).size=1 and !SIGN(node.right) then
14:            rightNodes ← Expand(node.right, direction=down)
15:            cluster ← merge(cluster, rightNodes)
           return cluster
```

At this point we have the ANDs and the input_set of their inputs. Using this information and an heuristic we reorganize the clusters. At this point we introduce the output_set(*n*) which is the set of PI that are present in the AIG if we cut node *n*. When rearranging a cluster the input_set and other_set change, the following function merges the input_set when merging two nodes and recomputes the other_set as the union of all input_set of other nodes.

Algorithm 4 Reorganizing Cluster

Require: All nodes have their `input_set` already computed

```
1:  function REORGANIZE(inputs, ands)      ▷ inputs of the k-AND, internal ANDs that form the k-AND
2:      groups ← inputs                          ▷ groups contains the set of nodes to be merged
3:      for i ∈ groups do                        ▷ Compute the other_set as the union of input_set
4:          for j ∈ groups where i ≠ j do
5:              groups[i].other_set ← MERGE(groups[i].other_set, groups[j].input_set)
6:      for i ∈ ands.size do
7:          pair ← PICKPAIR(groups)
8:          new_element ← MERGEPAIR(pair)
9:          groups ← groups.remove(pair)
10:         for j ∈ groups do                    ▷ The pair is already removed from groups
11:             new_element].other_set ← MERGE(new_element.other_set, groups[j].input_set)
12:         groups ← groups.add(new_element)
```

On the previous algorithm there are two new functions `PickPair` and `MergePair`. The first one chooses which pair of nodes of the cluster should be merged by using an `AND`. The function `MergePair` joins the `input_set` of both nodes and creates a new node with properties to use in the heuristic. The function `MergePair` can not compute the `other_set`, as it does not have access to other nodes, this is why the other set is computed afterwards in lines 10-11.

Algorithm 5 Merge pair of nodes in a clustering context

```
1:  function MERGEPAIR(pair)
2:      n ← EMPTYNODE
3:      n.input_set ← MERGE(pair[0].input_set, pair[1].input_set)
4:      n.input_set_increase ← |n.input_set| - max(|pair[0].input_set|, |pair[1].input_set|)
        return n
```

To choose a good pair of nodes to merge, we use the naive approach that takes all possible pairs and by means of an heuristic it selects the best candidate. This implementation can be further optimized caching the heuristic values of each pair, but practical experiments show that current implementation is quick enough for our tests and large AIGs.

Algorithm 6 Select a pair of nodes to merge in a clustering context

```
1: function PickPair(groups)
2:     best ← Pair
3:     found ← False
4:     for i ∈ groups do
5:         for j ∈ groups where i < j do          ▷ Merging groups (i, j) is the same as merging (j, i)
6:             n ← MergePair(groups[i],groups[j])
7:             for k ∈ groups where k ≠ i and k ≠ j do          ▷ Compute other set of node n
8:                 n.other_set ← Merge(n.other_set, groups[k].input_set)
9:             count ← |Merge(groups[i].other_set, groups[j].other_set)|          ▷ (1)
10:            n.other_set_decrease ← count - |n.other_set|
11:            n.beta ← |n.input_set ∩ n.other_set|
12:            if !found or IsBetter(n, best) then
13:                best ← n
14:                found ← True
       return best
```

(1) Here we compute the number of nodes that would be in other_set if these nodes where on different clusters. For example, if we have two nodes in a cluster and both have {2,3} as their input set, then their output_set will also have {2,3}. If we assume that there are no other nodes with {2,3} in their input_set, then joining them decreases the other_set by 2, this is in fact what we are computing: The union of the nodes will have other_set_merge = `other_set_1` $\bigcup$ `other_set_2` - {2,3}. Then `other_set_decrease` is 2.

