LiveSynth: Towards an Interactive Synthesis Flow

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

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

ABSTRACT

Currently, one of the major bottlenecks in digital design is synthesis. Each iteration of a design takes several hours to synthesize, putting pressure on designers to carefully consider when to submit jobs and wait for the delayed feedback. This delay is especially important in FPGA emulation, when synthesis is performed frequently while fixing the system functionality. This work proposes *LiveSynth*, a different approach for digital design with relatively quick feedback after small, incremental changes. Our approach delivers results with close-to-optimal quality–within a few seconds of processing time in most cases. *LiveSynth* was able to improve synthesis time by about 10x with minimal impact on QoR.

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. 2017. LiveSynth: Towards an Interactive Synthesis Flow. In *Proceedings of DAC '17, Austin, TX, USA, June 18 - 22, 2017,* 6 pages.
DOI: http://dx.doi.org/10.1145/3061639.3062275

1 INTRODUCTION

Synthesis is tedious and time consuming, especially during the timing/power closure cycle. Designers wait several hours for relatively small design changes to get synthesis results. We propose a different workflow that allow the designer to trigger synthesis results very frequently as the design is being modified. Most of the time, providing accurate results takes seconds instead of hours. This results in quick feedback to further optimize the design without degrading quality. Our proposed flow is an incremental synthesis flow on steroids.

Traditional synthesis flows contrast with the rapid development techniques popular in software engineering [9]. While most software engineers would consider hours of compilation unacceptable, this is the de-facto expectation in synthesis. Even though the EDA industry has been trying to address the problem of long synthesis times [1, 12], the current standards are either not fast enough or depend on manual interactions that often degrade design quality.

We expect designers' productivity to improve with an interactive synthesis environment. Our vision is of a "live" flow, where designers know right away how the change will affect Quality of Results (QoR). The flow is divided into two parts: an interactive, low-effort part, and a background high-effort part. The interactive aspect gives "live" feedback (within a few seconds) with good accuracy but not necessarily fully optimized designs. The background process has a slow turnaround time and optimizes the design while the human works in the next set of changes.

This flow allows more iterations per day, helping reduce the time for timing/power closure. Since iterations are fast, the designer can make more changes, and thus it is easier to track the impact of each change in the design. If the change did not positively impact QoR, it is easy and cheap to undo the change and proceed in another direction. In our vision, synthesis is triggered as the designer types or saves the file (as long as it is possible to parse the design). This guarantees small enough increments while avoiding the undesirable old habits of experienced designers. When there are no pending incremental small jobs, a background high-effort synthesis runs to improve the design quality. This background process aims to remove imperfections inserted by the live flow, thereby slowly improving the design implementation.

To support this development model, we propose *LiveSynth*, an incremental synthesis framework that provides results in a few seconds to half a minute. We target under 30 seconds because that is the time that the short-term memory lasts in humans. *LiveSynth* targets the front-end flow, and can be applied to ASICs and FPGAs. Our focus in this work is FPGAs. Major FPGA vendors already support incremental [1, 13] place and route steps that could be leveraged by our flow. *LiveSynth* is independent of the baseline tool.

Triggering synthesis over the whole design is widely adopted in industry and academia alike. Nevertheless, usually, at a given iteration, a designer is focusing on one small portion of the circuit. In traditional synthesis, even if a small portion of the design is changed, logic synthesis and placement are triggered for large blocks and require hours to complete. This is due to two main reasons: tools are not designed for incremental synthesis, and inter-module optimization has a significant impact in QoR.

We focus on highly optimizing the sub-region and triggering re-synthesis only when necessary, and not over the whole design. *LiveSynth* divides the design into multiple regions with invariant boundaries, *i.e.*, regions whose boundaries' functionality has not been changed during synthesis. These regions are smaller than user defined modules on average. When a change is made in the RTL description of the design, the synthesis flow needs only to find which regions were touched and replace them with the newly synthesized netlist.

