Technology Mapping Using Multi-output Library Cells

Alessandro Tempia Calvino, Giovanni De Micheli
Integrated Systems Laboratory, EPFL, Lausanne, Switzerland

***Abstract*—Technology mapping transforms a technology-independent representation into a technology-dependent one given a library of cells. Even if technology libraries contain multi-output cells, state-of-the-art techniques fully exploit single-output cells only. Multi-output cells have limited support in logic synthesis and are typically handled as white boxes once identified. This paper presents a scalable method to increase the support of multi-output library cells in technology mapping. Our contributions include 1) an approach to detect some multi-output cells, 2) a fast Boolean matching methodology, and 3) a technology mapping algorithm that supports multi-output cells. Unlike previous work, we address the mapping problem over the whole network. This has the advantage of optimizing area and delay without requiring many incremental steps. The experiments show that full adders and half adders are efficiently detected and mapped with an average area improvement of 7.48% when mapping for minimal delay compared to the default mapper in ABC. Moreover, our method improves the area of the synthesis flow in Yosys, which treats multi-output cells as white boxes, by 5% on average with a limited run time overhead.**

I. Introduction

Technology mapping is one of the fundamental steps in the realization of integrated circuits. It consists of translating a technology-independent representation of digital hardware into a connection of technology-specific components such as *standard cells* or *lookup tables* (LUTs). The problem of optimally mapping a Boolean function to a cell library is known to be intractable. Consequently, technology mapping is generally formulated as a series of local substitutions applied to a simple multi-level representation of logic called the *subject graph*. The objective of technology-independent logic synthesis is to obtain a compact subject graph in terms of size and depth.

Cell libraries, such as *standard cells*, define a set of pre-designed and pre-characterized primitives that are used as building blocks to create digital circuits. Typically, libraries contain simple cells (e.g., a NAND2), complex cells (e.g., a XOR3), multi-output cells (e.g., a full adder), and sequential elements. Commonly, technology mapping algorithms efficiently exploit technology libraries. However, multi-output cells are often neglected due to the complexity of detecting and evaluating them in the optimization loops of technology mapping. Only a few frequent multi-output cells, such as half adders and full adders, have partial support in industrial tools. These common elements are generally identified in ordinary logic blocks, such as adders, which can be extracted from a *register-transfer level* (RTL) description of digital hardware for which the mapping is known. Nevertheless, synthesis flows often decompose these cells to meet timing constraints. Consequently, it is crucial to re-detect and map multi-output cells starting from a gate-level description to recover area and power consumption.

Generalized matching (GM) [1] has been introduced as a multi-output matching technique that supports concurrent matching to multiple single-output cells or a multi-output cell. In [2], the authors propose an incremental remapping technique that utilizes GM on local small windows of already mapped logic. Their method supports multi-output cells and evaluates substitutions symbolically by solving a minimum-cost GM problem using *binary decision diagrams* (BDDs) [3] and *algebraic decision diagrams* (ADDs) [4].

The tool Yosys [5], which is part of the RTL to GDSII toolchain OpenLane, integrates a limited support of half and full adder cells. Yosys identifies adder cells from RTL or by performing circuit analysis. Adder cells are kept as *don't touch white boxes* during logic optimization and technology mapping. Hence, technology mapping is oblivious of multi-output cells resulting in a degradation of the potential delay, power, and area of the design.

In this paper, we describe an alternative method to increase the support of multi-output cells in the optimization loops of a technology mapping algorithm. Primarily, we address scalability since multi-output cells substantially increase the complexity of mapping to an intractable level. Our contributions include a fast multi-output cell detection methodology, an extension of Boolean matching, and a formulation of global and local area recovery heuristics that support multi-output cells. In contrast to [2], we tackle the global technology mapping problem instead of local remapping. This approach has the advantage of selecting cells and optimizing for inverters globally while meeting delay constraints without iterating through many incremental steps. Instead of BDDs and ADDs, we pre-compute a library to facilitate Boolean matching that can be rapidly accessed through canonization and hashing. Moreover, we use a cut-based method to enumerate multiple mapping options and cell selections. In area recovery, we employ a generalization of *area flow* [6], [7] and *exact area* [8] as heuristics to evaluate single- and multi-output cells.

Our implementation is open-source and available in the library *anonymous*. To the best of our knowledge, this is the first open-source implementation of a technology mapper for *standard cells* that integrates the support for multi-output cells.

In the experiments, we evaluate our approach using the ASAP7 cell library [9], which contains the *half adder* and *full*

adder cell. We compare our mapper against ABC showing a 7.48% area reduction on average when mapping for the minimal delay. When the delay is not constrained, our approach obtains a considerable area reduction of 7.42%. Furthermore, our method reduces the area by 5% on average compared to the method in Yosys in which adder cells are considered as white boxes during technology mapping. Our approach demonstrates scalability to large networks with an average run time increase of 8%. We consider this overhead fairly limited since large circuits are mapped in a few seconds. Finally, we discuss the usage of other multi-output cells.

II. Background

In this section, we introduce the basic notations and the necessary background related to logic networks, technology mapping, and equivalence classes.

