FADO: Floorplan-Aware Directive Optimization Based on Synthesis and Analytical Models for High-Level Synthesis Designs on Multi-Die FPGAs

LINFENG DU, The Hong Kong University of Science and Technology, Kowloon, Hong Kong
TINGYUAN LIANG, The Hong Kong University of Science and Technology, Kowloon, Hong Kong
XIAOFENG ZHOU, The Hong Kong University of Science and Technology, Kowloon, Hong Kong
JINMING GE, The Hong Kong University of Science and Technology, Kowloon, Hong Kong
SHANGKUN LI, Fudan University, Shanghai, China
SHARAD SINHA, Indian Institute of Technology Goa, Goa, India
JIERU ZHAO, Shanghai Jiao Tong University, Shanghai, China
ZHIYAO XIE, The Hong Kong University of Science and Technology, Kowloon, Hong Kong
WEI ZHANG, The Hong Kong University of Science and Technology, Kowloon, Hong Kong

Multi-die FPGAs are widely adopted for large-scale accelerators, but optimizing high-level synthesis designs on these FPGAs faces two challenges. First, the delay caused by die-crossing nets creates an NP-hard floorplanning problem. Second, traditional directive optimization cannot consider resource constraints on each die or the timing issue incurred by the die-crossings. Furthermore, the high algorithmic complexity and the large scale lead to extended runtime for legalizing the floorplan of HLS designs under different directive configurations.

To co-optimize the directives and floorplan of HLS designs on multi-die FPGAs, we formulate the co-search based on bin-packing variants and present two iterative optimization flows. The first (FADO 1.0) relies on a pre-built QoR library. It involves a greedy, latency-bottleneck-guided directive search and an incremental floorplan legalization. Compared with a global floorplanning solution, it takes 693X~4925X shorter search time and achieves 1.16X~8.78X better design performance, measured in workload execution time.

To remove the time-consuming QoR library generation, the second flow (FADO 2.0) integrates an analytical QoR model and redesigns the directive search to accelerate convergence. Through experiments on mixed dataflow and non-dataflow designs, compared with 1.0, FADO 2.0 further yields a 1.40X better design performance on average after implementation on the Alveo U250 FPGA.

CCS Concepts: • **Hardware** → **Reconfigurable logic and FPGAs**; **High-level and register-transfer level synthesis**; **Partitioning and floorplanning**; *3D integrated circuits*; *Software tools for EDA*; • **Computer systems organization** → **High-level language architectures**; **Data flow architectures**.

Authors' addresses: Linfeng Du, linfeng.du@connect.ust.hk, The Hong Kong University of Science and Technology, Kowloon, Hong Kong; Tingyuan Liang, tliang@connect.ust.hk, The Hong Kong University of Science and Technology, Kowloon, Hong Kong; Xiaofeng Zhou, xzhoubu@connect.ust.hk, The Hong Kong University of Science and Technology, Kowloon, Hong Kong; Jinming Ge, jgeab@connect.ust.hk, The Hong Kong University of Science and Technology, Kowloon, Hong Kong; Shangkun Li, skli20@fudan.edu.cn, Fudan University, Shanghai, China; Sharad Sinha, sharad@iitgoa.ac.in, Indian Institute of Technology Goa, Goa, India; Jieru Zhao, zhao-jieru@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China; Zhiyao Xie, eezhiyao@ust.hk, The Hong Kong University of Science and Technology, Kowloon, Hong Kong; Wei Zhang, eeweiz@ust.hk, The Hong Kong University of Science and Technology, Kowloon, Hong Kong.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1936-7406/2024/3-ART
https://doi.org/10.1145/3653458

Just Accepted

Additional Key Words and Phrases: High-Level Synthesis, Analytical Model, Design Space Exploration, Multi-Die FPGA, Directive Optimization, Floorplanning

1 INTRODUCTION

Guided by optimization directives, high-level synthesis (HLS) compiles high-level behavioral specifications to register-transfer level (RTL) structures, supporting hardware accelerators' ever-growing functional and structural complexity. The various directives contribute to a large design space to search upon. For example, there are 26 directives in Xilinx Vitis HLS 2020.2 [14], each with a set of parameters and can be applied at different levels or structures of the HLS source code. Previous works [25, 34, 35, 38, 39, 42, 46, 47, 52, 55] mainly use automated design space exploration (DSE) algorithms to search for the Pareto-optimal directive configurations, targeting the lowest latency (execution time in the number of clock cycles) under a specific resource constraint.

To deploy large-scale HLS designs on FPGAs, with consideration of chip yield in fabrication, larger FPGAs with multiple dies emerge based on 2.5D/3D integration techniques. However, the concomitant long net delay due to nets crossing die-boundaries harms the timing quality of the implemented designs. One of the multi-die packaging technologies is the Stacked Silicon Interconnect (SSI) from Xilinx [37], where a silicon interposer integrates multiple dies, also called super logic regions (SLRs). [4] states that the super long lines (SLLs) between dies cause ~1 ns delay, while [32] states that a typical medium-length routing wire within a single die has a 4X~8X shorter delay under the same manufacturing process.

Furthermore, [9] highlights a crucial factor limiting the maximum achievable frequency (Fmax) to ~300 MHz when applying optimizations, including floorplanning and pipelining on HLS dataflow designs—a handshake-based model for task-level parallelism. This limitation stems from the delay of SLLs and long routes detoured by specialized IP blocks close to the I/O banks. These findings underscore the untapped potential for timing improvement in multi-die FPGAs.

To mitigate the delay penalty on multi-die FPGAs, Xilinx proposes using the floorplanning method [19] to keep critical timing paths on a single SLR. However, the fine-grained gate/cell-level floorplanning is very time-consuming. In comparison, [9] requires that no function in the HLS dataflow region should spread over multiple SLRs and proposes a coarse-grained method to floorplan HLS functions at the SLR level to accommodate the large-scale dataflow designs and pipeline the wires crossing die-boundaries.

Min-cut floorplanning focuses on meeting the separate resource constraints on slots divided by SLR boundaries or I/O banks and the SLL number constraints between SLRs. However, an initial min-cut floorplan would not always support the latency-centric optimization of HLS directives. When a function's directive configuration changes, its resource also changes, and the original floorplan could be illegal because of resource over-utilization. For coordinating floorplanning with directive DSE, a straightforward solution is to call the global floorplanning repeatedly, e.g., using integer linear programming (ILP) solver [9], whenever a new directive is applied. This incurs an extended runtime of the DSE flow. Thus, we try to replace the global ILP solution with an incremental legalization algorithm to facilitate highly efficient integration of iterative directive search and floorplanning.

To address this challenge, we propose the automated floorplan-aware directive optimization, as Fig. 1 shows. On the one hand, we try to search toward the Pareto-optimal HLS directive configurations to minimize a design's latency under specific resource constraints. On the other hand, we legalize the floorplan after every change of resource utilization to maintain a high Fmax.

Fig. 2 shows our end-to-end solution—the FADO framework, with two versions of optimization flows. It co-optimizes HLS directives and floorplanning on multi-die FPGAs, thus benefiting both the latency and timing of HLS designs. We first formulate this complex co-optimization problem based on multi-choice multi-dimensional bin-packing (MMBP) [33], then develop a highly efficient synthesis-based iterative solution—the first flow, FADO 1.0. It involves a pre-processing stage where a Quality of Result (QoR) library is generated from the HLS synthesis report for each leaf function. During each iteration of the main greedy search, we evaluate the latency bottleneck

Just Accepted

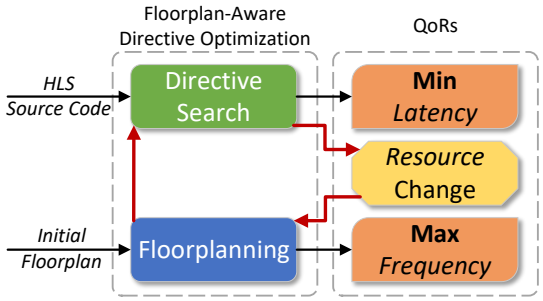


Fig. 1. The Main Idea of Floorplan-Aware Directive Optimization (FADO).

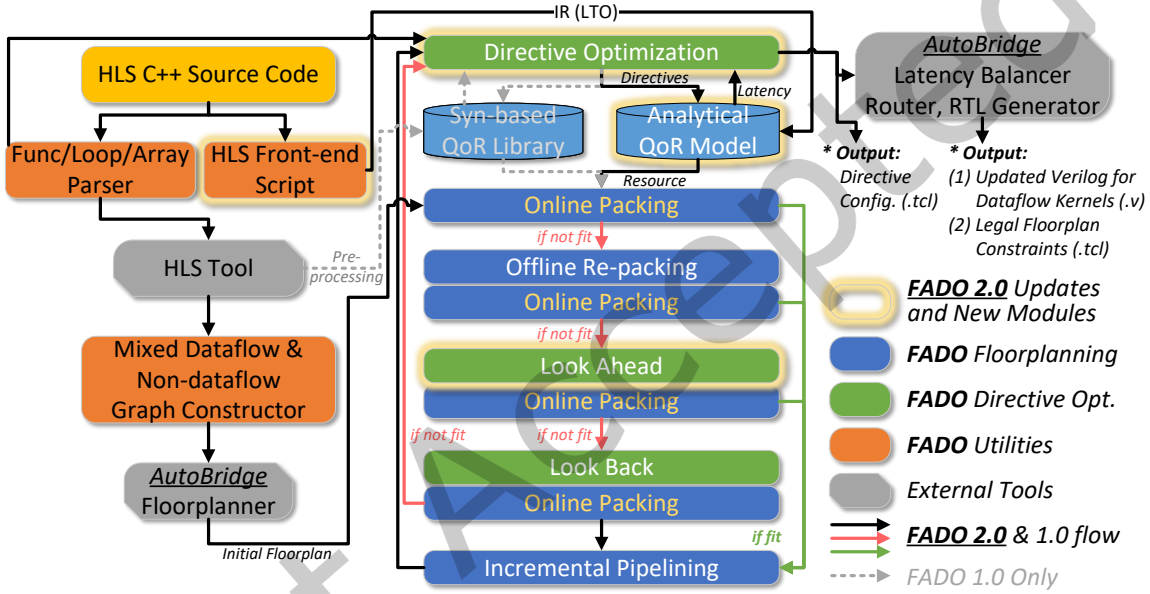


Fig. 2. Overview of Our FADO Framework, with New Modules in **FADO 2.0** Flow Highlighted.

from the QoR library and pick a more efficient directive configuration. Then, the resource utilization is passed to an incremental floorplan legalization. Such legalization applies both the worst-fit (WF) *online* bin-packing algorithm and the best-fit decreasing (BFD) *offline* algorithm [27]. The WF stage balances the resource utilization among slots on an FPGA to avoid congestion, while the BFD stage breaks the balance with minimum cost to enable the floorplanning of overlarge HLS functions. At the end of each iteration, FADO incrementally adds/updates/removes the pipeline logic along the long wires crossing die-boundaries. This incremental floorplan update is much faster than the global algorithms in previous works [9].

Experiments show that the synthesis-based FADO 1.0 improves the overall performance of designs deployed on multi-die FPGAs with co-optimization and achieves high speed with its customized iterative solution. It makes good utilization of the resources on FPGA under all constraints and proves to scale well to large accelerators with both dataflow and non-dataflow kernels.

Behind the high performance of FADO 1.0, the synthesis-based QoR library becomes the major limitation. Due to the long synthesis time of commercial HLS tools, it usually takes several hours to pre-process hundreds of

Just Accepted

design points. This timescale can cover all the individual directives, including unroll, pipeline, etc., for every branch in a loop tree. However, it should be more effective when considering (1) the interplay between multiple directives and (2) the combinations among different branches in a loop tree. As the size of the design space increases exponentially, the solution in FADO 1.0 is to sample among the reasonable design points following the descriptions in [14]. A key parameter of this sampling—the number of steps, is defined based on the range of parameters for the directives, including loop pipeline and unroll. This simple solution occasionally prunes the effective configurations, leading to sub-optimal results. Furthermore, the sampling leads to a random selection among BRAM, URAM, and LUTRAM without considering the balance in between, which limits the full utilization of on-chip resources.

To overcome the limitations above, we present an enhanced flow—**FADO 2.0** in this extension paper. It achieves better design performance by supporting a larger design space with a new analytical QoR model and developing a more intelligent search strategy accordingly. The new modules are highlighted in Fig. 2. Through a non-trivial development and calibration, we replace the QoR library and pre-processing stage with an analytical QoR model built based on COMBA [55]. The model takes in an intermediate representation (IR) and a directive configuration for an HLS function. Then, it outputs the estimated latency for guiding the selection of the next design point and the resource utilization to constrain the floorplanning. Accordingly, a more intelligent directive optimization strategy is modeled and designed, helping FADO 2.0 converge faster toward lower latency than the 1.0 flow. Moreover, the new DSE strategy balances the utilization among BRAM, URAM, and LUTRAM, contributing to a higher achievable frequency for most designs evaluated.

After the implementation of a set of optimized designs on the AMD Alveo U250 FPGA, HLS designs optimized by FADO 2.0 are observed to yield a better execution performance of 1.40X on average (not including one extreme case of 19.83X at most) compared with FADO 1.0. Compared to the new experimental results of analytical DSE with global floorplanning, the incremental methods help FADO 2.0 deliver an average of 2.66X better design performance. The latest version of FADO open source containing both 1.0 and 2.0 flow is maintained in https://github.com/RipperJ/FADO.

The core contributions in FADO 1.0 and 2.0 are as summarized below:

- To the best of our knowledge, we propose the first precise mathematical formulation of this directive-floorplan co-optimization problem on multi-die FPGAs. Accordingly, we design and implement the FADO framework, the first end-to-end solution to this problem.
- FADO supports two types of iterative, incremental flows: a synthesis-based solution (1.0) and an analytical model-based one (2.0). They improve both latency and timing of complex HLS designs within a very short runtime, achieving orders-of-magnitude speedup over global algorithms in prior works and better final design quality.
- Compared with [9], our FADO framework can automatically handle dataflow benchmarks and large-scale applications mixing dataflow and non-dataflow kernels.
- Compared with the best baseline using synthesis-based QoR library—directive search with the global ILP floorplanning and pipelining [9], FADO 1.0 achieves 693X~4925X speedup in exploration time with an even higher and near-optimal final design performance on FPGA for all six tested designs. The design performance measured with overall workload execution time on each design improves 1.16X~8.78X.
- In the second flow, FADO 2.0, we develop and calibrate a new analytical QoR model in place of the synthesis-based QoR library in FADO 1.0. It enables a much larger design space by exempting the time-consuming pre-processing stage.
- We implement a more intelligent directive search strategy that converges faster and better than the previous method relying on the QoR library. The new balancing mechanism among different types of on-chip RAMs also opens new optimization opportunities for timing quality.

Just Accepted

Table 1. Comparisons between FADO 2.0, FADO 1.0 and Previous Work

	Directive Search	Multi-die Floorplanning	Floorplan-aware Directive DSE (FADO 1.0)	Floorplan-aware Analytical Directive DSE (FADO 2.0)
QoR	latency, resource	timing (frequency)	latency, resource, timing	latency, resource, timing
Design Space	1. directives & parameters	2. SLR-level func location	1×2	**enlarged** $\mathbf{1 \times 2}$
DSE Efficiency	syn-based: slow; model[a]-based: fast	slow (SA, ILP, bi-partition, ...)	syn-based slow pre-proc, fast DSE	**analytical model**[b], fast DSE
Type of Benchmark	dataflow or non-dataflow	dataflow	mixed dataflow & non-dataflow	mixed dataflow & non-dataflow

[a] Generally with fixed data type and bitwidth. [b] Taking arbitrary data type and bitwidth as inputs.