Even though the each region is highly optimized, this process is much faster since the region is kept small. To be able to maintain QoR, especially delay, if part of the critical path is within the region, the neighboring regions are also included in the high effort synthesis. Special care is given to the case where multiple instances of a module exist in the design. If the region frontiers are within the module, the region can be optimized alone, which yields a faster process. In the case where the region frontiers are outside the module, each instance must be dealt with separately.

The *LiveSynth* flow includes a setup phase that performs a regular synthesis of the whole design and also finds invariant regions, which are used as incremental grains for the Live phase. When there is a change in the RTL, *LiveSynth* finds which regions were affected and synthesizes only them. The algorithms are designed so that *LiveSynth* does not traverse the whole graph. Even a linear algorithm would not be competitive with our synthesis timing.

Our results show that *LiveSynth* is able to reduce synthesis time by about 10x on average, but with high variation. *LiveSynth* is consistently faster than any of the previous approaches. Also, we only observe delay degradation in only a minority of design changes, and only with small magnitudes (<3%). Our main contributions are:

- First incremental synthesis flow that allows inter-module optimization
- Interactive synthesis methodology with fast feedback
- Incremental synthesis flow that is independent of a specific synthesis tool

2 RELATED WORK

An interactive synthesis flow was first proposed almost 30 years ago [8], motivated to improve timing closure in digital design. The authors claim that the flow needed under 30 minutes to evaluate relatively large designs (for the time), but could compute the effects in frequency of a small design change in only a few seconds of CPU time. Although the main motivation was timing analysis, the result is largely an incremental (though manual) synthesis flow. The whole circuit is kept in memory while the designer applies small changes to it.

More recently, incremental synthesis has been revisited. Dehkordi *et al.* [6] propose an incremental synthesis flow that partitions the design into regions that are synthesized independently. After a change, only the partition is re-synthesized. Due to the artificial partitioning, there is a significant hit on QoR depending on the parameters choice. Since a single set of parameters is not applicable to any design, the utility of the approach is limited.

Cheng and Singh [4] propose a line-level incremental synthesis flow that is implemented coupled with the Altera synthesis flow. Since this flow has access to internals of the synthesis flow, it is able to keep track of changes during the synthesis flow and reduces the setup overhead observed in our approach. The final proposal also incurs in the elaboration of the whole design at each change and launches the synthesis over the full design. Our approach is more efficient, since it reduces the amount of work both in the RTL elaboration and in the final synthesis. Our approach can also be used with different synthesis tools without accessing any code.

Another approach is presented by Brand *et al.* [2]. The main target is to reuse most of the logic, especially late in the design cycle. The motivation is that large changes in the netlist would greatly disturb the back-end synthesis, which is undesirable right before tape-out. Our approach does not look into minimizing the size of changes as much, since we are focusing more on the timing closure cycle. We also observe that this approach has a large area overhead, since the unused logic is kept in the design to reduce the impact on the implementation.

3 LIVESYNTH

LiveSynth works by creating an implementation for a modified RTL specification, utilizing as much as possible from a previous implementation for the original specification. Incremental flows rely on partitioning the design into regions that will be independently synthesized. Then, re-synthesis can be triggered in each region when a change occurs. Early flows depend on user-defined partitions, which are usually dependent on hierarchy and not optimal, since partitioning has an important impact on synthesis quality [6].

LiveSynth automatically defines regions of a few thousand gates that are used as incremental grains. To reduce the impact on QoR, *LiveSynth* finds invariant cones, *i.e.*, regions whose functionality do not change during synthesis. Intuitively, these cones define the regions across which no further optimization is possible (or necessary) during the initial synthesis. Although this is not always the case, these regions are a good starting point for the incremental phase of the synthesis. This is better than relying on a rather arbitrary hierarchical division, since it is well known that inter module optimization plays an important role in design optimization.

LiveSynth (Figure 3) is built on top of third-party tools. A setup pass is performed right after the initial synthesis to determine equivalence between specification and implemented netlist. This pass could be removed by integrating equivalence tracking into the synthesis step itself [4]. Still, since it is only executed once, the overhead from this pass is not a big problem.

