In Medio Stat Virtus*: Combining Boolean and Pattern Matching

Gianluca Radi, 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. This process is performed by means of local replacements that are extracted by matching sections of the subject graph to library cells. Matching techniques are classified mainly into *pattern* and *Boolean*. These two techniques differ in quality and number of generated matches, scalability, and run time. This paper proposes *hybrid matching*, a new methodology that integrates both techniques in a technology mapping algorithm. In particular, pattern matching is used to speed up the matching phase and support large cells. Boolean matching is used to increase the number of matches and quality. Compared to Boolean matching, we show that hybrid matching yields an average reduction in the area and run time by 6% and 25%, respectively, with similar delay.**

I. Introduction

Technology mapping is a crucial step in the realization of integrated circuits. Its primary goal is to translate a technology-independent representation of digital hardware into a connection of technology-specific components, such as *standard cells* or *lookup tables* (LUTs). However, the optimal mapping of an arbitrary Boolean function to a cell library is known to be an intractable problem. Hence, technology mapping is generally approached as a series of local substitutions applied to a multi-level representation of logic known as the *subject graph*. The key objective of technology-independent logic synthesis is to achieve a compact subject graph, both in terms of size and depth, to facilitate technology mapping and enhance quality.

Cell libraries, such as *standard cells*, define a set of pre-designed and pre-characterized logic primitives that are used as building blocks to create digital circuits. Technology mapping uses these blocks to cover the subject graph while minimizing a cost function, typically based on power, delay, and area. Mapping addresses two sub-problems: *matching* and *selection*. Matching involves associating sections of the subject graph with a list of cells that are functionally equivalent and capable of implementing those sections. Selection selects appropriate cells to cover the graph such that the target cost function is minimized.

In this paper, we focus on the matching problem. In early approaches, cells were represented using a graph, typically in the form of a NAND decomposition of the Boolean functionality. The matching task was then formulated as a (sub) graph isomorphism problem, particularly efficient when the decomposition graph is a tree. This form of matching is referred to as *pattern matching* [1]. However, this approach has several drawbacks. The most important one is that the graph decomposition is not canonical. Consequently, the number of possible graph decompositions can grow exponentially large for some cells, making it challenging to detect potential matches. Moreover, the matching process is significantly more involved when the decomposition graph is not a tree, such as in the case of XOR gates. Later approaches used *Boolean matching* [2] which is based on a canonical Boolean representation of the function, solving the limitations of pattern matching. Boolean matching inherently solves a tautology problem. Typically, this technique scales to cells up to 6 inputs, covering the majority of the cells present in standard cell libraries. Modern approaches use truth tables or BDDs as a canonical data structure.

In this work, we propose *hybrid matching* for technology mapping, an approach that combines the strengths of pattern and Boolean matching to achieve better quality. On the one hand, pattern matching finds application for many cells that are decomposable into NAND trees, such as ANDORs. Furthermore, modern cell libraries contain cells of 7 or more inputs, beyond Boolean matching capabilities (for reasonable run time). On the other hand, Boolean matching generally offers better quality for cells up to 6 inputs. By leveraging the benefits of both techniques, hybrid matching computes both pattern and Boolean matches and strategically combines them to achieve better quality and run time speedup.

We have integrated hybrid matching in the technology mapper in the logic synthesis library Mockturtle [3]. Similarly to previous work [4]–[7], in this paper, we work with a gain-based (load-independent) delay model. The methods discussed in this paper are anyhow compatible with physical-aware mappers since we focus on matching and not on covering. In the experiments, we compare hybrid matching to both Boolean and pattern matching showing that:

- Hybrid matching reduces the area up to 34% compared to Boolean matching, making efficient use of large gates. On average, it reduces the area by 6.15% while maintaining a similar delay.
- Hybrid matching reduces the area by 5% on average compared to pattern matching for better delay.
- Hybrid matching reduces the run time of technology mapping by 25% compared to Boolean matching.

II. Background

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

*Virtue stands in the middle

A. Notations and definitions

A *Boolean network* is 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 that have an outgoing edge towards n. Similarly, the *fanouts* of n is a set of nodes that have an incoming edge from n.

