From Logic to Gates: A Versatile Mapping Approach to Restructure Logic

Alessandro Tempia Calvino[1], Heinz Riener[1], Shubham Rai[2], Giovanni De Micheli[1]
[1]Integrated Systems Laboratory, EPFL, Lausanne, Switzerland
[2]Chair for Processor Design, TU Dresden, Dresden, Germany

***Abstract*—This paper proposes a versatile mapping approach for restructuring logic that has two objectives: i) it can map from one technology-independent graph representation to another; ii) it can map to a cell library. Our mapping approach for (i) uses a database of structures obtained using exact synthesis to map into a graph representation. Depending on the database, arbitrary 2-input and 3-input graph representations are supported (e.g. AIGs, XAGs, MIGs, etc.). The method is cut-based and uses Boolean matching to assign each cut to the available structures in the database before mapping. Consequently, the mapper is size- and depth-aware with respect to the final representation. Moreover, this process helps to mitigate logic sharing issues of LUT-based mapping approaches. The mapper supports delay- and area-oriented mapping for a global optimization objective. In the experiments, we show that our approach enables an average reduction up to 27.10% and 40.07% in size and depth respectively when mapping for size reduction from AIGs to MIGs in the EPFL benchmark suite. We then show that our approach leads to better results when used for logic restructuring reducing the average size by 10.24% and 14.63% more compared to LUT-based rewriting and cut rewriting methods. Finally, we map to a standard cell library and compare it to ABC showing an average improvement of 1.75%, 0.10%, and 18% in area, delay, and total run-time respectively.**

I. Introduction

Multi-level logic optimization is a fundamental step in the realization of competitive integrated circuits. State-of-the-art logic synthesis describes a circuit using a technology-independent representation, applies transformations to optimize mainly the size and the depth, and lastly, it maps the optimized logic to a technology-dependent representation.

Originally, 2-input NANDs and NORs together with inverters were used as graph representations thanks to their efficient implementation in CMOS technology. As logic synthesis evolved, the *And-inverter graph* (AIG) [1], consisting of 2-input AND gates and inverters, became the most common technology-independent representation. As an alternative, *Majority-inverter graphs* (MIGs) [2] have been proposed and motivated by a more expressive potential and by majority-based emerging technologies, e.g., quantum-dot cellular automata. Additionally, *Xor-And graphs* (XAGs) [3] and *Xor-Majority graphs* (XMGs) [4] have been proposed for their compactness in arithmetic circuits and as a basis for logic rewriting. Moreover, XAGs are commonly used to decrease the multiplicative complexity of a circuit [5]. Recent work investigated 3-input gates as new graph representations to address logic synthesis [6]. The first and only toolbox that supports optimization over multiple representations has been proposed in [7]. Since different graph representations are available to support logic synthesis, in this work, we investigate the mapping from one graph representation to another while optimizing the circuit for delay or area.

Mapping is the process of expressing a Boolean network using a set of primitives. In technology mapping, primitives depend on the target technology and are typically contained in a library such as *standard cells* or *field programmable gate arrays*. In the latter case, the logic primitive is the *lookup table* (LUT) which can express any function of k variables. We refer to a k-input lookup table as a k-LUT and to the process of mapping to k-LUTs as LUT mapping. In technology-independent mapping, primitives depend on the target representation. Typically, state-of-the-art technology-independent mapping relies on LUT mapping followed by a k-LUT decomposition using exact synthesis to obtain the target representation [8]. In this paper, we refer to this method as LUT-based mapping. Although in some cases a technology-independent mapping may be realized with a direct one-to-one replacement (e.g., from AIGs to MIGs [9]), this method does not provide an optimized representation and cannot be applied universally (e.g., from MIGs to AIGs). LUT-based mapping is often used also for logic rewriting. Haaswijk et al. implemented an optimization flow that used iteratively LUT mapping and exact k-LUT decomposition on MIGs [8] and XMGs [4]. A drawback of this methodology lies in the LUT mapping. LUT mapping aims at minimizing the size or the depth of the LUT circuit. By preferring larger LUTs to cover more logic, this approach loses information of the shared logic in the network. An example of this limitation is presented in the motivation section II-A.

To rewrite the network, other methods are also available in the literature. Rewriting [10] is a DAG-aware optimization method that aims at minimizing the size of a representation by replacing small parts of the network with smaller structures. The advantage of being DAG-aware is to be able to reuse existing logic and to exploit structural hashing [11]. The structures are typically contained in a database obtained using exact synthesis. This method greedily chooses the best local replacement but local decisions create replacement conflicts (e.g. two replacements cannot happen at the same time). Rewriting misses a *global view* that takes conflicts into account. An example of this limitation is presented in the motivation section II-A. The first proposed solution to tackle this problem has been proposed in [12]. The method annotates conflicts in a

conflict graph and then generates the final solution by solving a maximum weighted independent set problem on the graph.

In this work, we present a versatile mapping approach that can be used to: 1) map to a technology-independent representation; 2) rewrite the circuit for optimization; 3) map to a technology-dependent representation. We propose solutions to mitigate the logic sharing issues of LUT-based mapping and the global view limitation of logic rewriting. Our approach uses Boolean matching to associate each cut with a library of structures or primitives previous to mapping. The mapping can be delay- or area-driven with respect to the final implementation. A final DAG-aware rewriting iteration can be enabled to exploit structural hashing.

