REFINE: Runtime Execution Feedback for INcremental Evolution on FPGA Designs

Dongjoon Park
University of Pennsylvania
Philadelphia, PA, USA
dopark@seas.upenn.edu

André DeHon
University of Pennsylvania
Philadelphia, PA, USA
andre@acm.org

ABSTRACT

FPGA design optimization is challenging for developers for two main reasons. First, developers cannot easily identify a bottleneck of the design to know where to focus optimization effort to improve the application execution time. Second, slow, monolithic FPGA compilation makes evaluation of each design change costly. Together, these make FPGA development different and more challenging than traditional software development where software engineers are accustomed to using rich profiling tools to improve their designs through a series of quick, incremental refinements. To address these issues, we propose a fast bottleneck identification scheme using runtime feedback and separate FPGA compilation. Our scheme systematically identifies bottlenecks in streaming computations based on FIFO event counters extracted from hardware execution and guides developers to the operations that limit performance. We showcase our support for bottleneck identification with the fast, automatic design space exploration, iterating initial design points quickly with a separate, incremental compilation strategy. When the design reaches the point that latency cannot improve with the separate compilation approach, we migrate to the monolithic design flow that does not have the area overhead and communication bandwidth limit of separate compilation approach. Then, the remaining design space, if any, is explored with a monolithic flow. When tested on the AMD ZCU102 embedded platform with realistic HLS dataflow designs, our approach correctly identifies bottlenecks improving application latency 2.2–12.7× while reducing tuning time by 1.3–2.7× compared to monolithic flow.

CCS CONCEPTS

• **Hardware → Reconfigurable logic and FPGAs**.

KEYWORDS

FPGA, Incremental Refinement, Design-Space Exploration, Partial Reconfiguration

ACM Reference Format:
Dongjoon Park and André DeHon. 2024. REFINE: Runtime Execution Feedback for INcremental Evolution on FPGA Designs. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '24), March 3–5, 2024, Monterey, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3626202.3637560

FPGA '24, March 3–5, 2024, Monterey, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0418-5/24/03
https://doi.org/10.1145/3626202.3637560

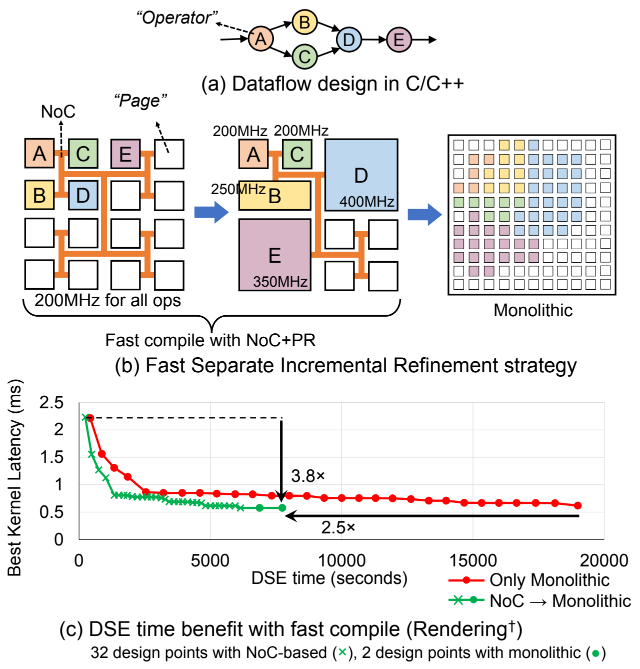
"Operator"
A
B
C
D
E
(a) Dataflow design in C/C++
NoC
"Page"
200MHz for all ops
200MHz 200MHz
250MHz
400MHz
350MHz
Monolithic
Fast compile with NoC+PR
(b) Fast Separate Incremental Refinement strategy
Best Kernel Latency (ms)
DSE time (seconds)
3.8×
2.5×
Only Monolithic
NoC → Monolithic
(c) DSE time benefit with fast compile (Rendering†)
32 design points with NoC-based (×), 2 design points with monolithic (●)

Figure 1: Fast incremental refinement strategy with runtime bottleneck identification

1 INTRODUCTION

FPGAs excel in many applications with massive parallelism and flexibility. The recent advancement of High-Level-Synthesis (HLS) lowers the entrance barrier of FPGA design so that software engineers without hardware expertise can program FPGA applications. Yet, designing an optimized FPGA design has never been an easy task even for experts.

Software engineers are used to profiling their code to identify where time is going in their applications, allowing them to identify bottleneck functions that can be revised to accelerate performance. Furthermore, when they make a change, incremental compilation allows them to quickly recompile only the functions that have changed and re-profile their designs. This allows the software engineering to quickly iterate and refine their applications to improve performance. In contrast, when the components of an FPGA application are concurrently resident on the FPGA, there is less visibility into which operators are limiting application performance. In addition, when the programmer makes a change, the entire FPGA design must be recompiled *monolithically*, an operation that can typically take hours. This prevents the FPGA developer from quickly identifying bottlenecks and iterating to improve the design.

To narrow the gap between FPGA development and software development, several works have recently proposed to utilize a packet-switched Network-on-Chip (NoC) and Partial Reconfiguration (PR), compiling each section of the device in parallel [28, 29, 37, 38]. Different *operators*, compute blocks in the dataflow graph, are mapped to different partially reconfigurable regions, *pages*, and operators communicate through the pre-built NoC overlay. This separate place/route and linking strategy, similar to software compilation, decreases FPGA compile time with a divide-and-conquer strategy and recompiles only the changed operators. However, programmers still do not have access to the inner state of the application and cannot identify what part of the codes to refine. The question is, *can we identify the bottleneck of the application along with the fast compilation to iterate through the initial design points quickly on the path to an optimized monolithic FPGA design?*

Fig.1 (b) shows the incremental refinement strategy integrated with our fast bottleneck identification based on runtime feedback from the hardware. The goal is to quickly map the HLS dataflow application on the FPGA with the previously suggested NoC-based strategy and to incrementally refine each operator. In this process, the bottleneck operator is identified on each iteration and the sizes of bottleneck operators may increase to exploit more parallelism on the hardware. However, the infrastructure added for the NoC based system may add its own bottlenecks and prevent the design from fully using all the FPGA resources for the application. When the design space for the bottleneck operator is all explored or the design is expected to use more resources than available on the PR pages, we remove the NoC and optimize for the final monolithic design, which directly connects the operators with FIFOs using a similar bottleneck identification scheme. Instead of using potentially inaccurate models for application performance or resource utilization, we always have a working copy of the application whose performance we can measure to provide feedback to identify the next bottleneck that needs attention.

