DynaRapid: Fast-Tracking from C to Routed Circuits

Andrea Guerrieri[†*], Srijeet Guha[*], Chris Lavin[§], Eddie Hung[§], Lana Josipović[‡] and Paolo Ienne[*]
[*]EPFL, School of Computer and Communication Sciences, Lausanne, Switzerland
[†]HES-SO Valais-Wallis, School of Engineering, Sion, Switzerland
[§]AMD Research and Advanced Development
[‡]ETH Zurich, Department of Information Technology and Electrical Engineering, Zurich, Switzerland

***Abstract*—Advancements in design automation technologies, such as *high-level synthesis* (HLS), have raised the input abstraction level and made the design entry process for FPGAs more friendly to software programmers. In contrast, the backend compilation process for implementing designs on FPGAs is considerably more lengthy compared to software compilation: while software code compilation may take just a few seconds, FPGA compilation times can often span from several minutes to hours due to the complexity of the underlying toolchain and ever-growing device capacities. In this paper, we present DynaRapid, a fast compilation tool that generates—in a matter of seconds—fully legal placed-and-routed designs for commercial FPGAs. Elastic circuits created by the HLS tool Dynamatic are made exclusively of a limited number of reusable components; we exploit this fact to create a library of placed and routed building blocks, and then stitch together instances of them as needed through RapidWright. Our approach accelerates the C-to-FPGA implementation process by a geomean 20× with only 10% of degradation in operating frequency compared to a conventional commercial off-the-shelf implementation flow.**

I. Introduction

Field-programmable gate arrays (FPGAs) are versatile computational devices; yet, the experience that they can offer to programmers critically depends on two challenging aspects: (i) the possibility for users to program these devices with common languages used for high-performance applications, such as C/C++, and (ii) a quick development turnaround from code changes to execution.

Progress in software-like design entry for FPGAs (referred to as *high-level synthesis*, HLS) has been significant for more than a decade [1], [2]. Typical modern FPGA accelerators consist of a relatively large and highly optimized infrastructure (e.g., customized memory hierarchy, host interfaces, job scheduling mechanisms) and of a multitude of processing elements (PEs) implementing one or more computing kernels. While the former is often handcrafted at register transfer level (RTL) and seldomly requires changes for a given application domain, PEs are usually small- to medium-size kernels where software development focuses and where the ability of HLS to extract operation-level parallelism has the best chances to excel. Yet, the time lag for developers between implementing code changes and starting to debug remains highly unsatisfactory. While common software compilation times are on the order of seconds to produce executable files after limited code changes, FPGA tools ordinarily take up to hours to produce a placed and routed design. A fast, software-like compilation flow for FPGAs is the goal we pursue in this paper.

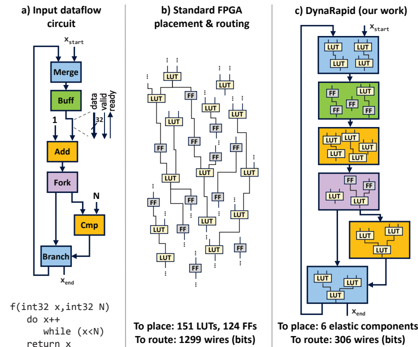

Fig. 1: HLS, Elastic Circuits, and FPGAs. (a) Some HLS tools compile C code into an elastic circuit made of handshaken components out of a finite set. (b) In a regular FPGA flow, one would synthesize the circuit into a fine-grained netlist of bit-level FPGA primitives that need to be placed-and-routed through a complex and time-consuming process. (c) DynaRapid leverages the peculiar structure of elastic circuits and preimplements every possible elastic component offline in isolation, prior to HLS compilation; during compilation, it only instantiates, stitches, places, and routes the appropriate word-level components for a faster and simpler implementation.

A. Exploiting Structure to Reduce Complexity

In recent years, a new approach to HLS has gained some popularity: instead of generating statically scheduled circuits, compilers produce *elastic* circuits [3], [4]. These circuits are composed of elastic (also called *dataflow*) components exchanging data through connections with handshake signals: there is no centralized controller and computation advances as soon as operands are available. One advantage of such HLS tools is that this technique may be better adapted to classic software applications and that less code refactoring may be needed to extract performance. But what makes this approach compelling for this work is that these circuits are created exclusively by interconnecting predefined components out of a limited set, without any glue logic whatsoever. Exploiting this high-level structure is the first ingredient of our recipe.

Practically any HLS tool eventually outputs its results in an RTL language. This needs to be passed through a logic synthesizer, technology-mapped onto lookup tables and other FPGA primitives, placed, and routed, before generating a configuration bitstream. This is an immensely complex process

1946-1488/24/$31.00 ©2024 IEEE
DOI 10.1109/FPL64840.2024.00014

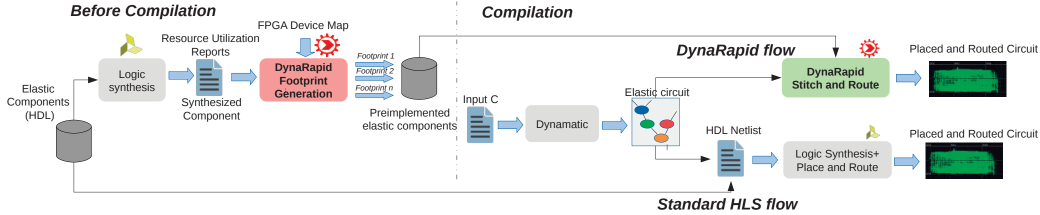

Fig. 2: DynaRapid. Our flow (top) contrasted to a typical flow (bottom). We exploit the modularity of elastic circuits produced by Dynamatic to build offline (hours) placed and routed components and, at compile time, we place and route coarser circuits with 1–2 orders of magnitude less components and only route intercomponent wires (seconds). The output of DynaRapid Footprint Generation (in pink) is a preimplemented set of all elastic components with a variety of footprints that can be relocated to many positions in the FPGA. The output of DynaRapid Stitch and Route (in green) is a fully placed-and-routed design checkpoint for AMD FPGAs.