- Through extensive experimentation, FADO 2.0 surpasses the analytical DSE with global floorplanning by an average of 2.66X better design performance. Meanwhile, it outperforms FADO 1.0 with a 1.40X better average performance of the optimized designs.

The subsequent sections of this paper are structured as follows. Section 2 summarizes the difference between FADO and previous works on directive search or multi-die timing optimization and introduces bin-packing basics. It also highlights the necessity and challenges of switching to an analytical model in FADO 2.0. Section 3 provides a motivation example to interpret the challenges and opportunities in directive-floorplan co-search. Then, Section 4 provides the ILP formulation of the co-search problem. Section 5 presents implementation details for each module in the framework, with emphasis on the newly introduced IR extraction (Sec. 5.1.3), analytical QoR model (Sec. 5.2.2), smarter DSE strategy (Sec. 5.3), etc. Section 6 analyzes the superiority of FADO 2.0 and 1.0 by comparing them with previous solutions on various HLS designs.

2 RELATED WORK

As shown in Table 1, FADO performs co-optimization considering latency, resource, and timing during floorplanning, while previous commonly used flows, either directive search or multi-die floorplanning, only targets one or two optimization objectives. The design space for multi-objective optimization is enormous, defined by the Cartesian product of HLS directive parameters and SLR-level function locations. Previous works may take a long time to traverse such a large design space. We can break down the long runtime into two parts. First, in directive search, synthesis-based methods are slow at either pre-processing a database/library or iteratively triggering the HLS tools. Second, to pursue high floorplan quality, the time complexity of mainstream solutions is also exceedingly high.

Accordingly, FADO tackles the QoR and runtime issues in two steps. First, with a one-off QoR library generation, our effective incremental floorplanning assists FADO 1.0 in achieving orders-of-magnitude speedup in search time. Moreover, existing floorplanning works for HLS designs [2, 9, 23] are dedicated to dataflow applications. FADO can automatically solve the floorplanning for non-dataflow functions by adding additional constraints. As the second step, to make the co-optimization flow more powerful and user-friendly, FADO 2.0 includes an analytical QoR model, which facilitates a smarter traversal throughout a broader design space and replaces the pre-processing stage for QoR library taking hours in the flow 1.0.

HLS Directive Optimization has been researched thoroughly. Below are the featured challenges.

- *Directives and their parameters contribute to an enormous design space [43].*

Just Accepted

FADO 1.0: We collect the QoRs from the HLS reports of individual leaf functions in large designs to construct a library. Then, it assists a latency-bottleneck-guided algorithm to minimize the overall latency and speed up the DSE effectively.

FADO 2.0: Pre-processing is exempted by implementing an analytical model. The inference time falls between milliseconds and seconds, greatly expediting the traversal across a more extensive design space.

- *Directive configurations have non-monotonic effects [43] on QoRs, which is also explained by [53] as inter-dependency among directives and structures in the source code.*

FADO 1.0: We define the look-ahead/back sampling strategies based on the QoR library to avoid getting trapped in local optima on the non-monotonic design space.

FADO 2.0: We formulate the interplay between the most effective directives. This analysis lays the foundation for the new DSE strategy without global QoR knowledge.

The general techniques [40] for HLS directive DSE includes (1) synthesis-based approaches with (1.1) meta heuristics [38, 39, 51] or (1.2) dedicated heuristics [34, 35, 43, 53], and (2) model-based approaches including (2.1) machine learning [31, 46, 54] and (2.2) graph analysis [20, 42, 47, 52, 55]. Prior works on directive search mainly optimize latency under an overall resource constraint of single-die FPGAs. To compare, multi-die FPGAs introduce separate constraints on each slot and between SLRs, as well as the vital timing issue because of long wires crossing die boundaries. Hence, we cannot directly apply the previous DSE algorithms to our co-optimization problem.

The directive search in FADO 1.0 partially belongs to (1.2) synthesis-based dedicated heuristics, which is generally accurate but incurs a long pre-processing time when triggering commercial HLS tools for building the QoR library. This overhead also limits the optimization opportunity to some extent. In this case, switching to model-based approaches is a direct solution because of the short inference time. However, machine learning models are easily trapped with accuracy and generality issues if they lack a comprehensive training dataset, even for the latest GNN models [8, 48], especially when the control and data flow of HLS designs varies. The GPU memory would severely limit the holistic mapping of a big CDFG to GNN or other models. Based on our attempts with some of the latest models, including GAT [44], GraphSage [11], etc., the GPU A100 40GB can only work for designs with no more than ~10K LUTs, under a typical setting of parameters including the number of layers, batch size, etc. This is far from predicting QoRs for the designs implemented on modern multi-die FPGAs, taking AMD Alveo U250 as an example, an FPGA with ~1.7M LUTs.

By contrast, analytical models are free from the scalability problem. They try to extract or recover the core flow in commercial HLS tools. The main difficulty of adopting an existing analytical model is the need for calibration due to the variance among different versions of commercial tools, e.g., Vivado HLS 2016.1 in [55], Vivado HLS 2019.1 in [52], and Vitis HLS 2020.2 in FADO 1.0 [7]. Besides, previous models generally consider only a specific data type, for example, a 32-bit integer or floating point [52, 55], while our work involves the use of arbitrary precision integers [15], float and double. All difficulties above call for migrating, calibrating, and generalizing an existing analytical model towards our case. This motivates the development of FADO 2.0.

Multi-die Timing Optimization can be classified by their objectives as Table 2 shows.

To mitigate the long delay crossing dies, existing solutions differ in their objectives, algorithms, and granularity. The fine-grained method maps primitives/cells to the tile/site level, which is time-consuming. [6] proposes optimization on the total wirelength and aspect ratio of face-to-face-stacked floorplans. [10] and [32] extend the P&R tool VPR [3] to multi-die scenario by adding parameters to the cost function including wire-cut ratio, delay increment, and number of cuts, while [32] also considers the congestion cost. [36] implements a partition-driven placer and an aspect-ratio-aware cut scheduling algorithm.

Among the coarse-grained works, [28] minimizes the total wirelength while reducing total and die-crossing delay. [9] applies ILP to minimize the number of die-crossing long wires. It runs iterative bi-partitioning rather than N-way partitioning and prefers the most balanced floorplan across all slots divided by die-boundaries and

Just Accepted

Table 2. Previous Works on Multi-die Timing Optimizations

Work	Objectives					Algorithm/Tool	Granularity
	Total Wirelength	Signal Delay	# of Cut	Routing Congestion	Aspect Ratio[a]		
[6]	✓				✓	Partition-based	fine
[28]	✓	✓				Force-directed, SA	coarse
[10, 32]		✓	✓		✓	*VPR*	fine
[36]			✓	✓	✓	Partition-based	fine
[2]				✓		ILP (balancing)	coarse (HLS)
[9]			✓			ILP (min-cut)	coarse (HLS)

[a] The aspect ratio of the bounding box of a design's floorplan or placement.

Just Accepted

I/O banks. [23] is an extension of [9] supporting an additional type of dataflow channel—the RAM-based buffer channel aside the original FIFO channel. [2] also applies ILP and resource balancing heuristics to partition dataflow accelerators and average congestion among different SLRs. It also discusses partitioning dataflow accelerators among multiple FPGAs through network interfaces. Although these coarse-grained flows reduce the problem size, it is still slow if called repetitively when integrated with directive search.

In FADO 1.0 and 2.0, we mainly compare with [9] on the efficiency of coarse-grained floorplanning for large-scale designs. We identify that the major frequency improvement in [9] comes from the insertion of pipelining logic, while min-cut floorplanning works as a legalization for logic resources and die-crossings. Thus, we replace the time-consuming min-cut ILP floorplanning with an incremental legalization algorithm, alongside a partial pipelining update to speed up the DSE without sacrificing floorplanning quality.

Knapsack [30] and **Bin-Packing Problems** [29] are a series of classic combinatorial optimization problems sharing a common ground with the directive-floorplan co-search. The basic version is the 0-1 Knapsack problem, where multiple items with different weights and values are to be packed in a knapsack, and a binary choice is made for packing the item or not. [22, 41] introduces the multiple-choice Knapsack problem, where the items are classified, and precisely one from each class is chosen to form a solution. The classes here map to the directive configurations for an HLS function in our problem. [24] introduces the multiple-dimensional Knapsack problem, where the weight of each item and the capacity of knapsacks are in vectors. This corresponds with the types and amounts of resources on each die of a multi-die FPGA. [1, 33] separately formulates the multi-choice multi-dimensional Knapsack problem (MMKP) and bin-packing problem (MMBP). The optimal solution to this problem can be found using branch-and-bound with linear programming, but the high time complexity does not support a large number of variables and equations. Another approximation is using greedy approaches, generally sorting items based on the values and the weights in a specific order. We thus formulate our problem based on the MMBP and combine online Worst Fit (WF) and offline Best-Fit Decreasing (BFD) [27] bin-packing heuristics to solve it efficiently. Here, in the online algorithm, the decision to pack an item is irreversible, and the next item is only visible after the previous packing gets settled. For offline algorithms, the values and weights of all items are visible from the very beginning, and we can sort the items to improve the packing quality [21].

3 MOTIVATION

To show the different challenges of the directive-floorplan co-search problem on multi-die FPGAs, we use a toy example with one step of floorplanning followed by directive optimization to explain why general floorplanning algorithms and heuristics will not collaborate with directive search.

Suppose we have a toy multi-die FPGA with two slots, each with a constraint of 70% of the total available resource, as suggested by [2, 9]. Although there are several different resources on a modern FPGA, such as Look-up Tables (LUT), Flip-Flops (FF), Digital Signal Processors (DSP), Block-RAMs (BRAM), UltraRAMs [13] (URAM), etc., in our experiments, we normalize and take the maximum among all resources as the final utilization ratio in evaluation.

Fig. 3 shows a design consisting of 2 dataflow and 1 non-dataflow kernel. A/B (or D/E) are functions connected by the FIFO channels in the same dataflow kernel (region), defined as an HLS function with the directive "DATAFLOW," achieving task-level parallelism within the function through a handshake-based model. C is a non-dataflow function connected through RAM to other kernels. The channel width between A and B is 16, and that between D and E is 8.

For latency optimization, different QoRs are caused by various directive configurations in HLS. For example, when applying a smaller initiation interval (II) to the directive *PIPELINE*, or a larger factor to the *UNROLL*, the latency of an HLS design tends to decrease, while the resource utilization is likely to increase. In this example, assume that we only have one directive—two configurations for each function independently, either with or without that directive. When the directive is applied to a function, its resource consumption increases and latency decreases, as shown in Fig. 3.

For timing optimization, a design's frequency can be ensured at a high level by pipelining as long as the following floorplan conditions are satisfied. The first is having a legal floorplan meeting the resource constraint on every single slot. Second, only FIFO channel connections are allowed to cross the slot boundaries (within a limit on total width not reflected in this toy example) because the handshake interface of FIFO is easy to pipeline. In contrast, the complex RAM interface cannot be pipelined. Thus, functions connected through RAM should be grouped and floorplanned on the same slot, while functions connected through FIFO channels can be partitioned on different slots.

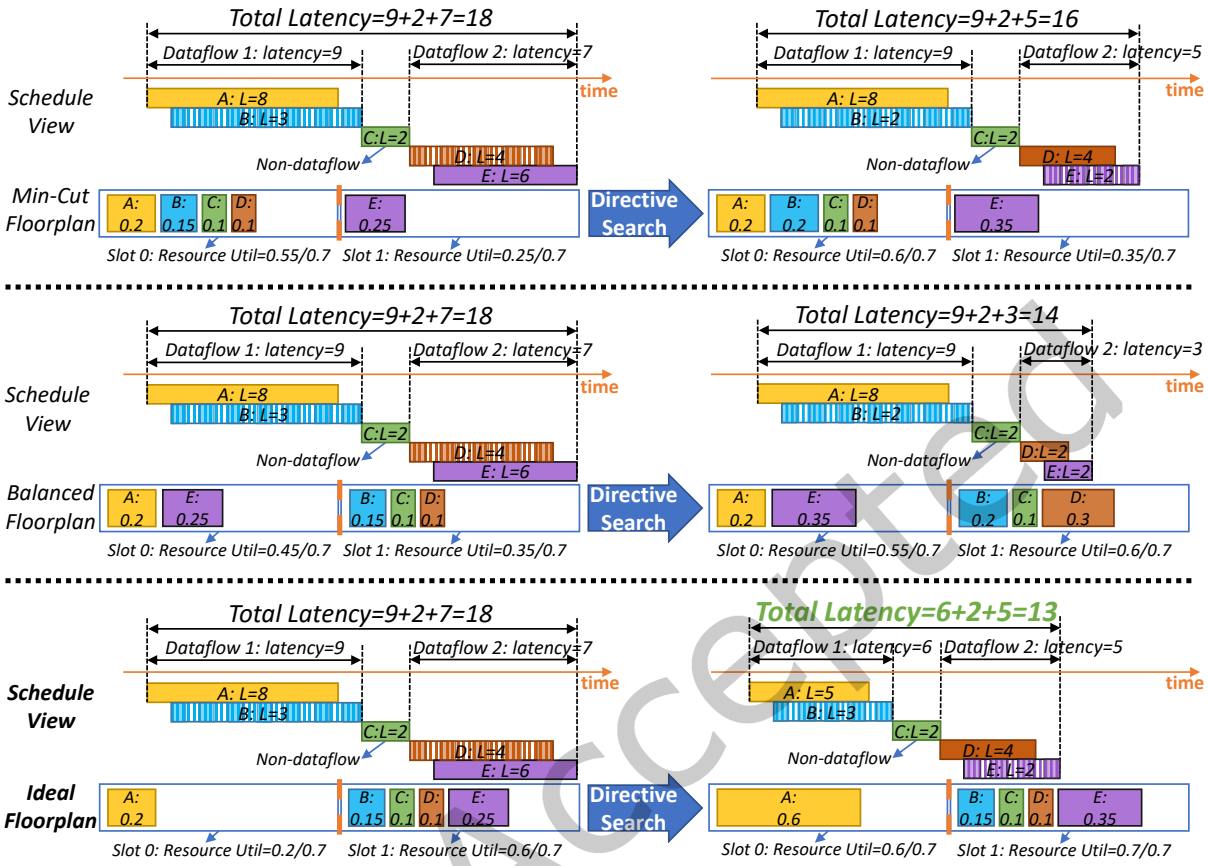
During directive DSE, we minimize the total latency of the design while ensuring the two floorplan conditions above. Suppose that every function has no directive applied at the beginning of DSE. We first find an initial floorplan using a specific algorithm and then try to improve some functions by applying their respective directive, subject to the resource constraint on each slot.

Fig. 3 shows the three floorplanning objectives compared in this toy example. The first minimizes the width of the FIFO channel crossing two slots (min-cut), as used in [9]. Since all functions cannot be packed in one single slot, the solution to the min-cut problem is 8, which is the channel width between D and E. Thus, function E should be assigned to the other slot. In this case, functions B and E can still fit into their slots when applying directives, and the final total latency is improved to 16.

The second floorplan refers to the resource balancing heuristics [2]. Since functions B, C, and D are grouped during floorplanning, as the largest one, they are initially floorplanned onto the other slot. After DSE, directives are applied for B, D, and E, and the total latency is improved to 14.

