Fast Hierarchical NPN Classification

Ana Petkovska*, Mathias Soeken*, Giovanni De Micheli*, Paolo Ienne*, and Alan Mishchenko†
*Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland
†University of California, Berkeley, USA
{ana.petkovska, mathias.soeken, giovanni.demicheli, paolo.ienne}@epfl.ch, alanmi@berkeley.edu

***Abstract*—Classifying functions according to some common properties into libraries of functions is an important step in many logic synthesis and technology mapping algorithms used in FPGA design flows. NPN classification is one of the frequently used classifications. Existing algorithms for NPN classification perform a sequence of steps to derive the resulting NPN class, but discard the intermediate results produced at the end of each step. The hierarchical method introduced in this paper uses the same sequence of steps, but it saves the intermediate results at each step and reuses them when classifying other functions. It is, on average, 3.7 times faster compared to a state-of-the-art non-hierarchical method, at the cost of a modest increase in memory needed to save the class hierarchy. The hierarchical approach enables a rapid exact NPN classification for functions up to 10 inputs—it exactly classifies one million 6-input functions in the same time as the heuristic state-of-the-art algorithm.**

I. Introduction

Negation-Permutation-Negation (NPN) classification assigns two Boolean functions into the same NPN class if one can be obtained from the other by negating (i.e., complementing) and permuting inputs, and negating the output. The number of NPN classes is much lower than the number of functions.

In practice, NPN classification reduces the size of the library of functions with some desirable properties used in synthesis [1], [2] and mapping [3], [4] for FPGAs. NPN classification is used, first, as a precomputation step, to build the library of known functions and second, as a Boolean matching algorithm returning the class for each function encountered while running the algorithm on a given design.

Due to its importance, NPN classification is a well-studied problem for which different heuristics and exact algorithms already exist [5–12]. Although some of these algorithms are composed of several methods that compute intermediate representations of the function, they do not save nor use this information. For example, if an algorithm consists of two methods for transformations, A and B, then they are executed one after the other for each function that has to be classified and the final result is a set of NPN classes, as shown on Figure 1a. The intermediate results from A are not saved for future calls, but only used as input to B and discarded.

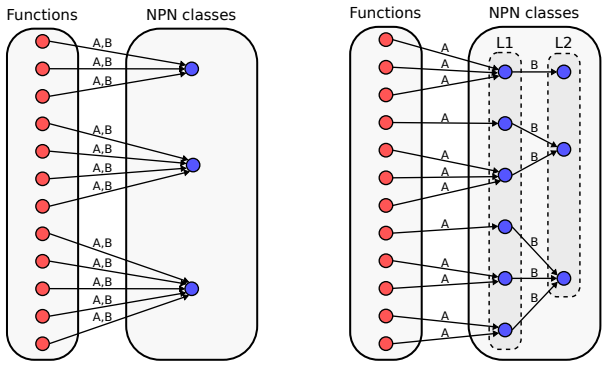
On the contrary, we propose to memoize intermediate representations by building a hierarchy of classes. For the previous example, as Figure 1b shows, the results from A form an intermediate level of classes, while the final classes are obtained by executing B additionally. For each class from the intermediate levels, we save information that leads to the final NPN class to which it would merge by executing the additional methods. Thus, once the hierarchy contains enough information, the final NPN class is obtained faster—in the best case, only the first method of the NPN algorithm is executed, while the following methods are required only if there is no match with an existing class at the corresponding level.

The idea of the hierarchical approach is independent from the NPN algorithm and can be applied to any algorithm composed of several methods. Moreover, it allows creating a hierarchy of different NPN algorithms—fast heuristics can be used in the higher levels as a preprocessing step to obtain intermediate classes, while slower (pseudo-)exact methods can generate the final NPN classes in the lowest level. In this manner, as an independent contribution of this paper, we propose an exact method that enables fast exact NPN classification when being used to obtain the final NPN classes in a hierarchy.

Following, Section II gives background information. In Section III, we describe the hierarchical NPN classification, and propose strategies for exact classification. Section IV gives the experimental results. Section V concludes the paper.

II. Background Information

In this paper, we use the terms *Boolean function*, *truth table*, *cofactor*, and *disjoint-support decomposition (DSD)* as defined by Huang et al. [12]. Following we give the terminology associated with NPN equivalence and canonical form.



(a) Flat organization. (b) Hierarchical organization.

Figure 1: NPN classification of Boolean functions. (a) In the existing algorithms, all classes belong to one level and functions are directly matched to these final NPN classes. (b) Having a hierarchy of classes allows matching the functions to intermediate classes at the first level (L1), which are further matched into final NPN classes from the second level (L2).