In the experiment we evaluate the versatility of the mapper and we compare it to state-of-the-art methods: 1) we map from AIGs to MIGs showing a size reduction up to 27.10% when mapping for size and a depth reduction of 45.17% when mapping for depth; 2) we evaluate our solutions to improve logic sharing and consider a global view by comparing to previous state-of-the-art LUT-based rewriting and logic rewriting obtaining better results in all the 19 optimizable benchmarks; 3) we map to a standard cell library and compare to ABC *map* showing an average improvement of 1.75%, 0.10%, and 18% in area, delay, and total run-time respectively.

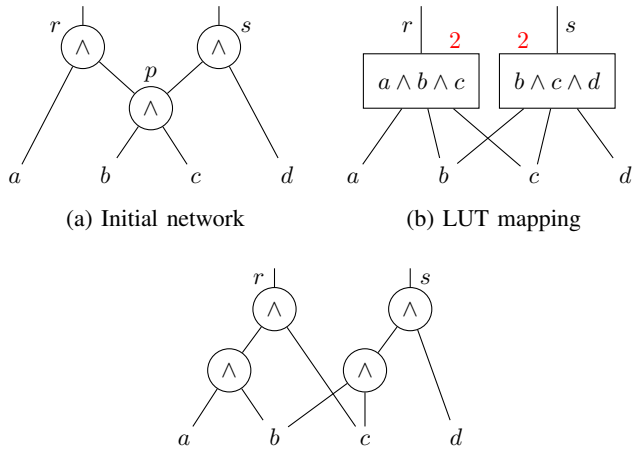
II. Motivations and Background

In this section, we briefly introduce the motivation, the basic notations, and the necessary background on technology mapping and optimization.

A. Motivation

In the introduction, we mentioned the limitations of previous mapping and rewriting approaches. In this section, we review the limitations in detail and we propose solutions.

1) Logic sharing: LUT-based mapping [8] consists of a LUT mapping followed by a k-LUT decomposition to obtain the target representation. LUT mapping is cut-based. Each k-feasible cut can be represented by a k-LUT so that LUT mapping consists of choosing a set of cuts to cover the network. LUT mapping aims at mapping the network by minimizing the number of LUTs or the depth. By preferring larger LUTs to cover more logic, local sharing of logic is often lost. In Figure 1a, an AIG network contains a shared node p. When the network is mapped to a 3-LUT network for size reduction, the network obtains the configuration in Figure 1b using the minimum number of 2 LUTs to cover the network. This operation loses the local information of the shared node p. When the LUTs are decomposed back to an AIG using exact synthesis, in Figure 1c, the two LUTs are matched to the same structure which creates an additional node with respect to the original network. As a remedy to mitigate this problem, we propose to assign a size and depth *weight* to each k-feasible cut of the network based on the matching structures obtained using exact synthesis. The weight represents the size and the depth of the structure. Our approach has the objective of minimizing the total weight in the cover. In Figure 1b the size weight is 4 (2 for each LUT). By minimizing the weight, the best solution maps each node with a LUT, preserving the shared node p, with a total weight of 3.



(a) Initial network (b) LUT mapping

(c) LUT decomposition with exact synthesis

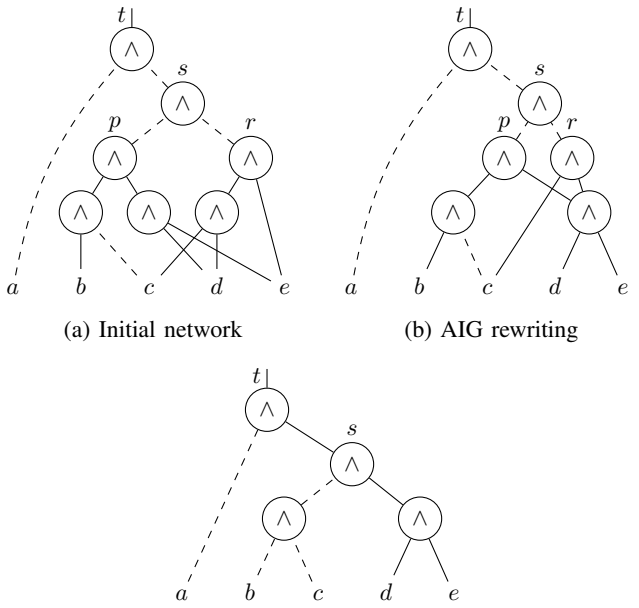
Fig. 1: Logic sharing limitation in LUT-based rewriting

2) Global view: Rewriting [10] is a cut-based optimization method that aims at minimizing the size of a representation by replacing small parts of the network with smaller structures. The approach chooses the best local replacements but local decisions create conflicts. An example is shown in Figure 2. We use 4-feasible cuts. Figure 2a shows the initial AIG network in which dashed lines represent negations. By rewriting the network, the best structure is obtained by replacing the cut (b, c, d, e) at root s. The implementation of this replacement depends on the substitution at the PO node t. In Figure 2b, rewriting selects the cut (a, p, r) at root t. Consequently, the best replacement at s cannot be used since s is already included in the best cut at t. AIG rewriting replaces the subgraphs rooted at p and r, thus leading to a size improvement of a single node. The best result in Figure 2c can be achieved by evaluating the conflicts globally. To mitigate this issue, we use global optimization methods typical of technology mapping such as area flow heuristic and exact area. The details are in Section III.