To compare, an ideal floorplan for this design is partitioning between A and B, and the best point improves total latency from 18 to 13 clock cycles, which is the minimum achievable latency for this case. If we have an ideal floorplan at the very beginning, it is natural for DSE to reach the optimal latency without any effort to change the floorplan. However, if we start with the other two floorplans, neither the min-cut nor the balancing algorithm can further improve the achieved latency. In our FADO framework, the incremental floorplanning algorithm smartly re-packs the functions from either of the two prior floorplans and always reaches the ideal

Just Accepted



HLS Function		Without Directive		With Directive	
		Resource	Latency	Resource	Latency
A		0.2	8	0.6	5
B		0.15	3	0.2	2
C		0.1	2	0.3	1
D		0.1	4	0.3	2
E		0.25	6	0.35	2

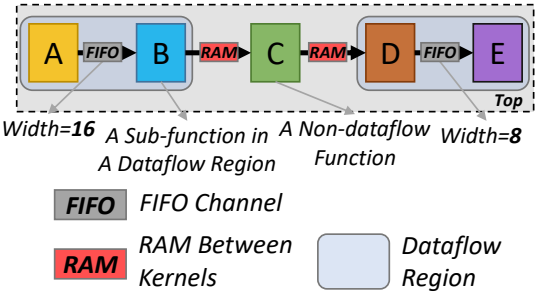


Fig. 3. A Toy Example with 2 Dataflow Kernels and 1 Non-dataflow, with Different Latency (L) and Resource Consumption (R). Three Different Floorplanning Methods and the Corresponding DSE Results Are Compared.

solution finally. Specifically, if FADO starts with the min-cut solution, A is identified as the latency bottleneck and has the top priority to apply the directive. When online packing finds no legal floorplan for A under the min-cut floorplan, the offline re-packing stage groups B, C, D, and E. Thus, the ideal floorplan is found, and the directive for A is successfully applied. It is a similar workflow if we start with the balanced floorplan. In Sec. 6, the control experiments on real benchmarks all start with a min-cut floorplan, to fairly compare the effectiveness of FADO with ILP floorplanning assisted DSE.

Just Accepted

From this example, we want to show that previous floorplanning techniques could fail to assist directive optimization on multi-die FPGAs. On the contrary, an improper floorplan could prune the high-performance points in a design space. That is why we propose floorplan-aware directive optimization, the FADO framework.

4 PROBLEM FORMULATION

Based on the multi-choice, multi-dimensional bin-packing problem, we formulate the directive-floorplan co-search problem on multi-die FPGAs. To accurately describe the problem and show the complexity compared with floorplanning in [9], we also present a ILP formulation below. Note that ILP is only used for a description of the problem. In the implementation (Sec. 5), we use approximation heuristics for the objective and each constraint instead of repetitively calling the ILP solver.

4.1 Symbol Definition

Table 3 shows the definition of all the constants and variables used in our formulation.

Table 3. Symbols Used in Problem Formulation

Symbols	Definition
m, n	The number of dataflow and non-dataflow kernels along the longest-latency path in an HLS design.
W, H	Width/Height of an FPGA (by number of slots).
R_{kt}	The total amount of resource t on slot k.
B_h	Total number of SLLs on a die boundary.
L_{ij}	The latency of the j-th function in the i-th kernel along the longest-latency path. $i \in \{1, 2, ..., m+n\}$, $j \in \{1, 2, ..., C_i\}$. $C_i = 1, \forall i \geq m+1$.
z_{ijp}	Suppose that function j or kernel i has Q_{ij} directive choices in total. $z_{ijp} = 1$ when directive choice p is applied to function j of kernel i. $p \in \{1, 2, ..., Q_{ij}\}$.
x_{ijk}	$x_{ijk} = 1$ when the sub-function j of kernel i is floorplanned to slot $k \in \{1, 2, ..., S\}$ among S slots.
r_{ijpt}	r_{ijpt} represents the consumption of resource type t of function j in kernel i, when directive choice p is applied. $t \in BRAM, DSP, FF, LUT, URAM$.
$e_{i1,j1,i2,j2,RAM}$ $e_{i1,j1,i2,j2,FIFO}$	$e_{i1,j1,i2,j2,*} \in \mathbb{N}$. It's positive when there exists a RAM or FIFO connection from function $j1$ in kernel $i1$, to $j2$ in kernel $i2$, with a width of $e_{i1,j1,i2,j2,*}$. $i1, i2 \in \{1, 2, ..., m+n\}$, $j1 \in \{1, 2, ..., C_{i1}\}$, $j2 \in \{1, 2, ..., C_{i2}\}$.
$b_{h(k,ks,kd)}$	$b_h \in \{0, 1\}$, it equals 1 when a source function on slot ks goes through the die boundary indexed with k and connects to a destination function on slot kd. $k = (k_x, k_y), k_x \in \{0, 1, ..., W-1\}, k_y \in \{0, 1, ..., H-2\}$.

4.2 ILP Formulation

The objective function in our problem minimizes the total latency of an HLS design. To show the generality of our problem, assume that we have a large accelerator containing multiple dataflow and non-dataflow kernels

Just Accepted

connected by RAMs. Minimizing the total latency is equivalent to minimizing the sum of latency of kernels L_i along the longest-latency path, as Eq. 1 shows.

$$\text{minimize} \sum_{i=1}^{N} L_i \tag{1}$$

Generally, a dataflow kernel's total latency is very close to the longest sub-function $\max_j L_{ij}$ in it, meanwhile relatively much smaller latency comes from the depth of dataflow—the cycle number it takes between each piece of data appearing at the beginning of the dataflow pipeline and at the end. We here approximate the total latency of a dataflow kernel i with the latency of the longest sub-function. Hence, for the longest-latency path with m dataflow kernels and n non-dataflow kernels, the objective function can be re-written as:

$$\text{minimize} \sum_{i=1}^{m} \max_j L_{ij} + \sum_{i=m+1}^{m+n} L_i \tag{2}$$

where i iterates on kernels, and j on sub-functions. There is no sub-function in non-dataflow kernels indexed from $m+1$ to $m+n$.

Table 4. Directives in the Design Space of FADO

Directives	Parameters
PIPELINE	Initiation Interval (II) (<int>: $\{MinII, ..., \min(4 \times MinII, IterLat)\}$)
UNROLL	Factor (<int>: $\{1, 2, 4, ..., LoopBound\}$)
ARRAY_PARTITION	Type (Block/Cyclic/Complete) Factor (<int>: in correspondence with the UNROLL factor) Dimension (<int>)
BIND_STORAGE	Implementation (BRAM/URAM)

4.2.1 Constraint 1: multi-choice packing problem. During the directive search, we have multiple choices of directives and parameters for HLS functions, loops, and arrays. The directive design space of FADO is shown in Table 4. For *PIPELINE*, the lower bound of II, $MinII$, is determined by recurrence and resource analysis, and it is also revealed by the HLS report when applying the minimum value possible for target II, i.e., 1. The upper bound considers the iteration latency and four times the $MinII$. For *UNROLL*, the applicable value for its factor ranges from 1 to the loop bound. For *ARRAY_PARTITION*, we consider the three types of partitioning schemes and the dimension of an array. For *BIND_STORAGE*, an array is implemented using either BRAM or URAM.

Every time we trigger the HLS, one group of directives and the corresponding QoR are applied for each function (including the loops and arrays within it), described as the constraint in Eq. 3.

$$\sum_{p=1}^{Q_{ij}} z_{ijp} = 1, z_{ijp} \in \{0, 1\} \tag{3}$$

4.2.2 Constraint 2: multiple bins. Multi-die FPGA is partitioned into several slots during floorplanning by the die boundaries and I/O banks. Eq. 4 guarantees that there's no duplicated or missing floorplan for each HLS function.

$$\sum_{k=1}^{S} x_{ijk} = 1, x_{ijk} \in \{0, 1\} \tag{4}$$

Just Accepted

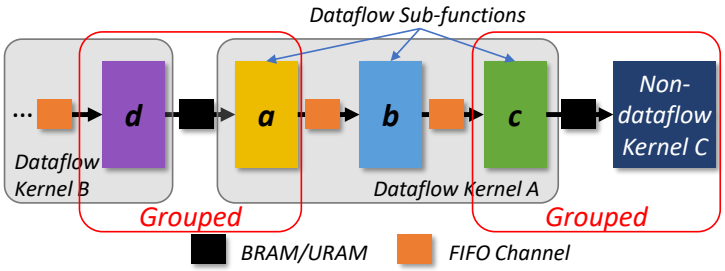


Fig. 4. Grouped Floorplan for RAM-Interfaced Functions.

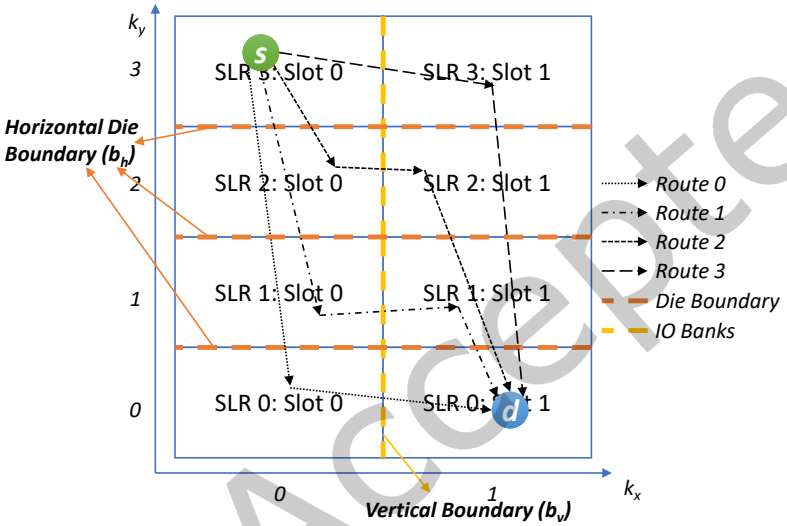


Fig. 5. An Example about Routes between Two Functions Placed on Separate Slots of Alveo U250 FPGA.

4.2.3 Constraint 3: multi-dimensional packing problem. In our problem, a single dimension of resource constraints corresponds with one type of resource on the multi-die FPGAs. The following constraint in Eq. 5 assures that functions on each slot with specific directive configurations respectively will not cause overflow in any type of resource.

$$\sum_{i=1}^{m+n} \sum_{j=1}^{C_i} x_{ijk} z_{ijp} r_{ijpt} \leq R_{kt} \tag{5}$$

4.2.4 Constraint 4: grouping the RAM-connected functions. Another particular type of constraint is introduced by the RAM connection between kernels, as shown in Fig. 4. Since the interface between RAMs and functions is not the handshake model and is difficult to pipeline, the RAM-connected functions are grouped and assigned to the same slot during floorplanning. As Equation 6 states, $e_{i1,j1,i2,j2,RAM} x_{i1j1k1} x_{i2j2k2} = 0$ guarantees that either there is no RAM connection between two functions, or they are not separated on two different slots.

$$\begin{aligned} &e_{i1,j1,i2,j2,RAM} x_{i1j1k1} x_{i2j2k2} = 0, (i1 \neq i2 \vee j1 \neq j2) \wedge k1 \neq k2 \\ &e_{i1,j1,i2,j2,RAM} \in \mathbb{N}_+, i1 \neq i2 \vee j1 \neq j2 \end{aligned} \tag{6}$$

4.2.5 Constraint 5: Limited number of SLLs. As Fig. 5 shows, Alveo U250 FPGA is vertically partitioned into two parts by the I/O banks and horizontally partitioned by die-boundaries into four parts. Suppose we have a source

Just Accepted

function "s" and a destination function "d" placed on SLR3:Slot0 and SLR0:Slot1, respectively. No matter which route between "s" and "d" is chosen, it crosses three horizontal die boundaries. For each boundary with vertical index k_y, the route passes through either the left half or the right half of it. The corresponding constraint is:

$$\sum_{k_x=0}^{W-1} x_{i1j1k1} x_{i2j2k2} b_{h(k,k1,k2)} \operatorname{sgn}(e_{i1,j1,i2,j2,FIFO}) = 1 \tag{7}$$

Since the number of SLLs is limited between two dies, we have the formulation in Eq. 8.

$$\sum_{i1,j1,i2,j2,k1,k2} x_{i1j1k1} x_{i2j2k2} b_{h(k,k1,k2)} e_{i1,j1,i2,j2,FIFO} \leq \beta B_h, \tag{8}$$

where β is the upper limit of SLL utilization. It is set to 90% in our implementation.

5 IMPLEMENTATION

To solve the ILP formulation above, we develop the FADO framework (Fig. 2), which can be split into the following parts.

Sec. 5.1 introduces the input generation for the core co-search of FADO, featuring the newly added HLS front-end script in this extension paper. Next, as the foundation of DSE, the pre-processing for synthesis-based QoR library in FADO 1.0 and the analytical QoR model in FADO 2.0 are introduced in Sec. 5.2. Then, we describe the objective and multi-choice constraint (Eq. 3) in the DSE algorithm (Sec. 5.3), highlighting the redesigned directive search strategy in FADO 2.0. Other constraints are included in the floorplanning algorithms (Sec. 5.4). Then, the optimized floorplan after each iteration will be passed to the incremental pipelining module as elaborated in Sec. 5.5. Last but not least, the interaction between FADO and external tools is explained in Sec. 5.6.

5.1 FADO Initialization

5.1.1 Common initialization in 1.0 and 2.0: As Fig. 2 shows, at the very beginning, the input to FADO is an HLS design without any directive of PIPELINE, UNROLL, ARRAY_PARTITION, and BIND_STORAGE. We parse the source code in "Func/Loop/Array Parser" to generate labels and hierarchy for nested loops. The labeled code is synthesized by an external HLS tool. Then, the initial HLS report is analyzed by a graph constructor, where connections through FIFOs and RAMs are identified. Next, the graph is passed to the min-cut floorplanner of AutoBridge [9] to generate an initial legal floorplan, which is passed on to FADO core flow for directive-floorplan co-optimization.

5.1.2 FADO 1.0: The results from "Func/Loop/Array Parser" help identify HLS functions of the same template. Then, we sample a group of efficient design points for each template, which means that unreasonable design points are excluded, such as mismatched factors for loop unrolling and array partitioning, and that the interplay between directives is included, e.g., outer loop pipelining infers automatically and fully unrolling the inner loops. These design points are then added into the QoR library *QoR_lib* during pre-processing.

5.1.3 FADO 2.0: HLS Front-end script. During the initialization stage, IRs are generated for each HLS function and will be directly reused later in the main search process together with a model and various HLS directives to infer QoRs. The analytical models in previous works [52, 55] only consider limited data types or bitwidths—32-bit integer or floating point. By contrast, FADO additionally considers 64-bit floating point and arbitrary precision integers, "ap_[u]int<W>" from the Vitis HLS library. This is achieved by a sequence of *clang*, *clang-tidy*, *llvm-link*, and *opt* instructions extracted from the "autopilot.flow.log" of Vitis HLS's front end. They can run in a stand-alone mode to get the link-time optimized (LTO) IR within milliseconds. We notice that the bitwidth of some IR instructions in the LTO stage could be different from that of IRs in later stages or the corresponding registers/signals in the generated Verilog. However, we cannot run till later stages because the HLS tool doesn't

Just Accepted

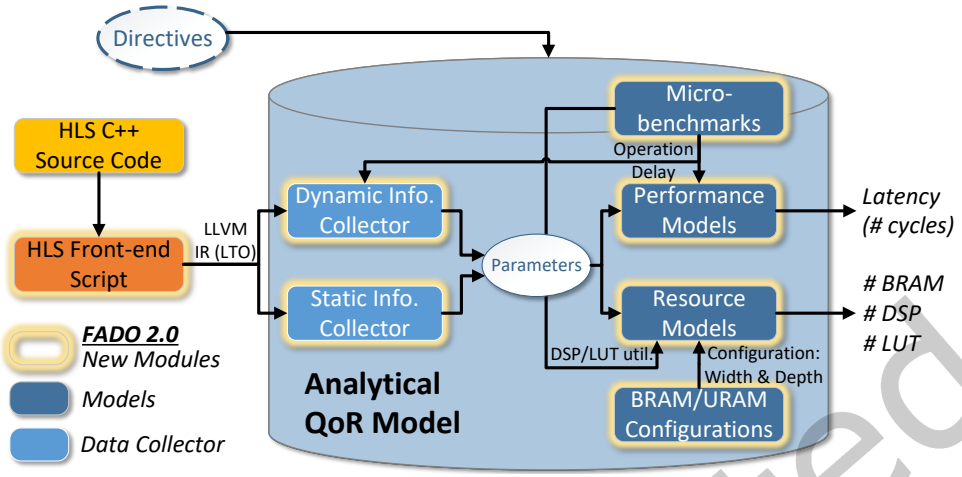


Fig. 6. The Updated Components in FADO 2.0's Analytical QoR Model.

support standalone execution of the middle/back end. Besides, the original intention of designing this model is to avoid the long synthesis time. Accordingly, we adopted some additional LLVM passes after the LTO pass sequence to reveal the correct bitwidth at front end. For example, a pair of "trunc" and "zext|sext" instructions can sometimes hide the real bitwidth. We accordingly adopt the "instcombine" pass to recover the correct bitwidth.

