IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 19, 2021, accepted March 2, 2021, date of publication March 19, 2021, date of current version April 13, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3067453

High-Level Annotation of Routing Congestion for Xilinx Vivado HLS Designs

OSAMA BIN TARIQ[1], (Graduate Student Member, IEEE), JUNNAN SHAN[1], (Graduate Student Member, IEEE), GEORGIOS FLOROS[2], (Member, IEEE), CHRISTOS P. SOTIRIOU[2], MARIO R. CASU[1], (Senior Member, IEEE), MIHAI TEODOR LAZARESCU[1], (Senior Member, IEEE), AND LUCIANO LAVAGNO[1], (Senior Member, IEEE)

[1]Department of Electronics and Telecommunications, Politecnico di Torino, 10129 Torino, Italy
[2]Department of Electrical and Computer Engineering, University of Thessaly, 382 21 Volos, Greece

Corresponding author: Osama Bin Tariq (osama.bintariq@polito.it)

ABSTRACT Ever since transistor cost stopped decreasing, customized programmable platforms, such as field-programmable gate arrays (FPGAs), became a major way to improve software execution performance and energy consumption. While software developers can use high-level synthesis (HLS) to speed up register-transfer level (RTL) code generation from C++ or OpenCL source code, placement and routing issues, such as congestion, can still prevent achieving an FPGA programming bitstream or dramatically reduce the FPGA implementation performance. Congestion reports from physical design tools refer to thousands of RTL signal names instead of developer-accessible identifiers and statements, considerably complicating the developer understanding and resolution of the issues at the source level. We propose a high-level back-annotation flow that summarizes the routing congestion issues at the source level by analyzing the reports from the FPGA physical design tools and the internal debugging files of the HLS tools. Our flow describes congestion using comments back-annotated on the source code and identifies if the congestion causes are the on-chip memories or the DSP units (multipliers/adders), which are the shared resources very often associated with routing problems on FPGAs. We demonstrate on realistic large designs how the information provided by our flow helps to quickly spot congestion causes at the source level and to solve them using appropriate HLS directives.

INDEX TERMS FPGA, HLS, routing congestion.

I. INTRODUCTION

Since transistor cost stopped decreasing in the latest technology generations [1]–[3], software performance and energy improvement through parallelism was considerably reduced. Application-driven circuit customization allows increasing performance at comparable prices, but rising mask costs restrict this solution to a few application domains, such as general-purpose machine learning, as witnessed by many startups working on specialized chips in this domain. Other application domains can get some improvements, for a limited time, from extreme customizing of the datapath and memory architecture on a *reconfigurable platform*, such as a field-programmable gate array (FPGA). This means using a hardwired finite state machine (FSM) instead of a fetch/decode/execute cycle for control, using a customized memory hierarchy instead of a cache, and using the exact bit width datapath required by the application at hand for each operator. All these combined can typically gain an order of magnitude in power consumption, and sometimes a similar performance gain magnitude, over graphical processing units (GPUs), and much more over central processing units (CPUs). Hence, several cloud computing service providers, such as Amazon [4], Microsoft [5], [6], Alibaba [7], and Huawei [8], offer virtual machines with FPGA accelerators.

However, accelerating datacenter applications using FPGAs requires a software-like development cycle to be economically appealing. The developers expect to compile hardware accelerators from high-level source code written in languages like C++ or OpenCL [9]. Even higher level languages, such as Python, can benefit from this trend through accelerated implementations of mathematical libraries, like BLAS [10].

Even though C++ or OpenCL high-level synthesis (HLS) has made extraordinary progress in recent years (both Xilinx and Intel/Altera offer such tools), the underlying

The associate editor coordinating the review of this manuscript and approving it for publication was Christian Pilato.

implementation flow still relies on synthesis, placement, and routing. Limited routing resources in FPGAs [in particular involving "hardened" components such as block random access memory (BRAM) and digital signal processing (DSP) units] make meeting clock cycle requirements particularly challenging [11]. Up to 50 % of the FPGA resources can be left unused to complete the routing and/or to meet the timing requirements.

Unfortunately, the modern physical design tools report problems in terms of register-transfer level (RTL) nets for both FPGAs and application-specific integrated circuits (ASICs). For HLS users, this is like a C compiler reporting errors referred to the generated assembly instructions. Software developers, like experienced hardware designers, do not write, and often cannot read the RTL code generated by an HLS flow while porting a complex software application or library to FPGA accelerators.

To fill this gap, we provide a set of tools that *trace back the origin of congestion issues to the HLS input code*, written in C++ or OpenCL. This is only a first step toward a *physically aware HLS tool*, but it helps to solve a real, stringent problem. We developed a set of algorithms that treat the HLS flow almost as a black box and use the limited relationship that the HLS tools, and in particular the Xilinx Vivado HLS on which our results are based, keep between the C++ or OpenCL source code and the final RTL.

We use this feedback and some basic knowledge of the mapping between source language constructs (e.g., arrays and on-chip BRAMs, or arithmetic operations and DSP units) to show how the source code and/or the synthesis directives can be modified to reduce congestion, hence improving the final clock frequency without losing in other respects, and even improving the throughput due to BRAM partitioning introduced to reduce congestion. Our main contributions are:

- A new method to trace back the nets in congested areas to the high-level source code independently of the HLS tool used for RTL generation or the placement algorithm.
- Techniques to separate the congested nets by the functional unit they belong to (e.g., multipliers, adders or dividers, memory units, and the related multiplexers), to help adopting the best strategy to alleviate the congestion at the HLS level.
- We demonstrate the usefulness of the information that we trace back by improving the allocation of limited FPGA resources, such as DSPs and BRAMs, and increasing design performance, such as the operating clock frequency.

II. RELATED WORK

Tracking accurate routing congestion causes can guide timing and placement optimization, assist efficient design space exploration, and if back-annotated on the high-level source code it can direct the designer toward the code sections that are responsible for most routing congestion. FPGAs, unlike ASICs, have limited routing resources, hence congestion is a very common cause of routing problems, and it needs to be dealt with at the abstraction level used by the designer [12], which for HLS means the C++ or OpenCL code. Routing congestion can lead to timing violations, and lower power and area performance in the post-layout design phase [12].

Several methods have been proposed to adapt the HLS scheduling and allocation algorithms [13]–[16] to generate layout-friendly RTL models. For example, cut sizes and graph embedding metrics can correlate with routability, allowing one to evaluate the impact of HLS to improve the generated RTL routability on FPGA [16].

However, these methods improve the RTL models or incorporate floor planning information into HLS instead of finding the actual congestion causes and reporting them in the source code. Hence, they can only solve the congestion issues which are caused by *bad scheduling and binding decisions*, rather than those that originate from *inherently difficult to route designs or improper optimization directives*. Our method focuses especially on the latter.

Wirelength, timing, and routability estimations are often tightly connected physically and algorithmically. Zheng *et al.* [17] proposed an iterative high-level synthesis flow with constraints adjusted using the estimates from the Altera Quartus fast placement and routing tool and a delay estimation model developed by the authors. The system can be used to direct the synthesis to improve the circuit latency. Guo *et al.* [18] observed that the delays induced by signal broadcasting, either by the control logic or the data path, reduce the operating frequency, but they are not included in the delay models of widely used HLS tools, e.g., Vivado HLS. The authors propose techniques to optimize the timing of the implicit broadcasts, such as broadcast-aware scheduling, redundant synchronization pruning, and skid-buffer-based pipeline control.