A *cut* C of a Boolean network is a pair (n, $\mathcal{K}$), where n is a node called *root*, and $\mathcal{K}$ is a set of nodes, called *leaves*, such that 1) every path from any PI to node n passes through at least one leaf and 2) for each leaf $v \in \mathcal{K}$, there is at least one path from a PI to n passing through v and not through any other leaf. The *size* of a cut is defined as the number of leaves. A cut is k-feasible if its size does not exceed k. A *trivial cut* of a node is a cut composed of only the node itself. A cut *covers* all the nodes encountered on the paths between the leaves and the root, including the root and excluding the leaves.

A *cover* of a Boolean network is a set of cuts such that 1) each node in the network is covered by at least one cut and 2) the root of each cut in the cover is either a PO of the network or a leaf of one or more cuts in the cover. A cover can be extracted top-down (in reverse topological order) by selecting cuts rooting in the POs and recurring on the leaves.

B. Cut enumeration

Cut enumeration [8] assigns a set of k-feasible cuts to each node in the subject graph. Cut enumeration starts at the PIs and proceeds in the topological order towards the POs. Processing in topological order guarantees that the cut computation of a node is performed after its fanins. For PIs, the list of cuts contains only the trivial cut. For an internal node n with two fanins a and b, its cut set $\Phi(n)$ is computed as follows:

$$\Phi(n) = \{\{n\}\} \cup \{u \cup v \mid u \in \Phi(a), v \in \Phi(b), |u \cup v| \leq k\}$$

Informally, merging two sets of cuts adds the trivial cut and the set of pair-wise unions of cuts belonging to the fanins. Only k-feasible cuts are kept in the set. The pair-wise union of two cuts belonging to the fanins is called *cut merging*.

C. 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 [9]. NP-equivalence classes are defined similarly without considering the output inversion.

D. Technology mapping

Technology mapping [10] is the process of expressing a Boolean network using a set of logic primitives contained in libraries such as *standard cells* or *field programmable gate arrays*. Before mapping, the Boolean network is represented as a *k-bounded* Boolean network called the *subject graph*. A k-bounded network contains nodes with a maximum fanin size of *k*. *And-inverter graphs* (AIGs) are typically used as subject graphs.

The subject graph is transformed into a mapped network by applying local substitutions to sections of the circuits defined by cuts and extracted using cut enumeration. Mapping addresses two sub-problems: *matching* and *selection*. Matching involves associating cuts with a list of cells that are functionally equivalent to the covered section. This process requires the construction of a specific cell library representation that is suitable for matching. Two main matching approaches exist, namely, *pattern* matching [1] and *Boolean matching* [2]. Selection chooses appropriate cells to cover the graph such that the target cost function is minimized. 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.

Delay-optimal technology mapping algorithms in a load-independent model have been proposed in [4], [5]. These methods have been extended to perform well on discrete-size cell libraries with a gain-based model [11]. Multiple heuristics for area minimization have also been proposed in [12], [13]. In [6], [14], logic decomposition is combined with mapping to reduce the structural dependency on the subject graph. In [15], mapping has been extended to support multi-output cells.

E. Disjoint support decomposition

The *disjoint-support decomposition* (DSD) is a special case of Boolean decomposition. A function f has a DSD decomposition if it can be decomposed such that

$$f(X) = h(X_1, g(X_2)), \quad X_1 \cup X_2 = X \; X_1 \cap X_2 = \emptyset$$

A Boolean function is called *full-DSD* if function g is recursively with disjoint support. Two efficient procedures can be used to find DSD decompositions using 2-input operators, namely *top-decomposition* [16] and *bottom-decomposition* [17].

Top-decomposition finds a decomposition using a 2-input operator $\odot$ applied to a support variable x_i and a remainder function g:

$$f(X) = x_i \odot g(X \setminus \{x_i\}), \quad x_i \in X$$

Bottom-decomposition finds two variables x_i and x_j that uniquely influence f through a 2-input operator:

$$f(X) = h(x_i \odot x_j, g(X \setminus \{x_i, x_j\})), \quad x_i, x_j \in X$$

For instance, the function $f = ((a \vee b) \wedge (c \vee d)) \wedge e$ is top-decomposable for variable e and it is bottom-decomposable for variables (a, b) and (c, d).