3.1 Incremental Synthesis

Any incremental synthesis approach looks into applying changes in the RTL specification of a design to an existing implementation. Conceptually, this process involves 4 netlists:

- *Spec0* and *Spec1*: are the netlists after elaboration (and before synthesis) for the original (*Spec0*) and modified (*Spec1*) RTL. We refer to these as elaborated netlists.
- *Impl0* and *Impl1*: are the synthesized netlists for the original (*Impl0*) and modified (*Impl1*) RTL. We refer to these as synthesized netlists.

The objective of incremental synthesis is to create *Impl1* that implements *Spec1* by utilizing as much as possible from *Impl0*. In *LiveSynth*, *Spec1* is not fully generated: only the modified files will pass elaboration, whereas the remainder of the modules are inferred from *Spec0*, since they did not change.

To avoid the need of arbitrarily defining incremental regions, which was shown to degrade synthesis quality [6], *LiveSynth* first synthesizes the entire design and then finds regions that can be used for incremental synthesis.

3.2 What size should the blocks be?

Partition size has a major impact on synthesis time, especially because synthesis time is not linear with design time. *LiveSynth* targets a "few seconds" synthesis time. To achieve that target, we need to define what design sizes would be feasible. To better understand how synthesis time varies with design time and thus define our target partition size, we perform a preliminary experiment. We synthesized different modules of different sizes in two synthesis tools, a commercial tool and Yosys [11], to evaluate how blocks of varying sizes would affect synthesis time (the synthesized blocks were sub-sets of our benchmarks, explained in Section 4). Since we are mostly interested in small blocks, we do not scale to large sizes.

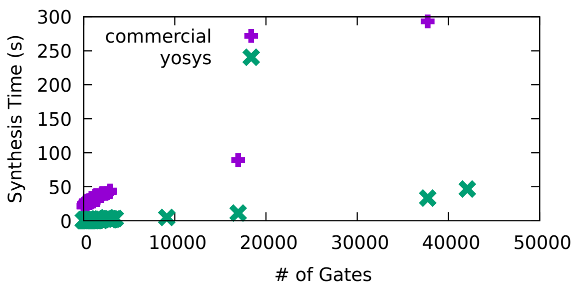


Figure 1: Synthesis time varies super-linearly depending on design size. In our tests, designs with less than $\approx 5k$ gates, had the least variation in synthesis time.

In these simple experiments, we observe that synthesis time varies super-linearly with design size[1]. Nevertheless, for small designs (under 5k gates) there is little variation in design time, whereas for designs too small ($< 1k$ gates), most of the time is consumed in tool overhead, which would be wasteful.

This data suggests that the $1k - 5k$ gates size offers a decent trade-off between amount of work done and runtime, and therefore, *LiveSynth* aims to use design partitions in this range.

3.3 What should constitute a block?

The choice of partitioning strategy has a major impact on synthesis time, area, and delay in incremental synthesis flows [6]. Choosing modules as blocks would prevent inter-procedural optimizations, and thus is not a suitable approach due to degradation of QoR [6]. Chen and Singh [4] propose a flow that triggers re-synthesis in the totality of the design after the modified region is included into the original design. Although this technique yields very good results for both area and delay, it comes at a relatively high cost in runtime. In some designs, the incremental synthesis takes as much as 77% of the original runtime. This penalty is due to the necessity to pass through the whole design at least once.

LiveSynth takes a different approach. We want to maximize design quality at the same time that we reduce the synthesis time. *LiveSynth* uses the concept of Invariant Cones to take advantage of the idea that further optimization is not possible (or needed) within the boundaries of that region. Our definition of Invariant Cone is not tied to module boundaries, and thus leverages intra-module optimizations. Since *LiveSynth* does not artificially define partitions, the QoR impact is substantially reduced.

Functionally-Invariant Boundaries (FIBs) [4] are the endpoints of invariant cones. A FIB is a net in the design whose functionality has not been changed during synthesis. This function is necessary, but the way it is calculated is unimportant. Global inputs and outputs are (always) FIBs. A change due a "don't care" condition is considered a functional change and thus, the node is not a FIB.