To illustrate our fast bottleneck identification and refinement capabilities, we use them in a simple, greedy automated Design Space Exploration (DSE) experiment. This experiment removes the human from the loop to focus entirely on (1) the accuracy of the bottleneck identification and (2) the time required to automatically identify bottlenecks and recompile refinements. Two major *knobs* in the design space are design parameters and kernel clock frequency. One example of design parameters are HLS pragmas that are added to the source code. In both the NoC-based system and monolithic system, we explore different kernel clock frequencies (200MHz–400MHz) per operator so that an operator with the lowest F_{max} does not limit the operating frequency of the rest of the design. Fig.1 (c) shows one of the case studies for the greedy, automated DSE. In our fast incremental refinement strategy (green), DSE starts with the fast, separate compile framework using a pre-built NoC. After each compilation, the design runs on the hardware, the bottleneck operator is identified, and the next design point is selected. In Fig.1 (c), both incremental (green) and monolithic (red) correctly identify the operators limiting performance and tune their parameters to achieve a 3.8× reduction in runtime. In the incremental strategy, the NoC-based fast compile iterates 32 design points (marked as ×) then iterates 2 design points (marked as ●) with the regular monolithic compilation all in less than 2.2 hours, leading to an overall 2.5× speedup in design tuning over a purely monolithic flow.

The contributions of this work include:

- We provide a fast, automatic runtime bottleneck identification scheme based on the FIFO counters, along with the tools to automate their insertion and bottleneck identification (Sec. 4).
- We demonstrate an incremental refinement strategy that iterates initial design points with the fast, NoC-based, separate compilation and migrates to the optimized monolithic FPGA design (Sec. 7).
- We enhance the NoC-based system including the support for the multiple clock frequencies for the pages (200MHz–400MHz), utilization of multiple NoC interfaces, and page assignment based on graph bi-partitioning (Sec. 5).
- We evaluate our incremental refinement strategy with automatic DSE case studies on Rosetta benchmarks [42] and FINN [4, 34] where bottleneck identification selects which operators and parameters to adjust to incrementally improve performance (Sec. 8). We show that the kernel execution time improves 2.2–12.7×, continually using the feedback from each compilation. The design space exploration time was 1.3–2.7× shorter than when the design is monolithically compiled every time.

We provide an open-source distribution for our tools.[1]

2 BACKGROUND

2.1 HLS Design Space Exploration on FPGA

There have been numerous works on HLS DSE [5, 11, 16, 23–25, 30, 32, 40]. HLS DSE is generally categorized into two approaches, model-based and synthesis-based. Model-based techniques build predictive models for performance and resource, so they can quickly search the design space. However, model-based techniques are inaccurate compared to synthesis-based methods that invoke HLS tools to evaluate the design point, and inaccurate models could lead to a suboptimal design, under-utilizing or over-utilizing resources for the given platform. Synthesis-based methods use results in the early stages of the hardware mapping like post-HLS estimates. They are more accurate than model-based methods at the expense of runtime, but there is still a huge gap between post-HLS estimates and the actual placed and routed designs [7]. The problem with post-HLS estimates is exacerbated for the data-dependent applications. For example, when there are variable loop bounds, AMD Vitis_HLS does not report the trip counts as the values are unknown at compile time [3, 6].

Our strategy (Fig. 1 (b)) is different from the aforementioned approaches because we place, route, and run the design on the FPGA and incrementally refine the design. We perform bottleneck-guided optimization, similar to the strategy used in [32]. Instead of relying on post-HLS estimates as done in [32], we use the feedback from the placed and routed design that runs on the hardware. [23] uses a domain-specific overlay to avoid repetitive FPGA compilation in DSE, but our approach accelerate FPGA compilation directly

[1]https://github.com/icgrp/prflow_REFINE

to the FPGA substrate, avoiding the overhead introduced by the domain-specific overlay.

2.2 Partial Reconfiguration

Partial Reconfiguration (PR) is a technology that reconfigures part of the reconfigurable fabric while other parts of the design are still operating [15, 39]. PR design consists of *static logic* that does not change, and *reconfigurable logic* that can be reconfigured. Compiling a small reconfigurable design on a small PR region should generally be much faster than compiling a large design on the entire chip due to the smaller problem size. AMD's *Abstract Shells* load the minimal logical and physical database to realize more of the potential benefit of mapping a small PR region. *Hierarchical PR* supports subdividing a single reconfigurable partition into multiple reconfigurable partitions [15, 39], leading to finer-grained PR.

2.3 Separate Compilation on FPGA

One of the reasons that current FPGA development is different from software engineering is the long FPGA compilation. Recently, many academic publications address slow FPGA compilation with a divide-and-conquer approach. The common idea is to split the device into multiple sections and compile each section in parallel since compiling small sections in parallel is faster than monolithically compiling the entire device, utilizing more cores in a multi-core workstation or compute server.

One main approach is to use a pre-routed overlay with PR. Authors in [29, 36–38] employ a pre-routed NoC or switchboxes to support separate compilations on partially reconfigurable pages. The advantage of this approach with a pre-routed overlay is that the global inter-page routing is not required after each page is separately compiled. This means that only the necessary pages can be recompiled similar to how software processors support separate compilation and linking so that only necessary files or functions are recompiled. The disadvantage of this approach is limited bandwidth and the area overhead introduced by the NoC.

Another main approach is to compile each *island* on FPGA in parallel and utilize an open source CAD tool like RapidWright [21] to stitch the compiled blocks together [14, 27, 33]. Authors in [14] demonstrate significant compile time speedup and improve the maximum clock frequency of the application. The advantage of this approach is that the design does not have extra design elements like NoC, which could result in bandwidth bottleneck or area overhead. The disadvantage of this approach is the global stitching process that integrates separately compiled islands which can take half an hour to an hour for current designs. As the size of islands decreases or the FPGA capacity grows, the final routing time is expected to grow. This limitation is detrimental especially in the incremental development since a very minor edit in the design would require top-level routing. In [13], a follow-up work of [14], authors show their work-in-progress effort to use pre-routed inter-island routing with PR to remove the final stiching time at the expense of an earlier global routing phase.

To support software-like incremental refinement, we build upon separate compilation using PR that recompiles the modules that change instead of going through the global routing. We adopt [28] that utilizes a pre-built NoC and Hierarchical PR pages because variable-sized pages can support continually changing sizes of operators in DSE (in Fig.1 (b), A,C are mapped to *single-sized* pages, B is mapped to a *double-sized* page and D,E are mapped to *quad-sized* pages). In [28], we demonstrate how the incremental development could be done, showing a single compile iteration could take less than 2 minutes when a single PR page is recompiled. However, the incremental refinement in [28] is manual, and bottleneck identification requires user insight.