III. Matching

In this section, we present our methodology for Boolean and pattern matching. Throughout this paper, we utilize the term *Boolean mapping* to denote a mapping algorithm based on Boolean matching and the term *structural mapping* for a mapping procedure based on pattern matching.

A. Boolean matching

In technology mapping, delay, power, and area can be minimized by exploiting different configurations of cells based on the $\mathcal{NPN}$-equivalence classes [7], [9]. Specifically, permutations increase the number of matches, and negations play a crucial role in the insertion of inverters. Boolean matching relates a canonical representation of a Boolean function to a list of cells that can implement it and is typically defined over $\mathcal{NP}(\mathcal{N})$-equivalence. In a Boolean mapper, Boolean matching is performed during the cut enumeration phase of mapping where a set of cells are associated to a cut given its function.

In line with previous work [7], we define a data structure that facilitates Boolean matching. Such a library is a hash table that relates the functionality represented as a truth table to a set of cells that can implement it. For each cell, the library also stores all its $\mathcal{NP}-$configurations. Given a function f of a cell, its $\mathcal{NP}-$configurations are all the input permutations and inversions applicable to f that generate functions in the $\mathcal{NP}-$equivalence class of f. Specifically, given a Boolean function to match, the library returns a set of cells in the $\mathcal{NP}-$equivalence class of the function along with their $\mathcal{NP}-$configurations.

B. Pattern matching

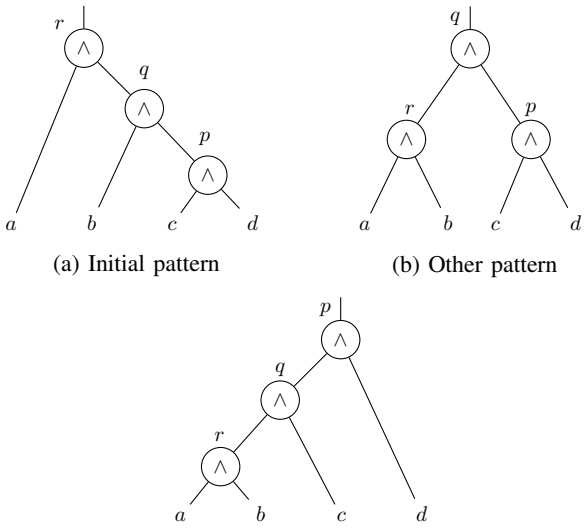
Pattern matching associates a set of cells to a (sub) graph by solving a graph isomorphism problem. A database contains a family of patterns for each cell. In modern technology mapping, a pattern is an AIG decomposition of a cell's function. A cell can be associated with a sub-graph in the subject graph if one of its patterns matches the sub-graph, i.e., it is structurally equivalent.

In technology mapping, sub-graphs to match are described by cuts and extracted using cut enumeration. When cell's patterns are only trees, pattern matching integrates readily with cut enumeration since the pattern identification algorithm is based on dynamic programming. Specifically, patterns at a node n can be identified during the cut merging operation by linking two input cut patterns using the top operation of node n.

To enable pattern matching, we first propose a method to derive the pattern database. Then, we show how to identify patterns during cut enumeration using pattern indexing.

C. Structural patterns derivation

In this work, we define patterns of a cell, called *structural patterns*, as *and-inverter trees* representing the disjoint support decomposition of the cell function. We restrict structural patterns to include only trees since the matching procedure is very efficient. Moreover, most of the cells in technology libraries are full-DSD, such as ANDs, and AND-ORs. A few exceptions are gates whose functions are not full-DSD, such as XORs, MUXs, and Majorities. Thus, it is not possible to produce a structural pattern for those functions, since the decomposition leads inevitably to a DAG structure. Nevertheless, non-full-DSD functions can be specifically identified on the subject graph via structural analysis prior to mapping. This feature is not part of our implementation.



(a) Initial pattern (b) Other pattern

(c) Other pattern

Fig. 1: Possible structural patterns for AND4.