A. Notations and Definitions

A *Boolean network* is modeled as a directed acyclic graph (DAG) with nodes represented by Boolean functions. The sources of the graph are the *primary inputs* (PIs) of the network, the sinks are the *primary outputs* (POs). For any node n, the *fanins* of n is a set of nodes driving n, i.e. nodes that have an outgoing edge towards n. Similarly, the *fanouts* of n is a set of nodes that are driven by node n, i.e., nodes that have an incoming edge from n. If there is a path from node a to node b, then a is in the *transitive fanin* (TFI) of b, and b is said to be in the *transitive fanout* (TFO) of a. The transitive fanin of b includes node b and the nodes in its transitive fanin, including the PIs. The *transitive fanout* of b includes b and all the nodes in its transitive fanout including the POs.

The *maximum fanout free cone* (MFFC) of a node n is a subset of the transitive fanin of n such that every path from the nodes in the MFFC to the POs passes through n. Informally, the MFFC of a node contains the node itself and all the logic exclusively used by the node. When a node is removed (or substituted) the logic in the MFFC can also be removed. The MFFC can be extended to operate on a set of nodes $\mathcal{S}$ such that every path from the nodes in the MFFC to the POs passes through at least one node in $\mathcal{S}$.

A *cut* C of a node n is a collection of nodes of the network, called *leaves*, such that every path from the PIs to node n passes through at least one leaf. Node n is called the *root* of C. The *size* of a cut is defined as the number of leaves. A cut is k-feasible if its size does not exceed k. The *volume* of the cut is the total number of nodes encountered on all the paths between the leaves and the root. A *multi-output cut* is an extension of a cut defined over a set of roots $\mathcal{L}$.

A *cover* of a Boolean network is a set of cuts such that each node in the network is contained in the volume of at least one cut and all the cuts in the set are leaves of other cuts in the set or are rooted in the POs. A cover can be extracted top-down (in reverse topological order) by selecting cuts starting at the POs and recurring on the leaves.

B. Technology Mapping

Technology mapping is the process of expressing a Boolean network in terms of a set of primitives defined 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*, which contains nodes with a maximum fanin size of *k*. *And-inverter graphs* (AIGs) are widely used as subject graphs.

A mapping algorithm can be summarized in three main steps: 1) computing the k-feasible cuts using a fast enumeration procedure [10], 2) matching cuts to the technology library using Boolean matching [1] or pattern matching [11], and 3) generating a cover of the graph while minimizing a cost function and satisfying the constraints. For details on covering, we refer the reader to [6], [12], and [13].

A *delay-oriented* mapping aims to reduce the delay of the slowest path in the cover. An *area-oriented* mapping aims to minimize the total area of the cover.

C. $\mathcal{NPN}$-equivalence classes

Two functions $f(x_1, \ldots, x_n)$ and $g(x_1, \ldots, x_n)$ are $\mathcal{NPN}$-equivalent if there exists an inversion of the inputs $\mathcal{N}_i : (x_i \rightarrow \overline{x}_i)$, a permutation of the inputs $\mathcal{P}_i : (x_i x_j \rightarrow x_j x_i)$, and an inversion of the output $\mathcal{N}_o : (f \rightarrow \bar{f})$ such that f and g can be made Boolean equivalent [1]. $\mathcal{N}$-, $\mathcal{P}$-, and, $\mathcal{NP}$-equivalence classes are defined similarly considering input negations, input permutations, and both input negations and permutations respectively.

Boolean matching is commonly defined in terms of $\mathcal{N}$-, or $\mathcal{P}$-, or $\mathcal{NP}$-, or $\mathcal{NPN}$-equivalence classes.

III. Detecting Multi-output cells

In this section, we present a method to identify multi-output cells in a Boolean network. This process requires 1) an elaboration of the cell library to be suitable for Boolean matching, 2) a multi-output Boolean matching method, and 3) a multi-output cut computation procedure.

A. Matching library generation

Given a cell library, we define a data structure that facilitates fast Boolean matching called *matching library*. The matching library links a Boolean function represented as a truth table to a set of cells that implements it.

In technology mapping, delay, power, and area can be minimized by exploiting different configurations of cells based on the $\mathcal{NPN}$-equivalence classes [13], [14]. Specifically, permutations increase the number of matches, and negations play a crucial role in the insertion of inverters.

For single-output library cells, the matching library is generated similarly to [14] by enumerating cell configurations based on $\mathcal{NP}$-equivalence classes. Hence, given a function f to match, the matching library returns a set of cells in the $\mathcal{NP}-$equivalence class of f. Each one of them has attached an $\mathcal{NP}$-configuration so that its functionality matches f. Since our implementation matches in two polarities (complemented

and uncomplemented), the output inversion is not considered at this stage.

For multi-output cells, the procedure is more involved. The functionality of a multi-output cell is defined through a set $\mathcal{R}$ of functions, one for each output pin. Due to this higher degree of freedom compared to single-output cells, the multi-output matching library utilizes two additional operators $\mathcal{N}_o$, representing output negations, and $\mathcal{P}_o$, representing output permutations. First, the classification of cells in $\mathcal{NP}-$equivalence classes is extended such that each configuration of input negations and permutations is applied concurrently to the functions in $\mathcal{R}$. Second, to have a fast matching, the $\mathcal{NP}-$configuration of the multi-output cell is canonized. This is achieved in two steps. First, each individual output function is negated to be *normal*, i.e., a Boolean function f is *normal* if $f(0, 0, \ldots, 0) = 0$. Second, a multi-output function is canonized by sorting output functions in lexicographical order.

