A Versatile Mapping Approach for Technology Mapping and Graph Optimization

Alessandro Tempia Calvino*, Heinz Riener*, Shubham Rai†, Akash Kumar†, Giovanni De Micheli*
*Integrated Systems Laboratory, EPFL, Lausanne, Switzerland
†Chair for Processor Design, TU Dresden, Dresden, Germany

***Abstract*—This paper proposes a versatile mapping approach that has three objectives: i) it can map from one technology-independent graph representation to another; ii) it can map to a cell library; iii) it supports logic rewriting. The method is cut-based, mitigates logic-sharing issues of previous graph mapping approaches, and exploits structural hashing. The mapper is the first one of its kind to support remapping among various graph representations, thus enabling specialized mapping to emerging technologies (such as AQFP) and for security applications (such as XAG-based design). We show that mapping to MIGs improves area by 10% as compared to the state of the art, and that technology mapping is 18% faster than ABC with slightly better results.**

I. Introduction

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

Originally, 2-input NANDs and NORs, together with inverters, were used as primitives in graph representations thanks to their universality. 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], [3] 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) [4] and *Xor-Majority graphs* (XMGs) [5] have been proposed for their compactness in arithmetic circuits and as a basis for logic rewriting. 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.

We tackle different mapping problems with a single implementation that, on one hand, achieves comparable results in technology mapping when compared to the ABC mapper, and on the other hand, achieves better results in graph mapping and logic restructuring compared to previous methods. Moreover, we present the first mapper that supports mapping from and to different graph data representations such as AIG, XAG, MIG, and XMG. This feature finds extensive applications in different technologies. For instance, the *Adiabatic Quantum Flux Parametron* (AQFP) [8] superconducting technology, and quantum-dot cellular automata [9] are inherently majority-based. Our tool provides an efficient rewriting to MIGs that is crucial for specialized tools such as [10] for AQFP design. Reconfigurable nano-technologies (RFET) make use of XMGs as an efficient representation to preserve self-duality [11]. In cryptography and security applications, XAGs are used to represent circuits and analyze the multiplicative complexity of Boolean functions which correlates with vulnerability against algebraic attacks [12]. Furthermore, additional applications are possible for logic optimization (e.g., logic rewriting), especially in arithmetic-intensive circuits. Since publicly available logic synthesis tools mostly rely on AIGs for logic optimization [13], this mapper provides a way to easily obtain a representation that is more suitable for a particular application while optimizing it. Additionally, we present technical improvements over previous logic restructuring methods on *logic sharing* and *global view*.

In the experiment, we evaluate the versatility of the mapper and compare it to state-of-the-art methods:

- We map to a standard cell library and compare to ABC *map* command [13] showing comparable results with an average improvement of 1.75%, 0.10%, and 18% in area, delay, and total run time respectively.
- We evaluate the mapper for logic restructuring on MIGs. We test our solutions to improve logic sharing and optimize with a global view by comparing to previous state-of-the-art LUT-based rewriting and cut rewriting. Our mapper improves the average size by 9.45% and 20.64% respectively obtaining considerably better results for all the benchmarks.
- We map from AIGs into XAGs and XMGs. We improve previous work on XMG size optimization using LUT-based rewriting in [5], [14] by 12.22% in geometric mean and 27.45% in size/depth product.

In summary, this mapper is the first tool to enable remapping among various graph-based representations and it enables faster technology mapping, as compared to ABC, for better or comparable quality of results.

II. Background

In this section, we introduce the basic notations and the necessary background on mapping and logic restructuring.

A. Mapping

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. Before 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 to the definition, other types of representation may be used such as XAGs.

The subject graph is transformed into a mapped network by applying local substitutions to sections of the circuits. These sections are defined by *cuts* [15]. 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 can be 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 other cuts 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 [16], [17], an optimal area mapping is NP-hard [18] and thus requires heuristics.

B. NPN-equivalence classes

Two functions $f(x_1, \ldots, x_n)$ and $g(x_1, \ldots, x_n)$ are NPN-equivalent 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 [19].