At this point, we already have the solution space with all the possible combinations of choices of pairs of elements to join. The next step is to define a heuristic that joins the pair of nodes that produces the best mapping. In this case the heuristic is to first do not increase the input_set of the two nodes, and if there are more than one solution, pick the one that reduces the most the other_set. In case that there are still more than one solution, it chooses the one with lower β, which is the node that better separates the internal function from the outside node's functions. If there are multiple options available it chooses the last one.

Algorithm 7 Heuristic

```
1: function IsBetter(a, b)          ▷ If true, pair a is better than b
2:     if a.input_set_increase < b.input_set_increase then return True
3:     if a.input_set_increase > b.input_set_increase then return False
4:     if a.other_set_decrease > b.input_set_decrease then return True
5:     if a.other_set_decrease > b.input_set_decrease then return False
       return b.beta > a.beta
```

4. Experiment Design

AIGROT is a transversal technique and its performance increases when used in combination of others state-of-the-art synthesis algorithms. After analyzing the available algorithms in ABC and trying several combinations that have shown to provide good results against the EPFL benchmarks, we developed the following iterative process Figure 31.

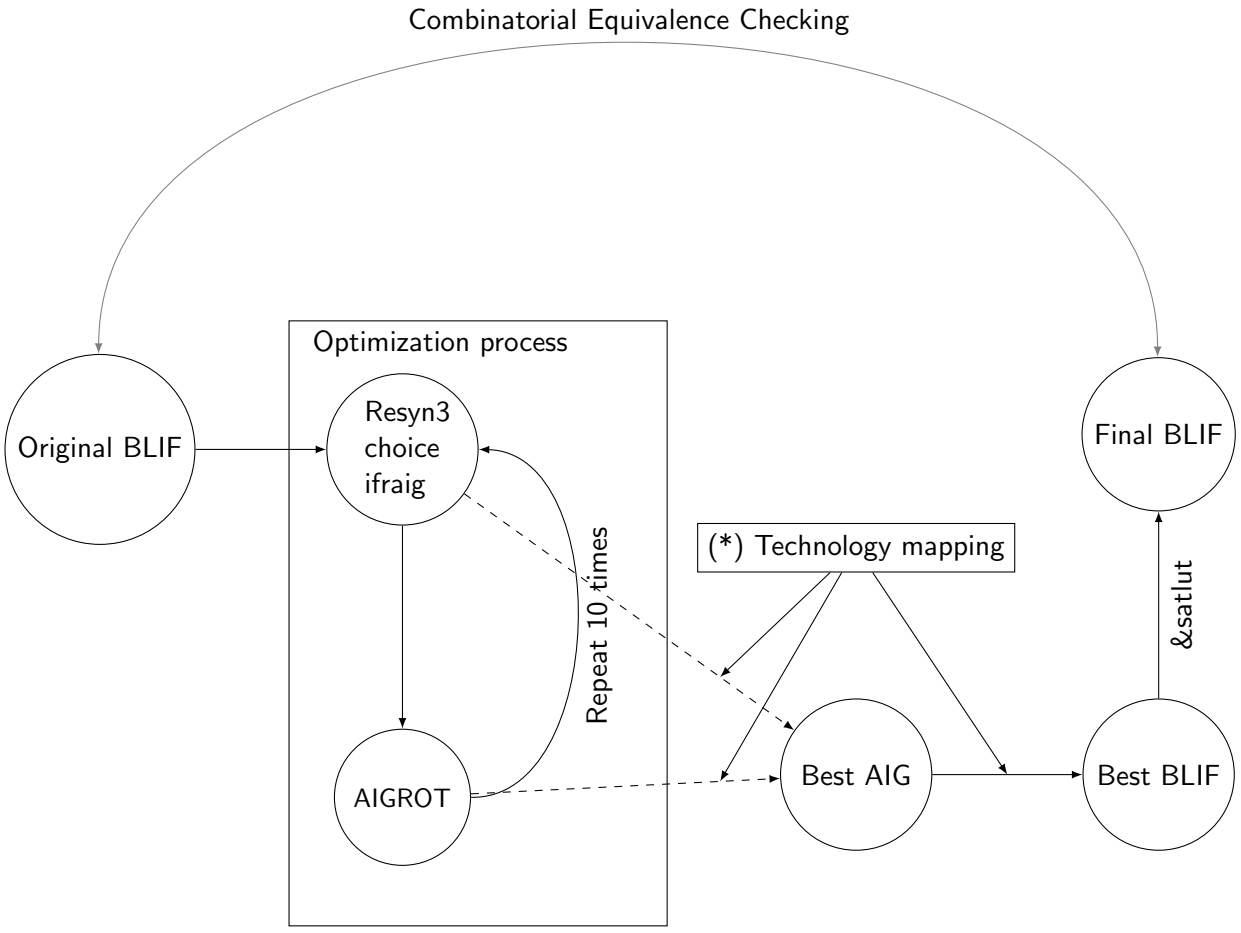


Figure 31: Experiment design

The process is divided in four principal blocks, the first one starts loading the original BLIF from EPFL Benchmarks, the second block is an iterative sequence of logic synthesis optimization, the third one is technology mapping optimization and the last step consists in checking that the final mapping is equivalent to the original circuit.

