SMatch: Structural Matching for Fast Resynthesis in FPGAs

Rafael Trapani Possignolo
Dept. of Computer Science and Engineering
University of California Santa Cruz
rpossign@ucsc.edu

Jose Renau
Dept. of Computer Science and Engineering
University of California Santa Cruz
renau@ucsc.edu

ABSTRACT

Designers wait several hours to get synthesis, placement and routing results even for small changes. Commercial FPGA flows allow for resynthesis after code changes, however, they target large code changes with not so effective incremental flows. We propose *SMatch*, a flow for FPGAs that has a novel incremental elaboration and novel incremental FPGA placement and routing that improves the state-of-the-art by reducing the amount of placement and routing work needed. We evaluate our approach against commercial FPGAs flows. Our method finishes synthesis, placement, and routing in under 30s for most changes of publicly available benchmarks with negligible QoR impact, being over 20× faster than existing incremental FPGA flows.

CCS CONCEPTS

• **Hardware** → **Methodologies for EDA**; **Logic synthesis**.

KEYWORDS

Incremental Synthesis, Electronic Design Automation, Design Productivity

ACM Reference Format:
Rafael Trapani Possignolo and Jose Renau. 2019. SMatch: Structural Matching for Fast Resynthesis in FPGAs. In *The 56th Annual Design Automation Conference 2019 (DAC '19), June 2–6, 2019, Las Vegas, NV, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3316781.3317912

1 INTRODUCTION

Engineers typically wait several hours for synthesis, placement, and routing. During timing closure and functional bug fixes project phases, this is done for relatively small changes. Incremental flows in commercial FPGA [2, 22] try to cut the time to generate an FPGA bitstream; however, those flows sacrifice runtime to guarantee same levels of Quality of Results (QoR) compared to a full synthesis, placement, and routing. Incremental flows are especially interesting for FPGAs on emulation platforms like Strober [12], where short implementation time could reduce the overheads involved in programming FPGAs during the evaluation of multiple similar RTLs, with negligible QoR impacts.

There is an important contrast of FPGA design with modern software engineering techniques that advocate for agile development cycles, even with live environments. Recently, *LiveSynth* proposed a workflow that triggers logic synthesis frequently for small changes and provides results in just a few seconds [18] for ASICs. A "live" flow for hardware can largely improve designers' productivity.

However, that approach only deals with the first step (synthesis), whereas placement and routing usually dominate the total time. In a quick evaluation with the Anubis [17] benchmarks, the commercial FPGA flows either had approximately the same time to perform synthesis, placement, and routing, or placement and routing take individually more time than synthesis. This means that placement and routing are the major runtime bottleneck in digital design flows.

This paper presents has three main contributions: (1) *SMatch*, a technique that leverages the netlist structure to speedup resynthesis in FPGAs; (2) a simple method for incremental elaboration that can be integrated with existing flows; and (3) an improved flow to reduce area overheads. The result is the first incremental flow that mixes logic and physical synthesis for faster synthesis results.

Briefly, *SMatch* aims to reduce the amount of LUTs that need to be placed and routed after a change is made in the code. Instead of running placement and routing for the whole modified portion of the circuit, *SMatch* analyses the netlist structure, regardless of functionality, to re-use existing placement and routing information whenever the structure of the netlist is unchanged. Then, placement and routing are applied to a smaller portion of the circuit.

Although theoretically this could be applied to ASIC synthesis, this novel technique is ideal for FPGAs, where LUTs have fixed placement and routing resources are also fixed. Changing the logic implemented by a LUT is simple and as long as the connections are the same, the final placement and routing should be the same, or at least very similar (depending on both the criticality of the path and the utilization of the region).

Incremental elaboration is not a novel problem. In software development, fine grain parser improves runtime for compilation [20]. The techniques are not directly applicable to RTL, however RTL elaboration is based on files changed. This is prohibitive for a live flow in single file generated RTL, like those generated by [4]. The proposed flow perform a module-based partitioning even in single file settings.

Our results show that our approach can be applied on commercial flows, and it is 5 − 21× faster than an commercial incremental flow, or 2× faster than the current state-of-the-art flow, *LiveSynth*, on average. *SMatch* is able to finish synthesis placement and routing in less than 30 seconds for 70% of the changes in the Anubis benchmark suite [17]. *SMatch* is faster than previous approaches in most of the cases, but it is never slower. We also note a slight degradation in QoR but not statistically distinguishable than previous approaches.