Initially, one structural pattern is derived for each full-DSD cell by applying recursively top decomposition and bottom decomposition for AND and OR operators. A key observation is that decomposition trees are not canonical. Hence, the same function might be represented by different structural patterns. For instance, an AND4 can be expressed as $(a \wedge (b \wedge (c \wedge d)))$, $((a \wedge b) \wedge (c \wedge d))$, or $(((a \wedge b) \wedge c) \wedge d)$ resulting in the three decomposition trees, depicted in Figure 1. Since matching is performed by comparing derived patterns with sub-graphs in the subject graph, generating multiple patterns for such cells is crucial as it translates into additional match opportunities. Thus, we employ an algorithm to derive the other possible patterns from the initial one.

The algorithm derives the different structural patterns for a cell by applying *associative moves* to its initial pattern. An associative move is a tree rotation applied to nodes with the associative property. Algorithm 1 presents the steps to derive multiple patterns. The algorithm recursively applies associative moves to every node of each computed structural pattern and terminates when no new pattern is generated. For instance, supposing the pattern shown in Figure 1a is the pattern initially retrieved by the decomposition, the pattern in Figure 1b can be obtained from the first one by applying an associative move (left rotation) to node r. In order to mitigate the number of derived structural patterns, hence limiting the run time for pattern matching, the algorithm filters structural patterns that are symmetric to others already found by canonicalizing the order of PIs and AND's input. Thus, in the case of the example of Figure 1, the pattern of Figure 1c would not be produced as it is symmetric to the one of Figure 1a.

Algorithm 1 Pattern Derivation

Input pattern og_pat, node $start\_node$, pattern_set set_pat

procedure DERIVE(og_pat, $start\_node$, set_pat)
  if is_pi($start\_node$) **then**
    return
  end if
  $l \leftarrow$ left_fanin($start\_node$)
  $r \leftarrow$ right_fanin($start\_node$)
  if not_negated(l) AND not_pi(l) **then**
    $r\_pat \leftarrow$ right_move(og_pat, $start\_node$)
    if check_symm(r_pat, set_pat) **then**
      add_pat(r_pat, set_pat)
      DERIVE(r_pat, $start\_node$, set_pat)
    end if
  end if
  if not_negated(r) AND not_pi(r) **then**
    $l\_pat \leftarrow$ left_move(og_pat, $start\_node$)
    if check_symm(l_pat, set_pat) **then**
      add_pat(l_pat, set_pat)
      DERIVE(l_pat, $start\_node$, set_pat)
    end if
  end if
  DERIVE(og_pat, l, set_pat)
  DERIVE(og_pat, r, set_pat)
end procedure

D. Pattern indexing and pattern table generation

To identify isomorphic patterns, each node in the structural patterns is assigned to an index. This index serves as a unique identifier for sub-patterns. In practice, if two pattern roots have the same index, it indicates that the patterns are isomorphic.

Patterns are processed in ascending order of size. The procedure indexes nodes of a pattern in topological order. Specifically, each PI is assigned an index of 1. To other nodes, the index is uniquely assigned based on the input indexes and polarities (structural hashing). The order of the inputs is canonicalized for permutation to remove symmetries (e.g., an AND between 1 and 2 is equivalent to an AND between 2 and 1). This procedure neglects the presence of negations and permutations at the PIs and the PO of a pattern. Hence, isomorphic structures in an $\mathcal{NPN}-$equivalence class share the same index.

To illustrate the indexing procedure, let us consider the simple cell library shown in Figure 2, composed of AND2, OR2, and AND4 cells. The first pattern to be indexed is the one associated with the AND2 cell and as such the index 2 is assigned to its only node. Afterward, the pattern corresponding to the OR2 cell is also assigned index 2, since the same structure has already been observed and the input and output negations are ignored. Next, the pattern in Figure 2c is processed assigning new indexes for AND3 (index 3) and AND4 (index 4). Finally, in Figure 2d another pattern of the AND4 cell is elaborated leading to a new index 5.
This procedure naturally exposes the relationship between structures and sub-structures. For instance, from the previous example, we can derive that an AND4 can be described as the AND between two AND2s or a PI and an AND3. From this information, we generate a hash table that expresses for each structure and substructure how they can be obtained by ANDing smaller substructures. Such a table is the *pattern table*. Additionally, we generate another hash table, named the *index table*, which relates the pattern indexes representing structural patterns to corresponding cells. Tables Ia and Ib show the pattern table and index table for the patterns shown in Figure 2.