For some circuits, performing this experiments on the entire EPFL Benchmark is a bit slow, especially commands `&satlut` with a large window and `cec` for `hyp` circuit, however in a personal computer it is possible to run all the benchmark in a couple of hours.

4.1 Converting BLIF to AIG

Berkeley Logic Interchange Format (BLIF) it is the common format in academic research to represent k-LUT mappings, to improve any circuit we first undo the mapping in order to obtain an equivalent AIG to perform logic synthesis. Using ABC this is a straightforward process, first load the BLIF using `read filename.blif`, then convert it to AIG using the `strash` command, and write the AIG file using `write_aiger filename.aig`.

4.2 Logic synthesis

This is the iterative part of logic optimization, the steps of actions that are repeated is `resyn3`, `choice`, `ifraig` and AIGROT. As explained in section 2, the number of nodes and depth of the circuit is not a perfect estimator of the quality of the mapping. To not lose intermediary AIG that produce good mappings, we cache the best circuit at each iteration before and after applying AIGROT. ABC allows different types of technology mapping, area oriented and delay oriented. We use both techniques as in some cases we reach better area using delay optimization. The commands in ABC to perform the technology mapping are `if -a -K 6` for area driven technology mapping and `if -K 6` for delay technology mapping optimization.

4.3 Technology mapping

At this point, we already have the best AIG for technology mapping, we map the circuit to 6-LUT and use the command `&satlut` for further optimization of the circuit. The size of the window is set to 128, and the number of conflicts is limited to 10.000. As the authors of this techniques comment on their paper, often using such large window is not worth the effort and using windows of size 32 produces similar results, but for the sake of the experiment we used 128.

4.4 Combinatorial Equivalence Checking

The last step is checking that everything works and that the final circuit is equivalent to the original one. This task is performed using the command `cec` from ABC.

5. Results

The method performs 10 iterations of optimization but usually after 3 or 4 full iterations the AIG stops improving. In the following Figure 32 we observe how the mapping of *Square root* circuit keeps improving until convergence. The number of steps is 20 because we store the AIG before and after applying AIGROT, this is what causes the small peaks and valleys.