2 RELATED WORK

Incremental synthesis tools are by no means new. The first incremental flow was reported over three decades ago [11], with the goal to incrementally calculate timing in a netlist. Although manual, the process was very similar in effect to what happens in incremental timing analyses [10, 13], where an initial solution is fully computed before changes are done in the netlist and the new solution is rapidly calculated using the initial results.

Incremental flows for synthesis have been proposed. Early [8] and still widely used [2] flows rely on pre-partition of the design, either manually or automatically. Each partition is independently synthesized, placed, and routed, and then the overall circuit is connected together. When changes are made, only the affected partitions are re-synthesized, placed, and routed, reducing the total time. However, the QoR is heavily dependent on the partitioning, and there does not seem to be a way of predicting which partitioning method is the ideal short of trying multiple partitioning strategies.

Traditional ECO approaches [7, 15] can also be classified as incremental synthesis, but the main goal of ECO flows is to reduce the amount of disturbance to an existing mask, usually late in the design process where it is costly to change the design. Therefore, the algorithms and methods used are well suited to reduce the amount of cells changed, wire re-routed and so forth, and not necessarily to keep overall quality and speed [5].

Post-synthesis partitioning methods [6, 18] first synthesize the design and then find suited partitions. Because these methods do not arbitrarily decide where to partition before synthesis, they have the advantage of minimizing QoR degradation. LLIR [6] removes the affected region from the synthesized netlist and replaces it with an unsynthesized netlist and synthesis started. This is faster than a regular synthesis because most of the design is already mapped and optimized and only a small portion requires work. *LiveSynth* [18] takes a different approach that synthesizes only the modified region, and then merges the netlists. If a critical path is hit, the neighbor region is also included. This type of methodology go hand-in-hand with revision control and code repository techniques that are finally becoming popular in hardware design. Partitioning can leverage an older revision of the code and be updated regularly depending on the rate of changes in the codebase.

Our approach is most similar to *LiveSynth*, but instead of fully removing the affected region, which requires placing and routing the region fully, we identify structurally matching regions and leverage placement and routing for them. Any non-matching in the original netlist is removed from the design and any new cell that is not matched is inserted, placed and routed.

3 STRUCTURAL MATCHING

In this section, we describe the key observations made that enable us to leverage the netlist structure matching to perform resynthesis, then we discuss how to handle retiming and recycling. We also discuss how we leverage incremental synthesis to limit the size of the region where our technique needs to be applied and finally present the final flow.

3.1 Structural matching of netlists

SMatch is based on two key observations: 1) placement and routing are agnostic to logic function and only depend on netlist structure and the physical dimensions of its components, 2) in FPGAs, the elements of a netlist can only be a handful of types, and thus there is a large number of equal objects in the netlist.

From the first observation, it is possible to conclude that structurally similar netlists will be placed and routed in similar ways. At this point, it is important to recognize that small variations in a netlist can cause important variations in the optimal result of placement and routing. However, modern incremental synthesis allows small degradations of QoR. Then, re-using an existing placement and routing will most likely yield similar QoR results.

In general, this could be applied to any netlist. However, for ASIC netlists, the standard cell libraries are usually rich in types and sizes of cells, and also, macros are often used to implement specific logic functions, such as arithmetic, which introduce heterogeneity to the netlist. This makes our second key observation useful. In FPGAs, there are only a few types of cells. Since LUTs with same number of inputs are physically equivalent, it is perfectly plausible to change the logic implemented by a LUT without the need to re-place and re-route it, as long as the physical and logical connections with adjacent LUTs are unchanged. Since not all LUTs will be matched, remaining LUTs need to be placed and routed following conventional place and route flow.