5.2 Pre-processing: Library vs. Model

5.2.1 Synthesis-based QoR Library. In a large-scale HLS design, many functions could be based on the same template. When running an HLS over the entire design, excessively long synthesis time is wasted on repeatedly analyzing functions from the same template. Besides, it is hard for existing graph analysis or machine learning methods to accurately predict the QoR of an HLS design with various coding styles, complicated control and data flow, and flexible compilation optimization.

Considering the small size of each function template, it only takes a short period to sample the most effective directive choices of them and build a function-level QoR library based on the HLS synthesis report to facilitate the whole workflow. To be specific, in FADO 1.0, we classify the functions by either the template of C++ generics or custom naming rules, e.g., all functions whose names match the regular expression *r"funcA_[0-9]_[0-9]"* will map to "*funcA*" in the QoR library. To apply directives to the C++ template, we follow [16]. As for custom regex, it is straightforward since they have distinct names. Accordingly, we only need to look up QoRs in the library during DSE instead of repeatedly running the HLS flow for an entire design.

5.2.2 Analytical QoR Model. In FADO 1.0, we sample the effective directives to speed up building QoR libraries to minutes or hours. This is where the completeness of the design space gets sacrificed to some extent, and this way of generating the QoR library is poor at generalization towards new designs. To completely avoid the pre-processing overhead and to generalize our framework, in FADO 2.0 flow, we build an analytical model based on COMBA [55]. Now, let us break down the model building into several steps and zoom in on the "Analytical QoR Model" part, as Fig. 6 shows.

First, re-build the micro-benchmarks. Due to the process advancement of FPGAs, the operation-level delay and resource model in HLS tools also change. For all the different types of operations in LLVM IR, we focus on those directly related to the on-chip resources, including most of the binary, logical, memory/addressing, and

Just Accepted

Algorithm 1: The Top-level of FADO Framework

Input: QoR_lib, $analytical\_model$, N: look-ahead step number, $flow$: search mode
Output: Optimal Directives, Legal Floorplan

while *True* **do**
  $(i_1, j_1) = \text{argmax}^1_{i,j}(L_{ij})$; // the longest functions
  $L_{(i_2,j_2)} = \max^2_{i,j}(L_{ij})$; // the 2nd-longest latency
  if $flow$ == *"FADO 1.0"* **then**
    $DS1$ = `Prune`$(QoR\_lib, (i_{max,1}, j_{max,1}), L_{i_{max,2},j_{max,2}})$;
    $DP = DS1[-1]$, break **if** DP is $None$; // Constraint Eq. 3
  if $flow$ == *"FADO 2.0"* **then**
    $DS2$ = `find_next_DP`$(analytical\_model, (i_{max,1}, j_{max,1}), L_{i_{max,2},j_{max,2}})$;
    $DP = DS2[-1]$, break **if** DP is $None$; // Constraint Eq. 3
    DP = `balance_BRAM_URAM_LUTRAM`$((i_1, j_1), DP)$;
  $fp\_success$ = `incremental_floorplan`$((i_1, j_1), DP)$;
  `exit_condition_check` $(fp\_success)$;

cast operations. For each of these operators, we build micro HLS benchmarks for each of the commonly used data types (int, float, double, etc.) and bitwidths (i16, i128, float32, double64, etc.), covering all the possible cases in the benchmarks to get tested. Then, HLS and logic synthesis for hundreds of these benchmarks can be done in parallel within several minutes, even for the largest bitwidth covered in our case, 512 bits.

Second, revise the data collectors. In COMBA [55], some configurations for arrays, loops, and HLS functions are manually computed and hard-coded into the C++ source code to adapt to specific HLS designs. In order to adapt this model to FADO's flow, the first step is to upgrade the IR parser in the "Static Info. Collector" to automatically extract the configurations from IR, such as parsing the metadata according to Vitis HLS's IR format and extracting the correspondence between loop labels and headers. As for the "Dynamic Info. Collector", we calibrate the operation chaining rules based on the pre-characterized operation delays in the first step. The delay of each operation is extracted from the verbose HLS scheduling report of the micro-benchmarks, where each operation in the finite-state machine has a predetermined delay value by the HLS tool.

Third, calibrate the resource and latency model. For BRAM and URAM, we update the configuration choices (width and depth) according to [18] to derive the correct utilization given the bitwidth and number of elements in an array. With the ground truth of DSP and LUT utilization from all the micro-benchmarks and some revision on the resource-sharing mechanism, we accurately predict DSP and LUT for most of our test cases. As for FF, since it is sufficient on FPGA [55] and seldom becomes the bottleneck during our experiments, we remove it from the floorplan constraint in FADO 2.0. Meanwhile, the latency (cycle) estimation is based on the delay of each operation determined in the second step. Accordingly, we calibrate the performance model for directives in Table 4 to improve latency estimation. The quality of latency estimation is analyzed in Sec. 6.4.2.

5.3 Directive Optimization

The top-level floorplan-aware directive optimization is described in Alg. 1. In every iteration, based on either the QoR library or the analytical model, we identify the functions with the longest and second-longest latency among the whole HLS design, i.e., the sub-function j_1 of kernel i_1 is the bottleneck, represented by $(i_1, j_1) = \text{argmax}^1_{i,j}(L_{ij})$, and the second-longest function is (i_2, j_2).

In FADO 1.0, we apply the latency-bottleneck-guided search [25, 55] in `Prune()`, which extracts all design points of (i_1, j_1) from the QoR library with smaller latency compared with the second-longest function's $L_{(i_2,j_2)}$.

Just Accepted

Algorithm 2: New Directive Search Strategy in FADO 2.0

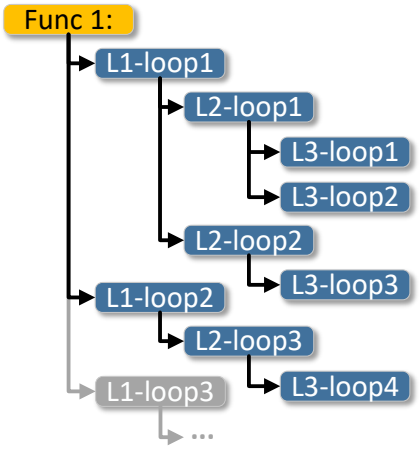
Input: $analytical\_model$, $(i_{max,1}, j_{max,1})$: the longest function, $L_{i_{max,2},j_{max,2}}$: second longest latency
Output: DS: all design point until DP as the last one to try floorplanning
// find_next_DP($analytical\_model$, $(i_{max,1}, j_{max,1})$, $L_{i_{max,2},j_{max,2}}$)
DS_until_DP = []; $single\_loop\_results$ = {};
// $single\_loop\_DS_{(i_{func},j_{func})}$: list[list[dict]]
// list: first level loops -> list: design points -> dict: directives and parameters
// $single\_loop\_index_{(i_{func},j_{func})}$: list[int], counting design points visited, initialized with 0
for $loop_i$ *in len*($single\_loop\_DS_{(i_{max,1},j_{max,1})}$) **do**
  if $single\_loop\_index_{(i_{max,1},j_{max,1})}[i] < len(single\_loop\_DS_{(i_{max,1},j_{max,1})}[loop_i])$ **then**
    $directives = single\_loop\_DS_{(i_{max,1},j_{max,1})}[loop_i][single\_loop\_index_{(i_{max,1},j_{max,1})}[i]]$;
    qor = run_analytical_model($directives$);
    $single\_loop\_results[loop_i] = (directives, qor)$;
if $single\_loop\_results$ **then**
  while *True* **do**
    $i, directives, qor$ = max($single\_loop\_results$, key = latency);
    $single\_loop\_index_{(i_{max,1},j_{max,1})}[i]$ += 1;
    if $qor[latency] < L_{i_{max,1},j_{max,1}}$ **then**
      append $directives$ to DS_until_DP;
    if $qor[latency] < L_{i_{max,2},j_{max,2}}$ **then**
      return(*sorted*(DS_until_DP)); // sorted by latency in decreasing order
    if $single\_loop\_index_{(i_{max,1},j_{max,1})}[i] < len(single\_loop\_DS_{(i_{max,1},j_{max,1})}[loop_i])$ **then**
      $directives = single\_loop\_DS_{(i_{max,1},j_{max,1})}[loop_i][single\_loop\_index_{(i_{max,1},j_{max,1})}[i]]$;
      qor = run_analytical_model($directives$);
      $single\_loop\_results[loop_i] = (directives, qor)$;
    else
      pop the $loop_i$ from $single\_loop\_results$;
Similar to the single-loop search above, search on $multi\_loop\_DS_{(i_{max,1},j_{max,1})}$;
return(DS_until_DP);

Just Accepted

These points form a next-step design space $DS1$ (a set of directive configurations and their respective QoRs). Although applying any of the configurations in $DS1$ to function (i_1, j_1) would make (i_2, j_2) the new latency bottleneck of the whole design, FADO will not make one-off latency improvements for bottleneck functions or choose the design point with the lowest resource utilization. On the one hand, aggressive latency improvement usually results in a dramatic increase in the resource utilization of current function(s), and potential latency improvement for future bottlenecks could be precluded because of a lack of resources. On the other hand, since resource utilization is calculated by taking the maximum utilization ratio among different resources, considering the non-monotonic design space, when utilization of one resource is minimized, others could still increase. Hence, we always assign the top priority to the design point DP, which has the largest latency among $DS1$, for further floorplan legalization. Note that functions having the same latency with (i_1, j_1) will be considered as a batch for efficiency.

In FADO 2.0, since we don't have the global knowledge of the design space from the QoR library, we need to redesign an effective search strategy, as Alg. 2 shows, to choose every next design point at each iteration. Given

that most HLS functions are made up of nested loop trees, we analyze all the loops within a function and derive two parts of the design space.



Order Index	Loops[a] whose directives are included
In $single\_loop\_DS_{(i_{func1},j_{func1})}$["**L1-loop1**"]	
0	L3-loop1, L3-loop2, L3-loop3
1	L2-loop1, L2-loop2
2	L1-loop1
In $single\_loop\_DS_{(i_{func1},j_{func1})}$["**L1-loop2**"]	
0	L3-loop4
1	L2-loop3
2	L1-loop2
In $single\_loop\_DS_{(i_{func1},j_{func1})}$["**L1-loop3**"]	
0	...

[a] Arrays accessed in the loops are also considered.

Fig. 7. An Example: The Level Order of Loops and Arrays Considered in $single\_loop\_DS_{(i_{func},j_{func})}$.

The first part is $single\_loop\_DS_{(i_{func},j_{func})}$. For each function, there is a list of first-level (outermost) loops on the loop tree. For each first-level loop, we build the list of directives by a reversed level-order traversal on its sub-loops, as Fig. 7 shows. For the list of directives indexed by loop L1-loop1, we start by checking the last-level (innermost) loops, L3-loop1, L3-loop2, and L3-loop3. After all configurations in the current level are finished, we move towards the second innermost level to check the directives of L2-loop1 and L2-loop2. Finally, we explore the directives of L1-loop1. This process repeats for every first-level loop. During the traversal, we take a record of the maximum loop bound of loops in each level, and compute the product from the last level to the first. If the cumulative loop bound exceeds a certain threshold (2048 in our experiments), the exploration of this subtree terminates, and we move on to the next first-level loop. This is to avoid unrolling the inner loops with an excessive factor.

Table 5. Exploration Order of Directive Configurations in Each Level within $single\_loop\_DS_{(i_{func},j_{func})}$

Order Index	Directive	Latency Model (Iter. Lat.=$iLat$, Tripcount=tc)	Parameter Order
1	Unroll[a] (factor = F)	$Lat_1 = \frac{iLat \cdot tc}{F}$	factor ↗
2	Pipeline	$Lat_2 = iLat + (tc - 1) \cdot ii$	II ↘
3	Unroll+Pipeline	$Lat_3 = iLat + (\frac{tc}{F} - 1) \cdot ii$	factor ↗, II ↘

[a] The parameters of array partition directives are set according to unroll and pipeline.

Given the traversal order on the loop tree, we consider the directive configurations for each level of loops in the descending order of latency, as Table 5 shows. This order is based on the assumption that ii is usually much smaller than the iteration latency $iLat$, and the tripcount tc is usually in the same order of magnitude with the unroll/partition factor F. Hence, one can add, remove, or change the order of any potential configurations to adapt to specific designs and trade off the size of the directive search space against the search time.

The second part, $multi\_loop\_DS_{(i_{func},j_{func})}$ is the Cartesian product of the directive choices of all first-level loops ($single\_loop\_DS_{(i_{func},j_{func})}[loop]$) in an HLS function. Specifically, to maintain the descending order of latency, we sort all elements in the $multi\_loop\_DS_{(i_{func},j_{func})}$ by the sum of all indices in their original $single\_loop\_DS_{(i_{func},j_{func})}$.

Just Accepted

Algorithm 3: Incremental Floorplanning

```
Input: longest_functions starting from (i1, j1), design point p
Output: fit
// incremental_floorplan()
1  fit = online_packing((i1, j1), p);
2  if not fit then
3      fit = offline_repacking();
4      if fit then
5          fit = online_packing((i1, j1), p);
6      if not fit then
7          if FADO 1.0 then
8              iterative look_ahead(N), online_packing;          // FADO 1.0 only
9          if FADO 2.0 then
10             iterative find_next_DP(...), online_packing;      // FADO 2.0 only
11         if not fit then
12             iterative look_back(), online_packing;
```

Based on these two orders, we redesign the `find_next_DP()` strategy (Alg. 2) for choosing every new design point. We start by checking the $single\_loop\_DS_{(i_{max,1}, j_{max,1})}$. Following the order in Table 5, we evaluate one new directive configuration for every first-level loop and choose the one with the minimum latency improvement (line 9, Alg. 2) at each step. Whenever the newly evaluated design point has a lower latency than the previous longest function, we append it to the *DS_until_DP* list (line 12, Alg. 2). When we find a design point with a latency smaller than the second-longest function, the search stops and the *DS_until_DP* is returned (line 14, Alg. 2) for floorplanning. Otherwise, we continue to check the *multi_loop_DS*. Finally, all the points within *DS_until_DP* could be used for `look_back()` in the next section.

After getting the design point *DP*, in line 10 of Alg. 1, we design a storage binding mechanism to balance the BRAM, URAM, and LUTRAM utilization. Since the analytical model in FADO 2.0 provides BRAM and URAM estimation, we will bind the storage type of a design point to LUTRAM when URAM and BRAM utilization are both higher than a certain threshold T_1. Meanwhile, the difference between LUT and URAM/BRAM is larger than another threshold T_2. We mainly use $T_1 = 0.45$ and $T_2 = 0.1$ in our design. If these conditions are not met, we decide between BRAM and URAM based on a lower total utilization ratio among the board after applying this design point. At this point, the updated *DP* is ready for floorplanning.