In the example in Figure 2, the synthesis process may change the implementation of the logic function $f = !(!a + bc)$ to $f = a \cdot !(bc)$. In this case, there are two Invariant Cones: $\text{fib}_1 = bc$ and $\text{fib}_2 = !a \cdot !\text{fib}_1$. Note that internal nodes in fib_2 presented logic changes and thus do not constitute an Functionally-Invariant Boundary.

[1]Note that other factors, such as target frequency, technology node, and synthesis flow, may also affect synthesis time and were not considered, since they are considered to be constant throughout this paper.

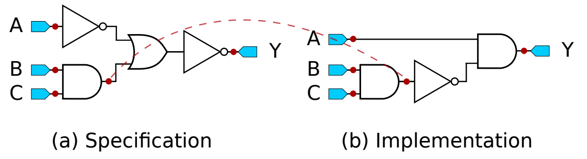


Figure 2: Functionally-Invariant Boundaries provide a natural boundary for incremental synthesis.

Table 1: Invariant Cones provide a natural boundary for incremental synthesis. Most of the Invariant Cones present in our benchmarks are smaller than the proposed target.

Invariant Cone Size	fpu	mips	or1200
< 200	1769	1237	643
$200 - 300$	99	73	172
$300 - 400$	938	35	156
$400 - 500$	1	2	185
$500 - 600$	649	11	74
$600 - 800$	34	316	63
$800 - 1000$	33	29	58
$1000 - 1500$	5	124	56
$1500 - 2000$	1	550	0
$2000 - 3000$	0	421	0
$3000 - 4000$	0	302	0
> 4000	0	115	0

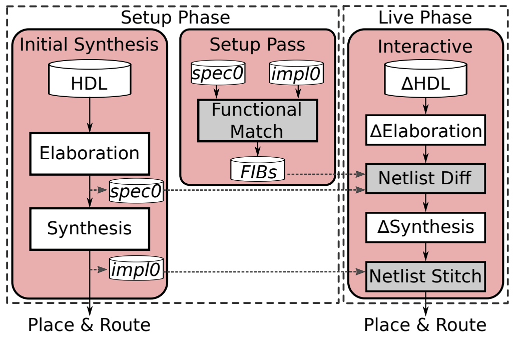


Figure 3: LiveSynth extracts a small subset of the design for synthesis and merges it back into the original synthesized netlist, quickly achieving results comparable to the non-incremental synthesis. Place and route are not included.

Table 1 shows statistics of the number of gates per Invariant Cone for our benchmarks (details in Section 4). We note that there is no clear trend in the distribution of cone sizes. Some cases, like the fpu, have smaller than ideal cone size and others, like the mips, have greater than ideal cone sizes. Our main conclusion from this observation is that it would be possible to merge a good number of cones in designs like the fpu and or1200 core, but ideally, the flow could leverage further splitting some blocks in the mips.

3.4 LiveSynth flow

The overall flow of *LiveSynth* is depicted in Figure 3, and consists of two phases: the Setup phase and the Live phase. The Live phase is split into three steps: *Netlist diff*, Partition Synthesis, and *Netlist stitch*. The Setup phase identifies FIBs (and respective Invariant Cones) between the *Spec0* and *Impl0* after the initial synthesis. After setup in the Live phase, when a change is made in the RTL, the

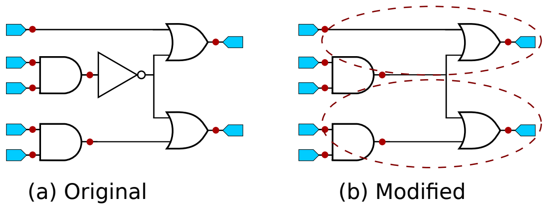


Figure 4: A single code change can impact multiple invariant cones that will need to be synthesized.

changed file passes elaboration, and the modified netlist is structurally compared to *Spec0*. The structural comparison (*Netlist diff*) only matches the portions of the netlist that are identical in their logic structure, and thus has linear complexity with the module size [4]. The main goal of this pass is to identify which Invariant Cones have been changed.

The final incremental synthesis region can include multiple cones. This is because a single code change may affect multiple cones, due to the overlapping nature of cones. This is depicted in Figure 4, which shows a single gate change in a design that affects two Invariant Cones (marked with the dashed ellipses).