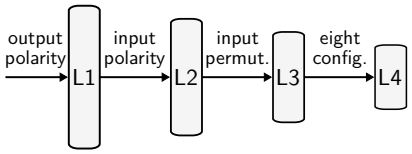


Figure 2: Hierarchical organization of the classes and methods of the heuristic algorithm for NPN classification.

Two single-output functions f and g are *NPN equivalent* if one can be obtained from the other by negating and permuting inputs, and negating the output. The *NPN class* of a function f with n inputs contains at most $n! \cdot 2^{n+1}$ functions that can be obtained from f. The *representative* of an NPN class is a function that is selected uniquely and gives the *NPN canonical form* of any function from the class. To find the NPN class of a function f, we should determine the required transformations, for which the truth table of f equals the truth table of the class representative. For a function $f(x_1, \ldots, x_n)$, a *permutation vector* $p = (p_1, \ldots, p_n)$ encodes the required permutation of inputs by recording the position for each input, and a *negation vector* $q = (q_0, q_1, \ldots, q_n)$ encodes the polarity for the output with q_0 and the polarity for the inputs on the corresponding positions with q_i, where $1 \leq i \leq n$. Initially, $p = (1, 2, 3, \ldots, n)$ as each input is on the initial position, and $q = (0, 0, 0, \ldots, 0)$ as the output and inputs are not negated.

III. Hierarchical NPN Classification

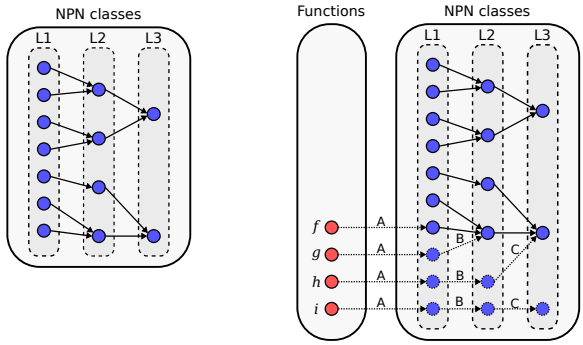
In this section, first, we describe the state-of-the-art heuristic algorithm [12] used to demonstrate our hierarchical approach. Then, we explain how to build a hierarchy of classes, and we propose a fast algorithm for exact NPN classification.

A. Methods for Heuristic NPN Classification

As the basis for our algorithm, we use the heuristic version of the state-of-the-art algorithm for NPN classification by Huang et al. [12], which is used in several recent publications [2], [4]. The first two methods of the algorithm determine the polarity of the output and the polarity of each input by counting the number of 1s in the truth table and cofactors, respectively. Then, the third method orders the inputs in increasing order by the number of 1s in the positive cofactor. However, if some inputs have equal number of 1s, then they form a *tied group* and their position is ambiguous. Thus, the fourth method performs a series of negations and permutations within each tied group to minimize the integer value of the function's truth table.

B. Building a Hierarchy of NPN Classes

The proposed hierarchical approach creates a hierarchy of NPN classes by saving intermediate results (i.e., truth tables). As Figure 2 shows, in a hierarchical version of the heuristic algorithm [12], each method creates one level of NPN classes in the hierarchy. For each level, a hash table stores the truth tables of the classes' representatives in that level. The hierarchy of classes allows to stop the execution as soon as the function equals a representative from a level of the hierarchy. For example, Figure 3 shows how a hierarchy is updated after the functions f, g, h, and i are matched and added. The function f shows the best-case scenario as it is matched in the first level, after executing A, while, the functions h and i show the worst-case scenario, when all methods are executed.



(a) (b)

Figure 3: Adding four functions f, g, h, and i to a hierarchy of classes using an algorithm that consists of three methods A, B and C. (a) The hierarchy of classes before adding the four functions. (b) The classes added to the hierarchy for the three new functions are given with dotted lines. Over the dotted lines we give the method executed for classifying each function.

In the hierarchical approach, when we have a match at some intermediate level m in the hierarchy, the computed vectors p_m and q_m encode only the transformations required to generate the representative of the intermediate class. The final transformations are generated by additionally applying $p_{m;c}$ and $q_{m;c}$ that transform the intermediate representative of the class c from level m to the representative of its final class. Thus, once we have a match, the final vectors are computed as $p = \text{permute}(p_m, p_{m;c})$ and $q = \text{permute}(q_m, p_{m;c}) \oplus q_{m;c}$ where $\text{permute}(a, b)$ orders the elements of a as defined by b.