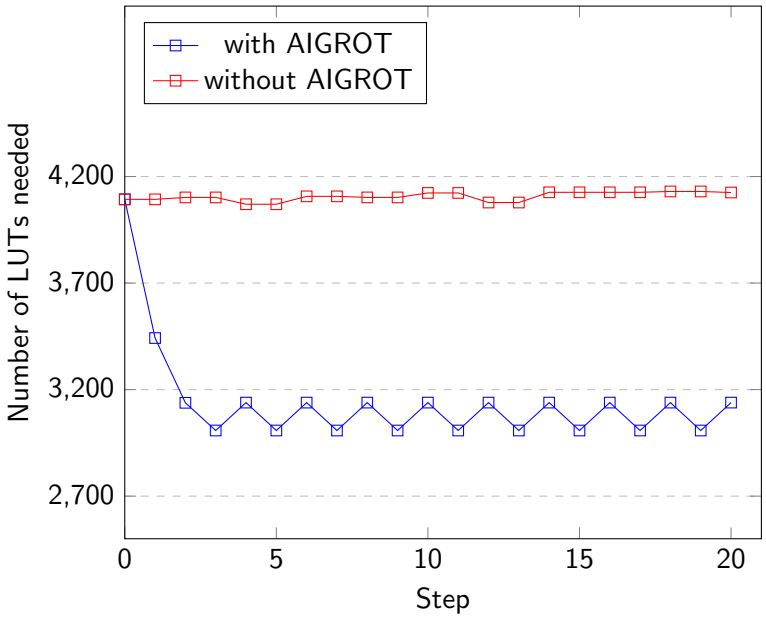


Figure 32: LUT mapping improvement on the *sqrt* circuit using *if -K* to get the statistics at each iteration, before and after AIGROT.

Previous Figure 32 shows how applying this technique improves both logic synthesis and technology mapping. In this particular case, we observe how AIGROT decreases the number of LUTs needed to map the circuit and then the next pass of *resyn3,choice,ifraig* deteriorates the solution, creating these small peaks. On other examples of these dataset, the peaks are made by AIGROT. This behaviour is anecdotal, as the usage of AIGROT usually improves the best solutions found or helps the other tools to find new local minimums.

In the following two tables we present the results of applying AIGROT among the techniques described in the previous section to the EPFL Benchmark. We split the results in two tables, the first one is area driven technology mapping and the second one is standard technology mapping. It's important to remark that best known results do not come from the same algorithm, in fact it is quite the opposite, most algorithms are specific and only improve a couple of mappings.

As the final LUT optimization provides a substantial improvement, we distinguish between the best mapping obtained using only logic synthesis (*resyn3, choice, ifraig, AIGROT*) from the overall best mapping using LUT optimization (*&satlut*).

File	Area	Depth	Area AIG Optimization	Area &satlut	Depth	Area improvement
Adder	185	119	244	192	64	-0.038
Barrel shifter	512	4	512	512	4	0.000
Divisor	3248	1194	4063	**3159**	1129	0.027
Hypotenuse	39826	4492	48051	**39799**	4931	0.001
Log2	6513	132	7473	6546	127	-0.005
Max	522	189	673	673	139	-0.289
Multiplier	4792	114	5672	**4594**	96	0.041
Sine	1205	61	1439	1240	65	-0.029
Square-root	3027	1096	3795	3104	1207	-0.025
Square	3232	76	3989	3241	67	-0.003
Round-robin arbiter	304	80	2599	2599	19	-7.549
Alu control unit	27	2	28	28	2	-0.037
Coding-cavlc	68	3	113	113	5	-0.662
Decoder	264	2	287	287	2	-0.087
I2c controller	200	10	285	283	10	-0.415
Int to float converter	24	4	46	46	8	-0.917
Memory controller	2019	21	10648	10532	48	-4.216
Priority encoder	100	26	237	197	56	-0.970
Lookahead XY router	24	5	71	71	10	-1.958
Voter	1279	19	1617	**1252**	17	0.021

Figure 33: Results using area driven technology mapping (6-LUT). The first two columns of area and depth belong to the smaller implementation currently know for each EPFL Benchmark. The third column is the number of 6-LUTs of the mapping of the best AIG that is obtained using logic synthesis. The following two columns are the final area and depth after using &satlut on the best AIG. Finally the last column is the improvement on the number of LUTs between columns 1 and 4.