Let us consider a canonization example on a half adder cell described by the functionality set $R = \{\text{“0110”}, \text{“1000”}\}$ which is composed of a XOR2 and an AND2 function and is represented using truth tables[1]. Let us select a configuration of the cell composed of negations over the input variables $\mathcal{N}_i = \{x_0x_1 \rightarrow \overline{x}_0x_1\}$ and no permutations $\mathcal{P}_i$. After applying the input $\mathcal{NP}$ operators, the resulting function is $\{\text{“1001”}, \text{“0100”}\}$. Finally, the configuration is canonized leading to the multi-output function $\{\text{“0100”}, \text{“0110”}\}$ where the XOR2 has been normalized and the outputs have been permuted in lexicographical order. The corresponding output operators are $\mathcal{N}_o = \{o_0o_1 \rightarrow \overline{o}_0o_1\}$ and $\mathcal{P}_o = \{o_0o_1 \rightarrow o_1o_0\}$ (the negation is applied before the permutation).

Multi-output cells are represented as a set of single-output cells each corresponding to an individual output pin. In this paper, we define to these cells as *virtual output-pin* (VOP) cells. VOP cells describe the pin-to-pin delay relation and the functionality of each output pin.

B. Multi-output Boolean matching

Boolean matching assigns to a function a set of gates that can implement it. In technology mapping, Boolean matching is performed on local regions of a network defined by *cuts*. For single-output cells, matching consists of a simple look-up of the cut function in the matching library. Similarly to [13], [14], our approach matches while considering two polarities for each gate (uncomplemented, complemented) to enable better logic sharing of inverters or avoid additional inverter delay costs.

For multi-output cells, matching assigns a multi-output cut to a set of VOP cells. The matching is achieved in two steps. First, the functions of the cut roots are normalized and permuted to be canonical according to the matching library rules. Second, the canonical function is looked up in the matching library. The output negations and permutations are then reverted to reassign the individual VOP cells to the corresponding cut roots.

[1]Truth tables are reported as bit-strings $b_{2^n-1} \ldots b_1 b_0$ where b_{2^n-1} (b_0) represents the output when all the inputs take value 1 (0).

Algorithm 1: Multi-output cells detection

```
1  Input  : Subject graph N, Maximum k, Maximum l,
           Matching library lib
2  Output: Set of multi-output cuts multi_cuts
3  cuts ← enumerate_cuts(N, k);
4  /* filter cuts based on individual multi-output cells */
5  cuts ← filter_match_cuts(cuts, lib);
6  /* hash the cuts based on the leaves */
7  cuts_h ← hash_cuts(cuts);
8  /* combine cuts sharing the same leaves and match */
9  multi_cuts ← combine_cuts(cuts_h, l, lib);
10 /* remove incompatible kl-cuts */
11 multi_cuts ← filter_multi_cuts(multi_cuts);
12 return multi_cuts;
```

C. Multi-output cut computation

The multi-output cut computation may require significant run time since the number of cuts grows significantly with respect to the number of nodes in a Boolean network. KL-cuts [15] is an algorithm that can be used to generate generic multi-output cuts. However, some specific cells, such as adder cells, can be identified using a much simpler methodology. Hence, in this section, we propose a multi-output cut enumeration that considers a class of cells in which each output has all the cut leaves in its support. These cuts describe cells such as half adder and full adder and can be extracted very rapidly throughout a Boolean network.

Algorithm 1 presents a high-level view of the steps to enumerate and match multi-output cuts. The inputs are a subject graph N, a maximum cut size k, a maximum cut merging value l, and the matching library lib. First, k-feasible cuts are enumerated for every node of the network and the associated function is computed. Second, filtering rules are applied to reduce the number of cuts to combine. Specifically, we select only cuts whose function is $\mathcal{NPN}$-equivalent to a VOP cell function, i.e., the cut may be part of a matchable multi-output cut. Next, cuts are arranged in groups such that each cut in a group shares the same leaves. This is achieved using a fast algorithm that hashes the leaves of cuts. Then, cuts in each group are combined up to l outputs and directly matched. Matched multi-output cuts are added to a list. Alternatively to cut hashing, kl-cuts [15] can be used to generate multi-output cuts.

While combining cuts, filtering rules are employed to remove *partially dangling* multi-output cuts. Specifically, a multi-output cut with a set of roots $\mathcal{L}$ is partially dangling if $\exists n \in \mathcal{L}$ s.t. $n \in \text{MFFC}(\mathcal{L} \setminus n)$. Informally, a partially dangling multi-output cut has an output pin that cannot be connected externally since it is only used in the volume of the cut.