where many NP-complete problems are solved. Obtaining the final result may take, as evoked before, from tens of minutes to hours. RapidWright [5] is an open-source Java framework that enables the direct manipulation of designs on AMD FPGAs and, in particular, makes it possible to relocate, compose, and stitch together preimplemented blocks on a specific device. Using RapidWright to exploit the modularity of elastic circuits is the second ingredient of our recipe.

Ultimately, the idea is to increase the granularity of the placed and routed elements (from LUTs and DSPs to elastic components), use a simple greedy placer for circuits now composed of 1–2 orders of magnitude less elements, and finally route only fewer intercomponent wires.

B. Dynamatic + RapidWright = DynaRapid

Figure 1a shows an elastic circuit implementing the functionality of the code snippet below it. The circuit is built out of a finite set of components: standard blocks that perform computation (e.g., the adder and the comparator in the figure), store data (e.g., a buffer), or steer data appropriately (e.g., the merge, that issues any one of its input values of `x` into the loop; the fork, that replicates `x++` to send it to the two successor units; the branch, that decides when to terminate the execution and issues the updated `x` to the appropriate successor). All units communicate through *elastic channels* with bidirectional handshake signals, indicating the validity of the data from the predecessor and the readiness of the successor to accept it. Several HLS approaches systematically and quickly produce such circuits from high-level code [3], [4], [6]–[8]; regardless of the exact generation strategy, they all employ a finite set of well-defined components and a consistent handshake protocol, just like the shown circuit.

Instead of producing and implementing the circuit using the ordinary FPGA toolflow as in Figure 1b, we want to preserve the elastic constructs and modularity: well-defined elastic components can be implemented *prior* to the compilation process; then, given a particular HLS-produced elastic circuit, these preimplemented components can be composed, stitched, placed, and routed to implement the application at hand. This makes the place-and-route process significantly simpler and faster, as Figure 1c suggests. Of course, such a complexity reduction and accompanying runtime savings do not come for free: as the elastic components are preimplemented, each of their logic functions is optimized in isolation and there is no longer an opportunity to perform powerful logic-synthesis optimizations across components. Qualitatively, the strategies of Figures 1b and 1c exhibit the very trade-off between compilation speed and optimization that one expects in software compilers between the compilation options `-O0` and `-O2`/`-O3`. This is exactly what we pursue.

In this work, we introduce DynaRapid, which leverages Dynamatic [9] to generate elastic circuits composed exclusively of components out of a library and combines it with the ability of RapidWright to relocate few placed and routed components and to perform only intercomponent routing. The rest of the paper is organized as follows: After an analysis of related work in Section II, we provide the necessary background on FPGA architecture and on RapidWright in Section III. In Section IV, we describe our process to preimplement offline our library of computational units that we will later place and route during compilation, following the strategy of Section V. We evaluate our approach in Section VI prior to concluding the paper.

II. Related Work

Several techniques aim to reduce the overall compilation time for FPGAs. Early approaches focused on algorithmic optimizations by tuning parameters to effectively trade off quality for improved runtime [10], [11]. Others attempted to minimize the total set of components to be placed by relying on macros [12] and macro prerouting [5], [13]. Yet, none of these approaches supports the full flow from C to placed-and-routed design because traditional HLS is ill-suited for these paradigms. More recent efforts by Xiao et al. [14] successfully leveraged modern HLS; they divided the FPGA into separately managed physical regions to allow independent logic to be mapped to the FPGA and accelerated the design process using partial reconfiguration [15]. Guo et al. [16] developed a parallel physical implementation of FPGA HLS-based designs called RapidStream; it mainly focuses on reducing the compilation runtime using a latency-insensitive

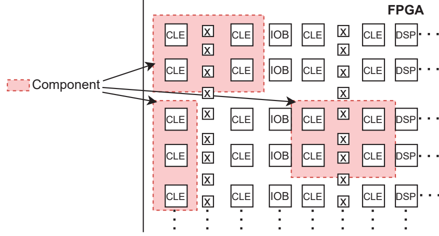

Fig. 3: Columnar FPGA Architecture. An FPGA is organized as a 2D array of various types of tiles, arranged in homogeneous columns of variable width. One of our *standard elastic components* is shown with two different *footprints* and, with the same footprint, in two different positions in the array.

approach to address the timing closure challenges of large designs.