2.4 Streaming Computations

We focus on streaming dataflow designs (Fig. 1 (a)) described at the C/C++ level that consists of operators and streaming links [9, 17]. Both the NoC-based system and the monolithic system include asynchronous FIFOs in the stream links between the operators so that operators can run at different rates. The NoC-based system has a pre-implemented NoC between the FIFOs to decouple the implementations of each operator while operators are directly connected with FIFOs in the monolithic system.

3 INCREMENTAL REFINEMENT STRATEGY

Our strategy is to start with the separately-compiled, NoC-based system design to evaluate the design on hardware. We support a fast incremental refinement approach by automatically identifying bottlenecks and incrementally recompiling only the operators revised to address the bottleneck.

We automatically add FIFO counters on the stream connections into and out of operators to monitor the design and identify bottlenecks. The support for bottleneck identification (Sec. 4) is the key component to guide users or an automated flow that other split FPGA compilation works lack.

When bottlenecks are identified, we can refine the operators or the NoC communication mapping for the streams between operators. To resolve the NoC bandwidth bottleneck, we support multiple NoC interfaces for an operator or operator merging to directly connect high bandwidth operators avoiding the need for the NoC on those links (Sec. 5).

The user or automation selects a new design point by refining either the bottleneck operator or the NoC bandwidth bottleneck. When a new design point is generated, we recompile only the necessary operators. This gives us a new design which can then be profiled with the FIFO counters to identify the next bottleneck.

This iteration continues until the design space is explored for the bottleneck operator or the design needs more resources than available in the PR pages. Finally, a design is compiled with the monolithic system that also integrates FIFO counters to identify the bottleneck (Sec. 6) and the iteration continues. This strategy can help the users to quickly iterate the important, initial design points, or the flow can be automated to output an optimized design in the given design space (Sec. 7, Sec. 8).

4 BOTTLENECK IDENTIFICATION

Building on prior work [6, 31], we insert counters on stream FIFOs and use those counters to identify likely bottlenecks. Operators in both the NoC-based system and the monolithic system have input and output FIFOs. In the NoC-based system, these FIFOs reside in the NoC interface which is then connected with the NoC. In the monolithic system, operators are directly connected with these FIFOs. In a streaming model, one or more operators could limit the throughput of the entire application. These bottleneck operators have lower throughput than their predecessors and their successors. As a result, they tend to have relatively empty output FIFOs and relatively full input FIFOs. Our tool instantiates different counters to monitor the status of the FIFOs and use them to identify the bottleneck operator or the NoC bandwidth bottleneck without any impact on the performance. The idea is similar to `-pg` option in the software compiler that causes each function to call `mcount` routine whose results are later profiled by a program like `gprof` [12].

[31] has bottleneck identification on a cluster of "Computing Elements" but does not attempt to systematically utilize the feedback to affect the next design point. [6] modifies HLS source codes to monitor the modules under analysis and detects the cause of the stall. We do not modify the source code; we identify bottlenecks in software based on the raw counter data collected. Unlike previous works, we demonstrate the incremental refinement on realistic applications using bottleneck identification (Sec. 8), and our work is embedded with the fast separate compile so that the runtime hardware execution feedback is obtained quickly.

4.1 Stall counters

Both the initial NoC-based system and the final monolithic system utilize *stall counters* to identify the bottleneck operator as shown in Fig. 2 (a). The input stall condition is defined as the state when *the input FIFO is empty and the user operator asserts a ready signal.* At a high level, it means that the operator wants to process the data, but the data is not available, so the operator stalls. Similarly, the output stall condition is defined as the state when *the output FIFO is full and the user operator asserts a valid signal for the data.* At a high level, it means that the operator wants to output the data, but the successor is still busy processing the previous data. The stall condition for the operator is asserted when *at least one input FIFO has a stall condition or at least one output FIFO has a stall condition.* Finally, the stall counter increments when the stall condition is asserted. We know that as the number of stall counters for an operator is low, the operator is likely to be the bottleneck because this operator is busy processing some data while other operators are waiting for the input data or waiting to output the data.

4.2 Full counters

In the NoC-based system, each operator has a single, limited-bandwidth input channel into the NoC and a single limited-bandwidth output channel; these can be narrower than the total input and output width needed by the application. This limited NoC bandwidth could limit the application performance. A NoC bottleneck can be detected similarly with full counters on the FIFOs associated with the stream links into and out of each operator. Full counters increment when the FIFO is full. In Fig. 2 (b), operator A sends data to operator B through the NoC. If operator A's output FIFO has large full counters and operator B's input FIFO has small full counters, we can assume that the NoC bandwidth could be a bottleneck. This means that operator A tries to send out the data often, but operator B does not receive the data at a similar rate. In our system, we consider there exists NoC bandwidth bottleneck *if the difference in the full counters (output FIFO's full counters of A - input FIFO's full counters of B) is large enough.*

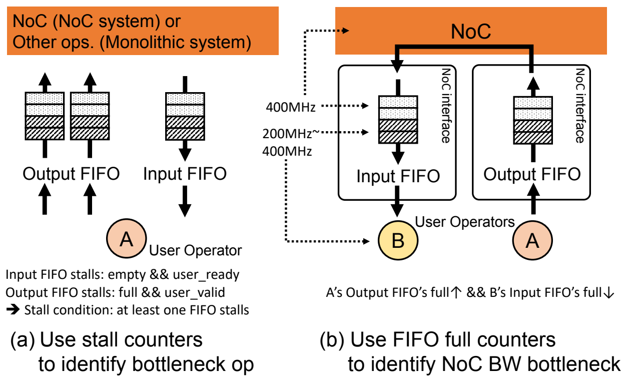
NoC (NoC system) or
Other ops. (Monolithic system)
Output FIFO
Input FIFO
A
User Operator
Input FIFO stalls: empty && user_ready
Output FIFO stalls: full && user_valid
➔ Stall condition: at least one FIFO stalls
(a) Use stall counters to identify bottleneck op
NoC
400MHz
200MHz~
400MHz
NoC interface
Input FIFO
Output FIFO
NoC interface
B
User Operators
A
A's Output FIFO's full↑ && B's Input FIFO's full↓
(b) Use FIFO full counters to identify NoC BW bottleneck

Figure 2: Bottleneck Identification with FIFOs

4.3 Resource usage

Logic related to FIFO counters is implemented in RTL along with NoC interface (NoC-based) or the top-level wrapper function (monolithic, Sec. 6). In the NoC-based system, a NoC interface and a user operator are mapped in a PR page. When the counters are set to 28-bit registers, a NoC interface with a single user input stream (32 bit) and a single user output stream (32 bit) costs about 700 LUTs, 1000 FFs, and 4 36Kb BRAMs including counters logic. Logic related to counters alone uses about 200 LUTs and 400 FFs. The size of single-sized pages in [28] is about 7,000–8,400 LUTs, so the counters logic is about 2% of the LUTs in a single-sized PR page. In the monolithic system, logic related to counters uses about 60 LUTs and 140 FFs per operator. One reason for the discrepancy in resource usage is that the NoC-based system uses about double the number of FIFOs than the number of FIFOs used in the monolithic system. For instance, for operators A and B in Fig. 4 (b), in the NoC-based system, they need one output FIFO for A and one input FIFO for B. In the monolithic system, on the other hand, one FIFO between A and B is enough. Among the benchmarks in our experiments (Sec. 8), Rendering has the largest number of streams (30 streams) thereby consuming the most resources for counters logic. In the final monolithic design of Rendering, counters logic is only 3.4% (1900 LUTs) of the final design's total LUT utilization, and only 5.3% (3500 FFs) of the FF utilization.