The last step of Algorithm 1 further filters cuts to be *compatible*. Two multi-output cuts C_i and C_j having the set of roots $\mathcal{L}_i$ and $\mathcal{L}_j$ are *incompatible* if $\mathcal{L}_i \cap \mathcal{L}_j \neq \emptyset$ and $\mathcal{L}_i \neq \mathcal{L}_j$. This filtering rule selects multi-output cells making sure they do not overlap if they do not share the same outputs. This constraint is crucial to limit the run time of technology mapping which would require undoing and re-

evaluating many mapping choices. Since incompatible cells are not very common in designs for typical cell libraries (that contain very few multi-output cells), we argue that the loss in quality deriving this filtering rule is limited. On the other hand, if we extend cell libraries to contain multi-output cells derived from the compression of random logic, the presented methodology would remove many possible matches affecting the potential quality. However, when multi-output mapping is used for re-mapping a small window of logic, the incompatible multi-output cell choices can be enumerated with a limited run time overhead.

IV. Multi-output cells support in technology mapping

In this section, we present how a technology mapping algorithm can be extended to support multi-output cells. Specifically, our method maps to multi-output cells only when the design cost improves compared to using single-output cells. Moreover, it handles the optimization of inverter cells also across multi-output gates and delay minimization.

Algorithm 2 shows the high-level pseudo-code of the mapper. All the steps are analyzed in detail in this section. First, the multi-output cuts are computed and matched using Algorithm 1. If multi-output cells have been already detected, e.g., they have been extracted from a *register-transfer level* (RTL) hardware description, they can be added at this step as multi-output cell choices for the mapper. Next, the network is sorted in a specific topological order guided by multi-output cuts. This is necessary to map the whole network in one forward and backward pass. Then, single-output cuts are computed using priority cuts [16]. At this step, our algorithm checks that the enumerated cuts contain also the single-output subcuts of the multi-output ones. This is desirable in some cases to reduce the impact of unmapping multi-output cells (a trivial decomposition can be used). Finally, the mapper covers the network by selecting a subset of cuts and associated cells.

Technology mapping consists of several iterations of mapping and covering. A mapping pass selects a candidate cell based on a cost function at each node. Covering extracts a complete mapping solution selecting a reachable subset of the cells starting from the POs. First, our implementation performs a round of delay-oriented mapping that selects for each node in topological order the cell with the smallest arrival time. This round finds the worst-case delay at the outputs and identifies critical paths. Following iterations have the objective of reducing area and/or power subjected to the delay constraints. In our technology mapper, we employ *area flow* [6] to globally optimize for the area and *exact area* [8] to locally refine the cover for area or power. While delay and *area flow* rounds are carried bottom-up (in topological order), exact area is carried top-down (in reverse topological order). Instead of propagating arrival times forward, exact area rounds propagate required times backward to select the candidate cells.

Algorithm 2: Technology mapping algorithm

```
1  Input  : Subject graph N, Maximum k, Maximum l,
     Matching library lib, cost function C
2  Output: Mapped network M
3  /* enumerate and match multi-output cuts */
4  multi_cuts ← compute_multioutput_cuts(N, k, l lib);
5  /* compute the constrained topological order */
6  topo_order ← constrained_topo_order(N, multi_cuts);
7  /* compute and match single-output cuts */
8  cuts ← compute_cuts(N, k, lib, multi_cuts, C);
9  /* cover the network and refine the mapping */
10 M ← cover(N, topo_order, cuts, multi_cuts, lib, C);
11 return M;
```

A. Constrained topological order

Technology mapping is generally a fast algorithm with a time complexity that is linear with respect to the number of nodes in the network. Ideally, we want to maintain the same complexity also when mapping multi-output gates. In technology mapping, as for many other synthesis algorithms, networks are stored in topological order to guarantee that when a node is processed, the nodes in its *transitive fanin* (TFI) have already been processed. For instance, this supports efficient propagation of arrival times while mapping. Our algorithm, presented in the next sub-section, follows this practice, i.e., nodes are mapped in topological and reversed topological order.

Let us consider a network and a topological order $\mathcal{T}$. Let us select three arbitrary nodes p, q, and t in $\mathcal{T} = \{0, \ldots, p, \ldots, q, \ldots, t, \ldots, m\}$ such that $p \in \text{TFI}(q)$, $q \notin \text{TFI}(t)$, and there is a 2-output cell that can implement p and t. Initially, all the nodes preceding t in the topological order, including p and q, are mapped using single-output cells. Next, p and t are mapped using the 2-output cell. Consequently, the arrival time computed at q may be invalid since node p in q's TFI changed the mapping. Moreover, the new mapping may unlock a different and more suitable mapping choice at node q. Thus, an algorithm that maps to multi-output gates might have to re-map some of the nodes between roots of multi-output cells in the topological order. However, we could avoid to re-process node q by picking a different topological order $\mathcal{T} = \{0, \ldots, p, \ldots, t, \ldots, q, \ldots, m\}$ where t precedes q. Our algorithm performs the topological ordering by positioning multi-output roots "close" to each other. In particular, given two roots p and t of a multi-output cut with $t > p$ in the topological order, it is always possible to have them topologically next to each other if $p \notin \text{TFI}(t)$ or $p \in \text{fanins}(t)$. Informally, if there is a path longer than 1 that connects two roots, there is at least a node that is between p and t in the topological order. In our topological sorting algorithm, when a node is a roots of a multi-output cut, first the TFI of all the roots are visited, then the roots are stored closed to each others. In our topological sorting algorithm, when a node is a root of a multi-output cut, first the TFI of all the roots is visited, then the roots are stored close to each other. This algorithm has the same computational complexity as a depth-first search.