5.4 Incremental Floorplanning

The initial floorplan input to FADO is generated by an iterative min-cut ILP bi-partitioning in the "AutoBridge Floorplanner [9]" shown in Fig. 2. During FADO's iterations, we first apply a resource-bottleneck-guided online WF algorithm. When this online packing fails to find a legal floorplan, an offline BFD re-packing compacts the existing floorplan before calling the online packing again. The definition of "online" and "offline" algorithms refers to [21]. During online packing, HLS functions are optimized one after another, and the previous floorplan of a function will be kept unchanged. In contrast, without applying new directives, the offline stage re-orders all functions by heuristics to improve the packing quality.

5.4.1 Online Packing. To avoid routing congestion from a high-abstraction view, the online packing tends to balance the utilization ratio among different resources and different slots, i.e., if a function fails to fit into its original slot after applying a new directive configuration, we try to floorplan it into other slots according to the

Just Accepted

Algorithm 4: Online Packing

Input: $longest\_functions$ and QoRs L_{ij}, R_{ij}, design point p
Output: fit, $incrfp\_list$

```
fit = False, incrfp_list = [], unfit_funcs = [];
for func in longest_functions do
    if func still fits in the current slot s_c then
        update directives for func, and resource util. for s_c;
        L_ij, R_ij = L_p, R_p; fit = True;
    else
        // check constraint Eq. 6, Eq. 7 and Eq. 8
        calculate overflow ratio, and sort other_slots by CR;
        for each s_o in other_slots do
            if no overflow when moving func to s_o then
                // check constraint Eq. 5
                update directives for func;
                update util. for s_o, s_c;                    // constraint Eq. 4
                append (func, s_o) to incrfp_list;
                L_ij, R_ij = L_p, R_p; fit = True; break;
        if not fit then
            append func to unfit_funcs;
if any func in unfit_funcs then
    clear incrfp_list; fit = False;
else
    update the floorplan according to incrfp_list;
```

non-decreasing order of critical resource (CR). CR refers to the type of resource having the highest overflow percentage among BRAM, DSP, FF, LUT, and URAM. If multiple slots have the same CR amount, we sort them by the average utilization of the other four non-critical resources. The online packing algorithm is shown in Alg. 4.

5.4.2 Offline Re-packing. Since `online_packing()` tends to spread functions evenly on each slot, and when there is an aggressive move in directive search with a sharp increase in resource utilization, the balance could preclude the new design point from taking effect. Thus, offline re-packing sorts all the slots SL_i by resource utilization in non-increasing order. Then, it respectively sorts all the functions F_{ij} on each slot SL_i by resource in non-increasing order as well. The re-packing starts with moving the F_{21} from the second fullest slot SL_2 to the fullest SL_1, and then the second largest function F_{22} from SL_2 to SL_1, etc. When SL_1 is full, or SL_2 is empty, we turn to move functions F_{31}, F_{32}, etc., from the third fullest slot SL_3 to SL_1, then to SL_2. A general step of re-packing the m-th fullest slot is to move $F_{m1}, F_{m2}, ..., F_{mn_m}$ to $SL_1, SL_2, ..., SL_{m-1}$ in turn.

Fig. 8 shows the floorplanning of 5 functions onto 4 slots. We reduce multiple resources to one dimension by taking the maximum ratio among them. Through directive search, the resource of the blue function expands from 0.3 to 0.6. However, since other gray functions were fixed during the online stage, the expanded function fits nowhere. To compare, the re-packing stage sorts the slots by utilization in non-increasing order and executes six trials in order. Trials 1/2 fail because slot 1 is full. During trial 3, a gray function is moved from slot 2 to slot 0. Trials 4/5 also fail because the destinations, slots 0 and 1, are full. Trial 6 is canceled because the destination, slot 2 has been found empty. With re-packing, the expanded function fits in slot 2 after a new round of online packing.

Just Accepted

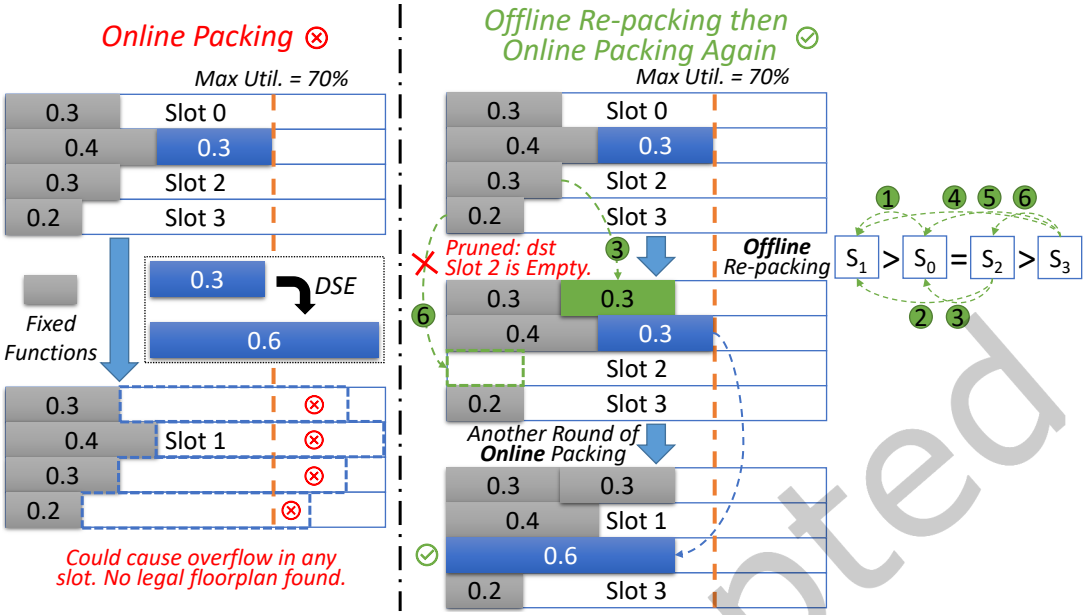


Fig. 8. An Example of Offline Re-packing.

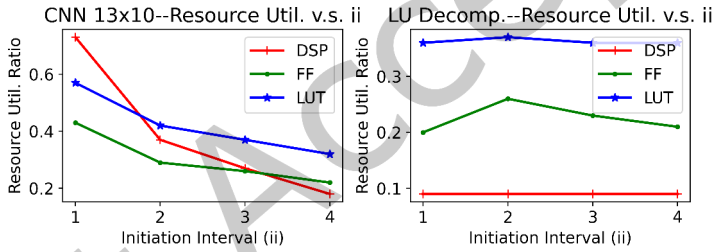


Fig. 9. Comparison of Resource Utilization vs. II between CNN and LU Benchmarks.

Re-packing applies different strategies for functions in dataflow kernels and non-dataflow kernels. The floorplan change is free for dataflow functions if the SLL utilization constraint is met. For non-dataflow parts, since their connections with other adjacent functions are not through the FIFO channel, the long wires could not be broken by inserting pipeline logic. Hence, instead of moving them, we force other dataflow functions with no RAM connection to different slots. Thus, FADO assigns more resources to the slots containing non-dataflow for further DSE.

5.4.3 Look-Ahead and Look-Back. If *DP* is still not successfully floorplanned after online packing and offline re-packing, we will keep trying other design points. This is because, in realistic HLS designs, greedy DSE and floorplanning could easily get stuck in local optima. Fig. 9 shows the different trend of resource utilization as the *PIPELINE* initiation interval (II) changes in two designs, CNN from [5] and LU from [45]. As II increases, latency also increases in both designs. It reduces the utilization of the three types of resources in the CNN benchmark because computation instances are shared among multiple cycles. However, in contrast, there are a lot of loop-carried dependencies in the LU benchmark; the results from the previous iteration cannot be directly

Just Accepted

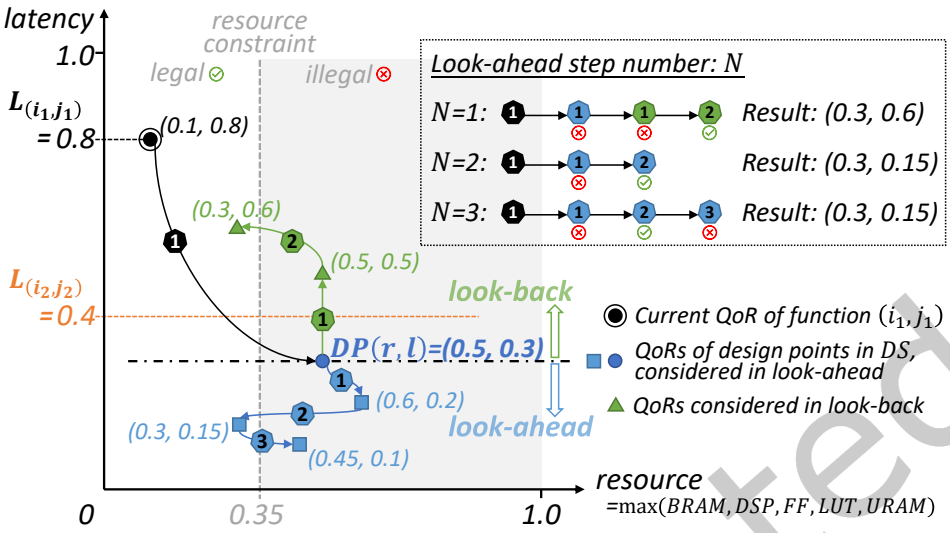


Fig. 10. Effect of Look-Ahead in FADO 1.0 Flow with Step # N, and Look-Back.

passed to the next, and extra logic is used to buffer the results. Together with the effect of resource sharing, the utilization first increases as II increases from 1 to 2, and then decreases when II continues increasing.

To handle the non-monotonic design space in the QoR library, FADO 1.0 proposes the sampling methods of `look_ahead()` and `look_back()`. When the first two stages of floorplanning—online packing and offline re-packing fail to find a legal floorplan for *DP*, we further check the floorplan for a limited number of design points with lower latency yet potentially fewer resources. This is referred to as the look-ahead stage. If it still fails in floorplanning, we turn to check the points with larger latency than *DP* in the look-back stage. These points are more likely to have lower resource utilization and a legal floorplan.

Here is an example of how FADO 1.0 samples within the QoR library. Fig. 10 shows a snapshot of directive search for the current bottleneck function (i_1, j_1). QoR values are normalized for clarity. The design point with the top priority for floorplan checking is *DP*, with a resource utilization of 0.5 and a latency of 0.3 (the longest latency smaller than $L_{(i_2,j_2)} = 0.4$). However, only 0.35 resource is left for the current function, and no legal floorplan is found for *DP* during online packing and offline re-packing. We now look ahead/back for other improvement opportunities with fewer resources. When we set the step number N to 1 during `look_ahead()`, the next design point consumes 0.6 utilization and also fails to be floorplanned. Thus, when `look_ahead()` also fails to find a point with a legal floorplan, `look_back()` traverses all the points with latency from 0.3 to 0.8. When N is set to 2 or 3, the directive configuration with a latency of 0.15 is found during `look_ahead()`.

For HLS designs, FADO 1.0 decides the step number N of `look_ahead()` by analyzing the range of parameters for *PIPELINE* and *UNROLL*, two of the most effective directives. We define the N as the largest number of different configurations on a single nested loop. To exclude the directive configurations that over-utilize resources, we check at most three levels for each nested loop from the innermost level. For each nested loop of n levels, we index the innermost loop with 1, and the outermost loop with n. (1) For directive *PIPELINE*, since [17] suggests a maximum loop bound of 64 for auto pipelining, we set the range of II to the logarithm of the minimum between 64 and the iteration latency *IL* from HLS report. (2) For directive *UNROLL*, similar to *PIPELINE*, we take the minimum between 64 and the loop bound *B*. (3) For the combination of *PIPELINE* and *UNROLL*, since all the inner

Just Accepted

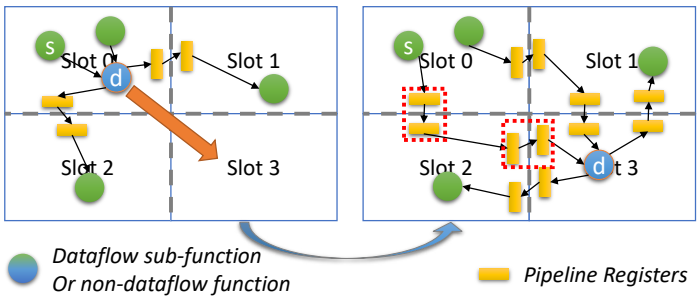


Fig. 11. Incremental Update of Pipeline Registers.

loops are completely unrolled when an outer loop is pipelined, we consider only the directive combination in 2 levels of loops. In all, the resulting step number N for a design with m nested loops is:

$$\begin{aligned} N_1 &= \max_{1\le i\le m} \sum_{j=1}^{\max(3,n_i)} \log_2 \min(64, IL_{ij}) \\ N_2 &= \max_{1\le i\le m} \sum_{j=1}^{\max(3,n_i)} \log_2 \min(64, B_{ij}) \\ N_3 &= \max_{1\le i\le m} \sum_{j=1}^{\max(2,n_i)} \log_2 \min(64, B_{ij}) \\ N &= N_1 + N_2 + N_3 \end{aligned} \tag{9}$$

On the flip side, the look-ahead sampling strategy for FADO 2.0 is more straightforward and free from the N above, because the sampling order shown in Table 5 secures a promising converging direction. We can keep calling `find_next_DP()` (line 10 of Alg. 3) until no new design point is available or the upper limit for loop tripcount is reached.

5.5 Incremental Pipelining

When floorplanning HLS functions from one slot to another, as Fig. 11 shows, each time a path crosses a boundary between two slots (SLR boundary or I/O banks), additional pipeline registers should be added beside the boundaries to break down the long wires. We here set a constraint of 90% (Eq. 8) for SLL utilization and incrementally update the pipelining logic of long wires crossing slot boundaries. As Fig. 11 shows, when function "d" is moved from Slot 0 to Slot 3, two groups of additional pipeline registers are added between "s" and "d."

5.6 Exit Condition and External Tools

During iterative optimization, when there is no legal floorplan found for the next design point of the current longest function or when no other directive configuration could improve the bottleneck's latency further, it is excluded in future iterations. FADO stops when there is no function left for the bottleneck analysis. Then, it dumps the optimal directives to a TCL file to guide the re-synthesis of the HLS code to generate a high-performance RTL design. FADO also delivers the final floorplan to the global router, latency balancer, and dataflow RTL generator within [9] to update the pipelining of dataflow kernels in the Verilog code and generate another TCL script to guide the floorplanning during implementation in Vitis.

Just Accepted

6 RESULTS

6.1 Benchmarks and Experiment Settings

To effectively evaluate FADO, the benchmarks' resource utilization and the number of HLS functions are essential metrics. If the resource utilization is too low, it would not challenge the quality of floorplanning. If only a few functions exist, it will not fully demonstrate the co-search efficiency. Both cases would reduce the co-search problem to a pure directive optimization problem. Hence, we mainly adopt large-scale open-source HLS designs with compatible interfaces for evaluation and filter out many commonly used but unsuitable benchmarks. To be specific, most of the designs in Vitis Libraries [49], CHstone [12], Rosetta [56], etc., occupy less than 10% of the resources on the Alveo U250 FPGA. They only have several functions to consider during coarse-grained floorplanning, which is not challenging even if we increase the design size, e.g., by applying a larger bitwidth. Besides, interface incompatibility makes it difficult to scale up by connecting multiple designs from these benchmarks. Hence, we generate large dataflow kernels *CNN*, *MM*, and *MTTKRP* using PolySA [5] and AutoSA [45]. For non-dataflow designs, we use *2MM*, *COV*, and *HEAT* from PolyBench [26], which are general programs also used in CPU, GPU, etc. To best show the generality of our solution, we assemble six large benchmarks mixing the dataflow and non-dataflow kernels above to evaluate the performance of our framework, as Fig. 12 shows. Their number of functions (dataflow sub-functions + non-dataflow) ranges from 175 to 350. When different directives are applied, their maximum utilization ranges from ~20% to over 80% of the on-chip resources within our designated dies after implementation. The kernels connect through RAMs, which enlarges the design space compared with a single dataflow kernel.