4.4 Limitations

It is possible to have a stall counter per input and output stream to identify the problematic stream for finer-grained analysis. For simplicity, we keep a single stall counter per operator and identify the bottleneck operator.

For the same run time, an operator running at 400MHz can have up to twice as large stall/full counters as an operator running at 200MHz. Therefore, we normalize counters by dividing them by the operating frequencies, but this simple approach may not be sufficient to reflect the differences in operating frequencies. Furthermore, an operator with different rates of input and output may be harder to classify correctly as the low rate side is less likely to fill a

FIFO than the high rate side. Although our bottleneck identification based on FIFO counters is an approximation, in Sec. 8, we show how our approach based on FIFO counters identifies bottleneck operators and resolves NoC bandwidth bottleneck in incremental development for realistic HLS designs, improving the application performance by 2.2–12.7×.

5 NOC-BASED SYSTEM

An assumption with using the incremental refinement strategy to accelerate optimized designs that will ultimately be implemented monolithically is that design points that the NoC-based system explores are similar to those the monolithic system would have explored. To narrow the gap between the two, we address the previous NoC-based system issues including the NoC bandwidth limit, NoC congestion and resource fragmentation.

5.1 Knobs in NoC-based system

As noted, the NoC-based system introduces its own performance artifacts and bottlenecks. In this section, we introduce potential optimizations and tuning knobs that can be applied to minimize the impact of the NoC. Developers can directly tune these knobs. However, since these optimizations require a detailed understanding of the NoC, it is less likely the software engineer will understand them and when they are necessary; even a hardware engineer thinking primarily about the final, monolithic design and not the details of the NoC overlay may find them difficult to understand. Consequently, it is particularly valuable to automate the tuning of these knobs as illustrated in the automated DSE case studies in Sec. 7 and Sec. 8.

5.1.1 Multiple NoC interfaces. [28] utilizes Hierarchical PR to support variable-sized (single, double, quad) PR pages by recombining smaller pages to create larger pages. Yet, in [28], the number of NoC interfaces is limited to 1 even after multiple PR pages are recombined. Because this limitation could lead to a NoC bandwidth bottleneck when the communication is heavy, we add support for multiple NoC interfaces for recombined pages. Fig. 3 (a) is the example that the user operator A is using two NoC interfaces when A has two input streams (32 bits, 64 bits) and three output streams (32 bits, 32 bits, 64 bits). The distribution of the streams to the multiple NoC interfaces is statically determined in a way to minimize the standard deviation of sums of the stream widths. In simple words, in Fig. 3 (a) case, we distribute two 32-bit output streams to one NoC interface and one 64-bit output stream to another NoC interface because the sums of the output streams per a NoC interface are equally 64. Future work can explore distributing the streams dynamically based on the operator's read and write rates.

5.1.2 Merging. If a NoC bandwidth bottleneck is detected and a larger number of leaf interfaces does not help, users or the automation script can simply merge two operators whose connection seems to suffer from the limited NoC bandwidth. This removes the NoC bandwidth limitation, at the cost of a larger page that will be slower to compile. In Fig. 3 (b), two operators A and B are merged, and 144 bits of data packets then do not need to be routed over the NoC. Among the two operators, the sender operator (A) and the receiver operator (B), the receiver operator (B) becomes the new

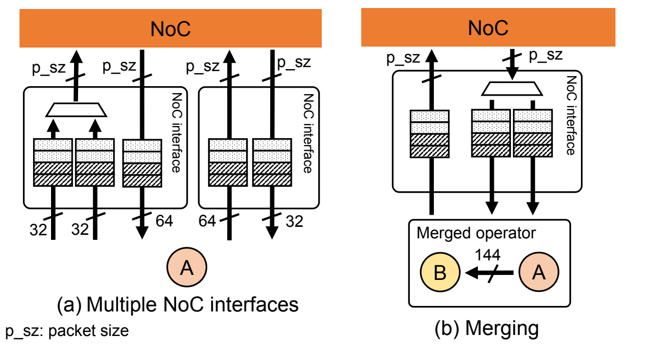

Figure 3: Knobs in NoC-based system to mitigate NoC bandwidth bottleneck

"representative" operator for the new merged operator and inherits the parameters from both operators.

5.1.3 Multiple clock frequencies. The reason multiple kernel clock frequencies is supported is not to close the gap between the NoC-based system and the monolithic system but just to create a more performant FPGA design with our incremental strategy. It also supports incremental refinement as we can recompile one operator at a time for a different frequency, rather than needing to recompile all the operators to try out an increased frequency. While [28] supports a single 200MHz clock for both NoC and user operators, we support 200MHz, 250MHz, 300MHz, 350MHz, 400MHz for user operators. NoC and NoC interfaces run on the maximum 400MHz, and asynchronous FIFOs are used to connect the NoC clock domain and the user operator's clock domain. We insert pipeline registers for the data signals and skid buffers for the ready signals between the NoC and the NoC interface to create a heavily pipelined system that runs at 400MHz. If we do not constrain the placement of the pipeline registers, these registers are randomly placed. Then, the tool (Vivado) could struggle to meet the timing when generating the NoC-based system's overlay, or even if it successfully generates the overlay, when user operators are mapped to the PR pages, the design may struggle to meet the timing since pipeline registers are not placed close enough. For this reason, we create *pblocks* right next to each PR page which constrain the placement of pipeline registers. Different clock frequencies significantly improve the application performance. In Sec. 8.3, the final designs' latencies are 1.3× (Rendering), 1.3× (Digit Recognition), 1.9× (Optical Flow), 2.2× (CNN-1), 2.2× (CNN-2) better than latencies when the clock frequencies are fixed to the lowest 200MHz.

5.2 Enhancements in NoC-based system

This section includes the pure enhancements for the NoC-based system over the previous works. The objectives of these enhancements are to make a smooth continuum between the NoC-based system and the monolithic system and to allow as many design points as possible to iterate in the NoC-base system.