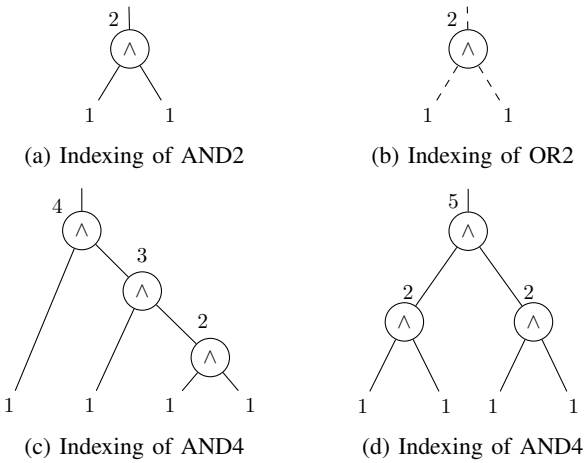


Fig. 2: Indexing of patterns.

Indexes	1	2	3	4	5
1	2	3	4	-	-
2	3	5	-	-	-
3	4	-	-	-	-
4	-	-	-	-	-
5	-	-	-	-	-

(a) Pattern Table

Indexes	Gates
2	AND2; OR2
4	AND4
5	AND4

(b) Index Table

TABLE I: Pattern and Index Table

E. Pattern matching in cut enumeration

During cut enumeration, each cut is assigned to a pattern index that identifies the underlying pattern covered by the cut. Differently from Boolean matching, this operation does not require computing the function of the cut. Instead, the pattern index is computed during the cut merging using the pattern table. Specifically, the pattern index of a cut c, obtained by merging two cuts u and v, is assigned by looking in the pattern table for an entry with the pattern of u and pattern v as fanins. Initially, a trivial cut is associated with the pattern index 1. Then cuts are computed in topological order and patterns are assigned.

Given a cut and its associated pattern index, pattern matching retrieves the set of cells using the index table. Since the cells are in the $\mathcal{NPN}-$equivalence class, correct permutation, and negations are applied during mapping.

IV. Hybrid Matching

In this section, we present our main contribution: *hybrid matching*, a matching algorithm that combines the Boolean and pattern matching techniques presented in Section III, addressing the shortcomings of both strategies and achieving

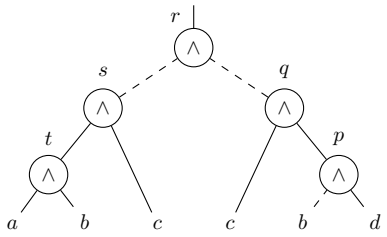


Fig. 3: Subject graph with functional redundancy

better quality-of-results. We first compare the advantages and disadvantages of Boolean and pattern matching to then present the algorithm for hybrid matching.

A. Boolean vs. Structural

Boolean matching typically yields results of superior quality compared to pattern matching for a few reasons. First, Boolean matching is not restricted to full-DSD functions. Second, Boolean matching inherently removes structural redundancies present in the subject graph. However, Boolean matching is also typically slower because it requires computing the Boolean function and matching in the $\mathcal{NP}-$equivalence class. Boolean matching in $\mathcal{NP}-$equivalence is typically addressed by enumerating all the $\mathcal{NP}-$configurations of the cells as explained in Section III-A. This procedure may generate up to $n! \cdot 2^n$ configurations for a cell with n inputs. This has two consequences. First, a Boolean library contains many configurations, leading to more matches and larger mapping time. Second, $\mathcal{NP}-$matching hinders scalability to cells of more than 6-inputs.

Although pattern-matching results are generally of inferior quality, the matching time is significantly smaller. Indeed, pattern matching supports larger cells because its matching complexity depends on the number of patterns generated which is typically small for full-DSD functions after filtering for symmetries. Informally, the number of minimum-size patterns for large full-DSD cells is significantly lower than the NP configurations needed by Boolean matching. Additionally, run time also benefits from the reduced number of matches per cut compared to Boolean matching. Because of this, the technology mapping algorithm is faster during selection. Moreover, despite Boolean matching's capability to detect functional redundancy in the subject graph, pattern matching can sometimes produce better results.