Pui *et al.* [19] proposed a multi-stage congestion-driven global placement, and a routing-source-aware detailed placement to reduce routing congestion for FPGA. The authors also note that blocks with high pin counts and large areas, such as DSPs and BRAMs, can significantly influence the global placement, increasing the half-perimeter wire length (HPWL). Based on this observation, we focus mostly on such blocks. Tatsuoka and Kaneko [20] devised a source-to-source compiler to detect wire congestion caused by multiplexers using pattern matching on program dependence graph, without going through logic synthesis and physical design.

Li *et al.* [21] proposed a routability-driven FPGA placer, UTPlaceF, which implements a routing congestion-aware depopulation technique and a hierarchical congestion-aware detailed placement technique to improve wirelength and routability. However, this mostly improves the placement algorithm. We focus on improving the HLS source code regardless of the underlying placement and routing algorithms.

Recently, the use of machine learning methods has gained popularity also in this domain [22]–[28]. Maarouf *et al.* [23] used tens of millions of routed grid cells from large FPGAs

for training and testing. They used features such as wirelength per area as well as pin count and cut nets per region, and applied various machine learning algorithms to estimate the actual congestion in each grid cell after placement, without the need to route the chip. Alhyari *et al.* [24] trained and evaluated a convolutional neural network model on tens of thousands of images based on several hundred benchmarks from Xilinx to predict the routability of the designs based on the congestion heatmap during placement. In the same team, Szentimrey *et al.* [27] combined a deep learning-based congestion management model with the congestion estimation approach proposed in [23] and the routability prediction algorithm from [24] and used them in the GPlace3.0 framework [29] to achieve better runtime and quality of results. Pui *et al.* [26] also used machine-learning-based models to estimate the routing congestion for a given circuit placement on an FPGA along with detailed placement techniques (two-step clock legalization and chain move) to better optimize wirelength and meet emerging clocking architectural constraints of modern FPGAs, like Xilinx UltraScale. Yu and Zhang [28] proposed generative adversarial networks to evaluate the full FPGA routing resource utilization and congestion, including the detailed routing. They devised the task as an image translation problem that includes features collected up to circuit placement. Routing requirement estimation techniques can analyze circuit netlists and provide a fine granularity distribution of interconnect requirements over the whole FPGA device [30].

All the approaches above focus on predicting congestion after placement, avoiding the need for routing. We exploit the routing algorithm of Xilinx Vivado to estimate the congestion information after global routing, which we use because it is *obtained faster than the actual post-routing delays, yet it correlates very well with the routing negative slack on FPGAs*.

Back-annotation has been used to map the various low-level aspects of hardware design to higher abstraction levels. For example, Tatsuoka *et al.* [12] and Goering [31] note that large multiplexers and demultiplexers in ASICs significantly contribute to congestion. They can be traced back to the source code either as back-annotations or as suggested changes to help the designers modify the code or the associated synthesis directives. Similarly, Lee *et al.* [32] generated functional hardware models back-annotated with cycle-accurate and data-dependent power and performance estimates at the intermediate representation (i.e., LLVM instruction) level. Another study used multiple features extracted from the HLS flow corresponding to operations at the LLVM level to train machine learning (ML) models to predict routing congestion [22]. It used various examples to generate a data set of thousands of samples for training. The tool can back-annotate the source code with congestion predictions to help identify early *potential* routing congestion sources. However, no further information is provided in the back-annotated code on what resources may have contributed to the congestion.

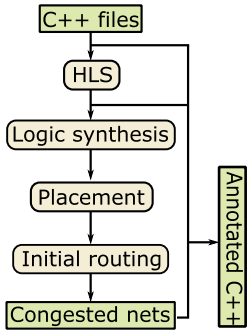


FIGURE 1. Design flow for back-annotation.

In summary, in this article, we use physical design tools (mostly Xilinx Vivado, although the method can be easily extended to cover other tools like Intel Quartus) as a black box. Rather than trying to predict the tool behavior (as the existing approaches listed above), we provide information at the source high level which can be used by the HLS users to iteratively improve the routability (and performance) of their designs. As we show in the extensive set of experiments that we performed, *reducing congestion dramatically helps achieve timing closure* on the FPGAs that we are targeting.

III. MOTIVATION

The most common FPGA design flow uses RTL models written in a hardware description language (HDL), which are then synthesized, placed, and routed. However, it is very time consuming and difficult to use efficiently by users unfamiliar with hardware architectures. Thanks to modern HLS tools, such as Xilinx Vivado HLS, the designers can easily program FPGAs using functional models written in high-level languages such as C++ or OpenCL. Hardware implementation (RTL generation) can also be easily directed using ''HLS pragmas''. However, to improve the design performance, users rely on the HLS tool reports that focus mostly on FPGA resource usage. Yet, poor high-level design choices can lead to routing congestion, which can significantly reduce the implementation performance (e.g., the FPGA working frequency) or the routing may fail altogether.

We describe a novel tool (Fig. 1) that provides back-annotations on high-level project sources in user-understandable terms on the possible causes of congestion, which should be addressed to improve the routability, hence the clock frequency.

IV. BACK-ANNOTATION – OUR APPROACH

A. GENERIC BACK-ANNOTATION FLOW

Our back-annotation flow includes five phases:

1) Extraction of the source-level debugging information. This information can be found in intermediate files generated during high-level synthesis. They contain information about the variables generated during synthesis and include the name and line number of their declarations in the source code.

2) Separation of the variables generated during the synthesis process based on their resource usage e.g., DSPs, BRAMs, and multiplexers.
3) Generation of a list of nets that cross congested FPGA tiles, weighed by relative significance (number of crossings) from the Vivado post-global routing database.
4) Cross-match the nets obtained from congested areas to the variable names generated during synthesis to find their source in the high-level code.
5) Back-annotation of the high-level source code lines that contribute the most to congestion, including the hardware resource types involved in congestion (essential to improve routability, as we will discuss later).

On-chip memories (implemented both as register files and BRAMs) and DSP units are the most common causes of congestion since they cannot be duplicated and moved around by the physical design tools as easily as look-up tables (LUTs) and registers. Tatsuoka *et al.* [12] and Goering [31] show the role of multiplexers in routing congestion. We observed that, in the case of designs synthesized with Vivado HLS, the congestion due to resource sharing multiplexers, and the resulting timing problems, can be significantly alleviated via a judicious choice of memory partitioning and DSP allocation directives.

B. XILINX VIVADO HLS AND VIVADO IMPLEMENTATION

Vivado HLS generates several intermediate files in the `<project>/<solution>/.autopilot/db` folder. The `.adb` files contain a control flow graph (CFG) describing the design at a high-level, close to the source code. Vivado HLS provides debugging data for each CFG node as an LLVM [33] instruction that produces a value flowing through the data path, hence closely associated with RTL nets and registers. It includes the source code line number, bit width, delay, and the RTL signal name. Source level data in `.adb` files are complemented by hardware resource level data in the `.rpt` report files that are also contained in the `<project>/<solution>/.autopilot/db` folder.[1] These report files contain additional useful information about multiplexers, such as the LUTs used and the input sizes. We then categorize them, by exploring the design connectivity, based on their relation with memories, functional units, and control. Using the project debugging files, we then find the source code lines and operations which feed these multiplexers. Note that, although these internal compiler files are not documented by Xilinx, their presence and format have been stable for several years. Moreover, we discuss in Section VII that other techniques, not based on the internal tool files, can also be used to achieve comparable results.