C. Towards a Fast Exact NPN Classification

The naïve way to perform exact NPN classification, for a function with n inputs, is to try exhaustively all $n! \cdot 2^{n+1}$ transformations, and select the one that leads to the minimum integer value of the truth table. However, this is computationally hard and prohibitively slow even for functions with few inputs.

The concept for building a hierarchy of classes is not limited to any method for NPN classification. We can easily extend the hierarchy with additional methods that refine the results by merging classes. Thus, to perform an exact NPN classification, we propose to pre-classify the functions by a fast heuristics, as proposed in Section III-B, and then to produce the final level of classes using an algorithm for exact NPN classifications as a last method of the hierarchy. The runtime benefit of such classification is presented in Section IV-B.

D. Group-Based Exact NPN Classification

Instead of executing the exhaustive exact NPN classification, we propose the following exact algorithm that uses the tied groups formed by the third method from Section III-A. After

TABLE I: Comparison of the runtime and memory usage of the algorithm `OrigH` and its hierarchical implementation `HierH`.

	#Vars	#Func	#Unique functions		#Classes	Time (s)		Speed-up	Memory (MB)		#Classes per level for `HierH`			
						`OrigH`	`HierH`		`OrigH`	`HierH`	1	2	3	4
	6	1M	40195	4.0%	204	0.28	0.10	2.80	0.03	0.44	38844	5845	1190	204
	8	1M	81864	8.2%	1344	0.80	0.22	3.64	0.13	4.00	81244	34994	4160	1344
Full	10	100K	19723	19.7%	1723	0.19	0.09	2.11	0.50	5.00	19699	10579	2651	1723
DSD	12	100K	20245	20.2%	3157	0.68	0.30	2.27	2.00	22.00	20239	11724	4394	3157
	14	10K	5270	52.7%	891	0.38	0.20	1.90	8.00	40.00	5270	2369	1094	891
	16	10K	3211	32.1%	1057	1.58	0.69	2.29	32.00	128.00	3211	2293	1332	1057
	6	1M	44493	4.4%	2254	0.36	0.10	3.60	0.03	0.63	43783	20590	5229	2254
	8	1M	77312	7.7%	14270	1.50	0.24	6.25	0.50	5.50	77188	57452	23336	14270
Partial	10	100K	17325	17.3%	6620	0.32	0.08	4.00	1.00	7.00	17317	14287	9194	6620
DSD	12	100K	18383	18.4%	8545	1.32	0.34	3.88	6.00	30.00	18381	15455	10952	8545
	14	10K	5930	59.3%	2482	0.53	0.31	1.71	8.00	48.00	5930	4655	3131	2482
	16	10K	5889	58.9%	3110	2.57	1.65	1.56	32.00	192.00	5889	5233	3744	3110
	6	1M	5909	0.6%	1748	0.32	0.09	3.56	0.03	0.19	5859	4506	2638	1748
	8	1M	5745	0.6%	2949	0.99	0.14	7.07	0.13	0.75	5730	5182	4091	2949
Non	10	100K	3926	3.9%	2016	0.49	0.05	9.80	0.50	2.00	3914	3613	3055	2016
DSD	12	100K	2517	2.5%	1409	1.41	0.16	8.81	2.00	8.00	2517	2345	1932	1409
	14	10K	1624	16.2%	980	0.50	0.11	4.55	8.00	32.00	1624	1522	1372	980
	16	10K	358	3.6%	282	2.69	0.28	9.61	32.00	128.00	358	346	325	282
Geomean						**0.70**	**0.19**	**3.72**	**1.19**	**8.57**				

this method, the position is deterministically defined only for untied inputs that form groups with one element, which we call *singleton inputs*. As the order of the groups is deterministically identified, it suffices to determine the order of the inputs within each group. So, instead of trying exhaustively all $n!$ permutations, we only need to try $\prod_{i=1}^{k} g_i!$ permutations, where k is the number of groups and g_i is the number of inputs in the group i, for $1 \leq i \leq k$, and $n = \sum_{i=1}^{k} g_i$. However, in some cases, the polarities of the output and singleton inputs are not uniquely defined. To minimize the integer value of the truth table, we have to try both of their polarities. The final truth table is not minimum, but it is minimal and unique given the permutations of the third method.

Finally, in the worst case, all inputs belong to one group and this algorithm has the same complexity as the exhaustive exact NPN classification, while in the best case, only singleton groups exist for which we require only 2^{n+1} transformations.