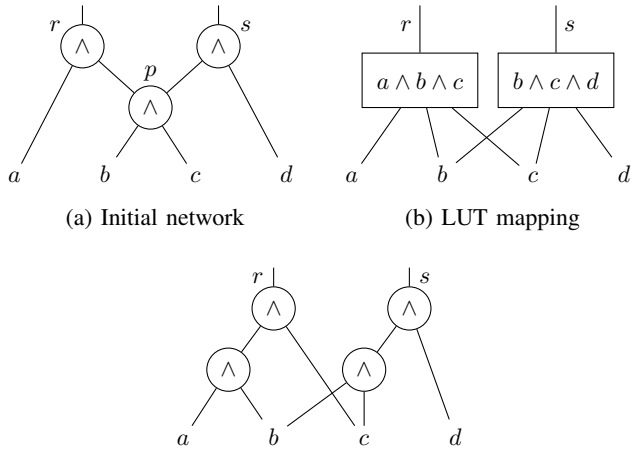
For n-inputs, 2^{2^n} different Boolean functions exist. Boolean functions can be partitioned into NPN classes. In particular, n-input Boolean functions can be classified into 14, 222 and 616126 classes, for $n = 3, 4, 5$ respectively.

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

C. Exact synthesis

Exact synthesis [20] 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 [5], [21] 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.



(a) Initial network (b) LUT mapping

(c) LUT decomposition with exact synthesis

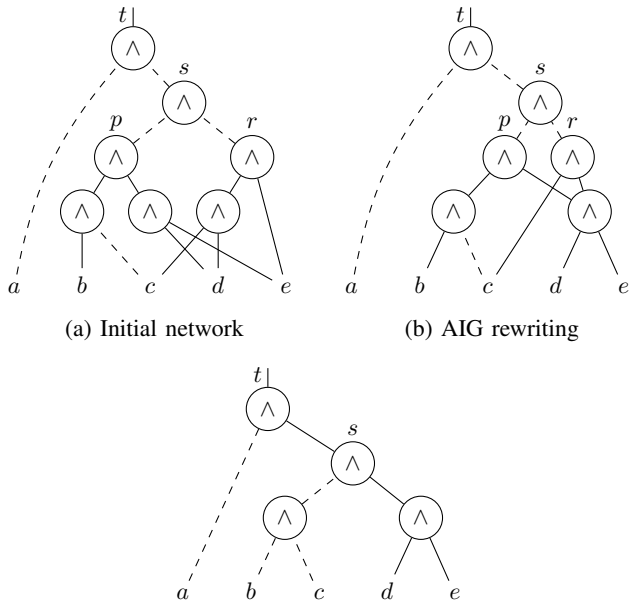
Fig. 1: Logic sharing limitation in LUT-based rewriting

D. Logic restructuring

LUT mapping [16] is a special case of technology mapping which covers a network using LUTs. State-of-the-art technology-independent mapping (or graph mapping) relies on LUT mapping followed by a k-LUT decomposition using exact synthesis to obtain the target graph representation [22]. We refer to this method as LUT-based mapping. LUT-based mapping is often used for logic rewriting by iteratively remapping the circuit. Previous work implemented optimization flows that used LUT mapping and exact k-LUT decomposition on MIGs [22] and XMGs [5]. LUT-based mapping suffers from a limitation that decreases the quality of results. LUT mapping aims at mapping a network by minimizing the number of LUTs or LUT levels. By preferring larger LUTs to cover more logic, the *logic sharing* of the original network is often lost. Hence, when the LUTs are decomposed using exact synthesis, more nodes than necessary are added to the network.

An example is shown in Fig. 1. In Fig. 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 Fig. 1b using the minimum number of two 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 Fig. 1c, the two LUTs are matched to the same structure which creates an additional node with respect to the original network. To describe structurally the logic sharing, a better mapping would use one LUT for each node of Fig. 1a.