Xilinx Vivado can include routing congestion reports in its design analysis. The reports have three congestion tables:

- Placer Final congestion
- Initial Estimated Router congestion
- SLR Net Crossing

Our *automatic* flow analyzes the congestion report and then sends TCL commands to Vivado physical design engine to dump into intermediate files the RTL net names that cross the highly congested areas reported by the initial global router congestion estimates.

The extracted RTL net names are weighed by the number of congested tiles they cross and are then matched with the source level references extracted from the `.adb` and `.rpt` files. The `.adb` files, as mentioned above, contain the information about the variables generated during synthesis, which includes the name and line number of their declarations in the source code.

Fig. 2 shows the detailed steps of the back-annotation flow. The light blue rectangle blocks represent either the files generated by Vivado HLS and Vivado. The purple oval blocks represent various parts of our back-annotation tool while the purple rectangle blocks represent the output files generated using the back-annotation tool. Algorithm 1 shows the operation of the back-annotation method. We split the RTL names of the synthesized design into four categories (the `for` loop on line 2), with special handling for two FPGA resource types that are particularly prone to causing problems during routing: DSP units, BRAM blocks, and multiplexer units connected to them. Note that the LUTs and the flip-flops (FFs) cause fewer problems because the physical design tools can replicate them to improve the routing. The rest of the RTL names are categorized as *other nets*. More precisely, we adopt the following procedure:

1) Collect in a global list (lines 2 and 3 in Algorithm 1) the RTL names and source line numbers from the `.adb` files.
2) Store the RTL names connected to DSP instances in the `.adb` files in the *dsp_nets_list* (lines 4 and 5 in Algorithm 1), with the source line numbers.
3) Store the RTL names connected to BRAMs in the `.adb` files (with ''addr'', ''load'', or ''store'' suffixes) in the *bram_nets_list* (lines 6 and 7 in Algorithm 1), with the source line numbers.
4) Store the multiplexers in the design along with their RTL names and source code line numbers. Note that a single multiplexer can be associated with multiple RTL names and multiple source code line numbers. For example, a multiplier instance using DSPs can receive inputs from multiple source code lines with multiplication operations.
5) Collect all other RTL names in the *other_nets_list* (line 11 in Algorithm 1), with their source line numbers.

Steps 1 – 5 were done by the ''Parser'' in Fig. 2. Then, we scan the nets in all congested areas and match them against the names collected in these lists as follows:

6) List all tiles belonging to the congested windows, reported by the Vivado congestion report, with level

[1]Note that these files are different from the similarly named `.rpt` files in the user-accessible design report directory, which contain less useful data for our purposes.

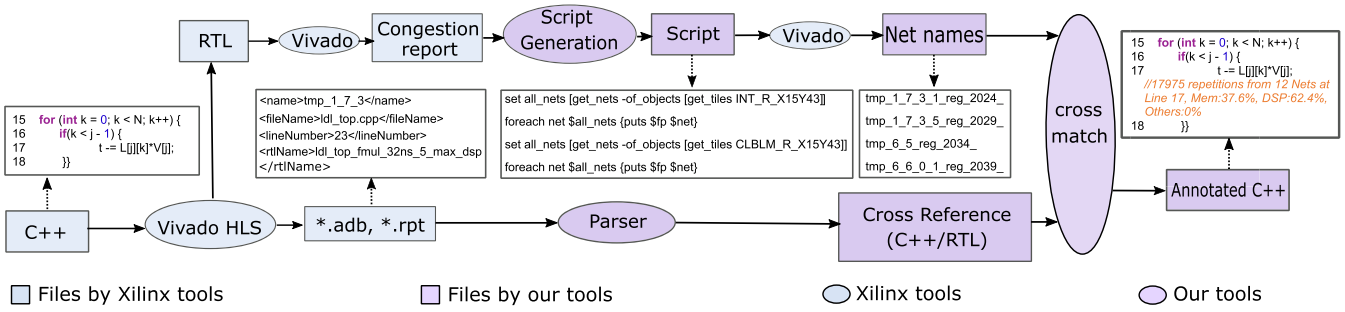


FIGURE 2. Our flow extracts design data from Vivado HLS project files and matches them with Vivado congestion reports. We then back-annotate the source code with synthetic congestion information.

3 and above and the nets passing through them (line 12 in Algorithm 1). The Vivado congestion report only lists the congestion windows and their corresponding congestion level. The "Script Generation" in Fig. 2 creates "Script" which has a sequence of commands for Vivado, to list tiles in congestion windows and the nets in them.

7) Attach source line numbers to each net crossing one or more tiles (line 13 in Algorithm 1) by removing one suffix at a time from their names (separated by '_') until the remaining name root matches an RTL name from the `.adb` files, which is associated to a source line number (lines 14 and 15 in Algorithm 1). For example, the net `L_6_3_loc_assign_2_reg_21300` matches the variable `L_6_3_loc_assign_2` in the `.adb` files, which has a source line number.
8) For any match, we increment the statistics associated with the (*name*, *line number*) set on the specific list: DSPs (from step 2), BRAMs (step 3), or *other* (step 5). If the *name* belongs to a multiplexer, we check in an additional step if it comes from a DSP- or memory-related multiplexer.
   Note that recognizing the memories (BRAMs) is more difficult because the `.adb` files do not include memory names. Hence we traverse the net hierarchy to determine if the net is connected to a BRAM.
9) Annotate the source code lines with the number and type of congested nets collected on lines 17, 19, 22, 24, and 26 of Algorithm 1.

Steps 7 and 8 were done at "cross match" stage in Fig. 2. We use the number of nets traversing congested tiles ("repetitions") as a measure of the congestion generated by the source code, while the type of resource involved in the congestion can hint the developer the actions to take to reduce it. In other words, "repetitions" is the sum of the weights of the nets corresponding to a source code line, where the weight of a net is the number of tiles crossed by the net. We empirically found that this correlates well with the amount of disturbance caused by a net, hence its impact on timing. Additionally, we generate a simple text report with details about the memories, the DSP usage in functional units, the multiplexers and their relation to the various parts of the source code, and the nets in the congested areas.

V. EXPERIMENTAL RESULTS

We demonstrate how the post-place and route congestion information can be correlated and back-annotated on the source code, and how we use it to resolve design congestion. We use a few *exemplary congested designs*, which can be easily explained in an article, but the tools and flow are fully general.

We examine the back-annotations and then infer changes to design source code or HLS directives based only on:

- specific knowledge of the source-level code;
- *generic knowledge about how the HLS and physical design of FPGAs work.*

Then, we verify that the physical design improves, i.e., the congestion is reduced, and iterate, if necessary.

We validate our tool using Vivado HLS and Vivado v2019.1, and we use SDAccel v2018.2 (which integrates Vivado HLS and Vivado) only for the convolution example in Section V-B.

The terms and abbreviations used in the following are explained in Table 1.

A. LDL DECOMPOSITION

LDL decomposition is a variant of the classical Cholesky decomposition. It is used for efficient implementation of many applications, such as direction of arrival (DOA) estimation [34], a hardware architecture for positive definite matrix inversion computation [35], and various numerical solutions, e.g., finance [36] and Monte Carlo simulations [37].

Our LDL implementation uses four floating-point array arguments: two 2D arrays and two 1D arrays. The implementation has an outer loop, which includes a sequence of several loops. Each inner loop performs multiplications, divisions, and subtractions. Application performance improved with a pipeline pragma outside the main loop to unroll large computations, and one of the 2D arrays is fully partitioned for concurrent data access (target FPGA `xa7s25ftgb196-2I`).