Using an example observed in our experiments, let us suppose that the subject graph presents the structure of Figure 3 and that the ASAP 7nm cell library [18] is employed. The structure presents functional redundancies and implements the following Boolean function: $(\overline{a} \vee \overline{b} \vee \overline{c}) \wedge (\overline{c} \vee b \vee \overline{d})$. Since pattern matching is purely based on the structure, it ignores redundancies and matches the subject graph to the 6-input OA33 cell, which implements the following Boolean expression: $(a \vee b \vee c) \wedge (d \vee e \vee f)$. Differently, Boolean matching detects a functional support of 4 variables and consequentially searches for 4-input cells to match. However, due to variable b being binate (present in two polarities), such a cell cannot be found, leading to a mapping that employs an AND3 and an AOI31 cell. This causes lower quality of results for both area and delay compared to pattern matching.

Algorithm 2 Hybrid Matching

```
1: Input: Boolean library bool_lib, pattern library pat_lib, Boolean
   network N, cut size Boolean k, cut size pattern l, bool
   do_pattern, bool do_bool
2: Output: Match Set match_set
3: if do_pattern then
4:     foreach node n ∈ N do
5:         cut_pat[n] ← pattern_cuts_merge(cut_pat, n, l)
6:         match_pat[n] ← pattern_match(cut_pat[n], pat_lib)
7:     end for
8: end if
9: if do_bool then
10:    foreach node n ∈ N do
11:        cut_bool[n] ← bool_cut_merge(match_set, n, k)
12:        match_bool[n] ← bool_match(cut_bool[n], bool_lib)
13:        match_set[n] ← union(match_pat[n], match_bool[n])
14:    end for
15: end if
16: return match_set
```

B. Integrating Boolean and pattern matching

Initially, both pattern and Boolean libraries are generated as described in Section III. In hybrid matching, two distinct phases of cut enumeration are performed, one for pattern matching and another for Boolean matching. Algorithm 2 shows the algorithm to compute cuts and matches. First, cuts for pattern matching are enumerated and matched for every node of the subject graph following the procedure of Section III-E. Subsequently, for every node of the subject graph, cuts for Boolean matching are enumerated and matched as explained in Section III-A. However, at this step, the algorithm joins the set of Boolean and structural cuts together with their matches. The union and cut merge operation follows the priority cuts paradigm [8] for which a limited number of cuts are saved for each node and are sorted according to a cost function depending on delay, area, and size. This is a crucial point in the algorithm, as this procedure allows us to combine the best results of both Boolean and pattern matching, overcoming the respective shortcomings. Note that Boolean cuts are computed starting from the merged cut sets. Additionally, Hybrid matching can optionally perform only Boolean or pattern matching, by selecting the parameters do_bool and do_pat.

Algorithm 2 requires two distinct phases of cut enumeration and matching since cuts for Boolean or pattern matching have different characteristics. Cuts for Boolean matching require the computation of the truth table, are limited to 6 inputs, and have redundancies removed in the support. Contrarily, cuts for pattern matching need only a pattern index, are canonicalized on symmetries, and keep functional redundancies. Given these differences, the two types of cuts are incompatible during the cut merging operation.

V. Experimental Results

In this section, we evaluate the performance of hybrid matching in a technology mapper. We compare it to the state-of-the-art Boolean mapper *map* implemented in *ABC*[1], which represents the baseline of our experiments. Furthermore, we compare the hybrid approach against our Boolean and pattern matching method by changing the parameters do_bool and do_pat of Algorithm 2.

A. Setup

The Boolean, structural, and hybrid matching methods have been implemented in C++17 as an extension to the technology mapper in [7] and are available in the open-source logic synthesis framework *Mockturtle*[2] [3]. For the experiments, we use the EPFL combinational benchmark suite [19] containing combinational circuits in the form of AIGs. All the results were verified using the combinational equivalent checker in *ABC*. We employ the ASAP7 7nm cell library [18], pre-processed by OpenLane[3]. The experiments were conducted on WSL (Windows Subsystem for Linux) version 1.0 on a 2.5GHz Intel i5 dual-core. For every benchmark, we provide the area and delay results and the total run time. The maximum cut size for Boolean matching is 6, and for pattern matching is 9. Moreover, a maximum of 16 cuts are stored for each node.