To restructure a circuit, another method is also available in the literature. Rewriting [21] 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 re-use existing logic and to exploit structural hashing [23]. The structures are typically contained in a database. The approach greedily chooses the best local replacements but



(a) Initial network (b) AIG rewriting

(c) Best structure with a global view

Fig. 2: Local view limitation in cut rewriting

local decisions create conflicts (e.g., two replacements cannot happen at the same time). The algorithm lacks of a *global view* to extract the best replacements globally.

An example is shown in Fig. 2. We use 4-feasible cuts. Fig. 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 with leaves $\{b, c, d, e\}$ at root s. The implementation of this replacement depends on the substitution at the PO node t. In Fig. 2b, rewriting heuristic selects the cut with leaves $\{a, p, r\}$ at root t since it has a greater or equal local gain compared to the other candidates at t. Consequently, the best replacement at s cannot be used since s is already included in the chosen cut at t. AIG rewriting replaces the sub-graphs rooted at p and r, thus leading to a size improvement of a single node. The best result in Fig. 2c can be achieved by evaluating the conflicts globally.

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 cell library. In the former case, it uses a database of pre-computed optimum structures (e.g., obtained using exact synthesis) to map or rewrite the network, in the latter case it uses a standard technology library. Our approach combines and extends state-of-the-art technology mapping [24] and logic rewriting [5], [25].

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, accurate decomposition costs (size and depth) are available during mapping. The cover is minimized using size and depth instead of the number of LUTs and LUT levels. This helps to better exploit shared logic as compared to LUT-based mapping (e.g., our mapper maps each node in Fig. 1a with a LUT since this cover has a lower decomposition cost than the one in Fig. 1b, preserving the shared node p). The mapper executes multiple mapping refinements, from global to local optimization. In this way, the mapper generates the cover globally accounting for shared logic and then optimizes it locally, in the MFFCs. This approach helps to choose better replacements with a global view (e.g., our method achieves the structure in Fig. 2c when mapping from the structure in Fig. 2a). Our mapper does not need to rewrite each cut. Nevertheless, an option exploits structural hashing during the last iteration to find shared nodes among the structures. This typical feature of rewriting is controlled by technology mapping algorithms to select the replacements.

Algorithm 1: Versatile Mapper

```
Input : Boolean network N, cut size k, library,
  cut_sorting_func, constraints, skip_delay, AreaGlobalIter,
  AreaLocalIter, AreaStrashIter, rw_limit
Output: mapped network M
cuts ← compute_cuts(N, k, cut_sorting_func);
match_cuts(cuts, library);
if !skip_delay then
    delay_oriented_map(N, cuts);
end
for i ← 1 to AreaGlobalIter do
    compute_required_times(N, cuts, constraints);
    global_area_oriented_map(N, cuts);
end
for i ← 1 to AreaLocalIter do
    compute_required_times(N, cuts, constraints);
    local_area_oriented_map(N, cuts);
end
M ← new_network();
foreach primary input i ∈ N do
    create_input(M, i);
end
if graph mapping and AreaStrashIter then
    compute_required_times(N, cuts, constraints);
    local_area_strash_oriented_map(N, cuts, M, rw_limit);
    remove_dangling(M);
else
    finalize_network(N, cuts, M);
end
return M;
```

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 some 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 four steps described in Sections A-D:

A) Library generation
B) Cut enumeration
C) Boolean matching
D) Mapping

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 different for technology or graph mapping since two different Boolean matching methods are used.

For cell libraries, the library contains all the NP-configurations of the gates. Given a gate with fanin size k, the maximum number of NP-configurations is $k! \times 2^k$. However, this number is often smaller due to function symmetries. The library stores the configurations of the gates and the associated functions. Note that for most standard libraries the number of entries is manageable. For the MCNC standard cell library [26], only 206 functions and 223 configurations are stored in the table.

For graph mapping, the pre-computed structures are partitioned into NPN-equivalence classes and saved in a database. Since the mapper matches both polarities, 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 would 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 inverter costs are supported.