Therefore, the main challenge is to find the largest structurally matching region after a change is introduced to the design. This process is illustrated in Figure 1, for simplicity and without loss of generality, we consider LUTs with two inputs, however the method is trivially extended to any number of input bits. The left figure shows an example of an initial netlist. Each LUT is named LUT_X with a unique ID, the function implemented by each LUT is indicated by a unique name. In the middle figure, the change added only affected the final function implemented by a single LUT, namely LUT_1, from f_1 to g_1. A regular flow would run placement and routing for all the LUTs in the design. *SMatch* proposes to not run placement and routing for any LUTs, and simply update the function implemented by LUT_1. Then, in the rightmost figure, the change made introduced an extra LUT, LUT_4. Since this is a new LUT, it will clearly need to be placed and routed. However, we note that the inputs for LUT_3 also changed, since it originally came from an input, but now comes from another LUT. Therefore, LUT_3 will also be placed and routed. LUT_1, however, also had its functionality changed, but will not require place and route, since its inputs and outputs did not change. One could argue that since LUT_3 will be most likely placed in a different position, this could affect the ideal placement for LUT_1, thus it should also be replaced. Although this is a valid observation, our results show that this was not necessary to maintain the QoR level.

Since this is a structural pass over both the netlists, it can be performed in linear time with the size of block considered. The method is explained in Algorithm 1. In the first loop (lines 5 to 17), candidate equivalent LUTs are found between the original design and the newly synthesized netlist. Since output are fixed and each net has a single driver, there is only one possible candidate LUT for the output nets. Then, for an arbitrary LUT throughout the netlist, we only consider the LUTs from the same input, that is, if the functionality is still the same, but the inputs are in a different order, the LUT will be marked as not equivalent. Then, in the second loop (lines 18 to 24), we verify that all the LUTs that were still not marked as not equivalent, have the same set (and order of inputs), otherwise the LUT is marked for placement and routing. It would be technically possible to allow flipped inputs, but this is left as future work.

After *SMatch*, matching LUTs have its functionality updated, and any additional LUT is placed and routed using the conventional

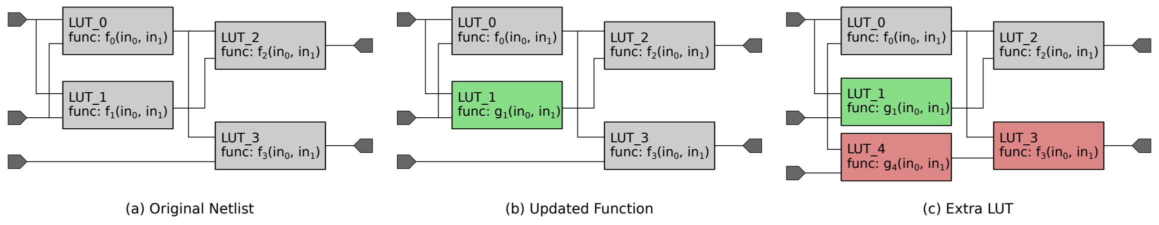


Figure 1: After synthesis of modified cones, some LUTs are structurally equivalent between the original and modified netlists. SMatch leverages that to reduce the amount of work needed to do during placement and routing.

place and route tools available in the flow. Any LUT that is no longer used in the new implementation of the circuit should be removed and made available before placement and routing of remaining LUTs, since it will open up space for new LUTs to be placed, which can improve QoR.

Algorithm 1 *SMatch* algorithm

```
1: procedure SMATCH(new_gates, old_gates)
2:     candidates ← map()
3:     matches ← ∅
4:     no_equiv ← ∅
5:     for all BFS from outputs(new_gates) do
6:         current ← BFS.next
7:         if is-output(current) then
8:             can ← same_output(old_gates)
9:         else
10:            can ← fan-in(candidates[fan-out(current)])
11:        end if
12:        if candidates[current] != can then
13:            no_equiv += current
14:        else
15:            candidates[current] ← can;
16:        end if
17:    end for//End BFS
18:    for all lut,candidate ← candidates do
19:        if fan-in(candidate) = candidates[fan-in(lut)] then
20:            matches += lut
21:        else
22:            no_equiv += lut
23:        end if
24:    end for
25:    return no_equiv, matches
26: end procedure
```

3.2 Handling retiming and extra registers

Retiming and Recycling can change the number and position of registers that divide pipeline stages in a circuit to improve timing closure [9, 14, 16]. Regardless of the exact technique used, here we are interested in handling those changes in an efficient manner. Moreover, moving a register or flop can be encoded as one removal from the original location and one addition to the new location.