B. Delay and Area-oriented Mapping

The results for delay-oriented mapping are shown in Table II. As can be seen, the hybrid mapper obtains an average area improvement of around 9% compared to the mapper in ABC, 6% compared to the Boolean mapper, and 4% compared to the structural mapper. This is made possible by the usage of both Boolean and pattern-matching cuts during mapping. On the one hand, in benchmarks such as *div*, *max*, and *sin*, the presence of large cuts extracted using pattern matching determines a considerable area reduction over the Boolean approach. On the other hand, for benchmarks such as *voter* and *hyp*, the employment of XOR and Majority cells is the reason for better results compared to the structural mapper. Delay results are similar between the hybrid, Boolean, and ABC mappers for most benchmarks. Only for a few of them, the hybrid mapper achieves slightly worse delay values, resulting in a negative average improvement over the Boolean mapper. This is mainly due to the absence of NP-configurations in structural patterns, resulting in fewer matches being found by pattern matching. Nonetheless, the small negative result is greatly outbalanced by the noticeable area reduction, which peaks at 34% in the case of *bar*. Regarding run time, the hybrid mapper achieves an average speedup of $1.27x$ compared to the Boolean mapper, and $1.54x$ compared to the ABC mapper. This is mainly caused by the employment of structural cuts. These cuts typically present fewer matches than the Boolean ones. Hence, on average, the number of matches per node reduces, decreasing also the run time during *selection*. Indeed, a structural mapper achieves the lowest run time.

The results for area-oriented mapping are shown in Table III. Similar considerations apply with the only difference of enhanced delay results when using hybrid matching.

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

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

[3]Available at: https://github.com/The-OpenROAD-Project/OpenLane

VI. Conclusion

In this work, we presented a method that combines the strengths of pattern and Boolean matching to achieve better quality. On the one hand, hybrid matching supports large standard cells. On the other hand, it offers superior quality compared to pattern matching. In the experiments, we integrated hybrid matching in a technology mapper showing better average area and run time results compared to Boolean matching for similar delay values. Moreover, hybrid matching proved to achieve results of superior quality for both area and delay compared to pattern matching on average.

Acknowledgments

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

References

[1] K. Keutzer, "DAGON: Technology binding and local optimization by DAG matching," in *Proc. DAC*, 1987.

[2] F. Mailhot and G. De Micheli, "Technology mapping using boolean matching and don't care sets," in *EURO-DAC*, 1990.

[3] 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.

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

[5] L. Stok, M. Iyer, and A. Sullivan, "Wavefront technology mapping," in *Proc. DATE*, 1999.

[6] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping," in *Proc. ICCAD*, 2005.

[7] 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.

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

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

[10] G. De Micheli, "Technology mapping of digital circuits," in *Proc. Advanced Computer Technology, Reliable Systems and Applications*, 1991.

[11] B. Hu, Y. Watanabe, and M. Marek-Sadowska, "Gain-based technology mapping for discrete-size cell libraries," in *Proc. DAC*, 2003.

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

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

[14] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, "Logic decomposition during technology mapping," *Trans. CAD*, 1997.

[15] A. T. Calvino and G. De Micheli, "Technology mapping using multi-output library cells," *Proc. ICCAD*, 2023.

[16] Bertacco and Damiani, "The disjunctive decomposition of logic functions," in *Proc. ICCAD*, 1997.

[17] V. Callegaro, F. S. Marranghello, M. G. A. Martins, R. P. Ribas, and A. I. Reis, "Bottom-up disjoint-support decomposition based on cofactor and boolean difference analysis," in *Proc. ICCD*, 2015.

[18] 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.

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

TABLE II: Results for Boolean, structural, hybrid matching for delay-oriented technology mapping. Green values represent dominant values for area or delay.