To show the scale of our problem, we visualize the HLS-function-level data flow graph of the *CNN*2+2MM*1* benchmark in Fig. 13. The two yellow bounding boxes mark the two *CNN13x2* dataflow kernels, each containing tens of sub-functions. The red circles on the top of this figure are the non-dataflow *2MM* kernel and the two RAMs connected to it. The RAM "temp_xin1_V_U" is connected to two input sub-functions of *CNN13x2* Kernel 1, and RAM "temp_xout0_V_U" is connected to one output sub-function of *CNN13x2* Kernel 0. Since their connections are not through FIFO channels, they are grouped during floorplanning and always placed in the same slot. As for a dataflow kernel, the green boxes are FIFO channels, and the blue circles are dataflow sub-functions. Dataflows can be partitioned, floorplanned, and pipelined on any slot as long as the resource constraints are met. The overall design space of FADO is the Cartesian product of directive space and floorplan space. For directive search, the space ranges from millions to billions in our benchmarks, considering the parameters in Table 4. For floorplanning, it maps hundreds of functions to four slots, and the space size is four to the power of hundreds.

We use the AMD Xilinx Vitis HLS 2020.2 for HLS synthesis and Vitis for implementation. We evaluate our framework on the AMD Alveo U250 FPGA, which contains eight slots defined by the 4 SLRs and an I/O bank

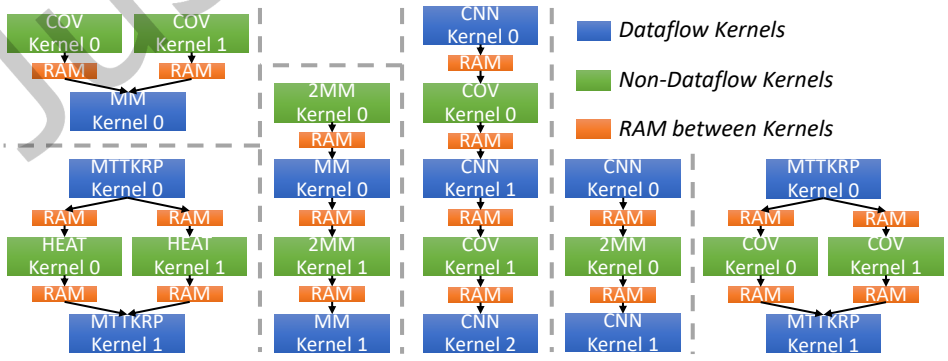


Fig. 12. The 6 Benchmarks Used to Evaluate FADO.

Just Accepted

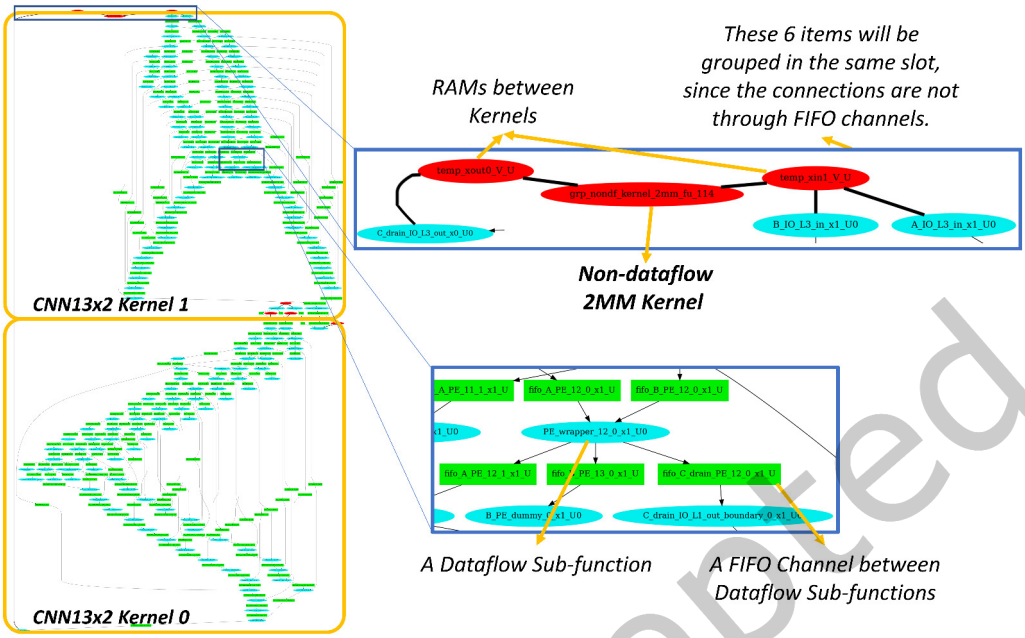


Fig. 13. The Scale of the CNN*2+2MM*1 Benchmark.

in the middle. Note that the rightmost column (~1/8) of clock regions is occupied by Vitis platform IP. Hence, the resource calculation excludes that column. We tightly limit the floorplanning of HLS designs to the ***lower*** half[1] (4 slots on SLR 0 and SLR 1) of the FPGA to post more challenges to the optimality of our results. In our experiments, we find that the resource constraint of 70% still leads to placement or routing failure sometimes. Hence, in FADO 1.0, we tighten the limit to 65% for each slot during DSE. Meanwhile, in FADO 2.0, the LUT utilization is sometimes higher than the actual number after logic synthesis because the resource sharing and operation chaining are not as comprehensive as the commercial tool. Together with the balance mechanism between BRAM, URAM, and LUTRAM, we loosen the limit to 80% or even larger for different types of resource utilization estimated by the model.

6.2 Comparative Experiments

Table 6 mainly compares FADO 2.0 and FADO 1.0 with different directive-floorplan co-search flows and the global floorplanning in [9]. We report the total runtime (consisting of the DSE time and the pre-processing time) and the quality of each implemented design with its resource, latency, maximum achievable frequency, and overall execution time. Among all metrics, overall execution time combines latency and timing quality, reflecting the ultimate design performance on FPGA. We highlight the best latency, Fmax, and overall execution time in each column.

In Table 6, the first row, "Original (directive-free)" shows the resource and latency of the 6 benchmarks when removing all directives and turning off every optimization, such as auto-pipelining. This configuration generally has the least resource utilization and thus is used as the starting point for directive-floorplan co-search. Before evaluating the automated co-search flows, two other configurations in the second and third rows also deserve our attention. Since AutoSA-generated designs are originally manually optimized with rich sets of directives,

[1]The lower half of the Alveo U250 FPGA, excluding the rightmost column of clock regions, contains 2016 BRAMs, 5184 DSPs, 1319040 FFs, 659520 LUTs, and 544 URAMs. These are the total resource considered in Table 6 and Table 7.

Just Accepted

we try to keep them, to not only observe the gap between our automated flow and the manual optimization, but also reveal the necessity of automating the directive-floorplan co-search. We have also turned on the auto-pipelining since PolyBench designs originally contained no directives. About "Original (directive-rich, no FP)", since the designs are not floorplanned, they can spread all over the 4 dies of the Alveo U250 FPGA. Manually optimized HLS directives indeed lead to the lowest latency in most (5 out of 6) cases, but it either results in sub-optimal frequency (e.g., *MM*1+COV*2* and *MM*2+2MM*2*) or implementation failures, such as over-utilization in BRAM (*MTTKRP*2+HEAT*2*), DSP (*CNN*3+COV*2*), and net conflicts during routing (*MTTKRP*2+COV*2*). As for "Original (directive-rich, AutoBridge FP)", when we constrain the designs to the lower half of the FPGA, only the smallest *CNN*2+2MM*1* can be successfully implemented. To briefly summarize, these two series with manual optimization are ***floorplan-unaware***. With a one-off latency optimization to the extreme, the aggressive resource expansion leads to sub-optimal frequency or implementation failure.

Then, we compare three types of automated co-search flows. The directive search in each flow either uses a synthesis -based QoR library for evaluation or relies on the analytical model.

The first type of baselines, "Initial FP -> Iterative Syn-/Ana-DO," perform the directive optimization using a one-off initial floorplanning. They apply the min-cut ILP floorplanning from [9] only once, and then all HLS functions' positions are fixed during the iterative directive search. The limited optimization opportunities caused by the fixed initial floorplan lead to an under-utilization of resources. This severely limits the latency optimization, resulting in the longest latency among all benchmarks. The synthesis-based one fails in the implementation of *MTTKRP*2+HEAT*2* because two *HEAT* kernels are floorplanned on the same slot, each having a large array using more than one column of BRAM or URAM, which triggers an exception during placement. Similar issues are found in the analytical baseline for the *MTTKRP*2+COV*2*.

The "Iterative (Syn-/Ana-DO + AutoBridge FP)" baselines run the min-cut ILP floorplanning iteratively after applying each new directive configuration. Note that the heuristics of look-ahead and look-back are also applied in these cases for fairness when compared with FADO. The synthesis-based one results in orders-of-magnitude longer search time than FADO due to repetitively calling the ILP solver when meticulously traversing the QoR library. In comparison, the analytical one converges in fewer steps with the redesigned directive search strategy (Alg. 2). Besides, the balance mechanism among BRAM, URAM, and LUTRAM reduces the utilization ratio of the critical resource in some cases, making it easier for the solver to reach feasible solutions. Meanwhile, since AutoBridge [9] applies iterative bi-partitioning rather than a one-off eight-way partitioning[2], optimality is not guaranteed. As reflected by the execution time, the design implementation quality of these baselines is inferior to the corresponding flow of FADO in all six benchmarks. In summary, these methods incur longer search time while still resulting in sub-optimal designs.

As for FADO 1.0, the online packing and offline re-packing strategies alternatively balance and compact the floorplan, contributing to better utilization of resources on multiple dies (the highest utilization ratio under resource constraint of 65% in five out of all six benchmarks). Accordingly, the high-quality floorplan strongly supports exploring a larger design space during the directive search. Thus, our FADO 1.0 achieves 33.12% smaller latency on average compared with the time-consuming "Iterative (Syn-DO + AutoBridge FP)" and attains the lowest latency for all benchmarks over all synthesis-based baselines. The latency improvement varies because of the nature of benchmarks—it is more significant when FADO 1.0 legalizes the floorplan for some bottleneck functions with a great latency-resource tradeoff, as the cases *MM*1+COV*2*, *MTTKRP*2+HEAT*2*, and *MTTKRP*2+COV*2* show. As for frequency, experiments show that when the utilization gets close to 65%, although the frequency could vary to some extent due to non-determinism in floorplanning and further implementation, our incremental solution still outperforms the baselines, with both a higher average Fmax of 290.96 MHz and lower variance.

[2]Eight-way partitioning runs even more than 10x slower compared with bi-partitioning in directive-floorplan co-search experiments using benchmarks above.

Just Accepted

Table 6. QoR Comparison among FADO 2.0/1.0 and Other DSE Strategies (Synthesis vs. Analytical)

Benchmarks	CNN*2+2MM*1 (175 functions)						MM*1+COV*2 (176 functions)					
	Res.[a]	Runtime[b]	Latency[c]	Fmax[d]	**Exe_time[g]**	**Norm.[h]**	Res.	Runtime	Latency	Fmax	**Exe_time**	**Norm.**
Original (directive-free)	28%	-	8933	-	-	-	20%	-	131,839	-	-	-
Original (directive-rich, no FP)	42%	-	91.4	302.39	302	1.24	115%	-	1,344	149.75	8,977	1.65
Original (directive-rich, AutoBridge FP)	42%	-	91.4	347.10	263	1.09	115%	-	1,344	*Failure*	-	-
Initial FP[e] -> Iterative Syn[i]-DO[f]	28%	2.24+t_p [k]	735	300.45	2,445	10.08	19%	0.16+t_p	131,839	282.86	466,094	85.90
Initial FP -> Iterative Ana[j]-DO	26%	4.44	8504	**400.80**	21,218	87.47	42%	13.65	2,700	305.25	8,846	1.63
Iterative (Syn-DO + AutoBridge FP)	48%	1658+t_p	92.7	235.89	393	1.62	41%	10307+t_p	2,549	278.84	9,141	1.68
Iterative (Ana-DO + AutoBridge FP)	42%	55.89	90.8	353.48	257	1.06	42%	37.42	4,777	153.66	31,090	5.73
Iterative (Syn-DO + Incr FP) (**FADO 1.0**)	55%	2.39+t_p	91.2	269.95	338	1.39	41%	2.17+t_p	**1,647**	274.47	6,001	1.11
Iterative (Ana-DO + Incr FP) (**FADO 2.0**)	43%	26.41	**87.2**	359.45	**243**	1.00	42%	35.49	1,683	**310.37**	**5,426**	1.00
Benchmarks	MTTKRP*2+HEAT*2 (242 functions)						MM*2+2MM*2 (350 functions)					
	Res.	Runtime	Latency	Fmax	**Exe_time**	**Norm.**	Res.	Runtime	Latency	Fmax	**Exe_time**	**Norm.**
Original (directive-free)	57%	-	8,147,919	-	-	-	40%	-	259,516	-	-	-
Original (directive-rich, no FP)	84%	-	63,753	*Failure*	-	-	79%	-	2,580	229.57	11,239	1.01
Original (directive-rich, AutoBridge FP)	84%	-	63,753	*Failure*	-	-	79%	-	2,580	*Failure*	-	-
Initial FP -> Iterative Syn-DO	46%	1.36+t_p	8,138,605	*Failure*	-	-	59%	1.13+t_p	258,842	274.10	944,335	89.39
Initial FP -> Iterative Ana-DO	54%	7.77	1,360,021	368.19	3,693,801	11.85	69%	31.75	260,710	399.36	652,820	58.69
Iterative (Syn-DO + AutoBridge FP)	62%	3554+t_p	598,532	159.97	3,741,525	12.00	60%	32656+t_p	67,652	*Failure*	-	-
Iterative (Ana-DO + AutoBridge FP)	86%	56.25	391,089	309.79	1,262,436	4.04	79%	143.77	6,112	**400.16**	15,275	1.37
Iterative (Syn-DO + Incr FP) (**FADO 1.0**)	63%	1.95+t_p	128,104	300.45	426,374	1.37	58%	6.63+t_p	66,158	300.00	220,527	19.82
Iterative (Ana-DO + Incr FP) (**FADO 2.0**)	74%	22.15	**102,982**	**330.36**	**311,726**	1.00	79%	76.37	**2,696**	242.37	**11,124**	1.00
Benchmarks	CNN*3+COV*2 (263 functions)						MTTKRP*2+COV*2 (242 functions)					
	Res.	Runtime	Latency	Fmax	**Exe_time**	**Norm.**	Res.	Runtime	Latency	Fmax	**Exe_time**	**Norm.**
Original (directive-free)	31%	-	18,130	-	-	-	38%	-	8,113,234	-	-	-
Original (directive-rich, no FP)	681%	-	259	*Failure*	-	-	155%	-	63,143	*Failure*	-	-
Original (directive-rich, AutoBridge FP)	681%	-	259	*Failure*	-	-	155%	-	63,143	*Failure*	-	-
Initial FP -> Iterative Syn-DO	39%	2.03+t_p	6,716	300.45	22,354	6.00	42%	2.18+t_p	8,113,234	300.45	27,003,607	130.37
Initial FP -> Iterative Ana-DO	34%	1.65	27,953	**355.37**	78,660	21.12	51%	3.54	1,962,240	*Failure*	-	-
Iterative (Syn-DO + AutoBridge FP)	62%	8301+t_p	1,278	222.01	5,754	1.54	61%	12627+t_p	562,017	300.45	1,870,585	9.03
Iterative (Ana-DO + AutoBridge FP)	86%	103.77	1,367	337.27	4,052	1.09	82%	50.96	339,519	*Failure*	-	-
Iterative (Syn-DO + Incr FP) (**FADO 1.0**)	63%	5.04+t_p	**1,233**	300.45	4,105	1.10	64%	4.89+t_p	126,921	300.45	422,437	2.04
Iterative (Ana-DO + Incr FP) (**FADO 2.0**)	86%	73.73	1,250	335.46	**3,725**	1.00	82%	21.22	**63,769**	**307.88**	**207,123**	1.00

[a] The max HLS-reported util. ratio among BRAM, DSP, FF, LUT and URAM. [b] DSE time in seconds (+t_p pre-processing time for syn-based methods) .
[c] Execution time of HLS designs in *kilo* clock cycles. [d] Maximum achievable frequency in MHz. [e] FP: Floorplanning. [f] DO: Directive Optimization.
[g] Overall performance (cycle #/frequency) of optimized HLS designs in microseconds (μs). [h] The overall performance *normalized* to FADO 2.0's result.
[i] Using synthesis-based QoR lib, corresponding with FADO 1.0. [j] Using analytical QoR model, corresponding with FADO 2.0 in this extension paper.
[k] t_p is the pre-processing time for generating the synthesis-based QoR library. It varies for different benchmarks, ranging from hours to 10s hours.