The key observation here is that, in FPGAs, adding and removing flip-flops is a simple operation due to the FPGA architecture and organization. FPGA LUTs are organized in slices (Xilinx FPGAs) [22], or equivalently ALMs (Intel FPGAs) [3]. The overall architecture of a block varies by vendor. In Xilinx slices, there are 4 LUTs (with 2 to 6 inputs), with hardened arithmetic logic, 4 flip-flops, and by-pass logic. LUTs can be used independently or combined into larger logic blocks (of up to 6 inputs). In any configuration, there is at least one flip-flop per LUT. In Intel FPGAs, each ALM contains an 8-input fracturable LUT, two full adders and four flip-flops. The fracturable LUT can be split into pairs of LUTs of up to 6-input. Inputs may be shared by each partition, if the combination exceeds the number of available inputs [1]. There is also at least one flop per LUT, regardless of how the fracturable LUT is configured. Also, in the case of both vendors, regardless of the flip-flop usage in each LUT, the routing resources from the output of the slice are the same. Figure 2 depicts a simplified version of the architecture of both Xilinx and Intel FPGAs.

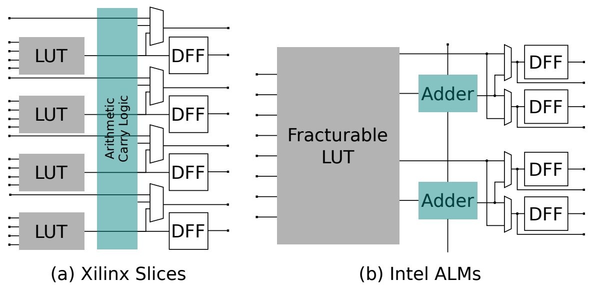


Figure 2: SMatch leverages the fact that each LUT has a flip-flop in its output that can be activated/deactivated without impacting which routing resources will be used.

Thus, *SMatch* can simply add or remove flip-flops at the output of each LUT to increase the amount of matching LUTs between two netlists. Simply put, during the *SMatch* pass, registers can be ignored, and a final pass over flops only need to enable or disable flip-flops to match the modified netlist. In a hypothetical architecture where there are not enough flip-flops, it would be possible to adapt *SMatch* to: while doing *SMatch*, verify whether a flip-flop is needed for each LUT and if so, verify if one exists, in this case, no extra work is needed. In case a flip-flop is needed and one is not available, the LUT is marked for placement and routing.

3.3 Incremental elaboration

In this paper, we call elaboration the process from parsing the source Verilog to creating a non-optimized netlist. For non-parameterized modules without defines, Verilog elaboration can be performed one module at a time. Since defines can have cross module impacts, most synthesis tools elaborate all the Verilog files without incremental support. Simulation tools still seem to re-parse all the Verilog inputs, but this is different because they do not do the second step of elaboration that is creating a netlist.

For more expressive HDLs like Chisel [4], a single Verilog file with several megabytes is generated after each code change, although there is no theoretical need for it. Moreover, even if multiple files were generated, it is often the case that spacing and comments cause files to be changed, even though the RTL did not change. For example, in Chisel all the assertion messages have a different text to notify the line of code.

Our incremental parser is able to: 1) split a multi-module file into multiple single-module files; 2) detect changes only if there were syntactic changes to the module. We do this by running the pre-processor, and then tokenization or lexical analysis. For each module, we remember the tokens inside the module, and tokens outside all the modules parsed before. We need to remember tokens outside the module because token in define structures can have side effects inside modules. If the token list is different from the previous version, a full elaboration is triggered.

Since Verilog can have comments, the lexer handle nested comments, and a simple pre-parser. The incremental elaboration splits any file into modules by looking for module definitions (in Verilog: “module” and “endmodule” keywords). Then, a parser compares tokens of the original an modified modules, thus comments and whitespaces are ignored. If there is a change in tokens, the module is flagged for elaboration. Cases like rename of wires are flagged as differences by this approach.

This limits the number of modules that need to be elaborated. Technically, our parser would be able to detect changes in sub-module structures and reduce even further elaboration, but that would require changes in the underlying synthesis tool to support sub-module elaboration, which is not done in this paper.

3.4 Partitioning the design

In theory, *SMatch* could be applied to an entire modified design, but that would be inefficient because it would require the synthesis of the whole design. Moreover, synthesizing the whole design, even after a small change, may yield important differences in the final netlist. Therefore, we limit the size of the netlist that needs to be considered by leveraging incremental synthesis [6, 15, 17].

