

IPRec and Isoblaze: Fuzzy Subcircuit Isomorphism for IP Detection in Physical Netlists

Dallin Dahl, Corey Simpson*, Keenan Faulkner, Brent Nelson, and Jeffrey Goeders

Department of Electrical and Computer Engineering

Brigham Young University

Provo, Utah, USA

djdahl@byu.edu, coreyrs@student.byu.edu, keenanrf@student.byu.edu, ben53@byu.edu, jgoeders@byu.edu

Abstract—Netlist reverse engineering has many uses, from detecting hardware trojans to recovering missing design source files. However, the basic problem of finding IP in a netlist has not been widely discussed. The problem boils down to subgraph isomorphism on graphs constructed from netlists. We present two approaches to identifying IP in larger circuits. IPRec focuses on exploiting hierarchy in the IP design and is a rather conservative approach, while Isoblaze focuses on local properties and connectivity and is more liberal in matching.

I. INTRODUCTION

Netlist reverse engineering is the process of understanding the structure and functionality of a hardware design from a low-level netlist. This netlist may originate from a physical ASIC design (eg GDSII design files, or delayering a part), or may be extracted from a bitstream of an FPGA using modern-day bitstream-to-netlist tools [1], [2], [3]. While reverse engineering a design can be motivated with malicious intent (eg. stealing IP, planning Trojan attacks), it is also a necessary technology for many useful and ethical purposes, including: detecting hardware Trojans, design equivalence checking, detecting IP infringement, recreating obsolete or missing design source files, and more [4], [5].

While a physical netlist may be functionally identical to the original RTL source, and may simulate with the same behavior, the netlist is missing significant information about the intent of the design, that is naturally present in the original RTL design. For example, a netlist obtained from an FPGA bitstream would have no net or instance names, no wire groupings or buses, no hierarchical information (including information about IP instances), no obvious distinction between control and data structures, and no functional or logical locality; all of which would be present in an original RTL design.

Recent research has explored methods to reconstruct this information, including techniques for identifying known circuit modules [6], locating larger macro objects such as register files, counters, adders, and subtractors [7], as well as reconstructing the grouping of signals that make up multi-bit words [8], [9], [10].

This work was funded by the Office of Naval Research and GrammaTech Inc. under grant N68335-20-C-0569.

*Simpson was with Brigham Young University at the time of this work, but is now with NVIDIA.

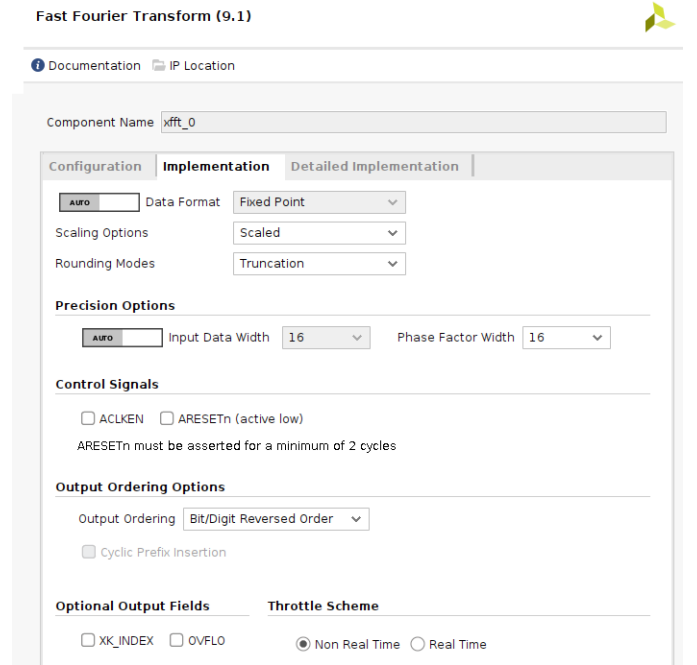


Fig. 1. FFT configuration options in Vivado

In this paper we focus on locating instances of known intellectual property (IP) cores in the netlist. In particular we present and evaluate algorithms for detecting *configurable* IP cores, which are common in FPGA designs. FPGA vendors include a large library of IP with their provided CAD tools, and each of these IP typically have multiple configuration options. For example, Fig. 1 shows a subset of the configuration options for the Fast Fourier Transform component (just one of three configuration tabs are shown in the figure).