Algorithm 3: Node mapping pass

```
1  Input : Subject graph N, Mapping M, Topological order
     topo_order, Cuts cuts, Multi-output cuts multi_cuts,
     Matching library lib, cost function C
2  Output: New mapping M'
3  /* compute the required time */
4  req ← compute_required(N, M, lib);
5  M' ← empty_mapping(N);
6  foreach Node n ∈ topo_order do
7      /* map node using single-output cells */
8      map_positive_polarity(M', n, cuts(n), lib, C, req);
9      map_negative_polarity(M', n, cuts(n), lib, C, req);
10     /* try to drop one polarity if there is enough slack */
11     select_polarity(M', n, req);
12     /* highest index output-pin of a multi-output cut */
13     if multioutput_root(n) then
14         map_multioutput(M', n, multi_cuts, cuts, lib, C,
             req);
15         remap_conflicts(N, M', n, cuts, lib, C, req);
16     end
17 end
18 return extract_cover(M');
```

Algorithm 4: Map multi-output cells

```
1  Input : Mapping M, Root n, Multi-output cuts multi_cuts,
     Cuts cuts, Matching library lib, cost function C, Required
     time req
2  Output: Modified mapping M
3  foreach Multi-output cut L ∈ multi_cuts(n) do
4      foreach Cell configuration G ∈ lib(L) do
5          /* Evaluate multi-output cell */
6          next_cell ← false;
7          foreach Root p ∈ L.roots do
8              if compute_arrival(M, L, G(p)) > req(p) then
9                  next_cell ← true;
10                 break;
11             end
12         end
13         if next_cell then
14             continue;
15         end
16         /* Evaluate improvement: area flow and delay */
17         if improvement(M, G, L, C) then
18             commit_cell(M, G, L);
19         end
20     end
21 end
22 return M;
```

B. Node mapping

Typically, technology mapping is carried out for several rounds to refine the cover according to specific cost functions. The first round is usually delay oriented with the objective of identifying the worst-case delay and critical paths. Following rounds optimize for area and/or power and are constrained on the maximum allowed delay. Commonly, mapping passes are carried bottom-up, while the cover selection and the required time propagation are carried top-down.

Algorithm 3 shows a forward pass to map nodes that can be found in a delay or area flow round. First, required times at each node are computed from the previous round if available. Then, each node is mapped in topological order. Node mapping works by initially covering the two polarities (complemented and uncomplemented) using single-output cuts. The best-fitting cell is chosen based on the selected cost function (e.g., delay) and the required time. If the latter allows it, only one polarity is implemented and the other is realized through an output inverter. Finally, multi-output-cell mapping is performed on the nodes that belong to a multi-output cut. However, multi-output mapping is evaluated for all the roots only when the highest-index root (with respect to the topological order) is processed. In this way, we ensure that all the roots of the cut have previously been mapped using single-output cells. This is necessary to compute accurate costs by comparing multi-output cell implementations to single-output ones. A multi-output cell is selected if it improves the cost function. If the multi-output mapping is successful, some nodes in the topological order among the roots of the multi-output cut might need to be remapped. This is to ensure that correct arrival times are propagated and that the best cell choices are made. Anyway, this step is often unnecessary if the multi-output roots are not leaves of a selected cut in the current mapping. Finally, a cover is extracted starting from the POs in reverse topological order and the mapping is returned.

During delay and area flow rounds, the mapping is improved by picking the best cell implementation at each node according to the selected cost function. The cover is only known after a top-down computation that selects the best cuts starting from the POs and recurs on the leaves. Consequently, after a bottom-up round not all the outputs of the multi-output cells are guaranteed to be used in the cover. This issue is not addressed during these iterations. Nevertheless, during exact area rounds the cover is known and is locally improved to guarantee that all the output pins of the selected cells are used, else nodes are re-mapped for a more suitable single-output cell implementation. Moreover, exact area rounds are carried out in reverse topological order to increase the likelihood of connecting all the output pins of multi-output cells. Contrarily to Algorithm 3, for exact area rounds, *multioutput_root*(n) is true for the lowest index output-pin of a multi-output cut.

The node mapping complexity is generally linear with respect to the number of nodes in the network. However, conflicts (line 15 of Algorithm 3) increase the complexity by having to re-process some nodes in the topological order among the multiple outputs. Anyhow, in practice no more than 0.5% of the nodes are ever re-mapped over the EPFL benchmarks [17].

C. Multi-output mapping evaluation

We evaluate multi-output cells by re-mapping simultaneously roots of a multi-output cut. The pseudo-code during an area flow round is shown in Algorithm 4. Initially, all the roots are assigned to a single-output cell. We evaluate a multi-output cell G by assigning to each root p the corresponding VOP cell ($G(p)$) and individually evaluating them. In particular,

the arrival time is checked against the required time. After the arrival time is checked, delay and *area flow* are used to evaluate the improvement and commit a multi-output cell if it has a better cost. A commit assigns VOP cells to each root. This process is repeated for all available cuts and cells.