DynaRapid and RapidStream goals are certainly aligned (reduce the time from high-level design to FPGA) and the methodology is similar (use RapidWright to stitch together different parts) but at different granularities: RapidStream stitches together a dozen of complex HLS-based user partitions, while we operate with hundreds of components that could be used to build the partitions themselves. RapidStream is about parallelism: split a design into relatively large partitions (each of the size of a significant FPGA portion) that can be placed and routed in parallel with a normal flow and stitched together later with RapidWright. In contrast, DynaRapid is about intrinsic structure: we take designs made of a limited set of small universal components, prepared once and for all in a library, and stitch them together with RapidWright profiting of the smaller number of components and nets to reduce runtime. Given such huge granularity differences and the different nature of the components, the problems we solve are in most cases completely different: for instance, our generation of various footprints for all our components (Section IV) and the placement problem (Section V-A) are totally unique to our work.

III. FPGA Architecture and RapidWright

Our goal is split into two tasks: One is the offline creation of an elastic component library, performed only once per device type and before invoking DynaRapid; it is a time-consuming batch job. The second task is the user design generation from C/C++ code and from the netlist of standard elastic components produced by our selected HLS tool, Dynamatic [9]. This is the task that matters to speed up. Figure 2 shows on the left the offline task, and on the right the online task.

Building the library relies on the regularity of FPGAs to create a limited number of preset implementations of all possible components that Dynamatic may instantiate. FPGAs typically have a fairly regular columnar architecture and AMD devices are no exception: they are constructed as a 2D array whose columns are made of identical *tiles* (e.g., LUTs, DSPs, memories) [17], as Figure 3 illustrates. Unfortunately, real devices have small irregularities in this array for a variety of reasons; we will discuss later how we account for this.

RapidWright [5] is a tool by AMD that allows the low-level manipulation of circuit implementations on AMD FPGAs. For the purpose of this work, it makes it possible to relocate preimplemented blocks to different but compatible areas of the device and route the connections between such preimplemented blocks to obtain a final working design. The blocks are previously synthesized, placed, and routed with the standard implementation flow (Vivado). The restricted areas used by Vivado to preimplement a design are called *Pblock*s. We limit ourselves to rectangular Pblocks made of a few rows of identical sequences of tiles. Figure 3 shows on the left an example of two implementations of the same component (requiring three logic resources CLEs, for example) in two different Pblocks (a 2×2 rectangle and a 1×3 one). Once we obtain an implementation of a component by restricting Vivado to a specific area (such as the 2×2 Pblock in the figure), RapidWright can relocate it into any other area composed of the same tiles, as suggested in the example. We will call different rectangular Pblocks for a component *footprints*.

IV. Library Generation

We need two qualitatively different explorations to build our library of preimplemented components: On one hand, we need to explore all possible different rectangular groups of tiles of an FPGA that can be used to map each of our components. On the other, we need to find all places where that particular Pblock can be relocated to or, in other words, identify all occurrences of the same rectangular group of tiles in the 2D array. Our main goal in the generation process is to obtain implementations that (i) have the smallest possible footprint, and (ii) can be placed in many locations on the FPGA.

An overview of our library generation flow is shown in the left part of Figure 2. The RTL description of each component used by Dynamatic is synthesized and technology-mapped independently; the resulting netlist of FPGA resources implementing a component is the input to the process. The final library contains, for every possible standard elastic component, a variety of possible rectangular areas occurring at least once on the target FPGA and where the component has been successfully preplaced and prerouted. As mentioned, we call these alternate implementations of a component *footprints*. Additionally, for each footprint of each component, the library contains a list of all possible valid placements of the preimplemented component. Naturally, producing all footprints and all possible relocation targets needs a detailed map of a particular target FPGA device; this can be obtained from RapidWright.

The rest of this section discusses the three main steps of the library creation: We first show how we generate all useful footprints for each component. We then discuss how to select the footprints to ensure that components can always be placed adjacent to each other without routing conflicts. Finally, we discuss our approach to expose the pins of each component in the footprint and thus ensure final routability.

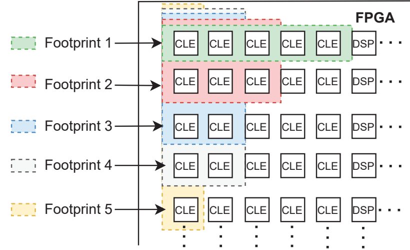

Fig. 4: Footprint Enumeration. For each component and from each starting tile in the FPGA array, we explore rectangular footprints of increasing height that are wide enough to have sufficient logic and routing resources to accommodate the fully routed component.

A. Footprints and Relocation

The search for all possible footprints is a relatively straightforward exhaustive enumeration of all possible rectangular areas of the target device containing at least the minimal resources needed to implement a component. To explore all possible positions and aspect ratios, the following process is repeated starting from any FPGA tile: For each height (i.e., number of rows) of the footprint and for each starting tile, a minimal width is identified by adding the minimum number of columns to the right of the starting tile so that the footprint contains at least the logic resources (CLEs, DSPs, etc.) needed for the component. Once this minimal rectangular tile area is identified, the component is placed and routed; if routing fails, the width is increased to bring in more resources, and the placement and routing are attempted again. Once routing succeeds, a new footprint for the component has been found and the process is repeated with one more row and, again, the minimum number of columns. Figure 4 illustrates this process.

Due to the FPGA's regularity, most of the explored candidate footprints, both failing and successful, end up identical. Thus, previous results are cached and, if the current footprint has already been encountered, the cached result is assumed and the repeated occurrence is recorded. While at the beginning and for starting tiles in the first row, most candidates require a full placement and routing run, soon thereafter all results can be found in the cache except for the occurrence of irregularities in the array, as those mentioned in Section III. The result is a set of footprints for each component and, for each footprint, a map of FPGA array positions where this footprint is valid.