B. Technology Mapping

Technology mapping is the process of expressing a Boolean network using gates from a technology library. Libraries contain primitives (e.g., NAND, NOR, LUTs) and/or complex functions (e.g., AOI21, MUX) that are specific to the target technology. During technology mapping, cells from the library are used to cover a Boolean circuit while satisfying the given constraints and minimizing some cost functions (e.g. on delay, area). Next, we introduce basic terminologies which are required for the discussion about mapping.

Before technology mapping, the Boolean network is represented as a *k-bounded* network called the *subject graph*. A k-bounded network contains nodes with a maximum fanin size of k. AIGs are typically used as subject graphs. Accordingly



(a) Initial network (b) AIG rewriting

(c) Best structure with a global view

Fig. 2: Local view limitation in cut rewriting

to the definition, other types of representation may be used like XAGs.

A *cut* C of a node n in the subject graph is a collection of nodes called *leaves* such that each path from the PIs to node n must traverse at least one leaf. Node n is the *root* of the cut. A cut is *k-feasible* if the number of leaves of the cut is less than or equal to the bound k. The number of leaves in the cut determines the size. A *trivial cut* is a special cut that contains exclusively the node n. Each non-trivial cut is typically associated with a truth table representing the function at its root considered from the leaves. Truth tables are used for Boolean matching i.e. to bind each cut to the cells of the technology library. A k-input lookup table (k-LUT) can implement any k-feasible cut. Thus, we may abstract each cut as a k-LUT implementing the corresponding truth table.

A *maximum fanout free cone* (MFFC) of a node n is a subset of the fanin cone containing only nodes such that every path from these nodes to the POs passes through n.

A *cover* is a set of cuts so that all the cuts in the set are leaves of another cut in the set or are rooted at the POs. A mapping algorithm selects a set of cuts to cover the subject graph. A *delay-oriented* mapping aims to reduce the delay of the longest path in the cover. An *area-oriented* mapping aims to minimize the total area of the cover. While a minimal-delay mapping is tractable and can be obtained in polynomial time using a dynamic programming approach when ignoring loading effects [13], [14], an optimal area mapping is NP-hard [15] and thus requires heuristics.

C. NPN-equivalence classes

Two functions $f(x_1, \ldots, x_n)$ and $g(x_1, \ldots, x_n)$ are NPN-equivalent [16] if there exists a permutation of the inputs ($x_i x_j \rightarrow x_j x_i$), an inversion of the inputs ($x_i \rightarrow \overline{x_i}$), and an inversion of the output ($f \rightarrow \bar{f}$) so that f and g can be made Boolean equivalent.

For n-inputs, 2^{2^n} different Boolean functions exist. Boolean functions can be partitioned into NPN-classes which is a much more compact set. For instance, n-input Boolean functions can be classified into 14, 222 and 616126 classes, for $n = 3, 4, 5$ respectively.

NP-equivalence classes are defined similarly without considering the output inversion.

D. Exact synthesis

Exact synthesis [17] is the problem of finding optimum representations of Boolean functions in terms of network primitives. Generally, the cost criterion is the size or the depth of the structure. Methods such as logic rewriting [4], [10] use exact synthesis to rewrite parts of the circuit with optimum implementations.

NPN classification supports exact synthesis by notably reducing the number of functions to be synthesized and stored. Due to the problem complexity and the double-exponential growth in the number of functions with respect to the number of variables, exact synthesis is generally limited to small functions of 4 variables.

III. Versatile Mapping

In this section, we describe our contribution. We present a versatile mapper that can map from a generic technology-independent representation (e.g., AIG, XAG, MIG) into another representation or a technology. In the former case, it uses a database of pre-computed structures (e.g. exact synthesis database) to map or rewrite the network, in the latter case it uses a standard technology library. Our approach takes inspiration from state-of-the-art technology mapping [18] and logic rewriting [4], [12].

Our mapper implements the best characteristics of these two methodologies and addresses the LUT-based mapping and cut rewriting drawbacks. Boolean matching is used to bind the cuts to the available structures or primitives. Thus, size and depth information are available when generating the cover. The cover is minimized using size and depth instead of the number of LUTs and LUT levels. The mapper executes multiple mapping refinements, from global to local optimization. The algorithm can take advantage of the local sharing of logic. Our approach does not need to rewrite each cut. Nevertheless, an option exploits structural hashing during the last iteration to find shared nodes among the structures.