These configuration options allow for control over which features are enabled in the IP, performance targets, resource usage types, and widths of data paths. Given the number of different configuration options for many IP, it is often not possible to enumerate and catalog all possible RTL implementations; the space of all possible configurations is simply too large.

Given this challenge, we present two different algorithms in this paper that take different approaches at solving this

problem. Neither algorithm is capable of performing perfect IP detection, but they aim to provide some level of classification of the IP in a larger netlist.

The main contributions of this work are:

- 1) An algorithm for detecting configurable IP cores in a netlist (*IPRec*), based on matching individual levels of hierarchy of the IP of interest. This algorithm is relatively slow, but capable of detecting configurations of the IP that it has never seen before.
- 2) A faster algorithm (*Isoblaze*) for detecting IP cores in a netlist, based on hashing cell properties and connection patterns. This algorithm can scale to more reasonable problem sizes, but is more reliant on seeing the exact configuration of the IP in the past.
- 3) An evaluation and comparison of the accuracy and runtime of the two approaches, in an experiment where five different IP cores are located in a larger bus-based processor design.

The remainder of this paper is organized as follows: Section II provides background information on subcircuit isomorphism. Section III provides information on previous work in circuit reverse engineering. Section IV describes the *IPRec* algorithm, and Section V describes the *Isoblaze* algorithm. Section VI presents the evaluation of the two algorithms, and Section VIII concludes the paper.

II. BACKGROUND

Subcircuit isomorphism is the application of subgraph isomorphism to circuits. In general, subgraph isomorphism is NP-complete, and while algorithms do exist to compute it ([11], [12], [13], [14], [15]), they are very intricate and have prohibitive worst-case performance.

To achieve more reliable performance, we have developed fuzzy approaches that approximate a rigorous subgraph isomorphism, while keeping runtimes drastically smaller. In addition, due to graph transformations during placing and routing, a rigorous subgraph isomorphism would often fail to find a match, while a fuzzy approach can often give partial matches. This is very helpful for finding maliciously modified circuits and performing other verifications.

III. PRIOR WORK

Prior work has focused mostly on recovering arbitrary structure from an unknown netlist, rather than identifying known subcircuit components.

Albartus et. al [9], Meade et. al [16], and McKendrick et. al [10] all focus on identifying multi-bit registers in an arbitrary netlist. Li et. al [8] identify words and word operations in combinational circuits in an effort to provide additional insight into a circuit’s operation. Our work primarily focuses on identifying instances of a known subcircuit in a larger design, rather than recovering arbitrary structure.

Li et. al [6] is similar to our approach, but focuses on a functional analysis using abstract components, while we use structural matching and concrete subcircuits. The functional analysis ignores any characteristics of the circuit itself, other

than its behavior, and therefore the abstract components are behavioral as well. In contrast, both *Isoblaze* and *IPRec* rely on physical structures to provide insight into the composition of a circuit, and thus depend on libraries of circuits rather than behaviors.

To our knowledge there is no other published work that addresses this exact problem. We hope that we can bring more attention to the area, and have others improve upon our work in the future.

IV. IPREC: HIERARCHICAL-BASED IP RECOGNITION

The goal of *IPRec* is to take a design netlist, along with a training set of randomly-configured IP, and to identify instances of the IP in the netlist. As explained previously, the netlist is assumed to be nameless and flat (as would be generated from analyzing an FPGA bitstream), meaning there are no net or instance names, no wire groupings, and no module/IP hierarchy. As such, the netlist can be represented as a simple directional graph with nodes representing cells and edges representing nets. The graph does contain cell/node properties (eg. LUT INIT values) that can be extracted from the netlist.

Although the search netlist is flat, the trained IP library contains rich hierarchical information, with many IP containing up to 10 layers of hierarchy. *IPRec* hinges on the key observation that while IP can be configured in various different ways, they are often composed of the same hierarchical modules. Stated another way, while the specific IP configuration present in the netlist may have never before been seen, each hierarchical module instance and its individual configuration may have been seen, and thus can be identified. We believe that by leveraging this hierarchical information, subgraph matching can be performed in a more tractable way (despite arbitrary subgraph isomorphism being NP-complete).

The first step in *IPRec* is to create a library recording the constituents and hierarchical relationships of module in the IP training data. Following that, *IPRec* gathers a list of all hierarchical modules and computes a “primitive span” metric for each module, which represents the largest spanning graph that can be made of primitive FPGA cells in the module (eg. LUTs, FFs), excluding instances of lower hierarchical modules.