B. Routing Resources

The footprints identified in the previous section have enough logic resources to implement our components. Yet, there is no guarantee that they have all the required routing resources inside the footprint: Switch boxes serve both the logic tiles to their right and left. Figure 3 illustrates this with the "X" marked elements corresponding to switch boxes. A switch box is implicitly included in the footprint when either its left or its right logic resource tile (CLE) is included in the footprint. Ignoring this aspect leads to invalid footprints, as illustrated in Figure 5(a): The footprint, shown as a purple rectangle, contains all the logic for the component; however, the adjacent switch boxes of its tiles are not explicitly included in the footprint even though a large portion of the routing requires them. This prevents the composition of such a footprint with others as routing resources used in an abutting component may overlap. Consider the example of Figure 5(b): the two elastic components have routing that spans beyond the footprints (green wires outside of the purple rectangles); when placed next to each other, the external wires overlap and the design is invalid (yellow lines show the conflicting routing resources).

To address this issue, we developed a heuristic to reject candidate footprints with adjacent routing resources that are not included in the footprint; when this happens, they are expanded to the next tile to implicitly include them. The result is in Figure 5(c): a different footprint for the component of Figure 5(b) uses the same logic tile rectangle (2×4) but only in positions where the routing resources are implicitly included.

C. I/O Pins

Our implementation of the elastic components is performed out of context which causes its I/O pins to be left unconnected. Simply placing and routing such a component may result in these floating I/O pins being placed deep inside the footprint where they may be unroutable when connected to other components. To prevent this issue, we devised a postprocessing technique to ensure routability of the component I/Os to the surrounding area. The idea is to connect the component I/Os to temporary external *flip-flops* (FFs) which imitate potential connections to other components, thus ensuring that inter-component connections are feasible. After place and route, removing these temporary FFs then leaves the I/O pins exposed at the edge of the footprint. This procedure is illustrated by an example in Figure 6. Note that the example is a very simple one: actual elastic components could have many more I/Os; for instance, a 32-bit fork with one input and six outputs has $(1 + 6) \cdot (32 + 2) = 238$ I/Os.

D. The Library

The end result of the library generation process is a large device-specific collection of preimplemented instances of all components Dynamatic might instantiate; they are generated for maximum speed and often achieve near-spec maximum operating frequency due to their small size and complexity.

To get a sense of the quality of our library, we evaluated the following metrics: *Density*, computed as the ratio between the logic resources needed and the total logic resources included in the footprint. *Relocatability*, indicating the number of possible locations in the FPGA that can host the given footprint. For a 2-input 32-bit adder, for example, eight possible footprints exist, ranging from 16% to 85% in density. Relocatability is larger than 60K on the target FPGA hosting 1728K LUTs.

To validate the correctness of our footprint construction, we tested our strategy as illustrated in Figure 7: We took some of our most complex components, placed them as close

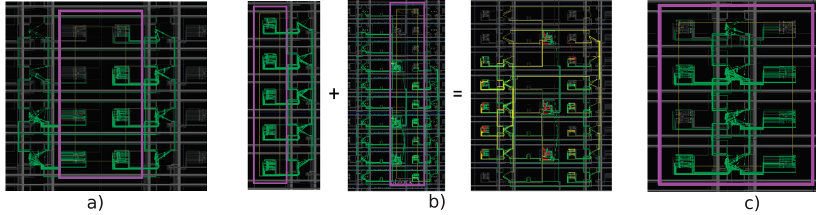

Fig. 5: Footprints with Valid Routing. (a) The purple box delimits the footprint and used routing resources (green diagonal lines) extend beyond it. (b) When routing resources of two components overlap, the implementation is invalid (yellow lines are disconnected wires). (c) The purple box delimits a footprint containing all wires and routing resources (green diagonal lines).

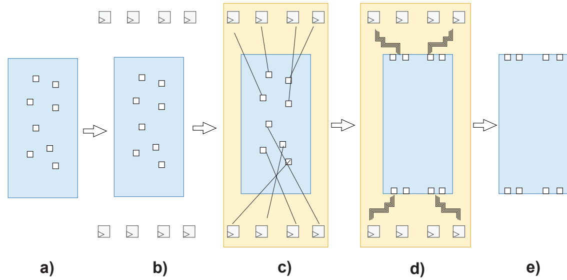

Fig. 6: Exposing I/O Pins. (a) I/Os are naturally scattered inside the footprint. (b) FFs are instantiated around the component. (c) FFs are connected to the component I/Os inside a larger area (Pblock). (d) The new design is routed within the larger Pblock. (e) The FFs and the larger Pblock are removed. The elastic component now has the I/Os exposed.

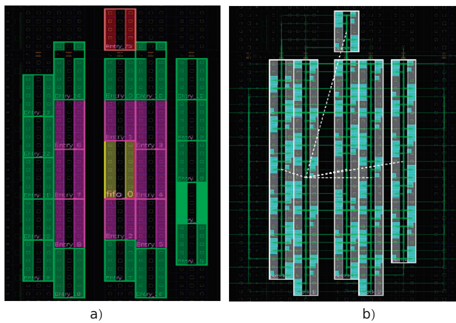

Fig. 7: Example of Routability Test. (a) Library components are placed on the FPGA. (b) The design is fully routed by Vivado with no routing errors.

as possible, and connected them randomly to each other. We successfully repeated this a few times for all components and thus gained confidence in the correctness of the process. The complete library generation process takes approximately 10 hours for all 74 Dynamatic 32-bit components on a Virtex UltraScale+ device (xcvu13p-fsga2577). Each component can have up to 50 different footprints, depending on the aspect ratio and the position on the FPGA array. The size of the complete library on disk is about 1 GB.