Listing 1 shows the back-annotated part of the LDL decomposition source. We note that congestion is due to memory accesses mainly in lines 20, 28, and partially in lines 17,

Algorithm 1: Code Back-Annotation Algorithm

```
procedure High-level-code-annotation
    // Collect all RTL nets by resource type
    for rtl_name in synthesized_design do
        add rtl_name, line_number to rtl_names
        if rtl_name associated in report to dsp_instances then
            add (rtl_name, line_number) to dsp_nets_list
        else if prefix(rtl_name) ∈ {addr, load, store} then
            add (rtl_name, line_number) to bram_nets_list
        else if rtl_name associated in report to Multiplexers then
            add (rtl_name, line_number) to multiplexers_list
        else
            add (rtl_name, line_number) to other_nets_list
    // Count congested nets by resource type and source line #
    for tile in congested_areas do
        for net crossing tile do
            while net ∉ rtl_names do
                remove_suffix(net)
            if (net) ∈ dsp_nets_list then
                matched_dsp [net, line_number]++
            else if (net) ∈ bram_nets_list then
                matched_bram [net, line_number]++
            else if (net) ∈ multiplexers_list then
                if (net) ∈ dsp_multiplexer then
                    matched_dsp [net, line_number]++
                else if (net) ∈ memory_multiplexer then
                    matched_bram [net, line_number]++
            else if (net) ∈ other_nets_list then
                matched_other [net, line_number]++
```

TABLE 1. Terms and abbreviations.

TNS	total negative slack
WNS	worst negative slack
Target	required clock period
PS	post-synthesis period
PI	post-implementation clock period
T_{exe}	execution time of the design on the FPGA

23, 31, 35. DSP resources (for floating-point operations) contribute to congestion mainly in lines 17, 23, 31 and partially in 32, while *Others*, i.e., LUTs and FFs used for implementing `fdiv` operations in line 32 and 35 are

```
17: V[k] = L[j][k]*D[k]; //7515 repetitions from 10 Nets, Mem:16%, DSP:84%, Others:0%
[...]

20: t = A[j][j]; //1625 repetitions from 8 Nets, Mem:100%, DSP:0%, Others:0%
[...]

23: t -= L[j][k]*V[j]; //17454 repetitions from 14 Nets, Mem:9%, DSP:91%, Others:0%
[...]

28: t = A[i][j]; //2320 repetitions from 28 Nets, Mem:100%, DSP:0%, Others:0%
[...]

31: t -= L[i][k]*V[k]; //15993 repetitions from 14 Nets, Mem:1%, DSP:99%, Others:0%
32: t -= L[i][k]/V[k]; //26184 repetitions from 12 Nets, Mem:0%, DSP:40%, Others:60%
[...]

35: L[i][j] = t / D[j]; //20807 repetitions from 17 Nets, Mem:5%, DSP:0%, Others:95%
```

LISTING 1. Congestion-related back-annotations on the unoptimized LDL decomposition source.

TABLE 2. FPGA resource usage for the LDL decomposition example.

		Slice	LUT	FF	DSP	BRAM	SRL
Original		2836	8778	11 131	12	0	101
Optimized	1	2618	8380	10 478	6	0	100
	2	3469	12 209	14 132	24	0	204

significant causes of congestion. From the text report generated by the tool we can also see further details, for example in line 31 how many nets involving DSP are because of multiplication operations and how many are because of subtraction operations. We address congestion in two steps i.e. first memory related and then operators related, to better see co-relation between congestion and trace-back. Most congested nets for lines 20 and 28 involve memory accesses. We partitioned the arrays to reduce memory port congestion and show the results on the first optimization line in Table 2 (BRAM usage is zero because the arrays are external to the synthesized block).

The remaining congestion after the first optimization is shown in Listing 2. The congestion due to memory accesses disappeared on lines 17, 28, 31 and 35 (but net repetitions slightly increased on line 20 because the array is now fully partitioned and implemented with FFs) and timing improved in terms of TNS and WNS but with higher T_{exe}(see Table 3). One consequence of addressing only the memory related issue (i.e., partitioning the memory) was that Vivado HLS by default allocated fewer DSPs to the design (see Table 2), aggravating the DSP resource allocation issue. We observe that the congestion related to DSPs is high on lines 17, 23, 31 and 32. Similarly, line 32 and 35 generate many congested nets due to `fdiv` operations. From the report generated by the tool we can see that the 86 % of the DSP nets on line 23 are because of `fsub` and 14 % are because of `fmul`, while 79 % belong to `fsub` and 21 % to `fmul` on line 31. This unbalance indicates clearly that more `fsub` units are needed.

```
17: V[k] = L[j][k]*D[k]; //6884 repetitions from 10
     Nets, Mem:0%, DSP:100%, Others:0%
[...]

20: t = A[j][j]; //2908 repetitions from 4 Nets, Mem:0%,
     DSP:0%, Others:100%
[...]

23: t -= L[j][k]*V[j]; //30140 repetitions from 20 Nets,
     Mem:4%, DSP:96%, Others:0%
[...]

31: t -= L[i][k]*V[k]; //28762 repetitions from 17 Nets,
     Mem:0%, DSP:100%, Others:0%
32: t -= L[i][k]/V[k]; //33492 repetitions from 17 Nets,
     Mem:0%, DSP:73%, Others:27%
[...]

35: L[i][j] = t / D[j]; //24544 repetitions from 18
     Nets, Mem:0%, DSP:0%, Others:100%
```

LISTING 2. Congestion-related back-annotations on the LDL decomposition source after the first optimization.

TABLE 3. Implementation performance for the LDL decomposition example.

	TNS (ns)	WNS (ns)	Target (ns)	PS (ns)	PI (ns)	Clock Cycles	T_{exe} (µs)
Original	−10.72	−0.424	5.0	4.26	5.42	253	1.37
Optimized 1	−5.92	−0.287	5.0	4.63	5.29	264	1.40
2	0.00	0.015	5.0	3.82	4.99	251	1.25

After allocating more operators using the pragma allocation, we both reduced the congestion and significantly improved the design timing (see Table 3).

It is worth mentioning that the tracing of congestion due to memory or DSPs helps to also reduce the multiplexer sizes. As mentioned previously, multiplexers are one of the main causes of routing congestion [12], [31]. We observed that after the first optimization, i.e. the partitioning of the arrays, all multiplexers related to memory vanished, but the remaining multiplexers related to DSPs were larger both in terms of LUTs and input size. After the second optimization, i.e. using the allocation pragma to allocate more operators, we had more multiplexers but they were much smaller in terms of LUTs and input sizes.

B. CONVOLUTION

The convolution kernel is one of the most important operations in popular machine learning algorithms for, e.g., machine vision and image recognition. Convolution kernels are both highly computationally intensive and highly parallelizable on FPGAs, to increase their throughput. They receive a feature map in input to which they convolve several (learned) filters to produce the output feature map.

The application was initially optimized for performance using loop tiling. The two innermost loops are fully unrolled, and the loop one level above is pipelined with an initiation interval of one (target FPGA `xcvu9p-flgb2104-2-i`).