On the previous Figure 33, we see that most results obtained by AIGROT are not far from best currently known mapping. Actually, we were able to improve the mapping of four circuits *Divisor*, *Multiplier*, *Voter* and *Hypotenuse*. However, it should be noted that in the case of *Hypothenuse*, our optimization pipeline is not improving the AIG and as we can see on column '&satlut' the improvement must be attributed to the '&satlut' tool. It is possible to further improve these mappings using more than one iteration of &satlut, but the improvement is a couple LUTs and we prefer to keep things simple. Notice also, that in all cases where the technique presented in this thesis improves the solution the delay of the solution is also reduced.

On the following table we repeat the previous process but using standard technology mapping instead of area driven technology mapping. One can expect to get worse values than before, since the goal is to reduce the area. However, using this method, we are able to reach a new best mapping for circuit *Square-root*. In general, results are worse, but still good considering that we are comparing against the best known mapping.

File	Area	Depth	Area (Phase 1)	Area (&satlut)	Depth	Area improvement
Adder	185	119	243	230	52	-0.243
Barrel shifter	512	4	512	512	4	0.000
Divisor	3248	1194	4672	4175	1091	-0.285
Hypotenuse	39826	4492	44625	40069	4328	-0.006
Log2	6513	132	7838	7272	101	-0.117
Max	522	189	757	735	88	-0.408
Multiplier	4792	114	5936	5480	71	-0.144
Sine	1205	61	1465	1386	47	-0.150
Square-root	3027	1096	**3008**	3008	1053	0.006
Square	3232	76	3967	3309	66	-0.024
Round-robin arbiter	304	80	2722	2722	18	-7.954
Alu control unit	27	2	28	28	2	-0.037
Coding-cavlc	68	3	118	115	5	-0.691
Decoder	264	2	287	287	2	-0.087
I2c controller	200	10	292	289	4	-0.445
Int to float converter	24	4	47	47	4	-0.958
Memory controller	2019	21	10964	10800	36	-4.349
Priority encoder	100	26	191	187	31	-0.870
Lookahead XY router	24	5	83	76	12	-2.167
Voter	1279	19	1489	1437	15	-0.124

Figure 34: Results using standard technology mapping (6-LUT). As in Figure 33, the first two columns of area and depth belong to the smaller implementation currently know for each EPFL Benchmark. The third column is the number of 6-LUTs of the mapping of the best AIG that is obtained using logic synthesis. The following two columns are the final area and depth after using &satlut on the best AIG. Finally the last column is the improvement on the number of LUTs between columns 1 and 4.

6. Conclusions and future work

6.1 Conclusions

AIGROT technology independent clustering algorithm that improves current state-of-the-art technology mapping by applying transformations to and-inverter graphs. Adding this technique to the set of algorithms for AIG optimization reduced the size of several circuits of the EPFL benchmark with little impact on runtime: The results improve the best known 6-LUT mapped circuits for *Divisor* reducing 2.7% the number of LUTs, *Multiplier* reduced by 4.1%, *Voter* 2.1% and *Square Root* a 6%. Our results show that this technique is not only useful to reduce the area, but that it can also reduce the delay. Additionally, in cases where our method is not able to beat the best current mapping, our results show that our solution quality approaches it.

Our contribution includes an heuristic to reorder the k-AND clusters using efficient data structures and fast algorithms especially designed to perform well in large circuits of millions of logic gates. Although this algorithm is specially locally biased, experimental results show that in an iterative process that contains other techniques, it is able to perform deep improvements to the overall circuit.

6.2 Future work

The circuits of the EPFL Benchmark may not be the best examples for discovering the true potential of this tool, most circuits are small and hard to further optimize using heuristics, since it is possible to apply computationally intensive algorithms to enumerate most combinations. Large circuits that contain larger than 10-AND clusters, are hard to optimize by the same means due to their intrinsic exponential computational complexity. It would be interesting to apply AIGROT to such circuits and study if the proposed heuristic improves the mappings. Along this path, it would also be interesting to implement this process in parallel treating multiple clusters at the same time.