V. Compilation

In the previous section, we described our strategy to create a library of individually placed-and-routed elastic components; we here discuss how to compose them into complete circuits on an FPGA. While library generation is an offline, once-per-device task and is therefore not time-sensitive, online compilation is the time-critical task. Dynamatic takes the C/C++ code and produces within seconds a netlist of elastic components implementing the required functionality. Our goal here is to produce a physical implementation of the circuit on an FPGA using the preimplemented components in our library. As with any backend flow, there are essentially two key actions needed: decide on the placement of the components from the library and route them. We will detail these in the coming sections.

A. Placement

The main challenge is to create a good placement for the preimplemented components on the FPGA array. There exists a wealth of placement algorithms targeting different optimization objectives (e.g., wire length, critical path delay, or routability); metaheuristics, such as simulated annealing, are a classic option. Not unique to our flow, our placement problem is made peculiar by a number of characteristics: (1) We can select different footprints for the same component, with a variety of aspect ratios and densities. (2) These footprints can be placed in a multitude of physical locations, different for each component and for each footprint. (3) Speed in performing the placement is of the essence, even at a price in quality of results. Therefore, we have opted for a fairly straightforward greedy approach.

We treat the circuit as an undirected graph, and we process the components in the elastic circuit produced by Dynamatic in a fixed order: Firstly, we identify the elastic component in the netlist with the maximum number of connections to others. It will be the first one placed; the center of this component becomes the reference point. Secondly, we perform a breadth-first traversal of the elastic graph from this component and place the rest of the circuit in the order of the traversal.

Once we have determined the order of placement, we center our placement on a particular reference tile of the FPGA array (the center of the FPGA or the center of the area where we want our design placed) and proceed as follows for each

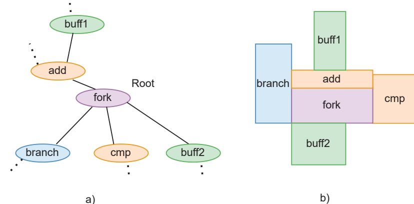

Fig. 8: Greedy Placement. (a) *fork* is the first unit placed because it has the largest number of connections to other units. (b) Components are placed in topological order starting from the *fork*, using the footprint that allows the closest placement to the centre and, in the case of ties, the tallest footprint.

component to place: (1) We start with a radius = 1. (2) We consider all footprints of the component to be placed and see if any can be placed such that its center is at a Manhattan distance from the reference smaller than or equal to the radius. (3) If none can, we increase the radius and go back to step 1. (4) If only one footprint is possible, we place the component and move to the next component, resetting the radius to one (step 1). (5) If multiple footprints are possible, we take one with the tallest aspect ratio, as this usually implies the best routability, or one of them at random, if they have the same aspect ratio. We continue with the next component (step 1).

Figure 8 shows a simple example of this greedy placement algorithm. A real placed design is shown in Figure 9 (top). In line with the goals of our placer, modules are placed adjacent to each other and with a preference for taller aspect ratios.

B. Routing

Individual components have been internally routed already, as discussed in Section IV; we now need to implement the inter-component connections specified in the netlist produced by Dynamatic. We base our component routing strategy on an improved variant of an existing router, RWRoute [18]. In contrast to vendor tools, RWRoute has a focus on runtime at the expense of solution quality—and this perfectly aligns with the goals of DynaRapid. To improve runtime even further, we use RWRoute in non-timing-driven mode. The variant of RWRoute that we employ (PartialRouter) attempts to preserve as much preexisting intra-component routing as possible, ripping up existing routing inside components only as a last resort. Experimentally, we have found this approach to be sufficient to reach full routability. Figure 9 (bottom) shows the layout of the final, fully placed-and-routed design.

VI. Evaluation

In this section, we evaluate the effectiveness of DynaRapid[1]. After discussing its implementation and our evaluation methodology, we compare the runtime and the resource usage of DynaRapid with that of the commercial FPGA backend when implementing the same elastic circuits. We break down

[1]DynaRapid is released as open-source and available for download at https://sites.google.com/view/dynarapid

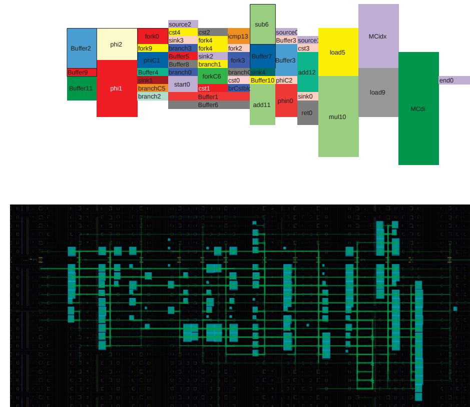

Fig. 9: Final Placed-and-Routed Design. The output generated by DynaRapid is a legal, fully routed design with no routing errors nor overlapping components.

the runtime into components to better understand the speedup sources and potential scalability weaknesses. We then compare the place-and-route quality of DynaRapid with that of the standard flow and contrast their placement densities. We conclude with some considerations on bitstream generation.