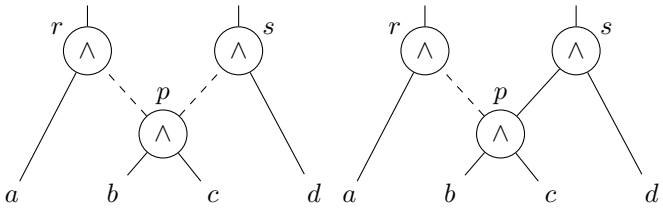
B. Cut enumeration

Cut enumeration computes a set of k-feasible cuts for each node in the subject graph (line 3 of Algorithm 1). The computation proceeds in topological order from the primary inputs (PIs) to the primary outputs (POs) as in [27]. The cut computation is independent of the graph representation and works for nodes with a variable number of inputs.

For each non-trivial cut, the corresponding truth table is computed. Truth tables are minimized by reordering variables and removing the ones without a functional support. This process eliminates "holes" in the truth table that prevent cuts from matching with the gates (which truth table is minimized). In this case, the support of the cuts is reduced accordingly. For instance, a cut C with leaves $\{l_1, l_2, l_3, l_4\}$ and truth table "$0F05$"[1] can be minimized to cut C' with leaves $\{l_1, l_3, l_4\}$ and function "31" since l_2 in C does not have functional support. An average of 0.06% additional cuts can be matched in the EPFL benchmark suite [28] using the MCNC library [26]. This number is not so small when considering the huge amount of cuts that are typically generated. Moreover, these cuts are often good since they contain internal don't care conditions.

During the enumeration phase, cuts are sorted on the fly based on their depth, area flow [29], 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 depth 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 [15]. 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 some of the first l selected cuts must match a function primitive.

[1]The truth table is represented in hexadecimal as a bitstring $b_{2^n-1} \dots b_1 b_0$.



(a) Inverter sharing (b) Optimal delay

Fig. 3: Advantages of matching in two polarities

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 in two polarities (uncomplemented, complemented). Considering both polarities for each cut function is necessary to enable logic sharing of inverters or avoid additional inverter delay costs [24].

In Fig. 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 complemented polarity match at p. Hence, r and s can share the complemented polarity 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 Fig. 3b, node p has two fanout of different polarities. If p is mapped with only one polarity, e.g., to an AND2 gate, the arrival time at node r would increase by an inverter delay. By matching both polarities 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 since it induces a duplication of the node p.

The Boolean matching technique (line 4 of Algorithm 1) is differentiated based on the mapping goal.

For 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.

For a technology-independent mapping, 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. Mapping

Delay-oriented mapping aims to cover the subject graph by selecting the gates that minimize the arrival time at each node. The computation (line 6 of Algorithm 1) 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 is selected. Both the uncomplemented polarity p and complemented polarity $\overline{p}$ 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 polarity p ($\overline{p}$) and d_{inv} is the inverter delay. This approach guarantees optimal delay mapping under the set of cuts. The area overhead is then addressed during area recovery once the required times at the nodes are known.

Area-oriented mapping or area recovery are performed in multiple passes over the nodes in the subject graph. In our approach, we use a first heuristic called *area flow* [29] for a global area optimization (line 10 in Algorithm 1) and a second method called *exact area* [15] for a local area optimization (line 14 in Algorithm 1). Our algorithm maps and adjusts the cover using these two methods iterated multiple times if necessary (lines 8-15 in Algorithm 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 polarity mapped per node to save area. The other polarity is obtained by adding an inverter on the output pin of the match. If the slack window is too narrow, both polarities are kept mapped. Depending on the mapping phase, the cost criteria to select the best gate are shown in Table I.