The approach in the experiment involves saving a copy of the best AIG between optimization steps. However, AIG with *choices* can be used to store equivalent representations of each cluster and let the technology mapper to chose the most convenient one. In combination with the above, it would be interesting to implement other heuristics and store all results inside the same AIG.

Most tools that are being used in the optimization pipeline preserve the delay, it would be interesting to combine AIGROT with other techniques that focus explicitly on reducing size and ignore the delay of the circuit. Also, mixing standard technology mapping with area driven technology mapping is a path that is worth researching.

Acronyms

AIG And-Inverter graphs. 1, 3–8, 10–15, 18–20, 28–33

AIGROT AIG Rotation. 1, 7–9, 14, 16, 18, 20–22, 28–31, 33

BBDD Biconditional Binary Decision Diagrams. 4

BDD Binary Decision Diagrams. 4

BLIF Berkeley Logic Interchange Format. 28, 29

CB Connection blocks. 5

CEC Combinatorial Equivalence Checking. 13, 28

CLB Configurable Logic Blocks. 5

CNF Conjunctive Normal Form. 10, 15

CPLD Complex Programmable Logic Devices. 6

CPU Central Processing Unit. 5, 10

DAG Directed Acyclic Graph. 5, 11

EDA Electronic Design Automation. 3

FPGA Field-Programmable Gate Array. 1, 3, 5–7, 10

FRAIG Functionally Reduced AIG. 13

GPU Graphics Processing Unit. 5

HAC hierarchical agglomerate clustering. 22

IC Integrated circuits. 3

LUT Lookup Table. 1, 5–8, 13–16, 18–20, 29–33

MCSP Minimum Circuit Size Problem. 3

MFFC Maximum Fan-out Free Cone. 11, 12, 14

MIG Majority-Inverter-Graphs. 4

MOSFET metal–oxide–semiconductor field-effect transistor. 3

PAL Programmable Array Logic. 6

PI Primary Input. 11

PO Primary Output. 11

PoS Product of Sums. 4

ROBDD Reduced Ordered Binary Decision Diagram. 4

ROM Read-Only Memory. 6

SAT Boolean Satisfiability Problem. 7, 10, 13–15

SMT SAT Modulo Theories. 10

SoP Sum of Products. 4, 6, 14

SW Switching boxes. 5

TT Truth Tables. 3, 4

VHDL Very High-Speed Integrated Circuit Hardware Description Language. 3, 10

7. References

[1] Gordon E. Moore. Progress in digital integrated electronics [Technical literaiture, Copyright 1975 IEEE. Reprinted, with permission. Technical Digest. International Electron Devices Meeting, IEEE, 1975, pp. 11-13.]. *IEEE Solid-State Circuits Society Newsletter*, 11(3):36–37, 2006.

[2] John M. Hitchcock and Aduri Pavan. On the NP-Completeness of the Minimum Circuit Size Problem. In *FSTTCS*, 2015.

[3] Armin Biere. The AIGER And-Inverter Graph (AIG) Format Version 20070427. 2007.

[4] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. Majority-Inverter Graph: A novel data-structure and algorithms for efficient logic optimization. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2014.

[5] Luca Amarú, Pierre-Emmanuel Gaillardon, Subhasish Mitra, and Giovanni De Micheli. New Logic Synthesis as Nanotechnology Enabler. *Proceedings of the IEEE*, 103(11):2168–2195, 2015.

[6] Thomas N. Theis and H.-S. Philip Wong. The End of Moore's Law: A New Beginning for Information Technology. *Computing in Science Engineering*, 19(2):41–50, 2017.

[7] A.H. Farrahi and M. Sarrafzadeh. Complexity of the lookup-table minimization problem for FPGA technology mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(11):1319–1332, 1994.

[8] Luca Gaetano Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. The EPFL Combinational Benchmark Suite. 2015.