A. DynaRapid Implementation

As shown in Figure 2, DynaRapid is run in two phases: (1) library generation, described in Section IV and performed offline once per FPGA device, and (2) compilation, described in Section V and executed for every circuit. DynaRapid is implemented in Java and uses the RapidWright API for device-specific tasks such as FPGA map extraction, component loading, stitching, and relocation, as well as for custom manipulations of the design (e.g., FFs insertion, Section IV-C). DynaRapid uses Vivado™ for the standard placement and routing tasks during the library generation process and for the integrity checks of the footprints. The design input to DynaRapid is the intermediate elastic graph output by an unmodified version of the open-source Dynamatic [9]. DynaRapid has available all necessary components in the preimplemented library (Section IV) and assembles them into a circuit following the strategy described in Section V. At the end of the design steps presented in Section V, a fully placed-and-routed design checkpoint can be generated and imported into the standard Vivado toolchain for timing analysis, bitstream generation, and integration with other PE kernels or shells to form a fully-featured design checkpoint.

B. Methodology and Benchmarks

The experiments are performed on a server Intel(R) Xeon(R) CPU E5-2698 v4 clocked at 2.20 GHz and provided with 256 GB of RAM, using all available threads. We compare DynaRapid with a commercial FPGA flow, Vivado 2023.1, targeting a Virtex UltraScale+™ xcvu13p-fsga2577. We provide as input to DynaRapid the elastic circuit from Dynamatic in DOT format. For the baseline, we supply Vivado with

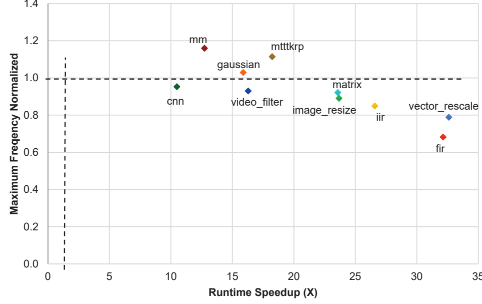

Fig. 10: Runtime Speedup vs. Performance (i.e., Frequency) Degradation of DynaRapid compared to Vivado.

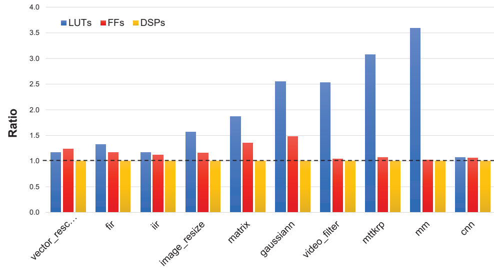

Fig. 11: DynaRapid Resource Overheads over Vivado.

the same elastic circuit description but translated into an RTL netlist of elastic components, as provided by Dynamatic. Since we target a fast implementation, we configure Vivado accordingly with the following runtime directives: For synthesis, we use the *Flow Runtime Optimized* strategies [19]. For placement and routing, we employ the option `-directive Quick`, offering the fastest non-timing-driven compile time to generate a legal design [20]. We compare the two generated placed and routed designs with Vivado using the standard commands `report_utilization` for resources and `report_timing` for the maximum frequency. We also use the command `report_route_status` to verify that all designs are fully routed without errors. Our benchmarks are typical HLS kernels from a recent work on dynamic scheduling [21], standard HLS suites [22], and PEs from RapidStream [16].

C. DynaRapid vs. a Commercial FPGA Backend

In this section, we compare DynaRapid with a commercial FPGA backend. Table I compares the implementation runtime and the circuit quality in terms of the operating frequency and resources (i.e., LUTs, FFs, and DSPs). We omit clock cycle counts as the implemented elastic circuits are identical: they execute in the same number of cycles and performance is thus proportional to the reported operating frequencies.

As indicated in Table I, DynaRapid completes the implementation, on average, 20× faster than the commercial flow. This is due to the fact that it leverages preimplemented implementations: the small number of coarser (word-level) components can be efficiently placed-and-routed on the FPGA fabric. In contrast, the commercial flow has to place and route netlists with at least an order of magnitude more (bit-level) components, and this at a significant runtime cost. Yet, its complete control on the circuit makes it possible to exploit the advantages of logic synthesis optimizations across components. DynaRapid, instead, synthesizes each component in isolation and cannot thus exploit this potential for optimization; this is reflected in a resource overhead of 1.8× on average. For the same reason, its routing flexibility is reduced and, thus, the achieved operating frequencies are lower; we will investigate this further in the following section.

D. Runtime Breakdown and Scalability

In this section, we analyze the dependence of DynaRapid's compilation time on benchmark complexity. The runtime breakdown for all benchmarks is reported in Table II. The time required for initialization, placement, and *stitching* (i.e., adding the connections between components) only slightly varies across benchmarks. As the number of components increases, the time to load them in slightly grows as well. Routing exhibits the largest runtime variations as the benchmark complexity increases (see cnn); yet, as discussed before, this is still significantly faster than the commercial flow for only a limited degradation in the final Fmax.

E. Place-and-Route Comparison

As mentioned in the previous section, Vivado can apply logic optimizations across elastic components, whereas DynaRapid can only optimize each component in isolation. It is thus reasonable to expect that the increased design size and complexity results in frequency degradation. In Table III, we compare the effectiveness of the two place and route processes, factoring out such logic optimizations across components.