The area flow (AF) of a node n involved in a multi-output cell G and a multi-output cut L is generalized as follows:

$$AF(n) = \frac{Area(G) + |L.roots| \times \sum_{l \in L.leaves} AF(l)}{\sum_{p \in L.roots} Refs(p)}$$

where *Refs* represent the estimated references of nodes in the cover. Initially, during the first round of the mapper, *Refs(n)* takes the fanout count of node n. Every following round it is updated using a linear combination of the previous value and the actual referencing in the cover. Note that if the generalized area flow formula is applied to a single-output gate, the equation reduces to the known formulation. In our implementation, each root of a multi-output cut references the leaves. This is why the area flow of a multi-output cut is multiplied by the number of roots. If AF is computed and exact references are known, the sum of AF over the POs gives the area of the cover.

In an *exact area* iteration, the cover is known from previous passes and it is locally improved. Specifically, for each node in the cover exact area chooses a gate such that the area in the MFFC of the node is minimized. This measure is extracted using a recursive cut referencing/dereferencing algorithm [8].

In exact area passes, a multi-output cut is evaluated only if all the output pins are referenced in the cover. This removes partially-connected multi-output cells originating from area flow iterations. Initially, all the roots of a multi-output cut are assigned to single-output cells. The roots are then recursively dereferenced to measure the exact area of the roots and remove them from the current mapping. A multi-output cell is then evaluated by checking the delay against the required time and measuring its MFFC area. If the cell replacement is accepted, the multi-output cut is referenced for each root, or else the previous implementation is restored.

V. Experiments

In this section, we present experimental results on using multi-output cells during technology mapping. We evaluate our method using the ASAP7 cell library [9], which defines two multi-output cells, namely the *half adder* and *full adder* with inverted outputs. For our experiments, we use the EPFL combinational benchmark suite [17] containing several circuits provided as *and-inverter graphs* (AIGs). The baseline has been obtained by applying the area-driven balancing algorithm available in *Mockturtle*[2] [18].

The mapper has been implemented in C++17 and is available as a command *emap* in the open-source logic synthesis framework *Mockturtle*. 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].

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

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

A. Comparing against ABC

In this experiment, we test *emap* for delay-oriented mapping against the default technology mapper in ABC (command *&nf -p*). We enable the use of multi-output cells in our mapper. We use the same settings for cut size and cut limit per node for both mappers.

Table I shows the results. We evaluate our mapper in terms of the area reduction and the geometric mean of the area and delay with respect to ABC. Our implementation effectively finds multi-output cells. Specifically, our mapper selects multi-output gates over non-critical paths to guarantee the minimal arrival time found during the first delay pass. Yosys and state-of-the-art mappers do not support this feature. In some benchmarks such as *adder*, no multi-output cells are used due to the delay constraints. In fact, multi-output cells have a higher delay than individual single-output cells in the ASAP7 library. While having a similar or better delay than ABC, our mapper has an average area reduction of 7.48%. For the *hyp* and *square* benchmarks, our mapper improves the area result of ABC by 20%. Our mapper has worse results than ABC in the *bar* benchmark. This is mainly due to worse cuts at the nodes. In fact, the benchmark does not contain any multi-output cells defined in the ASAP7 cell library. The experiment shows a 16.92% run time overhead compared to ABC which is reasonable considering that our implementation is generally 9% slower than ABC even when multi-output cell mapping is disabled.

B. Comparing to a two steps approach

In this experiment, we consider area-oriented mapping without delay constraints. We propose three flows. The first one performs a vanilla area-oriented mapping without mapping to multi-output cells. The second one emulates the approach in Yosys[4]. It is a two steps approach that detects full adders and half adders (command *extract_fa*), saves them as *don't touch white boxes*, and then maps the rest of the logic. We re-implemented the Yosys extraction algorithm following the detection algorithm presented in Section III which offers significantly better run time, scalability to large designs, and comparable results. Additionally, we use our mapper instead of ABC since our implementation typically provides 7% better area compared to *map -a* in ABC for area-oriented mapping. The third flow integrates multi-output detection and selection in technology mapping. We show that managing multi-output cells in a global technology mapping algorithm outperforms the two steps approach.

Table II shows the results. We use the same baseline as Table I to carry out the experiments. We evaluate the flows in terms of area and delay reduction with respect to the vanilla implementation. The two steps approach reduces the area of the vanilla implementation by 2.41% on average. This method is particularly run time efficient since technology mapping is decomposed into two independent steps. The multi-output approach is the best one with an average area

[4]Available at: https://github.com/YosysHQ/yosys

TABLE I: Comparing our mapper against ABC for minimal delay

Benchmark	Baseline		ABC command *&nf -p*			Our implementation			
	Size	Depth	Area	Delay	Time (s)	Area	Delay	Multi-output	Time (s)
adder	1020	255	1087.54	2520.72	0.06	1052.78	2514.41	0	0.03
bar	3336	12	2512.67	147.76	0.11	2835.74	151.76	0	0.08
div	45050	4405	46156.14	42422.38	1.82	41191.66	41964.47	4	2.11
hyp	214335	24801	197464.66	176397.09	8.16	157949.52	175299.08	12335	14.28
log2	31881	410	21997.83	3589.91	6.46	21612.64	3567.67	507	4.27
max	2865	229	2382.70	2114.45	0.19	2361.95	2106.08	0	0.11
multiplier	26943	266	24013.34	2596.60	1.04	19935.90	2576.03	652	1.56
sin	5383	186	5127.26	1614.09	0.71	5085.07	1583.31	49	0.74
sqrt	18372	6049	20576.44	43721.45	0.60	18066.04	43010.26	513	1.27
square	18264	250	13670.21	2446.85	0.64	10794.21	2440.58	1178	1.06
Improvement						-7.48%	-0.48%		+16.92%
Geomean			7614.05			7284.50			