The congestion back-annotation of our tool shows that source line 127 (see Listing 3) creates the most congestion because of memory and DSPs in nearly equal parts. The design already used $T_m \times T_n$ multipliers and adders and

```
120: for (int i=0; i<FILTER_SIZE; i++) {
121:    for (int j=0; j<FILTER_SIZE; j++) {
122:       for (int trr=0; trr<Tr; trr++) {
123:          for (int tcc=0; tcc<Tc; tcc++) {
124: #pragma HLS PIPELINE
125:             for (int too=0; too<Tm; too++) {
126:                for (int tii=0; tii<Tn; tii++) {
127:                   out[too][trr][tcc] +=
     filter_local[too*Tn+tii][i*FILTER_SIZE+j] *
     inp_image_local[tii][STRIDE*trr+i][STRIDE*tcc+j];
     //2806336 repetitions from 810 Nets, Mem:41.7%,
     DSP:58.3%, Others:0%
```

LISTING 3. Congestion-related back-annotation on the unoptimized convolution kernel source.

TABLE 4. Implementation performance for the convolution example.

	TNS (ns)	WNS (ns)	Target (ns)	PS (ns)	PI (ns)	Clock Cycles	T_{exe} (µs)
Original	−71 556	−2.43	4.0	N/A	6.43	1 177 465	7.57
Optimized (2 CU)	0	0.013	4.0	N/A	3.99	813 085	3.24

the arrays are already fully partitioned in the first dimension, hence allocating more resources or increase the array partitioning would not help much. We reduced congestion however by splitting the design into several smaller parallel modules, each with potentially less congestion because of the simpler RTL structure and lower throughput requirements.

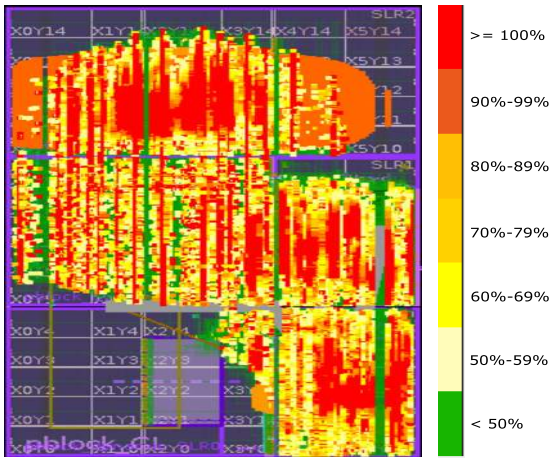
To do this, we reduced the loop unrolling factor and instantiated multiple module copies to preserve the overall performance. Specifically, we halved T_m and used two concurrent compute units (CUs). The congestion is markedly reduced (see Table 4), the target timing is met, and the performance increased by 57 %.

This reduced the routing congestion (as can be observed in Fig. 3 where the areas with more than 100 % interconnect density are significantly reduced) and the design satisfied the timing constraints (see Table 4) with almost unchanged resource usage (see Table 5).

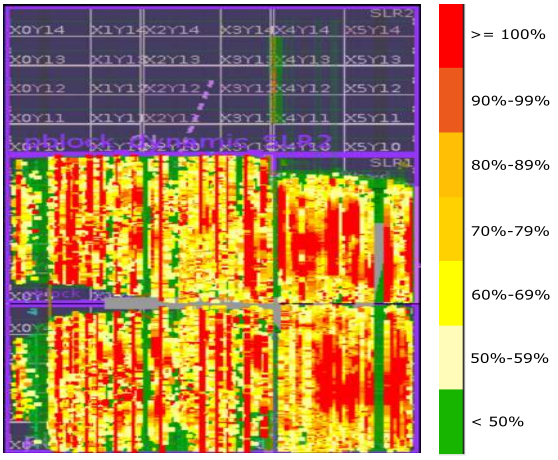
C. OPTICAL FLOW

Optical flow detects the movement pattern of objects between image frames, which is a vital and broadly used component for object detection and tracking in various image/video processing toolsets, such as OpenCV and the MATLAB Computer Vision toolbox.

We used the C++ model from the Rosetta benchmark [38], which is based on the FPGA-friendly Lucas-Kanade method [39]. It computes the movement of each pixel in five sequential image frames. The top-level function runs sequentially eight sub-functions creating a streaming dataflow pipeline between different stages of the algorithm. In this application, the main computing kernels are 1D convolution and Outer product. To optimize this application, we pipelined the outer loops to further improve the performance of the original code, and synthesized the design obtaining the resource utilization and performance shown in Table 6 and Table 7, respectively. On the target FPGA, `xcvu9p-flgb2104-2-i`, the timing requirement is not met due to routing congestion. Our tool traced the most congested nets to the code shown in Listing 4.



(a)



(b)

FIGURE 3. Interconnect density in percentage for the convolution example: (a) before and (b) after design optimizations.

TABLE 5. FPGA resource usage for the convolution example.

	LUT	LUT Mem	Reg	BRAM	DSP
Original	159 986	10 064	292 753	41	2564
Optimized (2 CU)	160 936	10 410	292 639	50	2564

TABLE 6. FPGA resource usage for the optical flow example.

	BRAM	DSP	FF	LUT	URAM
Original	2527	40	13 114	10 545	0
Optimized	2689	90	19 852	31 022	0

Lines 31 and 32 show that congestion is partially due to DSP utilization. As in previous cases, we used the allocation pragma to resolve it. Additionally, lines 125 to 127 show that many congested nets are due to memory. To resolve this, we used the appropriate array partition pragmas, depending on the array access patterns. After synthesis and implementation, we note that congestion was highly reduced and timing is almost met, as shown in Table 7. Moreover, the performance

```
31: x_grad += frame[r-2][c-i] * GRAD_WEIGHTS[4-i];
    //1134 repetitions from 2 Nets, Mem:0%, DSP:58%,
    Others:42%
32: y_grad += frame[r-i][c-2] * GRAD_WEIGHTS[4-i];
    //402 repetitions from 3 Nets, Mem:0%, DSP:10.5%,
    Others:89.5%
    [...]

125: acc.x += y_filt[r][c-i].x * GRAD_FILTER[i]; //3858
    repetitions from 9 Nets, Mem:58%, DSP:18%,
    Others:24%
126: acc.y += y_filt[r][c-i].y * GRAD_FILTER[i]; //4766
    repetitions from 7 Nets, Mem:75%, DSP:0%, Others:25%
127: acc.z += y_filt[r][c-i].z * GRAD_FILTER[i]; //5449
    repetitions from 7 Nets, Mem:88%, DSP:1%, Others:11%
```

LISTING 4. Congestion-related back-annotations on the unoptimized optical flow source.

TABLE 7. Implementation performance for the optical flow example.

	TNS (ns)	WNS (ns)	Target (ns)	PS (ns)	PI (ns)	Clock Cycles	T_{exe} (ms)
Original	−4947	−1.41	5.0	3.82	6.41	1 127 746	7.23
Optimized	−0.05	−0.04	5.0	3.82	5.04	1 127 738	5.69

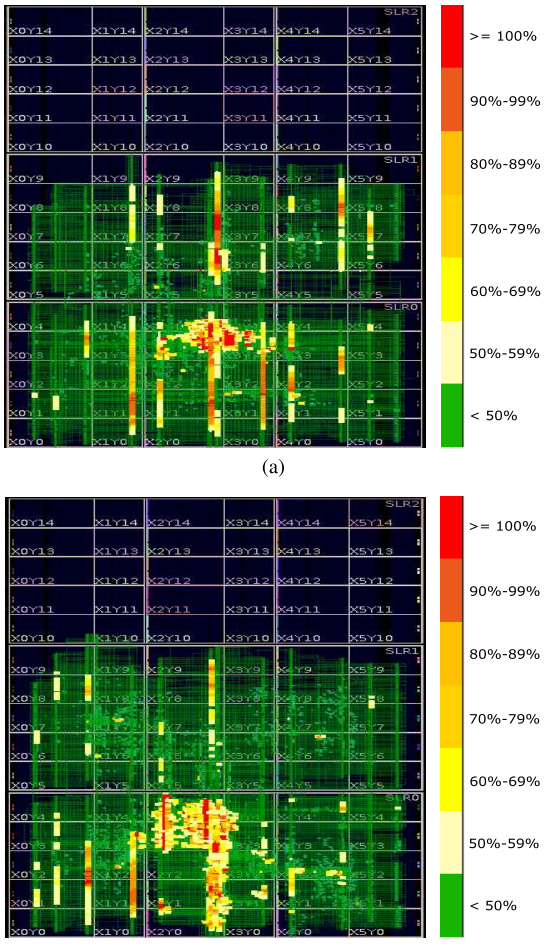
improved by 20 % thanks to the additional DSP resources and BRAM ports. The improvements can also be seen in Fig. 4 that shows the interconnect density before and after optimizations.