After *Netlist diff*, the extracted netlist containing all the modified Invariant Cones is synthesized. Then, the resulting netlist replaces the equivalent Invariant Cones in the original synthesized netlist. Note that only the small region that was modified is synthesized during the *LiveSynth* step, which is a key factor for synthesis speed.

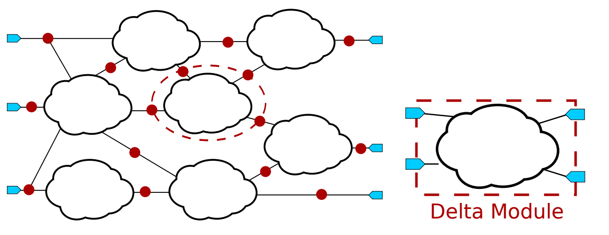
3.4.1 Setup phase. The main goal here is to find FIBs and which gates belong to each cone, as well as to how many cones a given gate belongs to. By knowing which gates belong to each cone, we avoid traversing the whole design when a change is made. Also, since cones may overlap, we only remove a gate from the design when it belongs to zero cones.

Since the structure of the logic changes during synthesis, it is not sufficient to simply compare the netlists. Thus, we rely on SAT solvers to compare the elaborated and the synthesized netlists. To reduce the search space, we assume that the synthesis flow has kept user-defined net names unchanged (except for appending instance names), which we have observed in all 5 flows tested (commercial and open source)[2]. We then compare the function implemented by each of the logic cones. To account for retiming (*i.e.*, changing of flop position) that may have occurred during synthesis, we just count the number of flops between each pair of FIBs. Although our results show that this is a very long step, it only needs to be performed once (prior to the execution of the flow), so this is not a huge problem. Also, this time could be mitigated with better integration with the synthesis flow to keep track of FIBs [4].

3.4.2 Live phase. After setup, the *LiveSynth* flow enters a interactive phase that provides designers feedback within a few seconds. This Live phase consists of cycling through three steps each time a designer makes a valid change.

The **Netlist diff** step finds which portions of the netlist have changed. We compare the modules that have been changed (identified by system time stamp) of *Spec1* with the original modules of *Spec0*. We traverse the netlist, starting at each FIB and going backwards, until a new FIB is found. If a difference is found, we mark

[2]It is fine to miss some equivalency between nets, this only would increase the size of regions, but would not jeopardize the method as a whole.



(a) Full Design **(b) Incremental Synthesis Region**

Figure 5: Instead of triggering synthesis in the whole design, LiveSynth extracts the region that needs to be synthesized. This is a key point for speed in our scheme.

the cone for synthesis. If the traversal does not spot a difference in the netlist, we ignore that region.

This structural comparison is fast since it only matches logic that is implemented in the exact same way. Note that to make this search fast, we assume that nets with the same ID are equivalent. Then, the search itself is responsible for proving that the two cones are structurally, and thus, functionally identical. The ID is the concatenation of instance names and the net name in the leaf instance. This allows for uniqueness of identifiers.

During *Netlist diff*, we also keep track of which gates are part of the cone, and thus we know which gates need to be synthesized when *Netlist diff* is done. The process is depicted in Algorithm 1.

Algorithm 1 *Netlist diff* algorithm

```
procedure DIFF(FIB old, FIB new)
    diff_cone ← Set.new
    same ← same_operation(old.op,new.op)
    for idx gets 0; idx < new.fanin.size; idx++ do
        if ! is_fib(fanin(new,idx)) then
            diff_cone.append(fanin(new,idx))
            same ← same & diff(fanin(old,idx),fanin(new,idx))
        end if
    end for
    return [same, diff_cone]
end procedure
```

After *Netlist diff*, the marked cones are extracted from the context of the design, and synthesized on their own (Figure 5) in **Partition Synthesis**. We carefully set nets as inputs and outputs to this new design to avoid them being optimized away. Since the block being synthesized does not necessarily begin and end in flops, we set input and output delays according to the ones reported in the original synthesis. This forces the synthesis to account for the delay of the logic that was not included in the block. Timing constraints are also set in accordance with the original design.