Our approach is based on functional invariant cones (cones whose functionality does not change during synthesis) [6], which provide limited impact on QoR and have are around a few thousand gates [18]. A net in the design is considered a Functional Invariant Boundary (FIB) if its functionality has not been changed during synthesis. By definition, global input and output pins are FIBs. From inputs and/or outputs, and traversing inwards into the design, it is possible to find internal nets with unchanged functionality. Figure 3 shows an example where synthesis changes the implementation of $f = !(!a + bc)$ using a couple LUTs with two inputs. In this example, there are two Invariant Cones: $\text{fib}_1 = bc$ and $\text{fib}_2 = !a \cdot !\text{fib}_1$. Internal nets in fib_2 changed logic and thus are not FIBs. In this example, each invariant boundary has a single LUT, but this is usually not the case. FIBs are found in our flow during a setup phase that is ran once and leveraged over multiple design changes.

Finding functionally equivalent regions requires formally proving that a region in the elaborated netlist is equivalent to a candidate region in the synthesized netlist, which is done using logic equivalency checkers. To limit the search space, we match net names and use them as candidate equivalents. Candidate equivalent nets are

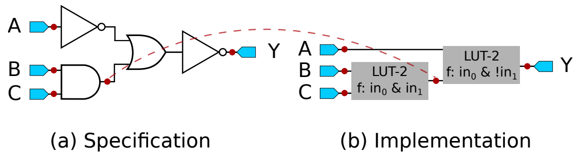


Figure 3: SMatch uses FIBs to define incremental synthesis regions.

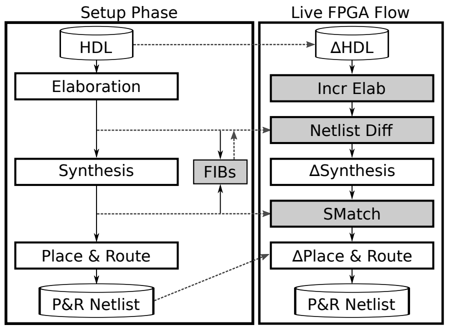


Figure 4: SMatch replaces placement and routing for a subset of cells changed in a design during incremental synthesis. This allows to reduce place and route time.

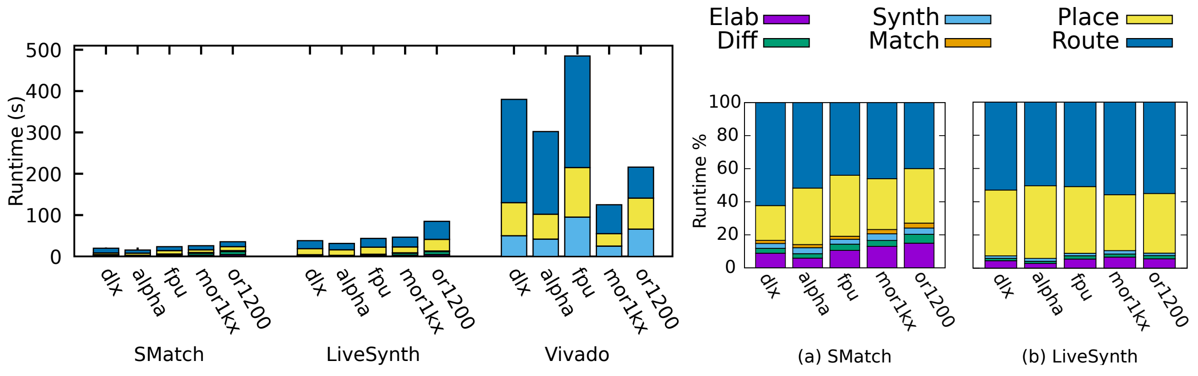
then formally compared for logic equivalence using standard tools. This is slow, but is done offline, during setup.

After a code change, *Netlist diff* (Algorithm 2) finds which blocks, to reduce the number of gates for *SMatch*, this algorithm is similar to the one proposed by *LiveSynth* [18] but targeting LUTs. A single code change can affect more than one cone since cones are overlapping. The original and modified netlist are traversed, starting by each invariant boundary and propagating backward until a FIB is found, if a difference is found the cone is marked different. Note that this is done once per FIB in the design and thus there is no cartesian product effect and the single gate comparison can be performed in constant time, therefore, *Netlist diff* has linear complexity. After *Netlist diff*, gates belonging to modified cones are synthesized outside of the context of the design and the synthesized and the original implementation of the design are fed to the *SMatch* step.