VI. INTERPRETATION OF ANNOTATIONS

As can be observed from the examples, the annotations belong to three categories: DSP-related, BRAM-related, and others (such as FF- and LUT-related). The DSP-related nets typically come from the implementation of mathematical operations (e.g., multiplication, addition, subtraction, division). Note that floating-point divisions, like in the LDL example, were implemented using FFs and LUTs, which can be traced back to the `fdiv` operation in the source code. So we can trace congested nets back to the source lines and operations regardless of the hardware resource used for its implementation. If a line in the source code contains multiple operations, our tool can identify the weight of the congested nets associated with each operation. We print this detail in a separate report, for readability.

Such detailed operation trace-back can guide the user to improve resource allocation. For example, when a DSP unit implementing a subtraction has a high number of repetitions, the user can direct the synthesis to allocate more subtraction units, e.g., using the allocation pragma. Similarly, if more repetitions in the *others* category are caused by a `fdiv` operation, the user can direct the synthesis to allocate more floating-point divisions. Besides improving resource allocation, this also breaks large sharing multiplexers into smaller ones, which contribute less to congestion.

If a large number of repetitions point to the BRAMs, the user can partition the arrays according to the data access order. Arrays implemented as BRAMs are reported in the *Mem* category, while arrays implemented by FFs are included in *Others*.



(a)

(b)

FIGURE 4. Interconnect density in percentage for the Optical Flow example: (a) before and (b) after design optimizations.

In summary, we can trace-back congested nets caused by arrays and operations regardless of what hardware resources are used for their implementation.

For some large very parallelizable designs like convolutions, if the trace-back points to the DSPs and BRAMs, and both blocks are already allocated and partitioned to fully exploit the available resources, we can split the design into several smaller ones that are working independently. Thus, each DSP unit will use a smaller amount of input data as operands, hence the multiplexer size will be reduced.

Note that *extracting congestion information from heat maps or reports containing hundreds or thousands of net names is absolutely non-trivial*, especially for large FPGAs, like the one used in the convolution example (Section V-B). This clearly shows the need for a back-annotation-based source-level congestion resolution flow, such as the one we propose in this article.

VII. BACK-ANNOTATION GENERALIZATION

Using the information from the `.adb` files, we can disambiguate well the attribution of nets to source code lines and variables, but this technique depends on the specific tool that we used (Xilinx Vivado HLS in this case). Alternatively, we can match unique source code identifiers directly with net names, without relying on tool-specific intermediate files. We thus tested also an independent method, based purely on replacing C++ source variable names with easily distinguishable ones, through hashing, and then looking for these names in the congestion reports. We used a tool that replaces the name of each variable with a unique hash, so that different variables with the same original name get different hashes (we used only 8 characters for readability).

Note that in our experiment we back-annotated the modified code just to show the effectiveness of the technique. In a better engineered version of the method, the back-annotation should be performed directly on the original source code.

The main steps of our procedure are:

1) Replace source code variable names with a unique hash.
2) Run the whole synthesis flow.
3) Obtain the congestion report.
4) Collect all net names in congested regions.
5) Select the nets matching the hashed source identifiers.
6) Count the congested nets matched by each identifier.
7) Associate with nets the source code line number that generated them and the number of congested tiles crossed.

Note that, as discussed above, we are interested in congested nets connecting to on-chip memories (BRAMs) or shared functions (floating-point operations and DSP units). We can easily identify the memories from the array names that are propagated into the netlist, since source code arrays that are mapped on memories, and hence BRAM-related congestion, can be traced back directly using this mechanism, without relying on intermediate files used by the HLS tool.

For nets associated with DSP units, we cannot trace back their congestion information based on variable names directly. We uniquely associate each operation (an integer or floating-point operation mapped to a shared DSP) by breaking a complex expression *where operations can be mapped to DSP units* into elementary assignments, hence with an associated source code variable. Note that this does not need to be done for the entire source code, but only for statements where *the sharing occurs*. In Xilinx Vivado HLS this can happen only for non-inlined user functions or floating-point operations. Then, we use the variable-to-shareable-operation one-to-one correspondence to trace the congestion information back to the source code, e.g., in Listing 5, the multiplication operation on line 28 is assigned to a new variable `o_76398e`. To trace back the nets from the DSP unit used on line 28, we count how many times the identifier `o_76398e` is found in the congested region. As mentioned above, we can automatically get information about the operations which are assigned to DSPs through the synthesis report. Hence, we can perform this manual breaking only for nets that are associated with shared DSPs. In future work, this step can also be trivially automated via a source-to-source translator.

```
23: o_f7efe8 = o_74d024[o_471974][o_471974]; //32031
     repetitions, Mem:100%, DSP:0%, Others:0%
    [...]

27: float o_bae153 = o_82649c[o_471974]; //2678
     repetitions, Mem:60%, DSP:0%, Others:40%
28: float o_76398e = o_1a7a49 * o_bae153; //435
     repetitions, Mem:0%, DSP:100%, Others:0%
29: float o_29c208 = o_f7efe8 - o_76398e; //4849
     repetitions, Mem:0%, DSP:91%, Others:9%
    [...]

35: o_f7efe8 = o_74d024[o_d95347][o_471974]; //32031
     repetitions, Mem: 100%, DSP:0%, Others:0%
    [...]

48: float o_75ede2 = o_f7efe8 / o_6d2ce5[o_471974];
     //598 repetitions, Mem:0%, DSP:0%, Others:100%
49: o_c3630f[o_d95347][o_471974] = o_75ede2; //18813
     repetitions, Mem:0%, DSP:0%, Others:100%
```

LISTING 5. Congestion-related back-annotations based on variable name hashing for the unoptimized LDL decomposition source.

```
23: o_f7efe8 = o_74d024[o_471974][o_471974]; //4115
     repetitions, Mem:100%, DSP:0%, Others:0%
    [...]

27: float o_bae153 = o_82649c[o_471974]; //6222
     repetitions, Mem:23%, DSP:0%, Others:77%
28: float o_76398e = o_1a7a49 * o_bae153; //748
     repetitions, Mem:0%, DSP:100%, Others:0%
29: float o_29c208 = o_f7efe8 - o_76398e; //1558
     repetitions, Mem:0%, DSP:52%, Others:48%
    [...]

35: o_f7efe8 = o_74d024[o_d95347][o_471974]; //4115
     repetitions, Mem: 100%, DSP:0%, Others:0%
    [...]

40: float o_9d068e = o_e7ff79 * o_036404; //4362
     repetitions, Mem:0%, DSP:100%, Others:0%
    [...]

48: float o_75ede2 = o_f7efe8 / o_6d2ce5[o_471974];
     //7241 repetitions, Mem:0%, DSP:0%, Others:100%
49: o_c3630f[o_d95347][o_471974] = o_75ede2; //22336
     repetitions, Mem:0%, DSP:0%, Others:100%
```

LISTING 6. Congestion-related back-annotations based on variable name hashing on the LDL decomposition source after the first optimization.