After the delta synthesis, the resulting netlist needs to be reattached to *Impl0* to create *Impl1* in the **Netlist stitch** step. Also, any unused nets need to be removed since synthesis will not be triggered over the whole design. Thus, we first inspect each gate in the original Invariant Cone and decrement its counter, removing from the design any gate that reaches the count of zero (Algorithm 2).

This procedure is sub-optimal for area, since it may result in redundancy. This overhead is small for each synthesis increment, but may accumulate over the course of multiple changes. However, note that a small hit in area (of around up to 5%, observed by [6]) is more tolerable than the same hit in delay.

Algorithm 2 *Netlist stitch* algorithm

```
1: procedure STITCH(Impl0, new_gates, old_gates, gate_count)
2:     for all gate ← old_gates do
3:         gate_count[gate.id]–
4:         if gate_count[gate_id] == 0 then
5:             remove(Impl0, gate.id)
6:         end if
7:     end for
8:     for all gate ← new_gates do
9:         insert(Impl0, gate.id)
10:    end for
11: end procedure
```

3.4.3 Dealing with delay degradation. To reduce delay penalties, when a critical path crosses the boundary of the changed region, the neighboring region is also included in for synthesis. This increases the runtime, but reduces delay impact on the final circuit. Another possibility would be to extend the partition definition, so the critical paths always lie within a region. One option not explored here is to trigger a second incremental synthesis when there is frequency degradation, however, it is not possible to know if the degradation is due to the flow or the change introduced.

4 EVALUATION SETUP

LiveSynth was implemented in Ruby 2.3 on top of Yosys, a tool based on ABC [3], targeting Xilinx FPGAs. Placement and Routing were done using Xilinx Vivado 2014.2, QoR results are reported after routing. We compare QoR with full synthesis for each change.

LiveSynth runtime is compared with with LLIR [4] and Rapid Recompile from Altera Quartus-II 2016.2. The experiments were run on 2 Intel(R) Xeon(R) E5-2689 CPUs at 2.60GHz, with 64GB of DDR3 memory, ArchLinux 4.3.3-3 server.

4.1 Benchmarks

We utilized three benchmarks: in-house Floating Point Unit (fpu), the open source MIPSfpga microMIPS core [7] (mips), the OR1200 RISC core [10] (or1200). To choose the benchmarks, we looked for open source benchmarks, possibly with public access to versioning control, that were large but would fit commercial FPGAs.

4.2 Change insertion

To emulate design changes, we inserted code changes to the benchmarks, using define statements. The changes can be divided into random synthetic, commented out code, and repository diffs.

Commented out code was used when available, following the method proposed in [5]. The same principle was applied to repository diffs when available. We looked for commits in nearby dates since we target small code changes. Commits that added entire modules or sub-systems were not considered. The idea of using commits from repositories tries to mimic "real-word" work.

To increase the number of changes, we also use synthetic changes. We used a pseudo-random number generator to select a file and a line of code. Then a change was made around that line. Changes include flipping bits, inverting conditions in if statements, inverting the order of concatenations, changing constant, changing expressions, and switching between constant and wire.

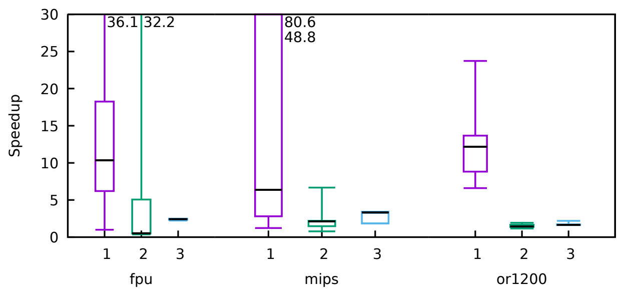


Figure 6: LiveSynth improves the synthesis speed by an average of ≈ $10x$ compared to a full synthesis. Each bar shows minimum, maximum and first, second and third quartiles for (1) LiveSynth, (2) LLIR and (3) Quartus. Values higher than the y-range are reported next to the bar.