The search algorithm then begins by choosing seed modules to serve as *anchor points*, from which the search will expand outward. These seed modules are chosen as modules with a large *primitive span*, that are also used in a large percentage of generated instances of the IP of interest. These anchor points serve as initial *hypothesis graphs*, and *IPRec* then searches for each hypothesis graph in the full netlist using a naive recursive approach. It is crucial to start with a hierarchical cell with a large primitive span since a large densely connected subgraph simplifies the graph isomorphism problem.

From each initial match, *IPRec* then attempts to grow the hypothesis graph, through a series of *descend* and *ascend* steps. In the descend step, the algorithm considers a child module instanced in the current hypothesis graph. Supposing

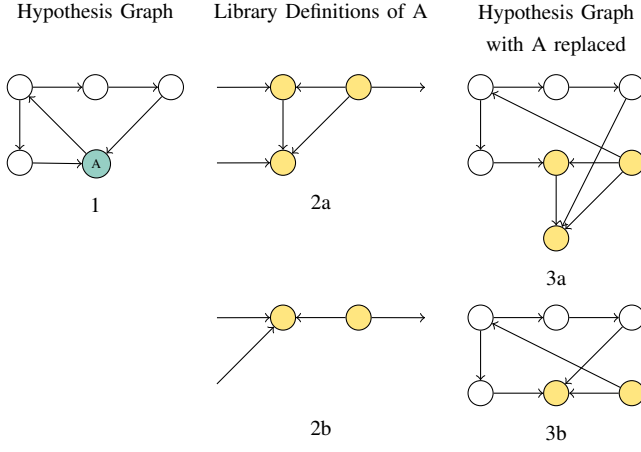


Fig. 2. IPreced descend step, from [17]

this is an instance of module A, the algorithm iterates through all possible implementations of module A from the IP training library, tentatively adds it to the hypothesis graph, and then does incremental matching against the full netlist using the same recursive approach from earlier. If only one possible implementation of A succeeds in matching the full netlist, then the hypothesis graph is permanently expanded to include this definition. However, if multiple different implementations of A succeed in matching the full netlist, then the decision is postponed. This is illustrated in Figure 2.

Matches to this algorithm are defined as having the same cell properties. Special consideration is given for LUT equations by anonymising all inputs, and removing Vivado-introduced don't care terms. This approach of anonymizing inputs overcomes input reordering introduced by place and route, and is very fast. However, it also loses information about which pins contribute to which terms in the logic equation. This is developed further in Isoblaze.

The *ascend* step follows a similar pattern, but traverses up the hierarchy, looking for modules in the IP training library that instance the module that is the current root of the hypothesis graph. For ascending, if there are several possible matches for a parent cell, the cell containing the most primitive cells is greedily committed to. This is to limit the search space to a reasonable size.

The outer loop of the algorithm repeatedly performs descend steps for all instanced modules in the hypothesis graph, followed by a single iteration of the ascend step. This is repeated until no more progress can be made.

The output of this algorithm is the largest hypothesis graph. Ideally, the output is the full hierarchical structure of an instance of the desired IP with no holes. However, in reality the graph may include holes, where no children cells matched, and may only reach partially up the hierarchy, if no parent cells matched.

IPRec assumes coverage over the hierarchical cell configuration space: that we have at least one instance of every

hash("LUT", "EQN:O6=A3*A2+A1")
hash("FDRE", "FFSR:LOW")
hash("LUT", "EQN:O6=A1+A4")
hash("FDRE", "FFSR:LOW")
hash("LUT", "EQN:O6=A1+A4*A2*A5")

Fig. 3. Dictionary entry for color and connectivity

configuration of every hierarchical cell in our library. In practice, this turns out to be relatively easy to achieve, since while the combinatorial explosion of configurations of an IP is intractable, focusing on hierarchical cells goes a considerable way towards linearizing this space.

In addition, good performance of IPRec depends on a relatively diverse distribution of cell properties, to allow for efficient matching of the initial anchor point.

IPRec also assumes that a given cell will always be placed on the same type of BEL. This turns out to be false, and was remedied in Isoblaze.

IPRec was implemented in Python to take advantage of the *iGraph* library for graph manipulation and Python's ease of development. More details of the IPRec algorithm can be found in Simpson's Thesis [17].