Moreover, since our incremental legalization leads to a minimum change of floorplan in each iteration of co-optimization, it is much more efficient than updating all functions' locations globally. This efficient legalization contributes to a speedup of 693X~4925X in the search time of the entire co-optimization when excluding the pre-processing overhead. With the FADO 1.0 optimization flow, the design implementation quality reflected in the overall design execution time is 1.16X~8.78X better than the best synthesis-based baseline.

After extending to the analytical FADO 2.0, although triggering the QoR model requires several more seconds in each iteration than directly looking up in the QoR library, our total runtime is still shorter than all baselines and even FADO 1.0. One reason is the removal of the pre-processing time t_p (hour-level). Another reason is the new smarter directive search strategy (Sec. 5.3). Compared with the synthesis-based counterparts, the DSE time of "Iterative (Ana-DO + AutoBridge FP)" is significantly shorter. This is because the new directive optimization converges more effectively, resulting in fewer rounds of ILP floorplanning (sec to 10s-sec-level each round), which is much more significant than the time difference between using the QoR library (μs to ms-level) and

JUST Accepted

the analytical model (ms to sec-level). On the flip side, the DSE time for FADO 2.0 is witnessed to have slightly increased compared with FADO 1.0. This is because the inference overhead of the analytical model is more significant compared with our efficient incremental floorplanning (μs-level).

The new exploration strategy of FADO 2.0 (Sec. 5.3) is more effective than the library-based greedy search in FADO 1.0 and further optimizes the latency to the extreme. On average, FADO 2.0 achieves 45.79% and 24.74% smaller latency than "Iterative (Ana-DO + AutoBridge FP)" and FADO 1.0, respectively. As for Fmax, the balanced storage binding helps FADO 2.0 to gain a 9.18% higher frequency. Altogether, the overall improvement in optimized design performance is **2.66X** over the strongest analytical baseline—"Iterative (Ana-DO + AutoBridge FP)." Compared with FADO 1.0, there is one outlier with 19.83X better design performance. The dramatic improvement comes from a new optimization opportunity opened by the enhanced search algorithm and storage balancing. Excluding this point, FADO 2.0 concludes with a **1.40X** better design performance over FADO 1.0 on average. This demonstrates the effective integration of incremental floorplanning with the analytical model and new search strategy.

Alongside the statistics above, we visually examine the implemented designs from the device view in Vivado, as Fig. 14 shows. By highlighting the leaf cells of each module (HLS dataflow sub-functions and non-dataflow kernels) in different colors, we use "Original (directive-rich, no FP)" and FADO 2.0 to demonstrate the importance of *directive-floorplan co-search*. Except for the medium-size *CNN*2+2MM*1* and large-size *MM*2+2MM*2*, we here add one additional *tiny* design, *SCMM* (modified from [50]), with only 10 functions to show FADO's capability of scaling down. *SCMM* and *MM*2+2MM*2* are two extremes. The former contains a small number of huge functions—its three main functions consume 52% of the total DSP available, and the latter comprises the largest number of functions with over 70% utilization of both LUT and DSP. When targeting a high frequency during implementation, these challenging cases can easily incur routing congestion or hold violations. Results prove that while "(directive-rich, no FP)" cases with manual directive optimization attain even slightly better latency, FADO 2.0 achieves superior frequency and overall design execution time by using only (lower) half of the FPGA. This is

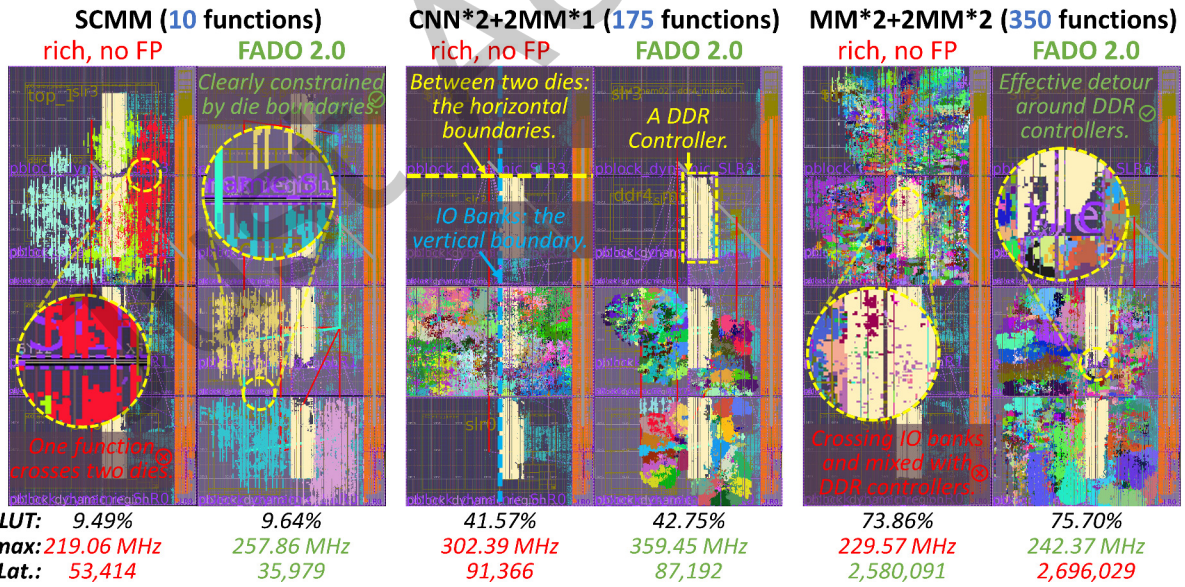


Fig. 14. Comparing "Original (directive-rich, no FP)" with FADO 2.0 from the Post-implementation Device View Using Three Designs with Various Number of HLS Functions and Resource Utilization.

Accepted

attributed to FADO's iterative and incremental search, which gradually takes full utilization of the designated dies without violating the floorplanning rules with regard to die boundaries and I/O banks.

6.3 Analysis of DSE Stages—Case Study (Syn-/Ana-based Search, CNN*2+2mm*1)

To analyze the effectiveness of the multiple stages in FADO, we visualize the directive-floorplan co-search process for the *CNN*2+2MM*1* benchmark using FADO 1.0/2.0 and the baseline Iterative (Syn-DO + AutoBridge FP) in Fig. 15. The horizontal axis takes the maximum utilization of resources on the FPGA, and the vertical axis shows the latency in the number of clock cycles. The *light cyan* points represent the whole directive design space

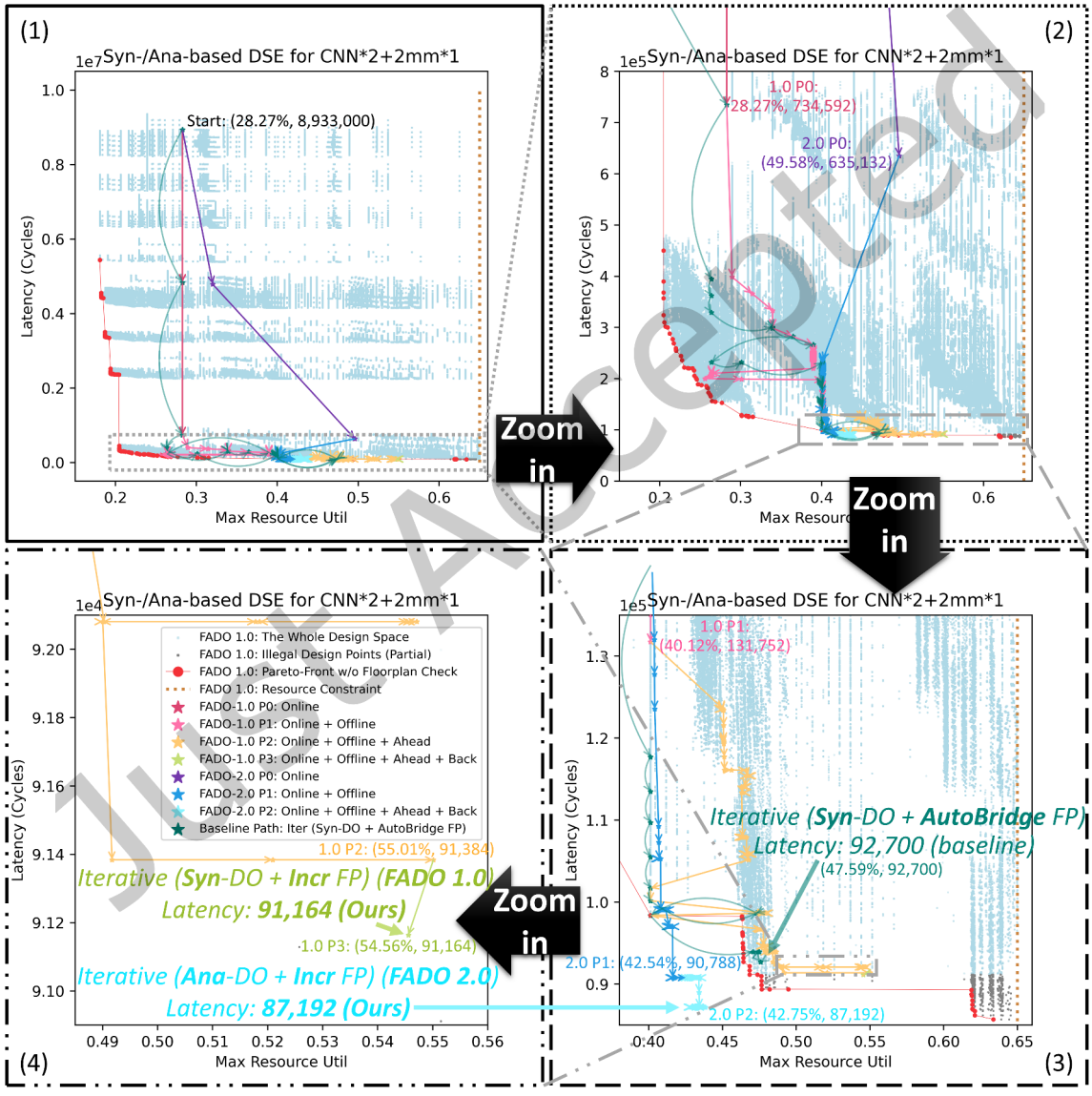


Fig. 15. Syn-/Ana-based DSE Stages and Results on the CNN*2+2MM*1 Benchmark.

Just Accepted

Table 7. Stages of Floorplan-Aware Directive Optimization

FADO 1.0												
Benchmarks	CNN*2+2MM*1		MM*1+COV*2		MTTKRP*2+HEAT*2		MM*2+2MM*2		CNN*3+COV*2		MTTKRP*2+COV*2	
Stages	Resource[a]	Latency[b]	Resource	Latency	Resource	Latency	Resource	Latency	Resource	Latency	Resource	Latency
Online	28.27%	735	40.66%	5,167	63.15%	163,241	40.28%	259,516	31.79%	4,718	62.95%	141,260
Offline	40.12%	132	40.66%	5,167	64.67%	153,927	40.28%	259,516	31.79%	4,718	62.95%	141,260
Ahead-Back	54.56%	91.2	40.66%	1,647	63.25%	128,104	57.53%	66,158	63.32%	1,233	64.49%	126,921
FADO 2.0												
Benchmarks	CNN*2+2MM*1		MM*1+COV*2		MTTKRP*2+HEAT*2		MM*2+2MM*2		CNN*3+COV*2		MTTKRP*2+COV*2	
Stages	Resource	Latency	Resource	Latency	Resource	Latency	Resource	Latency	Resource	Latency	Resource	Latency
Online	49.57%	635	42.28%	1,683	79.98%	548,302	79.01%	5,183	33.20%	17,605	81.79%	288,622
Offline	42.54%	90.8	42.28%	1,683	73.57%	102,982	79.01%	4,607	33.20%	17,595	81.79%	63,769
Ahead-Back	42.75%	87.2	42.28%	1,683	73.57%	102,982	79.01%	2,696	85.69%	1,250	81.79%	63,769

[a] The maximum HLS-reported util. ratio among BRAM, DSP, FF, LUT, and URAM. [b] Execution time of HLS designs in number of *kilo* clock cycles.

formed by the QoR library in FADO 1.0 without any floorplan legality check, with the *red* dots showing its Pareto front. Our search starts from the point with the highest latency (28.27%, 8,933,000).

In the **first** stage, the *cranberry* arrows (FADO 1.0 P0) show online floorplanning, which stops at (28.27%, 734,592) because of a sharp resource increase of the large non-dataflow kernel *2MM*. Similarly, the *purple* arrows (FADO 2.0 P0) reach (49.58%, 635,132) when configuring a different pipeline II for *2MM*. In the **second** stage, the offline re-packing of FADO 1.0 (*pink* arrows) clears out the dataflow sub-functions on the least-occupied slot and continues until (40.12%, 131,752), the top *pink* point in Fig. 15 (3). Meanwhile, FADO 2.0's offline floorplanning (*blue* arrows) helps the non-dataflow *2MM* find a design point with both less latency and resources. Guided by the directive search order in Table 5, FADO 2.0 continues with online and offline heuristics until breaking through the Pareto front (*red* dots) formed by FADO 1.0's QoR library and reaches (42.54%, 90,788). However, based on the QoR library, the next design point from the top *pink* point in sub-figure (3) for FADO 1.0 consumes significantly larger resources and triggers online and offline floorplanning failure. This forces FADO 1.0 to enter the **third** stage of look-ahead (*yellow* arrows). It continues sampling for points with less utilization of the current critical resource. As Fig. 15 (3) shows, with the help of look-ahead (*yellow* arrows), the search reaches (55.01%, 91,384), and **finally** stops at (54.56%, 91,164) after one additional step of look-back (the *light green* arrow in sub-figure (4)). To show the optimality of FADO 1.0's result within the design space constrained by the QoR library, we check the floorplan legality for all design points in the QoR library with less latency than our result of 91,164 cycles—all the *gray* points have no legal floorplan when running global ILP floorplanning solely. Freed from the QoR library, FADO 2.0's last step (the *cyan* arrow) significantly shortens the design's latency and converges to (42.75%, 87,192). By contrast, the baseline directive search (*dark green* arrows) with global ILP floorplanning stops earlier at (47.59%, 92,700).