The mapper is implemented in a flexible parameterized way so that it can switch to different cost functions for delay-oriented or area-oriented mapping. The pseudo-code is shown in Algorithm 1. The mapper maps for delay by executing a delay-oriented mapping followed by area-recovery iterations. Area-oriented mapping is achieved by bypassing the delay-oriented iteration or by relaxing the required time constraint. Our method follows equivalent steps for the technology-dependent and -independent mapping except for a few differences that will be fully covered in the next paragraphs. In

this section, the terms area, delay, and gates are equivalently used as size, depth, and structures respectively. The algorithm can be summarized in six steps described in Sections A-F in detail:

A) Library generation
B) Cut enumeration
C) Boolean matching
D) Delay-oriented mapping
E) Area-oriented mapping
F) Finalization

Algorithm 1 Versatile Mapper

1: **Input**: Boolean network N, cut size k, *library*, *cut_sorting_func*, *constraints*, *skip_delay*, *AreaGlobalIter*, *AreaLocalIter*
2: **Output**: mapped network M
3: *cuts* ← compute_cuts(N, k, *cut_sorting_func*);
4: compute_truth_tables(N, *cuts*);
5: match_cuts(*cuts*, *library*);
6: **if** *!skip_delay* **then**
7: delay_oriented_map(N, *cuts*);
8: **end if**
9: **for** $it \leftarrow 0$ to *AreaGlobalIter* **do**
10: compute_required_times(N, *cuts*, *constraints*);
11: global_area_oriented_map(N, *cuts*);
12: **end for**
13: **for** $it \leftarrow 0$ to *AreaLocalIter* **do**
14: compute_required_times(N, *cuts*, *constraints*);
15: local_area_oriented_map(N, *cuts*);
16: **end for**
17: $M \leftarrow$ finalize(N, *cuts*);
18: **return** M;

A. Library generation

We define a library as a hash table that is used to classify gates for simple and fast Boolean matching. Given a Boolean function represented as a truth table, the library returns, if possible, a set of gates that can implement that function. The library generation is differentiated based on if it operates on a technology library or a database of structures since two different matching methods are used.

1) Technology library: For fast matching, the library contains all the NP-configurations of the gates. Given a gate with fanin size k, the maximum number of NP-configurations is $k! \times 2^k$. Since $k \leq 6$, the maximum number of NP-configurations for a gate with $k = 6$ is of 46080. However, this number is often smaller due to function symmetries. For instance, for the gate AOI22 with $k = 4$, only 48 unique configurations are found instead of the possible 384. The library stores the NP-configurations of the gates and the associated functions. Note that for standard libraries the number of entries is manageable. For the MCNC standard cell library [19], only 206 functions and 223 configurations are stored in the table.

2) Database of structures: The database stores the precomputed structures in a hash table partitioned in NPN-equivalence classes. Since the mapper matches by phase, and automatically inserts output inverters, each entry must not implement an output negation. Given an entry S which implements the NPN-class representative function f, if S has a negated output, the output negation is removed and the entry is saved to the new class $\overline{f}$. Thus, NPN-classes are rearranged to NP-classes when necessary. In the library, the NP-configurations are not enumerated since the entries will be too many. Consequently, functions are matched by canonization (more details in Section III-C).

For each entry, the pin-to-pin delay and the area are computed given a cost function. The pin-to-pin delay describes the depth of the longest path from an input pin to an output pin. The area is defined as the size of the structure. Additionally, also the inverter cost is supported.

B. Cut enumeration

Cut enumeration computes a set of k-feasible cuts for each node in the subject graph (line 7 of Algorithm 1). The computation proceeds in topological order from the primary inputs (PIs) to the primary outputs (POs).

Let $M(V, E, Y)$ be a generic network where V is the set of nodes, E is the set of edges, and Y is the set of POs. Let $X \in V$ be the set of PIs of the network, and $N \in V$ the set of internal nodes ($V \setminus (X \cup \{1\})$). Let N_m be the subset of N containing the nodes with fanin size equal to m. Let $\Phi(n)$ represent the set of *k-feasible* cuts at node $n \in V$. We define recursively Φ as:

$$\Phi(1) = \{\{\}\}$$
$$\Phi(x) = \{\{x\}\} \qquad \text{for } x \in X$$
$$\Phi(n) = \{\{n\}\} \cup (\Phi(n_1) \otimes \cdots \otimes \Phi(n_m)) \qquad \text{for } n \in N_m$$

where $n_1, \ldots, n_m$ are fanins of node n, and the *merging* operation $\otimes$ is defined as:

$$A \otimes B = \{u \cup v \mid u \in A, v \in B, |u \cup v| \leq k\}$$

During the enumeration process, some cuts may be dominated. A cut is *dominated* if it is contained set-theoretically in another cut. Dominated cuts are bailed out during the enumeration to reduce the cuts considered during the merging operation. This process does not impact the quality of the mapping. For each non-trivial cut, the corresponding truth table is computed. For implementation details we refer the reader to [20].

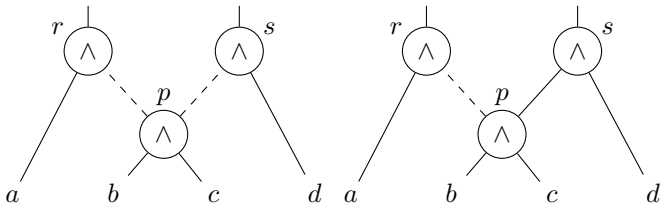
During the enumeration phase, cuts are sorted on the fly based on their depth, area flow and size. The cut prioritization is selected depending on the desired goal of the mapping. For a delay-oriented mapping, the sorting function primarily sorts for the delay while for area-oriented mapping, it orders primarily for area flow. To decrease the number of candidate cuts at each node, only a small number l is selected. On top of that, the trivial cut is added. This guarantees that at most $l+1$ cuts are saved at each node, so, for a node with fanin size equal to m, a maximum of $(l+1)^m$ cuts are enumerated. This technique is referred to as priority cuts [21], [22]. When using the mapper for technology mapping, the cut size is always the first criterion of selection. Ordering first by minimum size guarantees a feasible mapping if the technology library is complete (e.g. NAND2 and INV) since the first l selected cuts must contain a function primitive.