5.2.1 Page Heterogeneity. In the NoC-based system, PR pages are *heterogeneous*, which means that even if they are the same single-sized pages, available logic and routing resources are different. One reason for the heterogeneous PR pages is irregular columnar

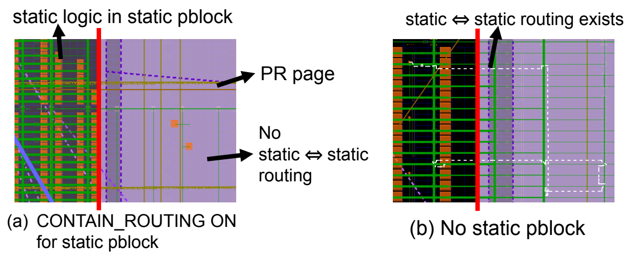
static logic in static pblock
PR page
No static ⇔ static routing
(a) CONTAIN_ROUTING ON for static pblock
static ⇔ static routing exists
(b) No static pblock

Figure 4: Snapshots of NoC-based system's PR page

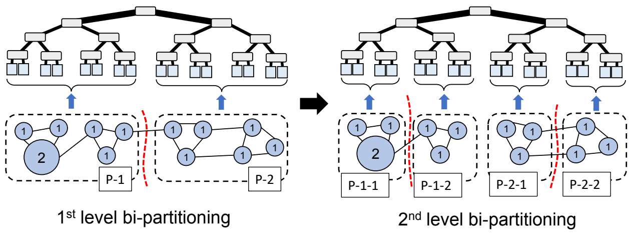

Figure 5: Page Assignment based on recursive bi-partitioning

resource distribution on modern FPGAs. Another reason is that modern PR technology from vendor tools allows the static design to route over reconfigurable regions [15, 39], thereby stealing routing resources from the PR pages and blocking some logic resources on the pages. This heterogeneity complicates the page assignment algorithm which will be explained in Sec. 5.2.3.

5.2.2 Static Routing over PR pages. We reduce the effects of static routing over PR pages by creating a pblock for non-pages elements (the NoC, AXI interconnect, peripherals) and setting the `CONTAIN_ROUTING` property on. This requires Vivado to create a hierarchy in the block diagram for non-pages logic and assign the newly created cell to the pblock. With this setting, routing whose source is in static design and destination is in static design (highlighted in white in Fig. 4 (b)) will be prevented from the reconfigurable regions. Fig. 4 (a) shows the snapshot of a PR page when the static logic is added to a pblock that has CONTAIN_ROUTING property on. The remaining green routing on the PR page shown is static⇔reconfigurable routing or global clock signals.

5.2.3 Page Assignment. [28]'s separate compilation framework runs HLS and logic synthesis for each operator in parallel. Then, it synchronizes to assign synthesized netlists to appropriate PR pages based on post-synthesis resource estimates to launch parallel placement and routing. To address heterogeneous page capacity, its *capacity-based* page assignment sorts the operators in descending order in size and assigns the "tightest" page for the operator. This greedy page assignment solely based on capacity could potentially lead to NoC congestion if logically adjacent operators are placed far apart from each other in the NoC. Butterfly Fat Tree (BFT) NoC [8, 18, 22] is used in [28], and a simple node placement to mitigate the congestion on BFT NoC is a placement based on recursive graph bi-partition. As BFT has a hierarchical structure, we can bi-partition the dataflow graph *to minimize the traffic between two partitions*, and we assign operators in one partition to one subtree and assign operators in another partition to another subtree. Fig. 5 shows how recursive bi-partition is used in the page assignment. The numbers inside the operators indicate "weights", the expected sizes of the PR pages that the operator will use based on the post-synthesis resource estimates. Although Fig. 5 shows only two levels of bi-partition, we keep bi-partitioning the subtrees until there is only one operator in the subtree. This approach could reduce unnecessary global traffic over the NoC. We choose the fast, heuristic-based page assignment algorithm because we want the algorithm to work reasonably well and the algorithm to run fast so that it does not slow the fast separate compile approach.

We use `metis` software [19] to perform graph bi-partitioning. Our input graph has weighted vertices that indicate the sizes of the pages (single, double, or quad) that each operator is expected to use. The capacity-based page assignment that assigns the tightest page to the operators runs first to assign weights to the operators. After every bi-partition, we make sure that operators are mappable in the subtree, and at this point, because of heterogeneity in the PR pages, the weights of the operators could increase (e.g., an operator that was mappable to a single page cannot find a large enough single page in the newly assigned subtree and now needs to be mapped to a double page). If the two partitions cannot be mapped to the initially assigned subtrees, we try swapping subtrees for each partition. While recursively bi-partitioning the graph, if there is one operator left to map in a subtree, instead of mapping to the tightest page, we assign a larger page in the subtree to the operator. If there is no valid page assignment using the recursive-bisection approach, the page assignment algorithm performs capacity-based page assignment. If the capacity-based mapping does not find a valid page assignment, we exit the page assignment, notifying the user or the automation script that there is no valid page assignment. In the incremental refinement scenario, if the previous page assignment is still expected to map the refined design, the algorithm uses the previous page assignment so that we do not unnecessarily recompile the unchanged operator in the different PR page.

5.2.4 IsFit classifiers. Simple resource counts are not sufficient to decide if a synthesized netlist will fit in a PR Region. Differences in routing complexity (e.g., Rent exponent [20]) will determine how tightly the resources (LUT, BRAM, DSP) can be packed into the PR region. This is further complicated by the loss of routing resources due to connections outside the PR region (Sec. 5.2.1). To determine whether a synthesized netlist fits a specific PR page or not, we train a classifier *per PR page* to predict whether a netlist would fail in the implementation or not. To generate datasets for the training, we traverse different parameter values for Rosetta benchmarks [42] and use integrated top functions of multiple submodules to create diverse designs as done in [41]. Features include post-synthesis resource estimates (LUT, BRAM and DSP), and complexity characteristics (Rent value, average fanout, and total instances). If all LUT, BRAM and DSP resource estimates are lower than 60% of the resources available in the PR page, we assume that the netlist can be successfully placed and routed. Otherwise, we use our classifiers to make a decision. For each PR page's training and test data, we select netlist with over 60% of the PR page in LUT utilization (post-synthesis estimates). The average number of training and test datasets for each page is 2,572 (80% training, 20% test), and place/route results become the labels. We use Random Forest model for all the classifiers and perform a grid search to find the best hyperparameter for each classifier. Our page assignment algorithm based on graph bi-partitioning and isFit classifiers finishes in less than a second.