TABLE II: Area-oriented mapping in different settings

Benchmark	Vanilla Mapping			Multi-output detection + Mapping				Multi-output Mapping			
	Area	Delay	Time (s)	Area	Delay	Multi-output	Time (s)	Area	Delay	Multi-output	Time (s)
adder	743.92	3910.79	0.03	596.02	7046.61	128	0.01	551.23	5376.32	128	0.03
bar	2062.66	209.60	0.08	2062.66	209.60	0	0.17	2062.66	209.60	0	0.10
div	29346.85	45188.06	1.91	28251.16	112512.28	3911	1.16	24486.23	90780.33	3904	2.02
hyp	157306.39	225518.08	11.73	156490.92	360506.91	20927	7.93	134586.34	218205.08	15853	13.65
log2	19031.21	4742.11	4.25	17561.92	8680.94	1461	4.34	15915.06	7393.68	1359	4.31
max	1860.41	2410.55	0.38	1860.41	2410.55	0	0.44	1860.41	2410.55	0	0.38
multiplier	16760.62	3638.24	1.68	15009.51	7701.35	1511	1.35	13219.61	6724.49	1467	1.66
sin	3585.97	2410.42	0.94	3361.07	3636.69	265	0.85	3132.99	3194.23	204	0.88
sqrt	12676.20	55493.75	1.42	14564.95	86093.17	1830	1.53	13107.36	64646.94	804	1.71
square	12966.38	3065.54	1.06	11600.65	7304.42	1471	0.56	9940.72	6974.56	1322	1.06
arbiter	7401.25	923.08	2.27	7401.25	923.08	0	2.80	7401.25	923.08	0	2.35
cavlc	441.37	221.24	0.03	446.03	217.10	7	0.04	446.50	223.54	2	0.02
ctrl	99.84	115.60	0.00	101.01	130.63	1	0.01	100.78	115.60	0	0.00
dec	289.26	56.63	0.16	289.26	56.63	0	0.18	289.26	56.63	0	0.16
i2c	911.19	210.04	0.09	911.42	210.04	2	0.12	911.19	210.04	0	0.10
int2float	146.50	187.80	0.01	146.50	187.80	1	0.01	146.50	187.80	0	0.01
mem_ctrl	30658.97	1530.06	3.23	30650.35	1512.75	45	5.27	30598.68	1512.75	5	3.47
priority	661.11	2475.18	0.07	661.11	2475.18	0	0.08	661.11	2475.18	0	0.07
router	171.69	374.68	0.09	178.22	708.09	38	0.02	171.21	522.58	15	0.07
voter	7190.60	895.47	0.59	6468.51	1372.72	1257	0.13	5534.16	1197.18	1183	0.62
Improvement				-2.41%	+44.02%			-7.42%	+26.27%		
Total			30.02			32855	27.00			26246	32.67

reduction of 7.42%, outperforming the two steps approach. The total run time increase is about 8% compared to the vanilla implementation which is reasonable considering the improvement in quality. Furthermore, if we consider only the arithmetic benchmarks (the first 10), our mapper improves the area by 12.7% against the 4.42% of the two steps approach.

While outperforming the two steps approach, the multi-output mapping generally uses fewer multi-output cells. Our implementation effectively leverages mapping heuristics to select the proper cells based on the global context. Moreover, even if the delay is not constrained, our approach often finds solutions with both better area and delay. Only the *sqrt* benchmark has 3.4% worse area when mapping using multi-output gates. Nevertheless, this is due to area recovery heuristics. Specifically, the multi-output mapper has 5.2% better area until the end of *area flow* rounds. However, following local rounds using *exact area* are considerably more effective on the vanilla flow (with a 13.37% area reduction) resulting in the mentioned quality difference. An interesting result is obtained on the *adder* benchmark. Although the two multi-output flows use the same number of multi-output cells, our proposed method achieves better area and delay by selecting different configurations of the multi-output cells (input and output negations) such that the number of inverters is globally minimized (192 instead of 256).

In this experiment, we have not constrained the delay during mapping for two reasons: 1) Yosys' method does not support effective delay minimization when adder cells are used (since they are considered as white boxes); 2) we want to test how much additional area is recovered with the proposed method against the state-of-the-art approach and Yosys. In addition, multi-output cells have higher propagation delay compared to single-output cell realizations. Hence, compared to the vanilla flow, delay increase is justified.