C. Boolean matching

Given a cut and the corresponding truth table, Boolean matching finds a set of gates that can implement that function. The pre-computed library of gates discussed in Section III-A is used to achieve that. In that section, we mentioned that the mapper matches by phases. Considering both phases for each cut function is necessary to enable logic sharing of inverters or avoid additional inverter delay costs.

In Figure 3a, node p has two negated outputs. Let us suppose that our library contains an AND2 gate and an inverter. Node p would be matched to an AND2 gate. Consequently, the mapper would insert two inverters on the edges (p, r) and (p, s) when mapping r and s with AND2 gates, creating an unnecessary inverter duplication. The adopted solution is to construct a gate composed of an AND2 plus an output inverter for a negative phase match at p. Hence, r and s can share the negative phase match avoiding the logic duplication. Although these redundancies could be removed with a circuit analysis after mapping, the mapper would be affected by wrong area estimations during the match selection phase leading to worse results.

Let us suppose now that the library contains also a NAND2 gate. In Figure 3b, node p has two fanout of different phase. If p is mapped with only one phase, e.g. to an AND2 gate, the arrival time at node r would increase by an inverter delay. By matching both phases separately using an AND2 and a NAND2 gate we could avoid an additional inverter delay. This operation is generally evaluated in terms of delay gain and area increase.



(a) Inverter sharing (b) Optimal delay

Fig. 3: Advantages of matching by phases

The Boolean matching technique is differentiated based on the mapping goal.

1) Technology mapping: Since the library contains all the configurations of all the gates, each compatible set of gates is obtained with a simple look-up in the library using the truth table. If a cut is not matched, it is ignored during the mapping.

2) Exact synthesis database: The library stores the database of structures in NP-classes. Boolean matching is achieved using function canonization to get the NPN class representatives f and $\overline{f}$ of the library and to match the gates. The canonization procedure finds the lexicographically smallest truth table (the NPN-class representative), the permutations, and the input negations to apply.

D. Delay-oriented mapping

Let the arrival time at a node n using a gate g matching a cut C be the sum of the pin-to-pin delays of g and the arrival time at the corresponding cut leaves of C. Delay-oriented mapping aims to cover the subject graph by selecting the gates that minimizes the arrival time at each node. The computation proceeds in topological order, over the internal nodes of the subject graph. For each node, the cut and the gate with the best arrival time are selected. In case of ties, tie-breakers are used as shown in Table I. Both the positive phase p and negative phases $\overline{p}$ of a node n are mapped separately if $t_a(n_p) < t_a(n_{\overline{p}}) + d_{inv}$ and $t_a(n_{\overline{p}}) < t_a(n_p) + d_{inv}$ where $t_a(n_p)$ ($t_a(n_{\overline{p}})$) is the arrival time of the best match at n with phase p ($\overline{p}$) and d_{inv} is the inverter delay.

After the delay-oriented pass, the cover is extracted by visiting in reverse topological order the nodes reachable from the POs using the best matches. The delay of the cover is defined as the latest arrival at the POs. To guide the area heuristics in the following steps and preserve the worst delay, the required times are computed by back-propagating the delay or the given required times from the POs to the PIs.

TABLE I: Gates selection criteria

Mapping Type	Cost criterion	Tie-breaker 1	Tie-breaker 2
Delay	arrival time	area flow	cut size
Global area	area flow	arrival time	cut size
Local area	exact area	arrival time	cut size

E. Area-oriented mapping

Area-oriented mapping or area recovery are performed in multiple passes over the nodes in the subject graph. Various heuristics that guide area minimization during technology mapping have shown good results [23], [24]. In particular, it has been shown in [22] that applying a first heuristic called *area flow* and a second method called *exact local area* leads to a good quality of the results. Our algorithm maps and adjusts the cover using these two methods iterated multiple times if necessary (line 10-17 in Algorithm 1)[1]. The area passes are constrained by the required time so that the worst-case delay is not increased. If the slack window is large enough, the algorithm tends to keep only one phase mapped per node to save area. The other phase is obtained by adding an inverter on the output pin of the match. If the slack window is too narrow, both phases are kept mapped.

1) Area flow: Area flow [24] for a node n estimates the area in its transitive fanin cone. During technology mapping, it can be formulated directly on a cut C with root n as follows:

$$AF(n) = [A_C + \sum_{i \in leaves(C)} AF(i)] \ / \ |fanout(n)|_{est}$$

where A_C is the area of a match for the cut C, and $|fanout(n)|_{est}$ is an estimation of the fanout size. For primary inputs or constants, the area flow is considered to be zero. Area flow is computed in a bottom-up traversal while mapping.

[1]Configuration details can be found in the experimental results section.