Now we analyze if the back-annotation through hashed variable names provides similar congestion annotations, hence resolutions, as the back-annotation through `.adb` files. In Listing 5, we can see the results of tracing back through variable name hashing for the example in Section V-A. In `ldl`, on lines 23, 27, and 35, we observe the congested nets related to memory accesses, which match exactly those reported by the `.adb` file method. Even though both the absolute numbers and the percentages on nets involved in congestion vary (in fact, the hashing-based method can trace back more congested nets than the `.adb` file method), the result for the designer is the same, identifying precisely the causes of congestion. The nets on lines 28 and 29 are attributed to the DSP resources. Note that the nets on line 48 are much fewer than those obtained via `.adb` backtracing (598 vs. 19 766), but in the following line we can see a high number of nets. This happens because the variable containing the result of division operation (`o_75ede2`) is assigned to the array `o_c3630f`. After partitioning all arrays, as in the `.adb`-based annotation, we can see the results in Listing 6. We can thus achieve the same result, as the congestion due to memory accesses is significantly reduced in lines 23 and 35 (the same array is causing congestion in both cases). We still observe a high number of congested nets due to DSP operations (which even increased in some cases, particularly on line 40 where it increased from 670 nets to 4362 nets), indicating the need to allocate more resources, as we did in the `.adb`-based method in Section V-A.

The technique discussed in this Section has only been partially automated, namely by automatically generating variable names. In the future, we can improve it to back-annotate the original source code, to provide better user support and replacing thus completely the one based on `.adb` files. For now, our goal was simply to illustrate that our technique is fully general because we obtain similar results without relying on tool-specific internal files and reports.

We also verified that this method works with the Intel HLS tool for their FPGAs (which incidentally uses the same LLVM front-end as Xilinx Vivado HLS). The hashed names of the variables related to both DSPs and on-chip RAM were found in the RTL. Since the messages and reports that are generated by physical design use the RTL names, it would be possible to complete a similar flow also for Intel FPGAs, and thus our method can support multiple vendors.

VIII. CONCLUSION

We showed that the back-annotation of post-placement global routing congestion information can be used to effectively and easily improve performance (in a particular clock period) by performing simple transformations using HLS directives.

Note that the recipes that we followed were relatively straightforward:

1) If congestion was due to BRAM, we partitioned it, typically resulting in increased performance.
2) If DSPs were involved, we either allocated more units or created one level of function hierarchy to simplify the scheduler's job and create a more regular netlist.

These recommendations can be provided directly to the designer, based on the findings of our tool, and may also be automated as part of future work. We believe that our approach can effectively help to solve a serious problem that affects fast HLS-based design flows for modern large accelerators implemented on FPGAs.

REFERENCES

[1] Z. Or-Bach. (2012). Is the cost reduction associated with IC scaling over? EE Times. Accessed: Nov. 13, 2020. [Online]. Available: https://www.eetimes.com/is-the-cost-reduction-associated-with-ic-scaling-over/

[2] K. Flamm, "Measuring Moore's law: Evidence from price, cost, and quality indexes," in *Measuring and Accounting for Innovation in the 21st Century*. Chicago, IL, USA: Univ. of Chicago Press, 2019.

[3] N. Thompson and S. Spanuth, "The decline of computers as a general purpose technology: Why deep learning and the end of Moore's law are fragmenting computing," Nov. 2018. [Online]. Available: https://ssrn.com/abstract=3287769, doi: 10.2139/ssrn.3287769.