Benchmark	**ABC *map***		**Boolean**		**Structural**		**Hybrid**	
	Area	**Delay**	**Area**	**Delay**	**Area**	**Delay**	**Area**	**Delay**
adder	92.53	2577.43	84.23	2573.43	89.83	2573.79	96.74	2572.58
bar	325.63	168.08	356.60	168.08	214.52	157.72	233.82	172.61
div	5597.18	43765.70	5362.89	43769.81	4626.64	44048.23	4587.79	43845.29
hyp	16373.78	195822.72	15158.98	195345.80	15997.90	195798.20	15329.77	195256.90
log2	2158.33	3954.51	2044.40	3846.34	2041.99	4109.75	1808.55	3913.82
max	226.59	2213.23	211.56	2213.23	195.03	2257.11	182.24	2232.52
multiplier	1942.83	2738.95	1909.41	2669.16	1900.97	2726.37	1686.94	2661.85
sin	425.40	1819.24	456.31	1760.28	434.31	1854.80	401.59	1817.37
sqrt	1838.48	47266.98	1883.23	47264.79	1911.20	47580.56	1838.26	47288.87
square	1193.80	2516.39	1104.55	2505.10	1218.89	2521.89	1123.76	2503.20
arbiter	766.68	898.75	766.32	898.75	766.32	898.75	766.41	898.75
cavlc	41.55	187.04	40.23	187.04	39.21	186.44	38.85	186.07
ctrl	8.76	102.49	8.26	102.49	10.63	120.31	8.29	101.21
dec	30.83	65.72	30.83	65.72	30.11	66.19	27.44	66.15
i2c	79.05	182.65	78.06	182.65	71.75	184.57	72.29	182.65
int2float	13.87	181.00	13.37	181.00	12.08	181.17	12.12	181.00
mem_ctrl	2761.82	1107.42	2733.28	1100.46	2575.61	1130.11	2578.26	1118.13
priority	86.64	2501.95	83.70	2501.95	81.87	2510.03	81.95	2501.95
router	18.89	278.53	19.36	274.05	20.23	278.83	19.14	274.05
voter	1522.27	741.93	1595.57	782.75	1948.72	749.02	1510.62	745.68
Total time (s)	34.20		21.50		4.78		15.61	
Ratio	1.000	1.000	0.970	0.994	0.949	1.014	**0.907**	0.998

TABLE III: Results for Boolean, structural, hybrid matching for area-oriented technology mapping. Green values represent dominant values for area or delay.

Benchmark	**ABC *map -a***		**Boolean**		**Structural**		**Hybrid**	
	Area	**Delay**	**Area**	**Delay**	**Area**	**Delay**	**Area**	**Delay**
adder	57.40	3548.84	57.40	3548.84	73.88	2946.34	57.40	3549.45
bar	191.81	238.53	156.90	218.45	143.64	201.59	143.64	201.59
div	4047.23	79696.65	3912.10	80724.15	3865.42	91466.62	3644.84	66682.48
hyp	14365.80	300293.70	13301.86	247283.50	14196.88	380409.30	13218.19	253042.40
log2	1637.35	6620.43	1532.25	5406.38	1663.60	7944.24	1525.40	6144.87
max	166.18	2862.23	158.84	2967.26	147.29	3779.87	143.52	2955.95
multiplier	1495.42	4985.99	1318.02	3581.94	1466.00	5710.86	1331.50	3619.38
sin	308.47	3053.09	287.06	2806.80	292.89	3481.79	281.03	2865.45
sqrt	1463.56	107677.55	1370.76	112605.30	1377.46	100853.50	1367.95	112662.50
square	1167.34	3711.58	1066.28	3512.69	1132.82	3612.90	1070.01	3626.51
arbiter	569.40	1018.69	557.84	999.87	557.84	1015.47	557.72	999.87
cavlc	8.16	131.57	36.16	233.30	34.75	247.51	34.93	242.30
ctrl	8.16	131.57	7.96	125.09	9.40	154.49	7.38	127.51
dec	27.5	85.83	27.14	72.30	27.14	72.30	27.14	72.30
i2c	78.20	219.62	74.04	256.62	68.20	266.62	68.99	265.66
int2float	12.88	205.02	12.39	206.51	11.34	210.38	11.27	217.47
mem_ctrl	2673.51	1800.28	2538.88	1975.53	2352.34	1679.64	2355.64	1727.51
priority	61.95	2795.01	59.48	3635.47	51.11	2943.53	51.06	2920.70
router	14.77	473.94	13.33	414.48	13.87	427.66	13.11	386.80
voter	905.61	1266.69	807.67	1197.85	862.19	1323.55	815.19	1170.45
Total time (s)	35.87		22.20		4.77		16.54	
Ratio	1.000	1.000	0.937	0.966	0.954	1.049	**0.902**	**0.947**