6 MONOLITHIC SYSTEM

The source codes for NoC-based system and monolithic system are the same, but in the monolithic system, each operator is directly connected through FIFOs. Like operators in the NoC-based system, operators in the monolithic system can also run on different frequencies (200MHz–400MHz), and the read and write frequencies for the FIFOs differ accordingly. HLS for operators compile in parallel as done in NoC-based system. Then, the generated RTL modules are collected and instantiated in a single top-level wrapper function with FIFOs and counter logic mentioned in Sec. 4. Logic synthesis and implementation are done monolithically. The monolithic system has similar AXI interconnect (runs 300MHz) and peripheral infrastructure to the NoC-based system. The monolithic infrastructure is a synthesized netlist that is placed/routed along with the generated monolithic top-level module.

7 AUTOMATED DSE CASE STUDY

Users can refine early designs based on runtime feedback information with the NoC-based system and then monolithically compile the final designs as our incremental refinement strategy suggests. While the strategy is still useful when the users hand-tune the design, we introduce automated DSE case studies that remove the human from the incremental refinement loop. Automation is useful, especially for software programmers who are not familiar with hardware design with multiple clocks or the NoC bandwidth bottleneck issue. The automated case further allows us to illustrate the potential impact on design iterations in the limit case where the time for developer changes is trivial.

7.1 DSE Experiment Overview

Fig. 6 shows the overview of our automated DSE experiment. User inputs of the system are parameter design space (`param_space.json`) that lists possible values for different parameters and HLS source code generator. We explore design parameters and kernel clock frequency per operator. Design parameters are application-specific like parallelization factor, and possible values for kernel clock frequency are 200MHz, 250MHz, 300MHz, 350MHz, and 400MHz. The HLS source code generator could be an external framework like FINN from AMD Research [4, 34]. Larger parallelization factor (e.g. 2) could mean either a larger operator (twice as large) or multiple (two small operators) operators. This transformation is described in the source code generator. The interconnection of the operators should also be defined in the source code generator.

The generated source codes are the input of either NoC-based flow or monolithic flow which is equipped with the counter-based bottleneck identification. After partial bitstreams (NoC-based flow) or a full bitstream (monolithic flow) are generated, the bitstreams are copied to the hardware along with the host executable. Then, performance metrics like latency and accuracy are measured. The

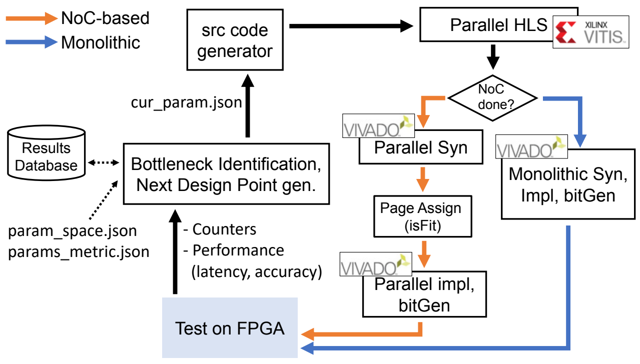

Figure 6: Automated DSE Experiment Overview

files containing performance metrics and FIFO counters are copied back to the host machine. The tuner records the results, identifies the bottleneck, and selects the next design point (Sec. 7.2).

7.2 Greedy Tuner

The tuning algorithm used in our case studies for both NoC-based system and monolithic system is as simple as identifying the operator with the least stall count and selecting the next design point that can improve the kernel execution latency. If the previous design point results in implementation failure, we revert to the most recent successful design point, and then select the next design point. Also, when the previous design point worsens the application latency, then we revert the design point. The previous latency could be just slightly worse than the best latency so far because of noise, so we relax the greedy nature of the algorithm by having a small margin (10% in the experiments). As explained in Sec. 4.2, NoC bandwidth bottlenecks can also be detected with full counters in the NoC-based system. If NoC bandwidth bottleneck is detected, then we increase the number of leaf interfaces or merge the two operators whose connection stream suffers from the limited NoC bandwidth as stated in Sec. 5.1. We prioritize NoC bottleneck over bottleneck operator so that any NoC bottleneck is resolved immediately, and the correct bottleneck operator is identified. When no NoC bottleneck is detected in the NoC-based system, or in the case of a monolithic system, we identify the bottleneck operator and select the design point that can improve the latency and has not been tried yet. When identifying the bottleneck operator, we search for the list of bottleneck operators that have similar small stall counts (10% in the experiments) instead of a single bottleneck operator with the least stall counters. In the NoC-based system, when we cannot improve the bottleneck operator anymore, then we move to a monolithic system. In the monolithic system, when we cannot improve the bottleneck operator anymore, we end the DSE.

There are design parameters that can improve the quality of the results but worsen the execution time. For example, in the K-Nearest-Neighbor (KNN) algorithm of Digit Recognition application in Sec. 8.3, a larger K value can improve the accuracy but increase the execution time. We provide `params_metric.json` that annotates the metrics the specific parameter can affect. For example, Digit Recognition's `params_metric.json` hints that K value is related to the accuracy and parallelization factor is related to the latency (`K_CONST: “accuracy”, PAR_FACTOR: “latency”`). We also provide the minimum accuracy for the benchmarks if applicable, and the tuner prioritizes to achieve the minimum accuracy instead of tuning for bottleneck operators to improve latency. Clock frequency is explored after design parameters are explored. Therefore, we tune for the design parameters like `K_CONST` first to achieve the minimum accuracy and tune for `PAR_FACTOR` and then clock frequencies.

Some benchmarks could have almost *identical* operators. For example, maybe designs have data parallel sections where one can allocate a number of identical data parallel operators. By default, our script updates parameter values of identical operators together since independently refining one operator at a time would unnecessarily take a long time. Nevertheless, if the computation is data-dependent, separately tuning each operator could be useful because one operator that has heavy computation load may need to run at 350MHz while another identical operator with a light load can run at 200MHz. In Sec. 8.3's Rendering benchmark, we will show the DSE results of both approaches.

8 EXPERIMENT

In this section, we evaluate the incremental refinement strategy with automated DSE case studies. We evaluate how both the fast, incremental flow and monolithic flow improve the application performance with our bottleneck identification. We also compare both flows in DSE time.

8.1 Experiment Setup

We create the NoC-based overlay on AMD ZCU102 evaluation board featuring UltraScale+ ZCU9EG FPGA. We use ZCU102 DFX platform [2] for the NoC-based system as we partially reconfigure each PR page separately in parallel. In the ZCU102 DFX platform we use, the dynamic region, the area that can be partially reconfigured, contains 262,496 LUTs, 1,752 18Kb BRAMs and 2,448 DSPs. We use the ZCU102 platform (non-DFX platform) for the monolithic system. 274,080 LUTs, 1,824 18Kb BRAMs and 2,520 DSPs are available in ZCU102 platform. We use Vitis 22.1 including Vitis_HLS and Vivado. We run the automated DSE experiments on a workstation equipped with the 3.4GHz AMD Ryzen 9 5950X 16 Core CPU with 32 processing threads and 128 GB of RAM.