During this process, the fanout size of the nodes is unknown until the whole network is covered. Let the reference count $ref(n)$ for a node n in the subject graph be the number of times n appears as a leaf or as a PO in the cover. Informally, the reference counter represents the fanout in the cover. It is computed with a reachability analysis from the POs to the PIs. During the first covering iteration, the estimations are initialized using the fanout sizes of the subject graph. In the next passes, the fanout size estimation at pass j is computed as a linear combination of the reference count and the previous estimation:

$$|fanout(n)|^{j}_{est} = \alpha \times ref(n) + (1 - \alpha) \times |fanout(n)|^{j-1}_{est}$$

where α is a factor that takes values between 0 and 1. In our implementation we used $\alpha = 1/3$. The factor α is decreased after each pass.

2) Exact area: Exact area [21] is a local refinement of the cut selection which is driven by the area in the MFFC. The area is locally reduced by selecting a cut so that the sum of the area of the best cuts in the MFFC is minimized. Given a current cover of the subject graph, the exact area for a node n, can be computed using recursive referencing and dereferencing shown in Algorithm 2. A recursive referencing (dereferencing) algorithm recursively explores the leaves in the MFFC of a current cover. First, the best cut at node n is recursively dereferenced to remove it from the current cover. Then, each cut with root n is recursively referenced and then dereferenced to measure the exact area in the MFFC. The new best cut is then selected accordingly to the last line of Table I and it is recursively referenced to insert it in the cover.

Algorithm 2 Recursive dereferencing and referencing

```
1: Input: node n, cut C
2: Output: exact area
3: function RECURSIVE_DEREF( n, C )
4:     area ← A_C;
5:     for each node i in leaves(C) do
6:         if decr_ref(i) == 0 then
7:             area = area+RECURSIVE_DEREF(i, best_cut(i));
8:         end if
9:     end for
10:    return area;
11: end function
12: function RECURSIVE_REF( n, C )
13:    area ← A_C;
14:    for each node i in leaves(C) do
15:        if incr_ref(i) == 0 then                ▷ post-increment
16:            area = area+RECURSIVE_REF(i, best_cut(i));
17:        end if
18:    end for
19:    return area;
20: end function
```

In technology-independent mapping, we extend exact area with an option for high-effort optimization that enables a rewriting of the l best cuts to exploit structural hashing. In this case, the area of the structures (A_C) is measured on the fly during mapping similarly to rewriting [10].

F. Finalization

In the finalization process, the resulting network is created using the computed cover and the associated gates (line 18 of Algorithm 1). In technology-independent mapping, once the mapping is finalized, the network is always strashed to remove redundant nodes.

IV. Experimental Results

In this section, we evaluate the versatility of the mapper and compare it to state-of-the-art methods. We first use the mapper to map from one representation to another. For this experiment, we map from AIGs to MIGs since MIG mapping and MIGs in general and are well investigated in the literature (e.g. [8]). We evaluate the mapper in different settings for depth and size optimization showing considerable depth and size reduction. Then, we evaluate the mapper for logic restructuring and we compare it to state-of-the-art LUT-based rewriting and cut rewriting. The results support the advantages of our mapper: exploiting logic sharing, being size and depth aware, and having a global optimization view. Lastly, we use the mapper to map to a standard technology library and we compare it to ABC.

The mapper has been implemented in C++ 17 in the logic synthesis framework *Mockturtle*[2] [25]. The experiments have been conducted on an Intel i5 quad-core 2GHz on MacOS. All the results were verified using the combinational equivalent checker in *ABC*[3].

A. Mapping into MIG

In this first experiment, we use the versatile mapper to obtain a MIG representation starting from an AIG. For the experiment, we use a database obtained with exact synthesis with size-optimum structures for the 4-input NPN classes. Up to 10 alternative structures are available for each NPN class. The mapper computes cuts of size 4 and stores up to 25 cuts per node. We compare using different settings for depth-oriented and area-oriented mapping. Depth mapping executes a delay mapping followed by one global area and two local area recovery iterations. Area mapping skips the delay mapping and performs two global area and two local area recovery iterations. Lastly, a high-effort area mapping additionally rewrites the best 8 cuts in the last local area iteration to exploit structural hashing. The benchmarks have been taken from the EPFL arithmetic benchmark suite [26] containing combinational circuits in the Aiger format.

The results are shown in Table II. We evaluate the quality of the mapper in terms of size and depth improvement over the baseline. Depth-oriented mapping reduces the depth by 45.17% while still improving the size by 23.52%. Area mapping improves the size by 25.32%. At the price of a considerably higher run-time, high-effort area mapping improves the size by an additional 1.78% thanks to structural hashing.

[2]Available at: https://github.com/lsils/mockturtle

[3]Available at: https://github.com/berkeley-abc/abc

TABLE II: Experimental results for mapping from AIGs to MIGs

Benchmark	Baseline		Depth mapping			Area mapping			High-effort area mapping		
	Size	Depth	Size	Depth	Time (s)	Size	Depth	Time (s)	Size	Depth	Time (s)
adder	1020	255	384	129	0.02	384	129	0.02	384	129	0.13
bar	3336	12	3016	12	0.07	3016	12	0.08	2693	14	0.39
div	57247	4372	57300	2217	1.60	53225	2467	1.57	50705	2359	10.89
hyp	214335	24801	136108	8762	8.34	136299	8911	8.35	134968	8903	53.60
log2	32060	444	24457	200	1.55	24419	204	1.54	24310	207	8.09
max	2865	287	2413	150	0.22	2413	152	0.19	2413	153	0.44
multiplier	27062	274	19716	133	1.18	19355	142	1.14	19317	143	6.06
sin	5416	225	4307	110	0.25	4274	126	0.24	4244	127	1.23
sqrt	24618	5058	23238	3366	0.76	21042	4933	0.74	20718	4208	4.90
square	18484	250	12179	126	0.75	12184	126	0.78	12048	126	3.83
Total					14.74			14.66			89.66
Improvement			+23.52%	+45.17%		+25.32%	+40.24%		+27.10%	+40.07%	