V. ISOBLAZE: HASH-BASED IP RECOGNITION

Isoblaze is intended to find a subcircuit using only local information to each BEL, specifically the BEL properties and the connectivity of that BEL, up to a certain distance. This allows for a much faster analysis, at the cost of a more approximate match.

Isoblaze performs its analysis in two stages. The first is a preprocessing stage, where all the relevant metadata of both the target design and the IP library is computed. The second is where the actual comparison between netlist and IP library takes place.

A. Netlist and Subcircuit Preprocessing

The preprocessing stage computes a color for each node from a hash of the BEL's properties. These properties include the logic equation for LUT cells, SR values and synchronicity for flip-flops, RAM-mode for RAMs, and so on for all configurable BELs. The goal is to hash everything that can be configured about a given circuit element, together with the type of circuit element.

Next, we compute a connectivity filter for each node based on the colors of its neighbors. Fig. 3 illustrates a single entry in this data structure. The key is represented as the top color, and the list of connectivities is indented from there. Each connectivity filter contains a set of colors, in this case both containing the same flip-flop color, but different LUT colors.

The connectivity filter is implemented as a Bloom filter. Bloom filters allow for constant-time membership queries and insertion, while providing compact storage. In addition, subset computations are reduced to bitwise operations, and are thus also constant-time.

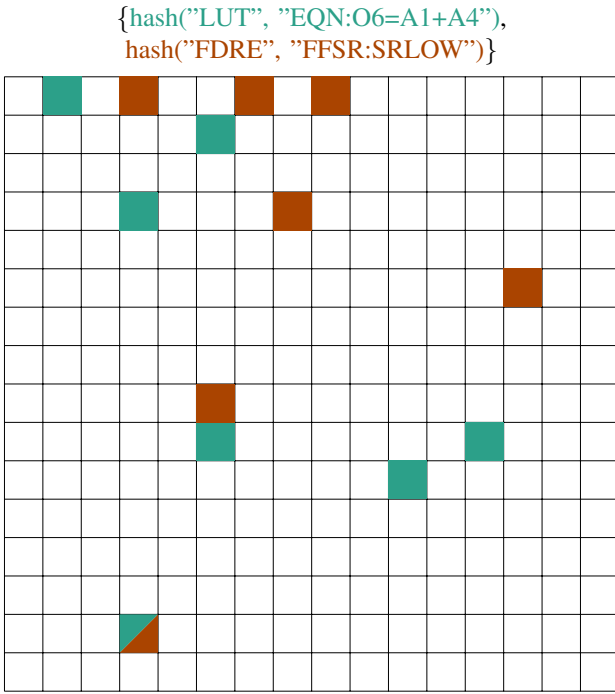


Fig. 4. 256-bit Bloom filter of 2 connected BELs

Bloom filters are characterized by the expected number of items stored in the filter n and the desired false-positive probability ϵ . Given these two quantities, the ideal width of a Bloom filter W , as well as the number of hash functions k can be computed according to the following formulas:

$$W = -1.44n \log_2 \epsilon \quad (1)$$

$$k = -\log_2 \epsilon \quad (2)$$

For this project, we decided on an error of less than 1%, and anticipated roughly 27 connections per BEL, based on the mean number of connections in an arbitrary design containing a MicroBlaze processor. This resulted in using seven hash functions and 256-bit wide filters. Fig. 4 illustrates a Bloom filter containing the colors for two BELs. This diagram illustrates that overlap is expected, but 100% overlap should be rare, due to the high number of hash functions and width of the filter. The grid represents all 256 bits of a Bloom filter, with the majority set to zero (white). The property lists above are hashed to derive the seven bits to turn on, which are then colored in the diagram according to which property list hashed to that index. Note that both property lists hashed to the same bit in the bottom-left corner. This illustrates the overlap that is expected, but not problematic.

To generate the entries for the Bloom filters, we use a sponge construction. This allows us to *absorb* as many bytes as provided into the hash, and then *squeeze* as many bytes as needed from that hash. It is based on a 64-bit xorshift pseudorandom number generator (PRNG), where the lower eight bits of the PRNG are xor-ed with the input byte after

```

for  $b \in T$  do
  if not  $D[b.c]$  then
    continue
  end if
  for  $x \in D[b.c]$  do
    if  $x \subseteq b.x$  then
      output( $b$ )
    end if
  end for
end for

```

Fig. 5. Isoblaze matching loop

which the PRNG is iterated once. The extraction takes place by repeatedly outputting the lower eight bits of the PRNG and iterating. Thus our colors are sponge constructions from which we can derive a pseudorandom sequence of bytes.