8.2 NoC-based Overlay and Monolithic Overlay

The NoC-based system consists of 20 single-sized pages (7,264–7,919 LUTs, 44–66 18Kb BRAMs, 44–88 DSPs), 11 double-sized pages (14,647–15,840 LUTs, 110–132 18Kb BRAMs, 131–154 DSPs) and 5 quad-sized pages (29,806–31,392 LUTs, 220–264 18Kb BRAMs, 264–308 DSPs). One double page is not subdivided into two single pages in the overlay used in this experiment, and this is why there are only 20 single pages instead of 22 single pages. Total 64% of LUTs, 78% of BRAMs and 63% of DSPs are available in the PR pages. An abstract shell for each PR page is generated accordingly, and synthesized operators are mapped to appropriate abstract shells with the page assignment algorithm in Sec. 5.2.3. We use a BFT NoC as done in [28]. The number of processing elements (PEs) in the NoC is 24, Rent's parameter p [20] is 0.67, and sizes of the packet and payload are 49 bits and 32 bits respectively. Two PEs are used for the NoC configuration and DMA. The BFT uses 11,297 LUTs, and other peripherals including AXI interconnect use about 27K LUTs. Similarly, the monolithic overlay uses about 23K LUTs.

Table 1: Resource utilization and DSE results

Benchmarks		LUT %	FF %	BRAM %	DSP %	lat.	t_{DSE}
Rendering	Init	3	2	6	0	3.9×	1.8×
	NoC Fnl	15	7	14	1		
	Mono Fnl	-	-	-	-		
Rendering†	Init	4	2	9	0	3.8×	2.5×
	Mono Fnl	20	12	11	1		
Digit Rec.	Init	9	5	21	0	12.7×	1.3×
	Mono Fnl	58	34	97	0		
Optical	Init	7	4	11	5	3.8×	0.9×
	Mono Fnl	15	12	14	10		
Optical‡	Init	7	4	11	5	3.9×	1.4×
	Mono Fnl	14	10	13	4		
CNN-1	Init	13	6	11	0	2.2×	2.7×
	Mono Fnl	20	14	13	0		
CNN-2	Init	17	8	11	0	2.2×	2.3×
	Mono Fnl	26	16	13	0		

Rendering†: Rendering when identical operators are separately tuned
Optical‡: Optical Flow with a lower accuracy target
lat.: improvement in Kernel Latency, t_{DSE}: speedup in DSE time

8.3 DSE time and Performance

Fig. 7 shows the benefit of our incremental refinement strategy in DSE time. Both the NoC-based system and monolithic system use FIFO counters to identify the bottleneck as discussed in Sec. 4. Fig. 7 records the best kernel latency. Tab. 1 shows the resource utilization of the incremental strategy at different stages: the initial design point (Init: application + NoC interfaces) with the NoC-based flow and the final design point after the monolithic flow is over (Mono Fnl: application + AXI interconnect + peripherals). The reason why “Mono Fnl” data for Rendering is not available is that the final design point of the NoC flow did not meet the timing for monolithic system. In such case, the final design point of the NoC flow is the best design, superior to the final design point of the monolithic-only flow. Tab. 1 also shows improvement in the application latency and DSE time. Since we consider the design that matches the minimum accuracy as a valid design, the latency improvement is calculated as the latency achieved by the monolithic flow that first matches the minimum accuracy divided by the latency achieved by the final design of the fast incremental strategy.

8.3.1 Rendering. The design space of the user parameters for Rendering from Rosetta benchmarks [42] includes parallelization factor for `rasterization` function and `zculling` function. As stated in Sec. 7.2, by default, we tune identical operators together. For example, `zculling`'s parallelization factor of 4 results in four `zculling` operators, and when one of the `zculling` operator is identified as a bottleneck, the tuner increases the clock frequency for all four identical operators together. However, data-dependent applications like Rendering may require independent tuning for identical operators even if independent tuning takes longer. Rendering in Fig.7 is the DSE results when identical operators are tuned together, and Fig.1 (c) in Sec. 1 is the DSE results when identical operators are refined separately.

Tab. 2 illustrates each step in Fig.1 (c). For the first few iterations, our greedy tuner increases the parallelization factor for

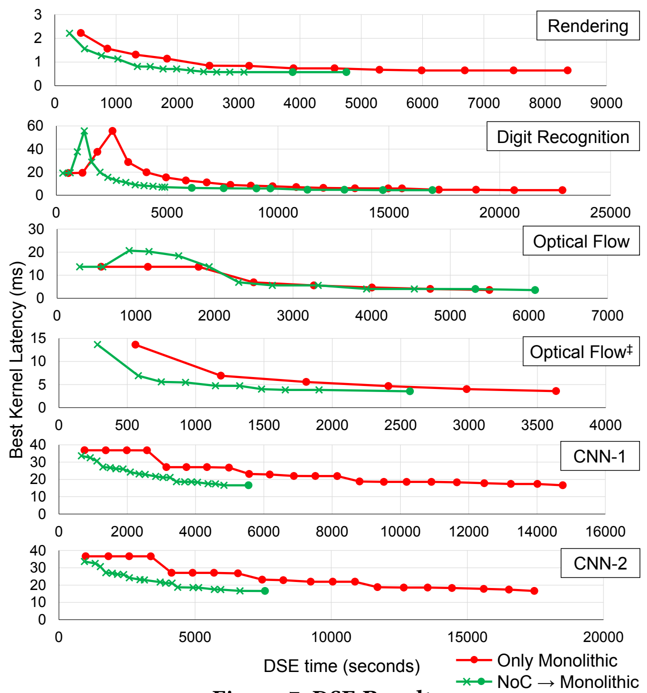

Figure 7: DSE Results

rasterization function and zculling function, generating new operators, and this is why the number of parallel compile runs increases. When the design parameters are all explored, clock frequencies are explored. DSE finishes because while the final design still points to rasterization and zculling as the bottlenecks (list of bottlenecks as discussed in Sec. 7.2), rasterization operators already reach the maximum kernel frequency and zculling fails at 400MHz. Our strategy achieves 1.8× and 2.5× faster DSE time in Fig.7 case and Fig.1 (c) case, improving 3.9× and 3.8× in application latency respectively.