Algorithm 2 *Netlist diff* algorithm

```
1: procedure DIFF(Netlist original, Netlist modified)
2:     diff_cone ← ∅
3:     same ← same_operation(original.op,modified.op)
4:     for idx gets 0; idx < modified.fanin.size; idx++ do
5:         if ! is_fib(fanin(modified,idx)) then
6:             diff_cone.append(fanin(modified,idx))
7:             same &= & diff(fanin(original,idx), fanin(modified,idx))
8:         end if
9:     end for
10:    return [same, diff_cone]
11: end procedure
```

The final flow is depicted in Figure 4. Invariant regions are found during setup step, run once after the initial synthesis and reused across multiple changes. When changes are being performed, the incremental step is used. It consists of three substeps: *Netlist diff*, synth, *SMatch*. *Netlist diff* compares the elaborated netlist from the original and modified elaborated netlists. Then, these modified



(a) Runtime **(b) Breakdown**

Figure 5: SMatch performs synthesis placement and routing in under 30s for most changes in the Anubis benchmark suite. This is faster than the incremental flow of Vivado and LiveSynth, the state-of-the-art academic incremental flow.

cones are synthesized, with aggressive optimization goals. Finally, during *SMatch*, the newly synthesized netlist is structurally compared against the equivalent region of the original synthesized netlist. Matching LUTs have their logic updated; while unmatched LUTs are removed, replaced with newly synthesized LUTS, that are then placed and routed. Both *Netlist diff* and *SMatch* are a simple pass over the graph, with simple comparisons across cells in each and therefore are linear with respect to the netlist size.

4 EVALUATION SETUP

SMatch was implemented on top of LGraph [19], in C++14, compiled with CLANG 5.0.0. The baseline synthesis flow is YOSYS [21] version 0.7+312, targeting Xilinx FPGAs. Placement and Routing were done using Xilinx Vivado 2017.2, QoR results are reported after routing. We compare QoR with full synthesis for each change. For the structural updates, the Vivado's TCL interface was used. Placement and routing for remaining cells was performed using Vivado's standard incremental flows. We compared the runtime of our approach with *LiveSynth* [18] and Vivado incremental placement and routing. The experiments were run on a server with 2 Intel(R) Xeon(R) E5-2689 CPUs at 2.60GHz, with 64GB of DDR3 memory, Arch Linux 4.3.3.

We used Anubis [17], a benchmark for incremental synthesis that includes both RTL code for five designs (DLX, ALPHA, FPU, MOR1KX, OR1200). Each benchmark includes around 20-30 code changes, taken from repository commits and commented out code. The changes vary from small localized changes (within a module) to adding and removing ports in modules and changes to modules that are instantiated multiple times in a design.

5 EVALUATION

Our evaluation begins by looking at the runtime of *SMatch*, considering synthesis, placement, and routing. Overall, *SMatch* has a runtime of under 30 seconds for synthesis, placement and routing, for most changes in the Anubis benchmark suite, with an average of around 21s. This is around 1.6 to 2×x faster than *LiveSynth*, and $5 - 21\times$ faster than the incremental mode of Vivado. Figure 5A reports the runtime for each flow, for each benchmark in the Anubis suite, averaged across all changes and broke down by step.

Most of the advantage of the approach proposed here comes from the *SMatch* algorithm. It adds to the synthesis runtime compared to *LiveSynth*, however *SMatch* has the advantage of reducing placement and routing. *SMatch* is able to finish more than 70% of the changes in less than 30s (versus only 31%). In Vivado, there is no incremental synthesis step, the incremental flow uses a full synthesis and then runs incremental placement and routing, trying to leverage existing results for those steps. This explains the large portion of synthesis for Vivado runtime results.

Figure 5B shows the percentage runtime breakdown for *LiveSynth* and *SMatch*, normalized to 100%. Most of the time is spent in placement and routing. Even though we minimize the amount of routing needed, we still rely on Vivado's placer and router, which even in incremental mode is meant to maximize QoR at all costs. This is true for both flows, but less so in *SMatch* since there are less wires to route. The main difference between Vivado flow and the other two is that since Vivado performs full synthesis, there is usually a larger number of affected gates.