This connectivity can be arbitrarily deep. Currently we have explored the algorithm with a connectivity distance of zero (only consider node colors) and a connectivity distance of one (node colors and connectivity of direct neighbors). Additional connectivity has the potential to reduce false positives further, but the cost of compute to perform connectivity analysis increases by roughly 10x for each additional level.

B. Subcircuit Matchings

The comparison stage iterates through all the BELs in the target design, together with their color and connectivities. The order is irrelevant, as the dictionary we are matching against doesn't maintain an order, and BEL has a unique identifier allowing it to be located in isolation.

Given the list of BELs in the target T and a dictionary of colors and connectivity D , Fig. 5 illustrates how Isoblaze identifies matching BELs. BEL b contains a color c and connections x .

This algorithm iterates over every BEL in the design, and checks to see if its color is found in the dictionary. If it is not, we skip it. Otherwise, we compare the connectivity of the BEL against the list of connectivities in the dictionary. If any of the connectivities in the dictionary are a subset of the connectivity of the cell, as computed by bitwise operations on the Bloom filter, then that cell is considered a match, and is output.

This algorithm depends on having relatively distinct node colors or connectivity. If there are many similarities in color and connectivity, false positives are likely to occur.

Our implementation of this algorithm was aided by being coded in Standard ML, using the MLton whole-program optimizing compiler. The presence of garbage collection and strong typing with inference allowed for quick development while still permitting significantly increased performance over Python.

C. LUT Comparison

When creating the color for LUT BELs, we perform three modifications that greatly increase our matching power. The

first is to ignore the logical size of the LUT, since placing and routing can change the logical size of the LUT without impacting its functionality. The second is to ignore an arbitrary “don’t care” term that is sometimes introduced by Vivado, as was done in IPRec. This term usually takes the form of $(A6 + \sim A6)$.

The final modification we make is to the LUT equation itself. During placing and routing, the input pins to a LUT can be reordered to ease routing congestion. However, these reorderings impact the logic equations significantly. In contrast to IPRec’s approach of anonymizing the inputs, in Isoblaze we first compute the number of variables involved in the equation, and construct a truth table containing only those inputs. Then we collect the minterms and for each input identify the number of minterms in which that input is true. To this list we append the number of minterms, and sort the resulting list. This list provides a representation of the equation that is stable across input-reordering.

As an example, given the equation $(A6 + \sim A6) * (A4 * A2) + A3$, we first strip the don’t care term, leaving us with $(A4 * A2) + A3$. We then simulate it and construct a three-input truth table. Ordering the inputs according to pin number, this results in the equivalent LUT init string of 8'b1101100. This indicates that there are five minterms. Of those five, input A2 is high for three of them, A3 is high for four of them, and A4 is high for three of them. The resulting list is then (3, 3, 4, 5). It is easily verified that for any reordering of the inputs, the list will remain identical.

D. BEL Transfer

Another issue we faced was due to the many-to-many relationship between primitive cells and BELs, an issue we term BEL transfer. Because we work primarily at BEL level, different BELs being used to implement the same cell results in different sets of properties coloring the BEL.

However, there is usually a common subset of properties between all the BELs available to implement a given primitive cell. We therefore added a list of special cases mapping a given cell and BEL combination to a defined subset of properties.

For example, a LUT can be implemented on a BEL of type LUT or LUT_OR_MEM. When a LUT_OR_MEM BEL is used there is a property indicating in which capacity it is serving. By limiting the properties output for LUTs to the equation, we become immune to this difference. This can also occur with flip-flops and other types of BELs.

VI. EVALUATION

A. Methodology

To characterize both of these algorithms, we generated a block design in Vivado containing the following Xilinx IP: a MicroBlaze soft processor, its associated debug module, a GPIO block, a clock wizard, and an AXI Interconnect bus connecting it all. We then created a library of 100 randomly generated instances for each of the five IP, along with two additional IP: an Accumulator and an AXI DMA Engine. This library was then matched against the block design using both

IPRec and Isoblaze. IPRec outputs the largest cohesive match it can find for a single IP, while Isoblaze outputs any BELs that appear to match any instance in the library. The output of each algorithm is detailed further in their respective sections,

We then evaluated the performance of each algorithm with respect to the quantity of BELs matched, missed (false negatives), and overshoot (false positives); and calculated a summary F1-score for each IP and algorithm. The runtime for each algorithm is also presented.