B. Logic restructuring

In this experiment, we compare our mapper to LUT-based rewriting and cut rewriting to optimize MIGs. The LUT mapping is realized with the synthesis package ABC [27] using the command `&if -a -K 4` followed by a node re-synthesis in Mockturtle that decomposes each LUT with a matching structure contained in the database. Rewriting is achieved using the standard cut rewriting algorithm [12] implemented in Mockturtle. The versatile mapper is set for a standard area-oriented mapping plus a high-effort rewriting of the single best-matched cut, for a low impact on performance. The experimental setting is equivalent to the previous one. The three restructuring methods are iterated until no improvement.

The results are shown in Table III. We evaluate the results in terms of size improvement with respect to the baseline. The versatile mapper obtains the best results in all the 19 optimizable benchmarks. While LUT-based rewriting is the fastest optimization method, the final average run-time is comparable with the one of our mapper when considering the optimization gain. Generally, our mapper converges in a couple of iterations. The results support our motivations and the proposed solutions to exploit shared logic and account for global optimization. Moreover, our mapper supports a considerable reduction in depth that the other methods cannot achieve.

C. Technology mapping

In this experiment, we use our mapper for technology mapping starting from an AIG representation. The mapping is delay-oriented using one iteration of global area followed by two iterations of local area. We use the MCNC standard cell library [19] to bind the network. We compare to the ABC command *map*. In this experiment, we compute cuts of size 5 storing a maximum of 25 cuts per node.

The results are shown in Table IV. We compare in terms of area and delay improvement with respect to the result in ABC. While the results are comparable, the versatile mapper improves the area by 1.75% on average with a better run-time. Although delay-oriented mapping can achieve optimal delay, the versatile mapper leads to better delay results for some benchmarks such as *log2* and *sin*. One possible explanation may have to do with the quality of cuts that are stored at each node.

V. Conclusion

In this work, we presented a versatile mapper for delay or area optimization that is independent of the underlying graph data structure and the target representation. Our approach better exploits the sharing of the logic with respect to LUT-based mapping thanks to a Boolean matching phase previous to mapping so that decomposition costs (area and delay) are evaluated directly during mapping. Consequently, the mapper is size- and depth-aware with respect to the final representation. Our mapper supports delay and area mapping for a global optimization objective. An option uses structural hashing during the last area iteration to exploit common nodes among the structures. The experiments show better results in logic restructuring compared to LUT- and cut-based rewriting methods over all the 19 optimizable benchmarks. Moreover, the mapper shows better run-time and an average improvement of 1.75% and 0.10% in area and delay for technology mapping when compared to ABC.

References

[1] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Trans. CAD*, 2002.

[2] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE Trans. CAD*, vol. 35, no. 5, pp. 806–819, 2016.

[3] I. Háleček, P. Fišer, and J. Schmidt, "Are XORs in logic synthesis really necessary?," in *IEEE Proc. DDECS*, 2017.

[4] W. Haaswijk, M. Soeken, L. Amarù, P. Gaillardon, and G. De Micheli, "A novel basis for logic rewriting," in *Proc. ASP-DAC*, 2017.

[5] E. Testa, M. Soeken, L. Amarù, and G. D. Micheli, "Reducing the multiplicative complexity in logic networks for cryptography and security applications," in *2019 56th ACM/IEEE DAC*, 2019.

[6] D. S. Marakkalage, E. Testa, H. Riener, A. Mishchenko, M. Soeken, and G. De Micheli, "Three-input gates for logic synthesis," *IEEE Trans. CAD*, 2020.

[7] H. Riener, E. Testa, W. Haaswijk, A. Mishchenko, L. Amarú, G. De Micheli, and M. Soeken, "Scalable generic logic synthesis: One approach to rule them all," in *Proc. DAC*, p. 1–6, Jun 2019.

[8] W. J. Haaswijk, M. Soeken, L. Amaru, P.-E. Gaillardon, and G. De Micheli, "LUT mapping and optimization for majority-inverter graphs," in *Proc. IWLS*, 2016.