In *SMatch*, the time spent in place and route is proportionally smaller than in *LiveSynth*. Placement account for only about 20% of the runtime and routing for an average of under 50% of runtime in the *SMatch* flow. To better understand where the speedups for our technique originate, we looked into the runtime of changing the functionality of a few hundred LUTs, changing their placement, and changing their routing, which are the main tradeoffs involved when performing *SMatch*. We performed a simple experiment where a design was fully synthesized, placed and routed, and then performed three operations independently, in a varying number of LUTs: change LUT functionality, re-place LUTs, re-route LUTs, for $\approx 100, 400, 1000$ and 5000 LUTs, typical range for incremental synthesis. Results are summarized in Figure 6.

We also compared the speedup of each incremental flow with the equivalent full flow. Thus, the incremental flow on Vivado was compared with a full synthesis, placement, and routing in Vivado. We also compared with full synthesis in Yosys, plus placement and routing in Vivado. *SMatch* is on average over 20× faster than a full synthesis, placement, and routing. It is at least 1.6× and up to 300× faster than full synthesis (maximum achieved when place and route are reduced to zero during the incremental phases).

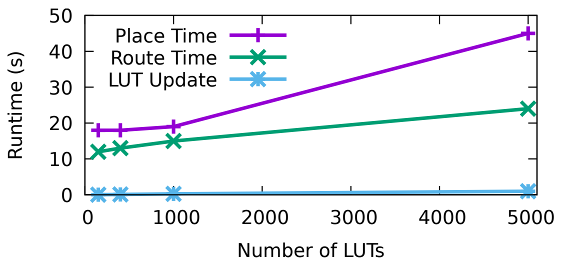


Figure 6: Place and route LUTs is orders of magnitude more expensive than simply update the functionality of an already placed and routed LUT. The speedups of *SMatch* come from that observation.

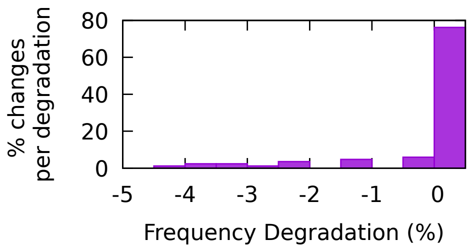


Figure 7: In more than 80% of the test cases Structural Matching delivers frequency within 0.5% of a full synthesis flow. The maximum decrease in frequency was of $\approx 5\%$.

The incremental mode of Vivado is only $30 - 80\%$ faster on average than the full mode, being slower in some cases. The main reason is that Vivado tries to maximize QoR, therefore, even the incremental placement and routing available in Vivado are sometimes as slow as the full synthesis.

We compared the QoR of *SMatch* against the full baseline flow (synthesis, placement, and routing). In some cases, there was increase in frequency of up to 3%, but we report those cases as 0% degradation, since those are not due to our technique. The maximum observed decrease in frequency was of $\approx 5\%$ and more than 80% of the changes had $< 0.5\%$ and only about 5% of the changes had a degradation of about 1% in delay(Figure 7). The high number of changes with no degradation is in parts due to some changes not affecting the critical path. This small degradation in quality is considered negligible and aligned with the goals of this work.

We also looked into area overhead. Our experiments revealed about $5 - 13\%$ increase in area for *LiveSynth*, but of only 4% for *SMatch* when comparing with the full synthesis flow. Our methodology creates inputs and outputs in the incremental region to preserve logic meaning and thus prevents the replication of gates that were used both by the incremental and the non-incremental regions.

The setup phase of *SMatch* includes a full synthesis, placement and routing, and finding invariant boundaries. Synthesis, placement and routing time are not considered overhead in the context of incremental flows. The routine to find the boundaries is the only added task. It requires a netlist after elaboration and a netlist after synthesis. In our experiments, we noticed that finding the boundaries takes about twice as much as the synthesis alone. For the benchmarks tested, that ranged from 120 to 480 seconds. However, this overhead can be amortized over multiple incremental changes, and thus can be considered negligible.