B. Results

The results of the experiment are tabulated in Table I. Note that IPRec never finished analyzing either the AXI Interconnect core or the MicroBlaze. The times listed are for how long the analysis ran before we gave up. IPRec on average has very little overshoot, which makes it useful in performing precise analysis. Everything that IPRec matches is almost certainly part of the IP. In contrast, Isoblaze tends to have much more overshoot. This overshoot is expected, due to similarities in BELs. By only looking at local properties and connectivity, with no concern for the global structure, we match additional cells. However, the overshoot drops by a factor of three with connectivity of 1 vs connectivity of 0, after controlling for outliers. This remains true for IP not present in the target design, such as the accumulator. The BELs in the overshoot category are almost entirely flip-flops and CARRY4 cells, which are not configurable and are thus indistinguishable from one another except on the basis of connectivity.

The outlier in overshoot is the Clock Wizard. The Clock Wizard has very few BELs and only one is configurable. Half of the BELs are buffer BELs. This leads to many false positives, since the fingerprint is so vague to begin with.

The runtimes between the two algorithms are telling. IPRec when it completes is on average about 338x slower than Isoblaze with zero connectivity, and about 15x slower than with a connectivity of one. If we allowed IPRec to run long enough to complete on both the AXI Interconnect and Microblaze IP cores, the average speedup would increase even further.

The F1-scores are better with Isoblaze than with IPRec across the board. With added connectivity, the Isoblaze algorithm becomes more precise, so the fact that the MicroBlaze F1-score is lower with higher connectivity is indicative that the particular configuration of MicroBlaze present in the test design was not generated in our library. When controlling for that factor, the average F1-score for connectivities 0 and 1 are 16.5 and 22.8 respectively. This indicates that higher connectivity is more effective at matching IP that are present in the design. Additionally, it indicates that Isoblaze can perform analyses with up to twice the matching power as IPRec.

It is interesting to note that although the F1-score decreased for higher connectivity on the MicroBlaze, it is still at almost 72%. This indicates that there is significant similarity between the IP in our library and that in our target design, and confirms the utility of the fuzzy approach.

TABLE I
RESULTS

Algorithm	IP	Miss	Overshoot	Match	Total	Time(s)	Precision(%)	Recall(%)	F1(%)
IPRec	AXI GPIO block	464	0	57	521	2439	100.00	10.94	19.72
	Clock Wizard	6	0	0	6	59	N/A	0.00	N/A
	MicroBlaze Debug Module	219	40	2	261	32	4.76	0.90	1.52
	AXI Interconnect	-	-	-	-	57180	-	-	-
	MicroBlaze	-	-	-	-	2700000	-	-	-
	Average					3718			10.62
	Accumulator	0	0	0	0	29	N/A	N/A	N/A
	AXI Direct Memory Access	0	34	0	34	62661	N/A	N/A	N/A
Isoblaze N=0	AXI GPIO block	24	1661	497	2182	1	23.03	95.39	37.10
	Clock Wizard	0	2493	6	2499	3	0.24	100.00	0.48
	MicroBlaze Debug Module	43	2671	184	2898	5	6.44	81.06	11.94
	AXI Interconnect	10	2962	184	3156	4363	5.85	94.85	11.02
	MicroBlaze	42	1049	2196	3287	1001	67.67	98.12	80.10
	Average					11			32.41
	Accumulator	0	1908	0	1908	0	0	N/A	N/A
	AXI Direct Memory Access	0	3197	0	3197	112	N/A	N/A	N/A
Isoblaze N=1	AXI GPIO block	9	472	512	993	11	52.03	98.27	68.04
	Clock Wizard	1	1078	5	1084	40	0.46	83.33	0.92
	MicroBlaze Debug Module	146	607	81	834	36	11.77	35.68	17.70
	AXI Interconnect	181	363	13	557	19333	3.46	6.70	4.56
	MicroBlaze	626	650	1612	2888	3306	71.26	72.03	71.64
	Average					252			32.57
	Accumulator	0	58	0	58	2	0	N/A	N/A
	AXI Direct Memory Access	0	775	0	775	763	N/A	N/A	N/A

VII. FUTURE WORK

One of the big limitations of this approach is that it doesn't handle optimizations of IP performed by the *Implementation* phase of the CAD flow, which performs optimizations depending on the context in which the IP is found. These optimizations include constant propagation and LUT combining, among others, that alter the graph beyond the reach of isomorphism. In the future, we plan on investigating approaches that will better handle these graph-altering optimizations.