To prevent Vivado from applying logic optimizations on the RTL version of the designs, we provide Vivado with the post synthesis gate-level netlists of DynaRapid's preimplemented components for place and route. Thus, the circuits placed and routed by the two flows are perfectly identical; the only difference is that DynaRapid will perform placement at a component level, whereas Vivado will place individual LUTs and FFs flattened across all components. Naturally, DynaRapid is still significantly faster.

F. Design Placement Density

In the previous section, we compared area consumption in terms of LUTs, FFs, and DSPs. This section quantifies the white space wasted among the used resources—that is, we measure *placement density* as the fraction of used resources within the rectangular bounding box around the compiled design. Figure 12 shows a series of placement density experiments: Firstly, we let DynaRapid do its placement and routing and observe how many discrete areas (i.e., *clock regions*) of the FPGA are used (Figure 12a). Then, we have Vivado perform its task using `-directive Quick` in the same FPGA region (Figure 12b); Vivado achieves a similar density as

TABLE I: DynaRapid vs. Vivado. The values are plotted in Figures 10 and 11.

Benchmarks	Components	Runtime (s)			Fmax (MHz)			LUTs			FFs			DSPs	
		Dyna-Rapid	Vivado	Speedup	Dyna-Rapid	Vivado	Ratio	Dyna-Rapid	Vivado	Ratio	Dyna-Rapid	Vivado	Ratio	Dyna-Rapid	Vivado
vector_rescale	50	15	489	33	140	179	0.79	626	534	1.17	687	552	1.24	3	3
fir	65	15	482	32	235	345	0.68	845	637	1.33	953	813	1.17	3	3
iir	91	19	505	27	173	204	0.85	1306	1113	1.17	1838	1644	1.12	6	6
image_resize	113	21	497	24	137	154	0.89	2376	1339	1.56	1521	1318	1.15	0	0
matrix	167	33	524	16	147	143	1.02	3912	2090	1.87	3164	2341	1.35	3	3
gaussian	178	21	495	24	138	149	0.92	3706	1455	2.55	2669	1810	1.47	3	3
video_filter	186	32	521	16	143	154	0.93	5985	2368	2.53	2969	2847	1.04	9	9
mm	319	40	509	13	72	63	1.16	14487	4038	3.59	3222	3138	1.03	3	3
mttkrp	201	26	474	18	95	85	1.11	6833	2226	3.07	1964	1832	1.07	6	6
cnn	790	50	524	10	152	160	0.95	10144	9449	1.07	12406	11731	1.06	18	18
GEOMEAN		**25**	**502**	**20×**	**138**	**149**	**0.9**			**1.82**			**1.16**		

TABLE II: Runtime Breakdown. All values are in seconds.

Benchmark	Init.	Placement	Loading	Stitching	Routing
vector_rescale	4.1	0.02	4.4	0.5	6
fir	4.2	0.02	4.0	1.2	6
iir	5.1	0.04	3.6	0.7	10
image_resize	5.6	0.05	6.0	0.8	9
matrix	6.1	0.06	6.3	1.1	20
gaussian	6.3	0.04	6.4	0.9	12
video_filter	7.5	0.37	6.3	1.1	17
mm	3.8	0.37	8.2	4.6	23
mttkrp	4.9	0.05	6.8	2.6	15
cnn	8.3	0.18	7.4	1.4	32

TABLE III: Comparison of the runtime and quality of DynaRapid with Vivado when given the identical netlist.

Benchmark	Runtime (s)			Fmax (MHz)		
	Dyna-Rapid	Vivado *(post-syn)*	Speedup	Dyna-Rapid	Vivado *(post-syn)*	Ratio
vector_rescale	15	420	28	141	167	0.85
fir	15	412	27	235	294	0.80
iir	19	420	22	173	217	0.80
image_resize	21	447	22	137	147	0.93
matrix	33	451	14	147	143	1.03
gaussian	21	435	21	138	145	0.95
video_filter	32	457	14	143	143	1.00
mm	40	499	12	72	91	0.79
mttkrp	26	478	18	95	100	0.95
cnn	50	457	9	152	196	0.78
GEOMEAN	**25**	**447**	**18×**			**0.9**

DynaRapid (around 30%). It is possible to constrain Vivado's design into a smaller region; for this, we manually reduced the region obtaining the smallest routable placement. Vivado thus achieves a density up to 65% in about 1.5× (685 seconds) the time of the loosely constrained implementation (Figure 12c) with 141MHz Fmax. Finally, we bounded our placer to the boundaries of the area used by Vivado for the smallest routable implementation; DynaRapid succeeds in matching the same density (Figure 12d), with only 1.1× the time (35 seconds) and 136MHz Fmax.

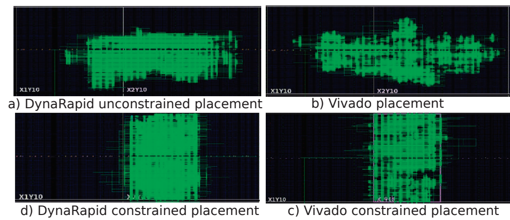

Fig. 12: Visualization of Vivado and DynaRapid placement density for the video_filter benchmark. (a) DynaRapid placement. (b) Vivado placement in the same region. (c) Vivado's manually obtained minimum footprint. (d) DynaRapid placement constrained to Vivado's minimum footprint.