6 CONCLUSIONS

SMatch is the first flow to propose structurally matching LUTs and only replacing their functionality to leverage existing placement and routing from previous runs. *SMatch* is up to 20× faster than existing incremental commercial flows with minimal QoR impact. It is based on the fact that the same placement and routing will yield good QoR if the structure and connections are the same.

SMatch is able to deliver most of the changes in the Anubis benchmark suite in under 30*s*. Future work will include looking for ways to further partitioning the synthesis blocks, but also at ways to handle unmatching LUTs without the need to fully re-place and re-route them, but with low impact in QoR.

ACKNOWLEDGMENTS

We like to thank the reviewers for their feedback on the paper. This work was supported in part by the National Science Foundation under grant CCF-1514284. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

[1] Altera Inc. 2006. Altera: FPGA Architecture White Paper. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01003.pdf. (Jul. 2006).
[2] Altera Inc. 2016. Quartus Prime Standard Edition Handbook Volume 1: Design and Synthesis. https://www.altera.com/en_US/pdfs/literature/hb/qts/qts-qps-handbook.pdf. (Mar 2016).
[3] Altera Inc., Intel. 2017. Cyclone V Device Overview. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_51001.pdf. (Dec. 2017).
[4] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a Scala embedded language. In DAC ’12.
[5] Daniel Brand, Anthony Drumm, Sandip Kundu, and Prakash Narain. 1994. Incremental Synthesis. ICCAD’94.
[6] Doris Chen and Deshanand Singh. 2011. Line-level Incremental Resynthesis Techniques for FPGAs. FPGA ’11.
[7] Jason Cong, Jie Fang, and Kei-Yong Khoo. 1999. An Implicit Connection Graph Maze Routing Algorithm for ECO Routing. ICCAD’99.
[8] Mehrdad Eslami Dehkordi, Stephen D. Brown, and Terry Borer. 2006. Modular Partitioning for Incremental Compilation. FPL’06.
[9] Ilya Ganusov, Henri Fraisse, Aaron Ng, Rafael T. Possignolo, and Sabya Das. 2016. Automated Extra Pipeline Analysis of Applications Mapped to Xilinx UltraScale+ FPGAs. FPL’16.
[10] Tsung-Wei Huang and Martin D. F. Wong. 2015. OpenTimer: A high-performance timing analysis tool. ICCAD’15.
[11] Norman P. Jouppi. 1987. Timing analysis and performance improvement of MOS VLSI designs. TCAD’87, vol. 6.
[12] Donggyu Kim, Adam Izraelevitz, Christopher Celio, Hokeun Kim, Brian Zimmer, Yunsup Lee, Jonathan Bachrach, and Krste Asanović. 2016. Strober: Fast and Accurate Sample-based Energy Simulation for Arbitrary RTL. ISCA’16.
[13] Pei-Yu Lee, Iris H. R. Jiang, Cheng R. Li, Wei-Lun L. Chiu, and Yu-Ming Yang. 2015. iTimerC 2.0: Fast incremental timing and CPPR analysis. ICCAD’15.
[14] Charles E. Leiserson and James B. Saxe. 1991. Retiming Synchronous Circuitry. Algorithmica, vol. 6.
[15] Nilesh A. Modi and Malgorzata Marek-Sadowska. 2008. ECO-Map: Technology remapping for post-mask ECO using simulated annealing. ICCD’08.
[16] Rafael T. Possignolo, Elnaz Ebrahimi, Haven Skinner, and Jose Renau. 2016. FluidPipelines: Elastic Circuitry meets Out-of-Order Execution. ICCD’16.
[17] Rafael T. Possignolo, Nursultan Kabylkas, and Jose Renau. 2017. Anubis: A New Benchmark for Incremental Synthesis. IWLS’17.
[18] Rafael T. Possignolo and Jose Renau. 2017. LiveSynth: Towards an interactive synthesis flow. DAC’17.
[19] Rafael T. Possignolo, Sheng H. Wang, Haven Skinner, and Jose Renau. 2018. LGraph: A multilanguage open-source database. WOSET’18.
[20] Tim A. Wagner. 1998. Practical Algorithms for Incremental Software Development Environments. Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/1998/5885.html
[21] Clifford Wolf. 2016. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/.
[22] Xilinx Inc. 2016. Vivado Design Suite User Guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_1/ug910-vivado-getting-started.pdf.