The numbers of changes per benchmark are: fpu (32), mips (32), and or1200 (20). Our experiments start with all changes deactivated, and each change is incorporated with respect to the original run, independently of other changes. The only restriction on the changes inserted is the ability to synthesize the design. Changes can be single- or multi-line but always affect a single file and module. However, a changed module can be instantiated multiple times.

5 EVALUATION

We begin our evaluation by showing the overall speedups achieved by *LiveSynth* and prior approaches for our benchmarks. Then, we provide a detailed runtime breakdown which allows us to better understand how time is being spent during *LiveSynth*. Finally, we provide QoR results to show the quality differences between a full synthesis and the incremental synthesis techniques studied.

5.1 Overall Results

Our experiments show that *LiveSynth* was able to reduce synthesis runtime by 10x (median value) when compared to the full synthesis (Figure 6). In absolute numbers, this means a reduction from around 40s to around 4s in *LiveSynth*, but only to around 20s when using LLIR. For Quartus-II, the reduction was from around 120s with full synthesis to around 45s in the incremental version. We note that *LiveSynth* only launches synthesis when the code change affects the elaborated netlist, which is not always the case.

Quartus-II has an almost flat improvement in synthesis time of around 2x, which is surprising due to the nature of the compilation. Since there are not many details on the implementation of the flow available, it is not possible to understand why this behavior occurred. Both *LiveSynth* and LLIR have large variation in the speedup results, which is expected since each change to the RTL has a different impact in the final design.

5.2 Runtime breakdown and Setup time

We also report the runtime breakdown of *LiveSynth*. The longest step of is the setup phase, results shown in Table 2. Finding FIBs is in general slower than the full synthesis, but as mentioned, this is not a huge problem since it is only performed once before design changes are made. We also emphasize that this could be mitigated with better integration with the synthesis step [4] or by implementing the flow in a faster language such as C++.

Table 2: LiveSynth requires a setup phase once per design.

Benchmark	Synthesis Time (s)	Find FIBs (s)
fpu	37	375
mips	90	1403
or1200	22	84

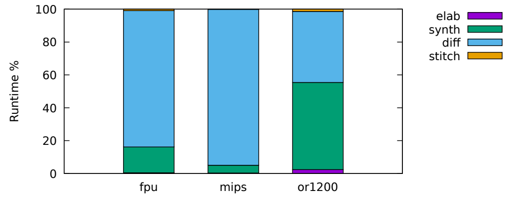


Figure 7: LiveSynth spends most of the time finding the difference between two designs. Synthesis time is around 10-40% of the total time depending on the benchmark.

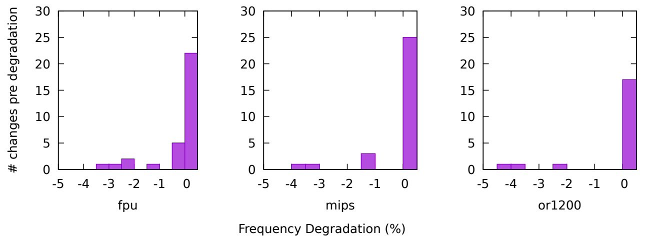


Figure 8: In most of the test cases, LiveSynth delivers the same frequency as a full synthesis. In the few cases where there were degradations, the hit on delay was around 4.5%.

The runtime breakdown for the incremental step is shown in Figure 7; a considerable amount of time is spent in finding *Netlist diff* (50-90%, depending on the benchmark). LLIR (not shown) uses 60-90% of the time in re-synthesis on average (across benchmarks). This is because LLIR requires a pass over the whole design for synthesis, which even if no work takes time. We note that the algorithm was implemented in Ruby, and runtime is expected to improve by a few times just by switching to a more efficient language such as C++. Stitch is not visible on the plot since it only takes a few milliseconds. Although *LiveSynth* requires a setup step, this setup is only executed once (before changes are inserted), and then multiple incremental synthesis steps can be performed without the need for running setup again.