[9] Robert Brayton and Alan Mishchenko. ABC: An Academic Industrial-Strength Verification Tool. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 24–40, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[10] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. Integrating logic synthesis, technology mapping, and retiming. In *Proc. IWLS'05*. Citeseer, 2006.

[11] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT*, 2003.

[12] Gilles Audemard and Laurent Simon. On the Glucose SAT solver. *International Journal on Artificial Intelligence Tools*, 27:1840001, 02 2018.

[13] He-Teng Zhang, Jie-Hong R. Jiang, and Alan Mishchenko. A Circuit-Based SAT Solver for Logic Synthesis. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–6, 2021.

[14] Andrew Mihal. A Difference Logic Formulation and SMT Solver for Timing-Driven Placement. 2013.

[15] Alan Mishchenko and Robert K. Brayton. Scalable Logic Synthesis using a Simple Circuit Structure. 2006.

[16] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. Improvements to Technology Mapping for LUT-Based FPGAs. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, FPGA '06, page 41–49, New York, NY, USA, 2006. Association for Computing Machinery.

[17] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. FRAIGs: A unifying representation for logic synthesis and verification. Technical report, 2005.

[18] Alan Mishchenko, Satrajit Chatterjee, Robert Brayton, and Niklas Een. Improvements to Combinational Equivalence Checking. In *2006 IEEE/ACM International Conference on Computer Aided Design*, pages 836–843, 2006.

[19] J. Cortadella. Timing-driven logic bi-decomposition. *IEEE transactions on computer-aided design of integrated circuits and systems*, 22(6):675–685, Jun 2003.

[20] P. Bjesse and A. Boralv. DAG-aware circuit compression for formal verification. *Proc. ICCAD '04*, pages 42–49.

[21] Luca Amarú, Mathias Soeken, Patrick Vuillod, Jiong Luo, Alan Mishchenko, Janet Olson, Robert Brayton, and Giovanni De Micheli. Improvements to boolean resynthesis. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 755–760, 2018.

[22] A. Mishchenko, S. Chatterjee, and R. Brayton. DAG-aware AIG rewriting: a fresh look at combinational logic synthesis. In *2006 43rd ACM/IEEE Design Automation Conference*, pages 532–535, 2006.

[23] Mathias Soeken, Alan Mishchenko, Ana Petkovska, Baruch Sterin, Paolo Ienne, Robert Brayton, and Giovanni Micheli. Heuristic NPN classification for Large Functions Using AIGs and LEXSAT. pages 212–227, 07 2016.

[24] Bruno Schmitt, Alan Mishchenko, and Robert Brayton. SAT-based area recovery in structural technology mapping. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 586–591, 2018.

[25] Luca Gaetano Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. EPFL Benchmark. `https://github.com/lsils/benchmarks`, 2015.

[26] Luca Amarú, Vinicius Possani, Eleonora Testa, Felipe Marranghello, Christopher Casares, Jiong Luo, Patrick Vuillod, Alan Mishchenko, and Giovanni De Micheli. LUT-Based Optimization For ASIC Design Flow. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 871–876, 2021.

[27] Lucas Machado and Jordi Cortadella. Support-Reducing Decomposition for FPGA Mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(1):213–224, 2020.

[28] Igor Lemberski, Artjoms Suponenkovs, and Marina Uhanova. LUT-Oriented Asynchronous Logic Design Based on Resubstitution. In *2019 14th International Conference on Design Technology of Integrated Systems In Nanoscale Era (DTIS)*, pages 1–4, 2019.

[29] Gai Liu and Zhiru Zhang. A parallelized iterative improvement approach to area optimization for lut-based technology mapping. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 147–156, New York, NY, USA, 2017. Association for Computing Machinery.

[30] Eleonora Testa, Luca Amarú, Mathias Soeken, Alan Mishchenko, Patrick Vuillod, Jiong Luo, Christopher Casares, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. Scalable Boolean Methods in a Modern Synthesis Flow. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1643–1648, 2019.