We extend exact area with an option for high-effort area optimization in graph mapping to exploit structural hashing to find shared nodes (lines 21-23 in Algorithm 1). Exact area 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 cut referencing and dereferencing. A recursive cut referencing (dereferencing) algorithm recursively explores the leaves in the MFFC of a current cover. The last local area iteration for graph mapping may include a rewriting of the l best cuts per node with structural hashing, called *rw_limit* in Algorithm 1, to find shareable nodes among the possible structures. In topological order, for each node, a candidate match is inserted in the network using one-level structural hashing by permuting and negating the inputs according to the NP transformation. Then, the number of added nodes is measured using node referencing and dereferencing similarly to rewriting [21]. Exact area is computed normally using the measured area value instead of the pre-computed area of the match. The match that minimizes the exact area at the node is selected. Structural hashing in exact area helps to select matches that share nodes with other structures in the cover. This method is particularly effective also when multiple alternative structures are available per NPN class and exact area with structural hashing is executed only on the previously selected best cut (*rw_limit* is 1).

In the finalization process, the resulting network is created using the computed cover and the associated gates (line 25 of Algorithm 1).

TABLE I: Gates selection criteria

Mapping Phase	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

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 to a standard technology library and we compare it to ABC [13]. In the second experiment, we use the mapper to optimize MIGs and we compare it to state-of-the-art LUT-based rewriting and cut rewriting methods. In the third experiment we map from AIGs into XAGs and XMGs. As baseline for all the experiments, we use the EPFL combinational benchmark suite [28] containing combinational circuits provided as AIGs.

The mapper has been implemented in C++17 in the open-source logic synthesis framework *Mockturtle*[2] [30] as a command *map*. 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. Technology mapping

In this first 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 [26] 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 II. We compare in terms of area and delay improvement with respect to the results in ABC. While the results are comparable, our versatile mapper improves the area by 1.75% on average with a better run time. Better run time is due to our simplified Boolean matching and cut prioritization that we use to store only 25 cuts. Our mapper leads to better delay results in two benchmarks thanks to the truth table minimization technique described in Section III-B that is not implemented in ABC.

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

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

TABLE II: Experimental results for technology mapping

Benchmark	I/O	Baseline		ABC map			Versatile mapper		
		Size	Depth	Area	Delay	Time (s)	Area	Delay	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%	

B. Mapping into MIG and 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 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 [25] implemented in Mockturtle (equivalent to AIG rewriting [21] but compatible with MIGs). 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. The versatile mapper is set for area-oriented mapping with one round of global area, two rounds of local area, and a high-effort round rewriting the two best-matched cuts, for a low impact on performances. The restructuring methods are iterated until no more improvement.

The results are shown in Table III. We evaluate the results in terms of size improvement with respect to the baseline. Our mapper obtains better results in all the benchmarks. From our experiments, mapping with structural hashing improves the size up to 10% more, and 1.23% on average, than standard area-oriented mapping. The results support our motivations and the proposed solutions to exploit shared logic and account for global optimization. Moreover, our mapper supports a reduction in depth that the other methods cannot achieve.

C. Mapping into XAG and XMG and logic restructuring

In this experiment, we show the versatility of the mapper to map into other graph representations starting from AIGs. For the experiment, we used two databases of structures for XAGs and XMGs obtained using exact synthesis and containing a single structure per NPN class. We run the mapper for area mapping until convergence using cuts of size 4. The results are shown in Table IV. The baseline is the same as the one reported in Table III containing only AND gates. The geometric mean is computed over size and depth. In the table, we compare our results with previous work on XMG optimization using rewriting in [5], taking the results using cuts of size 4, and in [14] that uses cuts of size 6. Our mapper obtains considerably better results in all the benchmarks in XMG optimization compared to the work in [5] when using the same cut size of 4. Moreover, our mapper obtains better results compare to the best results in previous work [5], [14] in 7 out of 10 benchmarks with an improvement of 12.22% in geomean and of 27.45% in size/depth product compared to [14]. Note that these results used LUT-mapping and exact synthesis on-the-fly on cuts of size 6. Consequently, their method needs a significant run time to compute the optimum structures on-the-fly since complete 6-input databases are too big to be pre-computed. Nevertheless, our method obtains better results for most benchmarks. This result shows again the advantage of our approach over LUT-based mapping.