G. Bitstream Generation

We successfully generated bitstreams from DynaRapid using Vivado, demonstrating that the implementations are not only valid but also passed necessary design rule checks (DRCs) for safe programming onto hardware. Although DRC and bitstream generation runtime are not the focus of this work, the FIR benchmark consumes 156 seconds in `write_bitstream` for a full device bitstream. Yet, considering the runtime advances shown by DynaRapid, we believe there exists a motivation for vendors to make DRC and bitstream improvements that will lead to much more performant solutions in the future.

VII. Conclusions

In this paper, we presented DynaRapid, a very fast compilation tool that enables the generation of fully placed-and-routed circuits in a matter of seconds. Our work leverages the modularity of recent HLS approaches that generate elastic circuits from C code to preimplement all standard elastic components so that they can be readily used across all applications. DynaRapid generates a fully placed-and-routed design in at most a few tens of seconds, bringing the compilation process to the same order of magnitude that software developers expect while compiling software code. We think that something along the lines of DynaRapid is essential to make FPGAs more appealing to software programmers.

References

[1] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–91, Apr. 2011.

[2] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 2, pp. 24:1–24:27, Sep. 2013.

[3] S. A. Edwards, R. Townsend, and M. A. Kim, "Compositional dataflow circuits," in *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*, Vienna, Sep. 2017, pp. 175–84.

[4] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in *Proceedings of the 26th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, Calif., Feb. 2018, pp. 127–36.

[5] C. Lavin and A. Kaviani, "RapidWright: Enabling custom crafted implementations for FPGAs," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines*, Sep. 2018, pp. 133–140.

[6] M. Budiu, P. V. Artigas, and S. C. Goldstein, "Dataflow: A complement to superscalar," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, Austin, Tex., Mar. 2005, pp. 177–86.

[7] A. Elakhras, A. Guerrieri, L. Josipović, and P. Ienne, "Unleashing parallelism in elastic circuits with faster token delivery," in *Proceedings of the 22nd Intl. Conference on Field-Programmable Logic and Applications*, Belfast, UK, Aug. 2022, pp. 253–61.

[8] R. Li, L. Berkley, Y. Yang, and R. Manohar, "Fluid: An asynchronous high-level synthesis tool for complex program structures," in *Proceedings of the 27th International Symposium on Asynchronous Circuits and Systems*, Beijing, China., Sep. 2021, pp. 1–8.

[9] L. Josipović, A. Guerrieri, and P. Ienne, "Dynamatic: From C/C++ to dynamically scheduled circuits," in *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Seaside, Calif., Feb. 2020, pp. 1–10.

[10] Y. Sankar and J. Rose, "Trading quality for compile time: Ultra-fast placement for FPGAs," in *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, Monterey, Calif., Feb. 1999, pp. 157–166.

[11] C. Mulpuri and S. Hauck, "Runtime and quality tradeoffs in FPGA placement and routing," in *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*, Monterey, California, USA, Feb. 2001, pp. 29–36.

[12] R. Tessier, "Fast placement approaches for FPGAs," *ACM Transaction on Design Automation of Electronic Systems*, vol. 7, no. 2, pp. 284–305, Apr. 2002.

[13] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "HMFlow: Accelerating FPGA compilation with hard macros for rapid prototyping," in *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, Salt Lake City, Utah, May 2011, pp. 117–124.

[14] Y. Xiao, E. Micallef, A. Butt, M. Hofmann, M. Alston, M. Goldsmith, A. Merczynski-Hait, and A. DeHon, "PLD: Fast FPGA compilation to make reconfigurable acceleration compatible with modern incremental refinement software development," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 933–45.

[15] Y. Xiao, D. Park, Z. J. Niu, A. Hota, and A. DeHon, "ExHiPR: Extended high-level partial reconfiguration for fast incremental FPGA compilation," *ACM Trans. Reconfigurable Technol. Syst.*, Sep. 2023.

[16] L. Guo, P. Maidee, Y. Zhou, C. Lavin, J. Wang, Y. Chi, W. Qiao, A. Kaviani, Z. Zhang, and J. Cong, "RapidStream: Parallel physical implementation of FPGA HLS designs," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Virtual Event, USA, Feb. 2022, pp. 1–12.

[17] *Xilinx Architecture Terminology*, AMD Inc., 2023. [Online]. Available: https://www.rapidwright.io/docs/Xilinx_Architecture.html

[18] Y. Zhou, P. Maidee, C. Lavin, A. Kaviani, and D. Stroobandt, "RWRoute: An open-source timing-driven router for commercial FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 1, Nov. 2021.

[19] *Vivado Design Suite User Guide: Synthesis*, AMD Inc., 2023. [Online]. Available: https://docs.xilinx.com/r/en-US/ug901-vivado-synthesis/Vivado-Preconfigured-Strategies

[20] *Vivado Design Suite*, Xilinx Inc., 2023. [Online]. Available: http://www.xilinx.com/products/design-tools/vivado.html

[21] L. Josipović, S. Sheikhha, A. Guerrieri, P. Ienne, and J. Cortadella, "Buffer placement and sizing for high-performance dataflow circuits," in *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Seaside, Calif., Feb. 2020, pp. 186–96.

[22] L.-N. Pouchet, *Polybench: The polyhedral benchmark suite*, 2012. [Online]. Available: http://www.cs.ucla.edu/pouchet/software/polybench