TABLE III: Experimental results for logic rewriting on MIGs

Benchmark	Baseline		LUT-based rewriting [8]			Cut rewriting [12]			Versatile mapper		
	Size	Depth	Size	Depth	Time (s)	Size	Depth	Time (s)	Size	Depth	Time (s)
adder	1020	255	385	130	0.05	893	129	0.08	384	129	0.06
bar	3336	12	2940	14	0.15	2952	15	0.71	2588	13	1.79
div	57247	4372	48827	4288	22.77	41553	2276	157.27	36858	2235	14.95
hyp	214335	24801	163398	9168	15.80	178736	9330	93.93	137048	8885	28.83
log2	32060	444	25651	247	3.91	30056	420	8.88	24295	206	3.20
max	2865	287	2446	248	0.35	2346	240	0.85	2171	162	0.96
multiplier	27062	274	20309	138	3.07	24829	271	12.37	19299	142	2.97
sin	5416	225	4560	159	0.44	5049	201	3.66	4196	122	1.14
sqrt	24618	5058	21002	6132	2.29	23889	4941	11.69	17355	3846	45.75
square	18484	250	14050	155	1.24	17669	163	8.85	11924	126	2.39
arbiter	11839	87	8769	62	0.87	11839	87	2.12	6996	59	1.57
cavlc	693	16	693	16	0.03	667	16	0.17	623	13	0.14
ctrl	174	10	131	6	0.04	126	9	0.03	114	5	0.01
dec	304	3	304	3	0.01	304	3	0.01	304	3	0.01
i2c	1342	20	1342	20	0.03	1272	22	0.24	1217	12	0.18
int2float	260	16	260	16	0.01	234	15	0.04	223	10	0.03
mem_ctrl	46836	114	44470	129	3.54	43651	137	13.69	41267	94	16.30
priority	978	250	978	250	0.02	843	248	0.37	808	117	0.24
router	257	54	228	55	0.04	218	53	0.09	207	42	0.04
voter	13758	70	8519	72	0.83	7293	69	4.24	5807	49	2.38
Total					55.49			319.26			122.95
Improvement			+16.59%	+15.09%		+12.20%	+9.78%		+26.83%	+36.01%	

TABLE IV: Experimental results for technology mapping

Benchmark	I/O	Baseline		ABC map			Versatile mapper		
		Size	Depth	Area	Delay	Total time (s)	Area	Delay	Total time (s)
adder	256 / 129	1020	255	1976	204.9	0.01	1975	204.9	0.01
bar	135 / 128	3336	12	5911	10.2	0.04	5911	10.2	0.05
div	128 / 128	57247	4372	124016	3516.5	1.20	127191	3516.5	1.34
hyp	256 / 128	214335	24801	435468	17520.6	7.35	429738	17520.6	5.59
log2	32 / 32	32060	444	55686	330.4	1.41	53778	329.8	1.02
max	512 / 130	2865	287	6186	208.4	0.06	5958	208.4	0.07
multiplier	128 / 128	27062	274	49597	210.9	1.05	47015	210.9	0.75
sin	24 / 25	5416	225	10690	154.3	0.24	10413	153.0	0.24
sqrt	128 / 64	24618	5058	44724	4235.8	0.58	44523	4235.8	0.71
square	64 / 128	18484	250	36321	199.4	0.74	35154	199.4	0.58
Total						12.68			10.36
Improvement							+1.75%	+0.10%	

[9] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *Proc. DAC*, 2014.

[10] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: a fresh look at combinational logic synthesis," in *Proc. DAC*, 2006.

[11] A. Mishchenko, S. Chatterjee, and R. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification," tech. rep., EECS Department, UC Berkeley, 2005.

[12] H. Riener, W. Haaswijk, A. Mishchenko, G. De Micheli, and M. Soeken, "On-the-fly and DAG-aware: Rewriting Boolean networks with exact synthesis," in *DATE*, Mar 2019.

[13] J. Cong and Y. Ding, "FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," in *Proc. ICCAD*, 1994.

[14] Y. Kukimoto, R. Brayton, and P. Sawkar, "Delay-optimal technology mapping by DAG covering," in *Proc. DAC*, pp. 348–351, 1998.

[15] A. H. Farrahi and M. Sarrafzadeh, "Complexity of the lookup-table minimization problem for FPGA technology mapping," *IEEE Trans. CAD*, 1994.

[16] L. Benini and G. De Micheli, "A survey of Boolean matching techniques for library binding," *ACM Trans. Design Autom. Electr. Syst.*, July 1997.

[17] W. Haaswijk, M. Soeken, A. Mishchenko, and G. De Micheli, "SAT-based exact synthesis: Encodings, topology families, and parallelism," *IEEE Trans. CAD*, 2020.

[18] S. Chatterjee, *On Algorithms for Technology Mapping*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2007.

[19] S. Yang, *Logic Synthesis and Optimization Benchmarks User Guide: Version 3.0.* Microelectronics Center of North Carolina (MCNC), 1991.

[20] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Improvements to technology mapping for LUT-based FPGAs," *IEEE Trans. CAD*, 2007.

[21] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," in *Proc. FPGA*, 1999.

[22] A. Mishchenko, Sungmin Cho, Satrajit Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts," in *Proc. ICCAD*, 2007.

[23] D. Chen and J. Cong, "DAOmap: a depth-optimal area optimization mapping algorithm for FPGA designs," in *Proc. ICCAD*, 2004.

[24] V. Manohararajah, S. D. Brown, and Z. G. Vranesic, "Heuristics for area minimization in LUT-based FPGA technology mapping," *IEEE Trans. CAD*, 2006.

[25] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, and G. D. Micheli, "The EPFL logic synthesis libraries," *CoRR*, vol. abs/1805.05121, 2019.

[26] L. Amarù, P.-E. Gaillardon, and G. D. Micheli, "The EPFL combinational benchmark suite," in *Proc. IWLS*, 2015.

[27] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification* (T. Touili, B. Cook, and P. Jackson, eds.), 2010.