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. Within one implementation, graph mapping, and logic restructuring. It is the first mapper that supports mapping among different graph data structures such as AIG, XAG, MIG, and XMG. Our approach better exploits the sharing of the logic as compared to LUT-based mapping by evaluating decomposition costs directly during mapping. It uses mapping algorithms for a global optimization. It can use structural hashing to exploit common nodes among the structures. The experiments showed better results in logic restructuring compared to LUT- and cut-based rewriting methods in all the benchmarks.

Acknowledgments

This research was supported by the SNF grant “Supercool: Design methods and tools for superconducting electronics”, 200021_1920981, and by the German Research Foundation (DFG), project “SecuReFET” 439891087.

References

[1] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, “Robust Boolean reasoning for equivalence checking and functional property verification,” *Trans. CAD*, pp. 1377–1394, 2002.

[2] 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.

[3] 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.

[4] I. Háleček, P. Fišer, and J. Schmidt, “Are XORs in logic synthesis really necessary?,” in *Proc. DDECS*, 2017.

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

TABLE III: Experimental results for mapping and rewriting MIGs

Benchmark	Baseline		LUT-based rewriting [22]			Cut rewriting [25]			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
Total					50.09			298.27			102.05
Improvement			+22.66%	+25.10%		+11.47%	+20.54%		+32.11%	+41.88%	

TABLE IV: Experimental results for rewriting XAGs and XMGs

Benchmark	XAG		XMG		XMG (k = 4) in [5]		XMG (k = 6) in [14]	
	Size	Depth	Size	Depth	Size	Depth	Size	Depth
adder	639	256	383	128	639	130	383	128
bar	3013	13	2944	14	3281	16	2149	14
div	29124	4316	17613	2300	29607	4371	37003	4243
hyp	158682	24912	114746	8984	155349	12507	99428	8755
log2	24330	327	21361	204	27936	275	22957	213
max	2766	234	1845	157	2296	296	1938	200
multiplier	18651	268	15642	134	17508	154	16357	133
sin	4259	175	3728	138	5100	176	3896	140
sqrt	12617	6122	9750	2431	20130	6031	17187	5169
square	13876	247	11250	126	15070	130	8325	156
Average	26,795.7	3.687.0	19,926.2	1,461.6	27961.6	2408.6	20,962.3	1,915.1
GeoMean	2,293.1		1,511.2		2,117.8		1,721.6	
Size · Depth	98,795,745.9		29,124,133.9		66,697,987.8		40,144,900.7	

[6] D. S. Marakkalage, E. Testa, H. Riener, A. Mishchenko, M. Soeken, and G. De Micheli, "Three-input gates for logic synthesis," *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*, Jun 2019.

[8] R. Cai, O. Chen, A. Ren, N. Liu, C. Ding, N. Yoshikawa, and Y. Wang, "A majority logic synthesis framework for adiabatic quantum-flux-parametron superconducting circuits," in *Proc. GLSVLSI*, p. 189–194, May 2019.

[9] K. Kong, Y. Shang, and R. Lu, "An optimized majority logic synthesis methodology for quantum-dot cellular automata," *IEEE Transactions on Nanotechnology*, vol. 9, no. 2, pp. 170–183, 2010.

[10] E. Testa, S.-Y. Lee, H. Riener, and G. De Micheli, "Algebraic and boolean optimization methods for AQFP superconducting circuits," in *Proc. ASP-DAC*, p. 779–785, 2021.

[11] S. Rai, H. Riener, G. Micheli, and A. Kumar, "Preserving self-duality during logic synthesis for emerging reconfigurable nanotechnologies," in *DATE*, 2021.

[12] J. Boyar, R. Peralta, and D. Pochuev, "On the multiplicative complexity of Boolean functions over the basis ($\wedge$, $\oplus$, 1)," *Theoretical Computer Science*, vol. 235, no. 1, pp. 43–57, 2000.

[13] 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.

[14] Z. Chu, M. Soeken, Y. Xia, and G. De Micheli, "Functional decomposition using majority," in *ASP-DAC*, pp. 676–681, 2018.

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

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

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

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

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

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

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

[22] 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.

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

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

[25] 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.

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

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

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

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

[30] 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.