IV. Experimental Results

ABC [13] contains implementation of the heuristic state-of-the-art algorithm [12] (command *testnpn -A 5 <input file>*) and the exhaustive exact algorithm (command *testnpn -A 1 <input file>*). We refer to these algorithms as `OrigH` and `OrigE`, respectively. The proposed algorithm from Section III-B, `HierH`, is the hierarchical version of `OrigH` and is added to ABC (command *testnpn -A 7 <input file>*). The exact algorithm from Section III-C, `HierE1`, is `HierH` with `OrigE` added as last method. Finally, `HierE2` differs from `HierE1` by using the algorithm from Section III-D as last method.

As benchmarks we use the sets of Boolean functions used by Huang et al. [12]. They are divided by their DSD properties [14] (functions with full DSD, partial DSD, and non-DSD), and by the number of input variables (even values from 6 to 16).

For the efficiency of the algorithms, we compare the runtime for Boolean matching while building a library of known functions. The algorithms classify each function, considering that the functions are produced on-the-fly as in practical applications, such as technology mapping [3]. A timeout is reported if the runtime exceeds 12 hours. We also compare the memory required for the hash tables that store the class representatives. Additionally, for the hierarchical algorithms, a permutation and a negation vector are saved for each class. But, this memory is negligible compared to the one required for the hash tables, and thus, it is omitted from the results.

A. Comparison to a State-of-the-Art Heuristic Algorithm

Table I compares the runtime and memory usage of the algorithm `OrigH` and its hierarchical implementation `HierH`. Since both versions of the algorithm execute the same steps to generate the representative of each function, the final number of classes, "#Classes", is identical for both versions. This proves the correctness of the hierarchical algorithm `HierH`.

Regarding the runtime, "Time (s)", on average over all runs, `HierH` is 3.7 times faster than `OrigH`. Regarding memory usage, "Memory (MB)", `OrigH` saves only the last level of classes, and thus it uses, on average, 7.2 times less memory than `HierH`, which saves the classes derived from each method. But, practically, the maximum increase in memory is only 160 MB, which is easily affordable nowadays.

B. Hierarchical Exact NPN Classification

Table II compares the runtime and memory usage of the exact algorithms `OrigE` and `HierE1`. As before, both `OrigE` and `HierE1` generate the same final classes, and thus the number of classes is identical for both algorithms.

In ABC, the command for `OrigE` first removes the duplicate functions, and then classifies only unique functions. Yet, this is only possible if all functions are precomputed, but in practical applications they are usually generated on-the-fly. To make a fair comparison, we evaluate both cases under "Unique

TABLE II: Speed-up for exact NPN classification of function with 6 and 8 inputs.

	#Vars	#Func	#Classes	All functions Time (s) OrigE	All functions Time (s) HierE1	All functions Speedup	Unique functions only Time (s) OrigE	Unique functions only Time (s) HierE1	Unique functions only Speedup	Memory (MB) OrigE	Memory (MB) HierE1
Full DSD	6	1M	191	1983.03	0.39	5084.69	69.61	0.45	154.69	0.03	0.47
	8	1M	1274	>12h	725.91	-	43486.15	716.72	60.67	0.13	4.13
Partial DSD	6	1M	2103	2345.11	4.00	586.28	73.11	4.14	17.66	0.03	0.66
	8	1M	13923	>12h	7310.89	-	41012.60	7718.16	5.31	0.50	6.00
Non DSD	6	1M	1673	1992.16	2.71	735.11	8.98	3.09	2.91	0.03	0.22
	8	1M	2836	>12h	1534.50	-	3111.02	1509.05	2.06	0.13	0.88
Geomean				**2100.26**	**57.04**	**1298.90**	**795.58**	**60.30**	**13.19**	**0.08**	**1.07**

TABLE III: Group-based exact NPN classification.

	#Vars	#Func	#Classes	Time (s)	Speedup
Full DSD	6	1M	191	0.20	1.95
	8	1M	1274	59.34	12.23
	10	100K	1707	9229.80	-
Partial DSD	6	1M	2103	0.50	8.00
	8	1M	13923	136.27	53.65
	10	100K	6494	6304.21	-
Non DSD	6	1M	1673	0.26	10.42
	8	1M	2836	19.21	79.88
	10	100K	1904	4456.96	-
Geomean				**46.66**	**14.29**

functions only" and "All functions", respectively. As shown, `HierE1` is significantly faster when the functions are generated on-the-fly, and it is 13 times faster if only unique functions are considered. However, `HierE1` still requires more than 12 hours to classify the sets of 10-input functions.