5.3 QoR degradation

Finally, we investigate the QoR after *LiveSynth*. Some losses are expected due to the nature of our approach. Our results were compared against the regular synthesis of the full design and show that for most of the design changes, there are no degradations in delay. The maximum delay degradation was around 4.5% due to the incremental flow. Figure 8 shows the distribution of frequency degradation per change for each benchmark. In some cases, we observed a slight increase in frequency, but since it comes from noise in the synthesis tool we do not report it.

6 CONCLUSIONS

Slow turnaround time for synthesis is one of the main bottlenecks in hardware design productivity. We believe that an interactive synthesis flow is possible and would reduce design time by allowing faster iterations between code changes and results.

We present *LiveSynth*, an incremental synthesis flow independent of specific tools. *LiveSynth* leverages natural invariant boundaries to reduce the impact of splitting the design into regions while minimizing the impact on QoR. *LiveSynth* minimizes the amount of work that needs to be done by: 1) only elaborating RTL files that were changed by the designs, and 2) avoiding launching synthesis over the whole design. When a critical path lies within the boundaries of the incremental region, *LiveSynth* includes neighboring regions to reduce the hit on frequency.

Our results show that *LiveSynth* is able to reduce synthesis time by an average of 10x. We also show that *LiveSynth* has small impact on delay (frequency) for only a few design changes but always smaller than 5%. Our future work will look into partitioning blocks further and applying disjoint decomposition techniques that can split a block without compromising QoR. This should improve runtime in larger blocks observed in some of the benchmarks and thus further improve our results. We also want to leverage incremental placement and routing already present in FPGA flows.

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 grants CNS-1059442-003, CNS-1318943-001, CCF-1337278, and 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. 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).
[2] Daniel Brand, Anthony Drumm, Sandip Kundu, and Prakash Narain. 1994. Incremental Synthesis. In Proc. of the 1994 IEEE/ACM Int'l Conf. on Computer-aided Design (ICCAD '94). IEEE Computer Society, Los Alamitos, CA, USA, 14–18. http://dl.acm.org/citation.cfm?id=191326.191338
[3] Robert Brayton and Alan Mishchenko. 2010. ABC: An Academic Industrial-strength Verification Tool. In Proceedings of the 22Nd International Conference on Computer Aided Verification (CAV'10). Springer-Verlag, Berlin, Heidelberg, 24–40. DOI: https://doi.org/10.1007/978-3-642-14295-6_5
[4] Doris Chen and Deshanand Singh. 2011. Line-level Incremental Resynthesis Techniques for FPGAs. In Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '11). ACM, New York, NY, USA, 133–142. DOI: https://doi.org/10.1145/1950413.1950442
[5] K. Constantinides, O. Mutlu, and T. Austin. 2008. Online Design Bug Detection: RTL Analysis, Flexible Mechanisms, and Evaluation. In Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41). IEEE Computer Society, Washington, DC, USA, 282–293. DOI: https://doi.org/10.1109/MICRO.2008.4771798
[6] Mehrdad Eslami Dehkordi, S.D. Brown, and T. Borer. 2006. Modular Partitioning for Incremental Compilation. In Field Programmable Logic and Applications, 2006. FPL '06. International Conference on. 1–6. DOI: https://doi.org/10.1109/FPL.2006.311202
[7] Imagination Inc. 2016. MIPSfpga microMIPS Core, v1.3. (2016). https://community.imgtec.com/downloads/mipsfpga-getting-started-v1-3/.
[8] Norman P Jouppi. 1987. Timing analysis and performance improvement of MOS VLSI designs. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 6, 4 (1987), 650–665.
[9] Robert Cecil Martin. 2003. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall PTR, Upper Saddle River, NJ, USA.
[10] OpenRISC. 2016. OR1200 IP Core. (2016). https://github.com/openrisc/or1200 https://github.com/openrisc/or1200.
[11] Clifford Wolf. 2016. Yosys Open SYnthesis Suite. "http://www.clifford.at/yosys/". (2016).
[12] Xilinx Inc. 2015. Vivado Synthesis - Strategies for reducing run time. http://www.xilinx.com/support/answers/62215.html. (2015).
[13] Xilinx Inc. 2016. Vivado Design Suite User Guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_1/ug910-vivado-getting-started.pdf. (Apr 2016).