VIII. CONCLUSION

The goal of this research was to identify and evaluate approaches for subgraph-isomorphism for use in IP identification in circuit netlists. We developed 2 novel algorithms to that effect, and evaluated their performance in finding IP as well as rejecting absent IP.

As a first attempt, IPRec represents an important perspective at the problem of netlist reversal. The idea of employing fuzzy subgraph isomorphism to circuits was then carried forward into the implementation of Isoblaze. These tools will likely only form parts of a complete netlist reversal toolchain, but with the advances in Isoblaze, identifying known subcircuitry is now a feasible task.

IX. ACKNOWLEDGEMENTS

We would like to thank Jason Dickens from GrammaTech Inc. for his technical support and guidance.

REFERENCES

- [1] F4PGA. "F4pga/prjxray," Accessed: Jun. 23, 2020. [Online]. Available: <https://github.com/f4pga/prjxray>
- [2] H. Yu, H. Lee, S. Lee, Y. Kim, and H.-M. Lee, "Recent advances in FPGA reverse engineering," *Electronics*, vol. 7, no. 10, p. 246, Oct. 2018.
- [3] C. Wolf. "YosysHQ/icestorm," Accessed: Oct. 31, 2021. [Online]. Available: <https://github.com/YosysHQ/icestorm>
- [4] S. E. Quadir et al., "A survey on chip to system reverse engineering," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 13, no. 1, 6:1–6:34, Apr. 13, 2016.
- [5] E. Cahill, B. Hutchings, and J. Goeders, "Approaches for FPGA design assurance," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 3, 28:1–28:29, Dec. 28, 2022.
- [6] W. Li, Z. Wasson, and S. A. Seshia, "Reverse engineering circuits using behavioral pattern mining," in *International Symposium on Hardware-Oriented Security and Trust*, Jun. 2012, pp. 83–88.

- [7] P. Subramanyan et al., “Reverse engineering digital circuits using structural and functional analyses,” *IEEE Transactions on Emerging Topics in Computing*, vol. 2, no. 1, pp. 63–80, Mar. 2014.
- [8] W. Li et al., “WordRev: Finding word-level structures in a sea of bit-level gates,” in *International Symposium on Hardware-Oriented Security and Trust (HOST)*, Jun. 2013, pp. 67–74.
- [9] N. Albartus, M. Hoffmann, S. Temme, L. Azriel, and C. Paar, “DANA universal dataflow analysis for gate-level netlist reverse engineering,” *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 309–336, Aug. 26, 2020.
- [10] R. McKendrick, C. Simpson, B. Nelson, and J. Goeders, “Leveraging FPGA primitives to improve word reconstruction during netlist reverse engineering,” in *International Conference on Field-Programmable Technology (ICFPT)*, Dec. 2022, pp. 1–5.
- [11] J. R. Ullmann, “An algorithm for subgraph isomorphism,” *Journal of the ACM*, vol. 23, no. 1, pp. 31–42, Jan. 1, 1976.
- [12] J. R. Ullmann, “Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism,” *ACM Journal of Experimental Algorithmics*, vol. 15, 1.6:1.1–1.6:1.64, Feb. 7, 2011.
- [13] L. Cordella, P. Foggia, C. Sansone, and M. Vento, “A (sub)graph isomorphism algorithm for matching large graphs,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 10, pp. 1367–1372, Oct. 2004.
- [14] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, and A. Ferro, “A subgraph isomorphism algorithm and its application to biochemical data,” *BMC bioinformatics*, vol. 14 Suppl 7, S13, Suppl 7 2013.
- [15] C. McCreesh, P. Prosser, and J. Trimble, “The glasgow subgraph solver: Using constraint programming to tackle hard subgraph isomorphism problem variants,” *Graph Transformation*, vol. 12150, pp. 316–324, May 31, 2020.
- [16] T. Meade, Y. Jin, M. Tehranipoor, and S. Zhang, “Gate-level netlist reverse engineering for hardware security: Control logic register identification,” in *IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2016, pp. 1334–1337.
- [17] C. Simpson, “Towards trojan detection from a raw bitstream,” Ph.D. dissertation, Brigham Young University, Mar. 23, 2022.