Table 7 shows the DSE results of different optimization stages in FADO 1.0 and 2.0. Note that the stages run sequentially in each iteration, and the latency/resource in this table is not the result of each stage acting alone, except for "Online." For example, the stage "Ahead-Back" includes the joint effort of (1) online packing, (2) offline re-packing followed by another round of online packing, and (3) look-ahead + look-back followed by online packing, as described in Alg. 3. It is possible that we only use (1) or (1)+(2) for some iterations while using (1)+(2)+(3) in the worst cases. The QoR of each stage shown in Table 7 measures the legal design point with the smallest latency achieved before calling the next stage for the first time. For example, the results in "Online" are the legal point with the smallest latency achieved before the first time calling `offline_repacking()`.

For some benchmarks, e.g., *MTTKRP*2+HEAT*2* in FADO 1.0 results and *MM*2+2MM*2* in FADO 2.0, each stage is more effective than the previous one on avoiding local optima. However, the offline method fails to improve the results in some other cases, such as *MM*1+COV*2* and *CNN*3+COV*2* in both FADO 1.0 and 2.0. This problem

Just Accepted

happens when there are oversized design points incurred by aggressive directive configurations, such as fully unrolling a loop or completely partitioning an array. For example, the non-dataflow *COV* kernel consumes 30 DSPs without any directive. However, when we unroll the loop containing the multiplication operation, the DSP number increases to 1920, more than the total number of DSPs available in any slot. Thus, the offline stage fails to optimize the floorplan, and the bottleneck DSP utilization remains the same value during DSE in *MM*1+COV*2*. For *CNN*3+COV*2*, since *COV* kernel has a longer latency than *CNN*, significant improvements are enabled by the look-ahead and look-back.

6.4 Analytical QoR Model Evaluation

6.4.1 Prediction Efficiency. To reason the necessity of having an analytical model, alongside our motivation of removing the time-consuming pre-processing stage, the most fundamental reason is that the HLS synthesis stage is too slow. Thus, we extracted 36 different HLS functions from all benchmarks, to compare their synthesis time in Vitis HLS 2020.2, and the inference time in our analytical model. Because directives usually complicate the latency/resource computation and incur a longer runtime for analysis, we choose two design points for each function. One point has no directive; the other is the best point encountered during the search in FADO 2.0. Experiments show that our analytical model's average speedup (Vitis HLS synthesis time/estimation time) is 231.84X and 208.80X respectively for the unoptimized and optimized cases.

6.4.2 Prediction Quality. Model estimation for resources is generally more accurate and reliable than latency. As mentioned in previous works, the accuracy of DSPs and BRAMs is guaranteed. The estimation error (mostly over-prediction) for LUTs is, to some extent, tolerated by tuning the resource constraint. However, our operator delays and chaining rules differ significantly from the previous results in [52, 55] because of the gap between different versions of HLS tools together with the new data types and bitwidths considered. If the prediction of a module's LUT number is 10% more than the actual utilization, a secure resource constraint would still help find a legal floorplan. However, even if the latency estimation of one design point is only two cycles off from the ground truth, it could possibly reverse the search order of this point with another one, and the DSE process would miss an optimization opportunity.

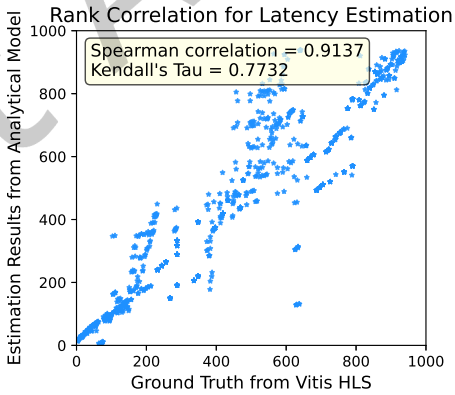


Fig. 16. Rank Correlation for Latency Computed by Vitis HLS and Our Analytical Model.

Hence, we validate our model's latency estimation quality in the following way. First, we randomly extract ~1000 directive configurations for various functions from all the benchmarks. Then, we run Vitis HLS synthesis and model-based prediction for them and rank the latency results respectively, as Fig. 16 shows. The Spearman coefficient of 0.9137 and Kendall's Tau of 0.7732 show that the predicted results strongly correlate with the

Just Accepted

ground truth from Vitis HLS. This prediction quality assures us that FADO generally converges in a very efficient direction.

7 CONCLUSION

Our work produces the FADO framework, an open-source, end-to-end solution that co-optimizes the directives and floorplan of HLS designs implemented on multi-die FPGAs. It provides two optimization flows, either with a synthesis-based library (1.0 version) or with an analytical QoR model (2.0 version). FADO integrates a latency-bottleneck-guided directive optimization and an incremental floorplanning algorithm mixing various bin-packing heuristics. On the one hand, our incremental floorplanning significantly improves over the corresponding synthesis/analytical-version global ILP floorplanning [9]. On the other hand, our co-optimization enables full utilization of resources on multiple dies and greatly benefits both the latency and timing. Among all six large-scale benchmarks mixing dataflow and non-dataflow kernels, FADO 1.0 optimizes their execution time by 1.16X~8.78X compared to the synthesis-based DSE with global floorplanning. Further, the optimized HLS designs from FADO 2.0 achieve 2.66X better design performance over the analytical-based solution with global floorplanning and, meanwhile, 1.40X over FADO 1.0.

ACKNOWLEDGMENTS

This work is partially supported by RGC General Research Fund 16214123, and AI Chip Center for Emerging Smart Systems (ACCESS), Hong Kong SAR. Many thanks to all the anonymous reviewers for their valuable comments.

REFERENCES

[1] Md Mostofa Akbar, Eric G Manning, Gholamali C Shoja, and Shahadat Khan. 2001. Heuristic solutions for the multiple-choice multi-dimension knapsack problem. In *International Conference on Computational Science*. Springer, 659–668.

[2] Tobias Alonso, Lucian Petrica, Mario Ruiz, Jakoba Petri-Koenig, Yaman Umuroglu, Ioannis Stamelos, Elias Koromilas, Michaela Blott, and Kees Vissers. 2021. Elastic-DF: Scaling performance of DNN inference in FPGA clouds through automatic partitioning. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 15, 2 (2021), 1–34.

[3] Vaughn Betz and Jonathan Rose. 1997. VPR: A new packing, placement and routing tool for FPGA research. In *International Workshop on Field Programmable Logic and Applications*. Springer, 213–222.

[4] Raghunandan Chaware, Kumar Nagarajan, and Suresh Ramalingam. 2012. Assembly and reliability challenges in 3D integration of 28nm FPGA die on a large high density 65nm passive interposer. In *2012 IEEE 62nd Electronic Components and Technology Conference*. IEEE, 279–283.

[5] Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-based systolic array auto-compilation. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[6] Yangdong Steven Deng and Wojciech Maly. 2003. Physical design of the "2.5 D" stacked system. In *Proceedings 21st International Conference on Computer Design*. IEEE, 211–217.

[7] Linfeng Du, Tingyuan Liang, Sharad Sinha, Zhiyao Xie, and Wei Zhang. 2023. FADO: Floorplan-Aware Directive Optimization for High-Level Synthesis Designs on Multi-Die FPGAs. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 15–25.

[8] Lorenzo Ferretti, Andrea Cini, Georgios Zacharopoulos, Cesare Alippi, and Laura Pozzi. 2022. Graph neural networks for high-level synthesis design space exploration. *ACM Transactions on Design Automation of Electronic Systems* 28, 2 (2022), 1–20.

[9] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. 2021. Autobridge: Coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die fpgas. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 81–92.

[10] Andre Hahn Pereira and Vaughn Betz. 2014. Cad and routing architecture for interposer-based multi-fpga systems. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*. 75–84.

[11] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[12] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. 2008. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1192–1195.

Just Accepted

[13] Xilinx Inc. 2016. UltraRAM: Breakthrough Embedded Memory Integration on UltraScale+ Devices. https://docs.xilinx.com/v/u/en-US/wp477-ultraram
[14] Xilinx Inc. 2020. Vitis High-Level Synthesis User Guide. https://docs.xilinx.com/r/2020.2-English/ug1399-vitis-hls/Optimization-Directives
[15] Xilinx Inc. 2020. Vitis High-Level Synthesis User Guide. https://docs.xilinx.com/r/2020.2-English/ug1399-vitis-hls/Overview-of-Arbitrary-Precision-Integer-Data-Types
[16] Xilinx Inc. 2020. Vitis High-Level Synthesis User Guide. https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Applying-Optimization-Directives-to-Templates
[17] Xilinx Inc. 2020. Vitis High-Level Synthesis User Guide. https://docs.xilinx.com/r/2020.2-English/ug1399-vitis-hls/Automatic-Loop-Pipelining
[18] Xilinx Inc. 2021. UltraScale Architecture Memory Resources User Guide. https://www.xilinx.com/support/documents/user_guides/ug573-ultrascale-memory-resources.pdf
[19] Xilinx Inc. 2022. Floorplanning With Stacked Silicon Interconnect (SSI) Devices. https://docs.xilinx.com/r/en-US/ug906-vivado-design-analysis/Floorplanning-With-Stacked-Silicon-Interconnect-SSI-Devices
[20] Hyegang Jun, Hanchen Ye, Hyunmin Jeong, and Deming Chen. 2023. Autoscaledse: A scalable design space exploration engine for high-level synthesis. *ACM Transactions on Reconfigurable Technology and Systems* 16, 3 (2023), 1–30.
[21] Richard M Karp. 1992. On-line algorithms versus off-line algorithms: How much is it worth to know the future?. In *Algorithms, Software, Architecture: Information Processing 92: Proceedings of the IFIP 12th World Computer Congress, Madrid, Spain*, Vol. 1. 416.
[22] Hans Kellerer, Ulrich Pferschy, and David Pisinger. 2004. The multiple-choice knapsack problem. In *Knapsack Problems*. Springer, 317–347.
[23] Moazin Khatti, Xingyu Tian, Yuze Chi, Licheng Guo, Jason Cong, and Zhenman Fang. 2023. PASTA: Programming and Automation Support for Scalable Task-Parallel HLS Programs on Modern Multi-Die FPGAs. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 12–22.
[24] Ariel Kulik and Hadas Shachnai. 2010. There is no EPTAS for two-dimensional knapsack. *Inform. Process. Lett.* 110, 16 (2010), 707–710.
[25] Tingyuan Liang, Jieru Zhao, Liang Feng, Sharad Sinha, and Wei Zhang. 2019. Hi-clockflow: Multi-clock dataflow automation and throughput optimization in high-level synthesis. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–6.
[26] Tomofumi Yuki Louis-Noel Pouchet. 2016. PolyBench/C. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/
[27] EG Co man Jr, MR Garey, and DS Johnson. 1996. Approximation algorithms for bin packing: A survey. *Approximation algorithms for NP-hard problems* (1996), 46–93.
[28] Fubing Mao, Wei Zhang, Bo Feng, Bingsheng He, and Yuchun Ma. 2016. Modular placement for interposer based multi-FPGA systems. In *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*. IEEE, 93–98.
[29] Silvano Martello and Paolo Toth. 1990. Bin-packing problem. *Knapsack problems: Algorithms and computer implementations* (1990), 221–245.
[30] Silvano Martello and Paolo Toth. 1990. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc.
[31] Pingfan Meng, Alric Althoff, Quentin Gautier, and Ryan Kastner. 2016. Adaptive threshold non-pareto elimination: Re-thinking machine learning for system level design space exploration on FPGAs. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 918–923.
[32] Ehsan Nasiri, Javeed Shaikh, Andre Hahn Pereira, and Vaughn Betz. 2015. Multiple dice working as one: CAD flows and routing architectures for silicon interposer FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 5 (2015), 1821–1834.
[33] Boaz Patt-Shamir and Dror Rawitz. 2010. Vector bin packing with multiple-choice. In *Scandinavian Workshop on Algorithm Theory*. Springer, 248–259.
[34] Nam Khanh Pham, Amit Kumar Singh, Akash Kumar, and Mi Mi Aung Khin. 2015. Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 157–162.
[35] Luca Piccolboni, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P Carloni. 2017. COSMOS: Coordination of high-level synthesis and memory optimization for hardware accelerators. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 5s (2017), 1–22.
[36] Chirag Ravishankar, Dinesh Gaitonde, and Trevor Bauer. 2018. Placement strategies for 2.5 D FPGA fabric architectures. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 16–164.
[37] Kirk Saban. 2011. Xilinx stacked silicon interconnect technology delivers breakthrough FPGA capacity, bandwidth, and power efficiency. *Xilinx, White Paper* 1, 1 (2011), 1–10.
[38] Benjamin Carrion Schafer. 2017. Parallel High-Level Synthesis Design Space Exploration for Behavioral IPs of Exact Latencies. 22, 4, Article 65 (may 2017), 20 pages. https://doi.org/10.1145/3041219
[39] Benjamin Carrion Schafer, Takashi Takenaka, and Kazutoshi Wakabayashi. 2009. Adaptive Simulated Annealer for high level synthesis design space exploration. In *2009 International Symposium on VLSI Design, Automation and Test*. 106–109. https://doi.org/10.1109/VDAT.

Just Accepted

2009.5158106

[40] Benjamin Carrion Schafer and Zi Wang. 2019. High-level synthesis design space exploration: Past, present, and future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2019), 2628–2639.

[41] Prabhakant Sinha and Andris A Zoltners. 1979. The multiple-choice knapsack problem. *Operations Research* 27, 3 (1979), 503–515.

[42] Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. 2022. Automated accelerator optimization aided by graph neural networks. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 55–60.

[43] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2022. AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 27, 4 (2022), 1–27.

[44] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[45] Jie Wang, Licheng Guo, and Jason Cong. 2021. Autosa: A polyhedral compiler for high-performance systolic arrays on fpga. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 93–104.

[46] Zi Wang, Jianqi Chen, and Benjamin Carrion Schafer. 2020. Efficient and robust high-level synthesis design space exploration through offline micro-kernels pre-characterization. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 145–150.

[47] Nan Wu, Yuan Xie, and Cong Hao. 2021. Ironman: Gnn-assisted design space exploration in high-level synthesis via reinforcement learning. In *Proceedings of the 2021 on Great Lakes Symposium on VLSI*. 39–44.

[48] Nan Wu, Hang Yang, Yuan Xie, Pan Li, and Cong Hao. 2022. High-level synthesis performance prediction using gnns: Benchmarking, modeling, and advancing. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 49–54.

[49] Xilinx. 2022. Vitis Accelerated Libraries. https://github.com/Xilinx/Vitis_Libraries/

[50] Xilinx. 2023. Vitis Data Center Acceleration Examples: Stream Chain Matrix Multiplication (C). https://github.com/Xilinx/Vitis_Accel_Examples/blob/main/host_xrt/kernel_chain/README.rst

[51] Sotirios Xydis, Christos Skouroumounis, Kiamal Pekmestzi, Dimitrios Soudris, and George Economakos. 2010. Efficient high level synthesis exploration methodology combining exhaustive and gradient-based pruned searching. In *2010 IEEE Computer Society Annual Symposium on VLSI*. IEEE, 104–109.

[52] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2022. Scalehls: A new scalable high-level synthesis framework on multi-level intermediate representation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 741–755.

[53] Cody Hao Yu, Peng Wei, Max Grossman, Peng Zhang, Vivek Sarker, and Jason Cong. 2018. S2FA: An accelerator automation framework for heterogeneous computing in datacenters. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[54] Georgios Zacharopoulos, Andrea Barbon, Giovanni Ansaloni, and Laura Pozzi. 2018. Machine learning approach for loop unrolling factor prediction in high level synthesis. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 91–97.

[55] Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang, and Bingsheng He. 2017. COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 430–437.

[56] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. 2018. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software-Programmable FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)* (Feb 2018).

Just Accepted