C. Hierarchical Group-Based Exact NPN Classification

As `HierE2` performs less transformation whenever the inputs are divided in more than one group, and as Table III shows, it is 14.3 times faster than `HierE1`. Remarkably, for the sets with 6-input functions, on average, `HierE2` produces the exact classification in 0.30 seconds, while the state-of-the-art heuristic `OrigH` requires 0.32 seconds. But, even `HierE2` is not scalable enough for classifying functions with more than 10 inputs and calls for a smarter method for exact classification.

V. Conclusion

In this paper, we present a new approach for fast NPN classification based on building a hierarchy of classes. The approach is general and improves the runtime of different algorithms for NPN classification for a modest increase in memory usage. But, NPN-based applications in logic synthesis [1], [2] and technology mapping [3], [4], [15], which are used in FPGA design flow, can afford to use up to 160 MB more memory to benefit from the speed-up in runtime. Finally, it is astonishing that the proposed hierarchical algorithm for exact classification is as fast as a state-of-the-art heuristic when they perform Boolean matching for one million 6-input practical functions. But, exactly classifying functions with more than 10 inputs still is a problem that we plan to address in the future.

Acknowledgments. This work was partly supported by NSF/NSA grant "Enhanced equivalence checking in cryptoanalytic applications" at University of California, Berkeley, and partly by H2020-ERC-2014-ADG 669354 CyberCare.

References

[1] A. Kennings, A. Mishchenko, K. Vorwerk, V. Pevzner, and A. Kundu, "Efficient FPGA resynthesis using precomputed LUT structures," in *Proceedings of the 20th International Conference on Field-Programmable Logic and Applications*, Milano, Aug. 2010, pp. 532–37.

[2] W. Yang, L. Wang, and A. Mishchenko, "Lazy man's logic synthesis," in *Proceedings of the International Conference on Computer Aided Design*, San Jose, Calif., Nov. 2012, pp. 597–604.

[3] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts," in *Proceedings of the International Conference on Computer Aided Design*, San Jose, Calif., Nov. 2007, pp. 354–61.

[4] A. Mishchenko, R. Brayton, W. Feng, and J. W. Greene, "Technology mapping into general programmable cells," in *Proceedings of the 23rd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, Calif., Feb. 2015, pp. 70–73.

[5] E. Goto and H. Takahasi, "Some theorems useful in threshold logic for enumerating Boolean functions," in *International Federation for Information Processing Congress*, Aug. 1962, pp. 747–52.

[6] M. A. Harrison, *Introduction to Switching and Automata Theory*. New York: McGraw-Hill, 1965.

[7] L. Benini and G. De Micheli, "A survey of Boolean matching techniques for library binding," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 2, no. 3, pp. 193–226, 1997.

[8] U. Hinsberger and R. Kolla, "Boolean matching for large libraries," in *Proceedings of the 35th Design Automation Conference*, San Francisco, Calif., Jun. 1998, pp. 206–11.

[9] D. Debnath and T. Sasao, "Efficient computation of canonical form for Boolean matching in large libraries," in *Proceedings of the Asia and South Pacific Design Automation Conference*, Yokohama, Japan, Jan. 2004, pp. 591–96.

[10] A. Abdollahi and M. Pedram, "A new canonical form for fast Boolean matching in logic synthesis and verification," in *Proceedings of the 42nd Design Automation Conference*, Anaheim, Calif., Jun. 2005, pp. 379–84.

[11] D. Chai and A. Kuehlmann, "Building a better Boolean matcher and symmetry detector," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Munich, Mar. 2006, pp. 1079–84.

[12] Z. Huang, L. Wang, Y. Nasikovskiy, and A. Mishchenko, "Fast Boolean matching based on NPN classification," in *Proceedings of the IEEE International Conference on Field Programmable Technology*, Kyoto, Dec. 2013, pp. 310–13.

[13] *ABC: A System for Sequential Synthesis and Verification*, Berkeley Logic Synthesis and Verification Group, Berkeley, Calif., Mar. 2016, `http://www.eecs.berkeley.edu/~alanmi/abc/`.

[14] R. Ashenhurst, "The decomposition of switching functions," in *Proceedings of the International Symposium on the Theory of Switching*, Cambridge, Mass., Apr. 1957, pp. 74–116.

[15] F. Mailhot and G. De Micheli, "Algorithms for technology mapping based on binary decision diagrams and on Boolean operations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 5, pp. 599–620, 1993.