C. Discussion

In this section, we showed the potential of supporting multi-output cells in a technology mapper. Our method effectively

maps to multi-output gates when convenient achieving a substantial area reduction both in area-oriented and delay-oriented mapping. However, the ASAP7 technology library, as other open source libraries such as SKY130, contains only half and full adders. We predict that by notably increasing the number of multi-output cells, the mapper would be less efficient at handling them. This is mainly due to a very large number of matches and *incompatible* cuts (explained in Section III-C) that must be filtered to achieve scalability to large designs. We verified this statement by crafting 120 multi-output cells obtained by merging single-output cells included in the ASAP7 library assuming they occupy 20% less area. Our mapper correctly matches many multi-output cells. However, due to cut filters and the algorithm design for scalability, the proposed approach does not effectively exploit the benefits of that many matches leading to limited area reduction compared to using only half and full adders. This is a trade-off between run time and potential quality. In this work, we addressed scalability primarily. In particular, we obtained a quick algorithm with a similar run time to state-of-the-art mappers. Additionally, the majority of the multi-output cells in the crafted test library represent the compression of random logic. As such, these cells do not lead to any structural advantage during global technology mapping.

To mitigate the decrease in quality when many multi-output cells are defined, we propose to limit the number of cells handled by a global technology mapper. The selected ones should have specific regular structures able to affect the global mapping. In the experiments we showed the potential of integrating half and full adders. Additionally, we noticed that it is much more effective to handle cells that compress random logic later in the flow through incremental local remapping steps as performed in [2].

Despite these findings, the algorithms presented in this paper can support multiple multi-output cells and are not restricted to only half adders or full adders. If we limit the mapper to work on a small logic cone (e.g., in the order of tens of nodes), cut filters can be relaxed to include incompatible cuts. KL-cuts [15] can be also used to enumerate more multi-output cuts. Conflicting configurations of multi-output cells can be enumerated and mapping can be rapidly performed for each one of them, also in parallel. Then, the best solution is committed. This method would be a part of a re-mapping engine that extracts windows of mapped logic and re-maps them using a better implementation. Moreover, this approach would extend the remapping algorithm in [2] that focuses only on one multi-output cell at the time. The integration of this mapper in a remapping engine is out of the scope of this paper and will be addressed in future work.

VI. Conclusion

In this paper, we proposed a scalable technique to increase the support of multi-output library cells in technology mapping. To the best of our knowledge, this is the first open-source implementation of a technology mapper that supports multi-output gates. In contrast to existing work on remapping [2], we addressed the mapping problem globally over the whole network. First, we proposed a method to identify multi-output cells by enumerating multi-output cuts and employing multi-output Boolean matching. Second, we generalized area recovery heuristics to support multi-output cells in the cost computation. Last, we developed an enhanced technology mapping algorithm integrating multi-output cell detection and selection. Experimental results showed that half and full adders are efficiently used, notably improving the area in delay- and area-oriented mapping by more than 7% on average with a limited run time overhead. Moreover, we showed that our approach outperforms the method implemented in Yosys.

Acknowledgments

This research was supported by the SNF grant "Supercool: Design methods and tools for superconducting electronics", 200021_1920981, and Synopsys Inc.

References

[1] L. Benini and G. De Micheli, "A survey of Boolean matching techniques for library binding," *ACM Trans. Design Autom. Electr. Syst.*, July 1997.
[2] L. Benini, P. Vuillod, and G. De Micheli, "Iterative remapping for logic circuits," *Proc. IEEE Trans, CAD*, vol. 17, no. 10, pp. 948–964, 1998.
[3] R. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.
[4] R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebraic decision diagrams and their applications," in *Proc. ICCAD*, pp. 188–191, 1993.
[5] C. Wolf, "Yosys open synthesis suite." https://yosyshq.net/yosys/.
[6] V. Manohararajah, S. D. Brown, and Z. G. Vranesic, "Heuristics for area minimization in LUT-based FPGA technology mapping," *IEEE Trans. CAD*, 2006.
[7] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," in *Proc. FPGA*, 1999.
[8] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Improvements to technology mapping for LUT-based FPGAs," *IEEE Trans. CAD*, 2007.
[9] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "ASAP7: A 7-nm finFET predictive process design kit," *Microelectronics Journal*, 2016.
[10] P. Pan and C.-C. Lin, "A new retiming-based technology mapping algorithm for LUT-based FPGAs," in *Proc. ACM/SIGDA Sixth International Symposium on FPGA*, 1998.
[11] L. Stok, M. Iyer, and A. Sullivan, "Wavefront technology mapping," in *Proc. DATE*, 1999.
[12] D. Chen and J. Cong, "DAOmap: a depth-optimal area optimization mapping algorithm for FPGA designs," in *Proc. ICCAD*, 2004.
[13] S. Chatterjee, *On Algorithms for Technology Mapping*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2007.
[14] A. T. Calvino, H. Riener, S. Rai, A. Kumar, and G. De Micheli, "A versatile mapping approach for technology mapping and graph optimization," in *ASP-DAC*, 2022.
[15] O. Martinello, F. S. Marques, R. P. Ribas, and A. I. Reis, "KL-cuts: A new approach for logic synthesis targeting multiple output blocks," in *Proc. DATE*, pp. 777–782, 2010.
[16] A. Mishchenko, Sungmin Cho, Satrajit Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts," in *Proc. ICCAD*, 2007.
[17] L. Amarù, P.-E. Gaillardon, and G. D. Micheli, "The EPFL combinational benchmark suite," in *Proc. IWLS*, 2015.
[18] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, S.-Y. Lee, A. T. Calvino, D. S. Marakkalage, and G. D. Micheli, "The EPFL logic synthesis libraries," *CoRR*, vol. arXiv:1805.05121v3, 2022.