[4] J. Varia and S. Mathew, "Overview of Amazon Web services," Amazon Web Service, Seattle, WA, USA, Tech. Rep., 2014, pp. 1–22. Accessed: Nov. 13, 2020. [Online]. Available: https://media.amazonwebservices.com/AWS_Overview.pdf
[5] *Microsoft Azure Cloud*. Accessed: Nov. 13, 2020. [Online]. Available: https://azure.microsoft.com/
[6] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–13.
[7] Alibaba Group. *Alibaba Cloud*. Accessed: Nov. 13, 2020. [Online]. Available: https://www.alibabacloud.com/
[8] *FPGA Accelerated Cloud Server-Huawei Cloud*. Accessed: Nov. 13, 2020. [Online]. Available: https://www.huaweicloud.com/
[9] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016.
[10] T. De Matteis, J. de Fine Licht, and T. Hoefler, "FBLAS: Streaming linear algebra on FPGA," 2019, *arXiv:1907.07929*. [Online]. Available: http://arxiv.org/abs/1907.07929
[11] S. Yang, A. Gayasen, C. Mulpuri, S. Reddy, and R. Aggarwal, "Routability-driven FPGA placement contest," in *Proc. Int. Symp. Phys. Design*, Apr. 2016, pp. 139–143.
[12] M. Tatsuoka, R. Watanabe, T. Otsuka, T. Hasegawa, Q. Zhu, R. Okamura, X. Li, and T. Takabatake, "Physically aware high level synthesis design flow," in *Proc. 52nd Annu. Design Automat. Conf.*, Jun. 2015, p. 162.
[13] J. Wu, C. Ma, and B. Huang, "Congestion aware high level synthesis combined with floorplanning," in *Proc. IEEE Pacific–Asia Workshop Comput. Intell. Ind. Appl.*, vol. 2, Dec. 2008, pp. 935–938.
[14] Y. Wang, J. Bian, Q. Wu, and H. Hu, "Reallocation and rescheduling after floor-planning for timing optimization," in *Proc. 5th Int. Conf.*, vol. 1, 2003, pp. 212–215.
[15] W. E. Dougherty and D. E. Thomas, "Unifying behavioral synthesis and physical design," in *Proc. 37th Conf. Design Automat.*, 2000, pp. 756–761.
[16] J. Cong, B. Liu, G. Luo, and R. Prabhakar, "Towards layout-friendly high-level synthesis," in *Proc. ACM Int. Symp. Int. Symp. Phys. Design*, 2012, pp. 165–172.
[17] H. Zheng, S. T. Gurumani, K. Rupnow, and D. Chen, "Fast and effective placement and routing directed high-level synthesis for FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2014, pp. 1–10.
[18] L. Guo, J. Lau, Y. Chi, J. Wang, C. H. Yu, Z. Chen, Z. Zhang, and J. Cong, "Analysis and optimization of the implicit broadcasts in FPGA HLS to improve maximum frequency," in *Proc. 57th ACM/IEEE Design Automat. Conf. (DAC)*, Jul. 2020, pp. 1–6.
[19] C.-W. Pui, G. Chen, W.-K. Chow, K.-C. Lam, J. Kuang, P. Tu, H. Zhang, E. F. Y. Young, and B. Yu, "RippleFPGA: A routability-driven placement for large-scale heterogeneous FPGAs," in *Proc. 35th Int. Conf. Comput.-Aided Design*, Nov. 2016, pp. 1–8.
[20] M. Tatsuoka and M. Kaneko, "Wire congestion aware high level synthesis flow with source code compiler," in *Proc. Int. Conf. IC Design Technol. (ICICDT)*, Jun. 2018, pp. 101–104.
[21] W. Li, S. Dhar, and D. Z. Pan, "UTPlaceF: A routability-driven FPGA placer with physical and congestion aware packing," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 4, pp. 869–882, Apr. 2018.
[22] J. Zhao, T. Liang, S. Sinha, and W. Zhang, "Machine learning based routing congestion prediction in FPGA high-level synthesis," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 1130–1135.
[23] D. Maarouf, A. Alhyari, Z. Abuowaimer, T. Martin, A. Gunter, G. Grewal, S. Areibi, and A. Vannelli, "Machine-learning based congestion estimation for modern FPGAs," in *Proc. 28th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2018, pp. 427–4277.
[24] A. Alhyari, A. Shamli, Z. Abuwaimer, S. Areibi, and G. Grewal, "A deep learning framework to predict routability for FPGA circuit placement," in *Proc. 29th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2019, pp. 334–341.
[25] Z. Qi, Y. Cai, and Q. Zhou, "Accurate prediction of detailed routing congestion using supervised data learning," in *Proc. IEEE 32nd Int. Conf. Comput. Design (ICCD)*, Oct. 2014, pp. 97–103.
[26] C.-W. Pui, G. Chen, Y. Ma, E. F. Y. Young, and B. Yu, "Clock-aware ultrascale FPGA placement with machine learning routability prediction," in *Proc. 36th Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2017, pp. 929–936.
[27] H. Szentimrey, A. Al-Hyari, J. Foxcroft, T. Martin, D. Noel, G. Grewal, and S. Areibi, "Machine learning for congestion management and routability prediction within FPGA placement," *ACM Trans. Des. Automat. Electron. Syst.*, vol. 25, no. 5, pp. 1–25, Oct. 2020.
[28] C. Yu and Z. Zhang, "Painting on placement: Forecasting routing congestion using conditional generative adversarial nets," 2019, *arXiv:1904.07077*. [Online]. Available: http://arxiv.org/abs/1904.07077
[29] Z. Abuowaimer, D. Maarouf, T. Martin, J. Foxcroft, G. Gréwal, S. Areibi, and A. Vannelli, "GPlace3. 0: Routability-driven analytic placer for ultra-scale FPGA architectures," *ACM Trans. Des. Automat. Electron. Syst.*, vol. 23, no. 5, pp. 1–33, 2018.
[30] P. Kannan and D. Bhatia, "Interconnect estimation for FPGAs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 25, no. 8, pp. 1523–1534, Aug. 2006.
[31] R. Goering, (Dec. 2014). *Front-End Summit: Avoiding Routing Congestion With High-Level Synthesis*. [Online]. Available: https://community.cadence.com/cadence_blogs_8/b/ii/posts/front-end-summit-avoiding-routing-congestion-with-high-level-synthesis
[32] D. Lee, L. K. John, and A. Gerstlauer, "Dynamic power and performance back-annotation for fast and accurate functional hardware simulation," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, 2015, pp. 1126–1131.
[33] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, 2004, pp. 75–86.
[34] A. A. Hussain, N. Tayem, and A.-H. Soliman, "LDL decomposition-based FGPA real-time implementation of DOA estimation," in *Proc. 52nd Asilomar Conf. Signals, Syst., Comput.*, Oct. 2018, pp. 1163–1168.
[35] C. Ingemarsson and O. Gustafsson, "Hardware architecture for positive definite matrix inversion based on LDL decomposition and back-substitution," in *Proc. 50th Asilomar Conf. Signals, Syst. Comput.*, Nov. 2016, pp. 859–863.
[36] P. Dellaportas and M. Pourahmadi, "Cholesky-GARCH models with applications to finance," *Statist. Comput.*, vol. 22, no. 4, pp. 849–855, Jul. 2012.
[37] X. Wang and I. H. Sloan, "Quasi-Monte Carlo methods in financial engineering: An equivalence principle and dimension reduction," *Oper. Res.*, vol. 59, no. 1, pp. 80–95, Feb. 2011.
[38] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez, W. Wang, and Z. Zhang, "Rosetta: A realistic high-level synthesis benchmark suite for software programmable FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2018, pp. 269–278.
[39] Z. Wei, D.-J. Lee, and B. E. Nelson, "FPGA-based real-time optical flow algorithm design and implementation," *J. Multimedia*, vol. 2, no. 5, pp. 1–8, 2007.

OSAMA BIN TARIQ (Graduate Student Member, IEEE) received the M.S. degree in electronic engineering with specialization in embedded systems from the Politecnico di Torino, Italy, where he is currently pursuing the Ph.D. degree with the Department of Electronic and Telecommunications Engineering. His research interests include artificial intelligence and machine learning applications, indoor localization, and high-level synthesis.

JUNNAN SHAN (Graduate Student Member, IEEE) received the B.S. and M.S. degrees from the Politecnico di Torino, Italy, where she is currently pursuing the Ph.D. degree with the Department of Electronics and Telecommunications under the supervision of Prof. Mario Casu and Prof. Luciano Lavagno. Her research interests include electronic design automation, system-level design, low-power, high-performance computing, and high-level synthesis.

GEORGIOS FLOROS (Member, IEEE) received the B.Sc. degree in computer science and the M.Sc. degree in electrical engineering. Since 2013, he has been working with FPGAs on topics like data acquisition networks, software defined radios, and cryptography.

CHRISTOS P. SOTIRIOU received the B.Eng. degree in computer science and electronics, and the Ph.D. degree in computer science from The University of Edinburgh, Scotland, in 2001. He is currently an Associate Professor with the Department of Electrical and Computer Engineering, University of Thessaly. His research interests include design methodologies for synchronous or asynchronous digital circuits and systems, electronic design automation (EDA) algorithms, tools and flows for digital circuit implementation, physical design, lower-power design, reliability and power, performance and area (PPA) optimisation. He received the Qualcomm Faculty Award twice, in 2019 and 2020, and has collaborated with several large corporations on EDA tools and flows development, circuit implementation, and physical design.

MARIO R. CASU (Senior Member, IEEE) received the Ph.D. degree in electronics and communications engineering from the Politecnico di Torino, Torino, Italy, in 2001. He is currently an Associate Professor. His past work focused mostly on latency-insensitive design of systems-on-chip (SoC) and on networks-on-chip. His research interests include systems-on-chip with specialized accelerators, system-level design and design methodology for FPGAs and ASICs, and embedded machine learning. He is also interested in the design of circuits, systems, and platforms for industrial applications (biomedical, automotive, food). He regularly serves for the Technical Program Committee of International conferences, such as DAC, ICCAD, and DATE.

MIHAI TEODOR LAZARESCU (Senior Member, IEEE) received the Ph.D. degree in electronics and communications from the Politecnico di Torino, Italy, in 1998. He was a Senior Engineer with Cadence Design Systems and founded several startups. He currently serves as an Assistant Professor. He has coauthored more than 60 scientific publications, four books, and international patents. His research interests include design tools for reusable WSN platforms, sensing, indoor localization, and data processing for the IoT, low power embedded design, high-level HW/SW co-design, and high-level synthesis.

LUCIANO LAVAGNO (Senior Member, IEEE) received the Ph.D. degree in electrical engineering and computer sciences from the University of California at Berkeley, Berkeley, CA, USA, in 1992. He was an Architect of the POLIS HW/SW co-design tool. From 2003 to 2014, he was an Architect of the Cadence CtoSilicon high-level synthesis tool. Since 1993, he has been a Professor with the Politecnico di Torino, Italy. He coauthored four books and more than 200 scientific articles. His research interests include synthesis of asynchronous circuits, HW/SW co-design, high-level synthesis, and design tools for wireless sensor networks.