Table 2: DSE trace for Rendering[†]

Iteration Count	Compile Time	Bottleneck	Design Point	Best Latency	# parallel runs	Flow
1	237s	None	init	2.23ms	5	NoC
2	241s	rast2_i1	PAR_RAST = 2	1.55ms	4	NoC
3	273s	zcul_i1	PAR_ZCUL = 2	1.27ms	8	NoC
4	270s	rast2_i1	PAR_RAST = 4	1.13ms	9	NoC
5	320s	zcul_i2	PAR_ZCUL = 4	0.81ms	13	NoC
6	170s	rast2_i1	clk = 250MHz	0.81ms	1	NoC
7	195s	rast2_i2	clk = 250MHz	0.81ms	1	NoC
8	131s	rast2_i4	clk = 250MHz	0.81ms	1	NoC
9	191s	zcul_i3	clk = 250MHz	0.78ms	1	NoC
...	...	...	...	...	...	...
34	865s	zcul_i3	clk = 400MHz	0.58ms	1	Mono

Rendering[†]: Rendering when operators are separately tuned

8.3.2 *Digit Recognition.*

The design space of the user parameters for Digit Recognition from Rosetta benchmarks includes parallelization factor KNN algorithm and K value. In the experiment, we set the minimum accuracy of 0.94, so our greedy tuner navigates the parameter (K value) to meet this accuracy first and then tunes for latency. This is why we see an increase in latency for the first four iterations. Our strategy achieves 1.3× faster DSE time compared to the monolithic flow while improving 12.7× in application latency.

In Digit Recognition, our strategy shifts to the monolithic system relatively quickly compared to other benchmarks, and this is because our tuner explores the user parameters (parallelization factor) first and then explores kernel frequencies. In Digit Recognition, we have 10 identical operators for the entire tuning and tune them together. The NoC-based system reaches to the parallelization factor that requires more BRAMs than available in the PR pages, and the design is migrated to the monolithic system. Then, different clock frequencies are all explored in the monolithic system.

8.3.3 *Optical Flow.*

The design space of the user parameters for Optical Flow from Rosetta benchmarks includes parallelization factor and width of OUTER_WIDTH variable that affects the accuracy of the application. Similar to Digit Recognition, it takes three iterations to reach the user-defined minimum accuracy and then tunes for the latency. Optical Flow is the application that our incremental strategy takes longer than the monolithic flow to reach the final design. In the incremental strategy, for the total 11 iterations spent with the NoC-based flow, 3 of them were to mitigate the limited NoC bandwidth by merging operators (Sec. 5.1.2). These three iterations are "extra" that are not necessary for the monolithic flow. Moreover, as the operators are merged to resolve the NoC bottleneck, the size of the merged operator becomes large, and the benefit of the fast separate compilation approach is reduced. Monolithic flow identifies one obvious bottleneck operator (tensor_weight_y) which is still the bottleneck when it is tuned to run with the maximum 400MHz. Optical Flow[‡] is the version when the accuracy target is relaxed so that OUTER_WIDTH variable does not increase and does not cause NoC bottleneck in the DSE. In this version, our strategy achieves 1.4× faster DSE time than the monolithic flow.

8.3.4 *CNN.*

To implement a Convolutional Neural Network (CNN) on FPGA, [4, 10, 26, 34, 35] use a streaming architecture that each layer has its own processing engine instead of having a single processing engine for all layers. As the input of the separate FPGA compilation framework is a dataflow graph, the streaming architecture naturally fits with the separate compilation. Therefore, instead of original Binarized Neural Network (BNN) design in Rosetta Benchmark, we use FINN open-source framework [4, 34] to generate neural network designs.

In our demonstration, we use FINN to generate HLS codes. In the generated HLS codes, performance and resource utilization for each layer can be controlled by parameters like PE and SIMD. We receive a hint from this configuration to set the starting point of the applications instead of starting from the minimum PE and SIMD. We create small CNNs with 6 convolutional layers and train the networks for CIFAR-10 dataset. CNN-1 has 1 bit for both weight quantization and activation quantization, and CNN-2 has 1 bit for weight and 2 bits for activation. The design space of the user parameters for CNN benchmark includes SIMD values for convolution modules and PE values for matrix multiplication modules. Our DSE system achieves 2.3–2.7× faster DSE time compared to the monolithic flow while improving 2.2× in application latency. The starting configuration is set by FINN assuming a single clock for all the operators. During DSE, as we increase the frequency of the bottleneck layer, the bottleneck moves to the different layer, exploring new SIMD values or PE values. The final designs in both CNN-1 and CNN-2 identify the

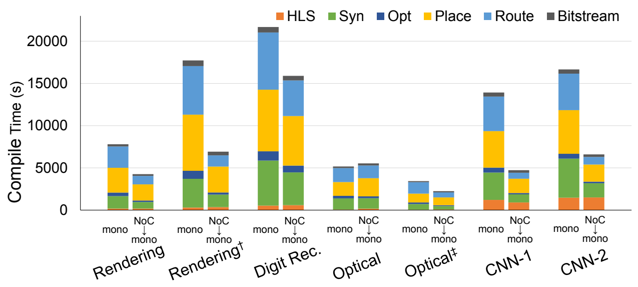

Figure 8: Compile Time Breakdown

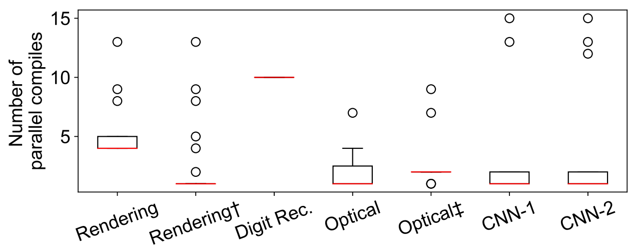

Figure 9: Number of parallel incremental page compile jobs in the NoC-based system

first convolutional layer as the bottleneck operator which already reaches the maximum SIMD value and maximum clock frequency.

8.4 Compile Time Analysis

Fig. 8 shows the compile time breakdown for both the monolithic flow and the incremental strategy. Time to read Vivado design checkpoints and *phys_opt_design* is omitted in Fig. 8 for brevity. As expected, the incremental strategy reduces compile time in all phases from HLS to bitstream generation except for the Optical Flow benchmark.

In CNN benchmarks, HLS takes 16–20% of the entire compile with our strategy whereas HLS takes only 3–5% for other applications. Long HLS runtime in FINN-generated HLS codes is a known issue which the authors in [1] resolve with “RTL weights” instead of embedding weight constants in HLS codes. If HLS runtime decreases with RTL weights, we expect to see even more speedup in DSE time. For example, if we exclude HLS time in DSE time, our incremental strategy achieves 2.9× faster DSE time for CNN-1 benchmark. While we can support RTL weights for necessary modules, we keep all the source codes at the HLS level to be consistent with other benchmarks.

8.5 Incremental Compilation

Fig. 9 shows the distribution of number of parallel compile jobs in the NoC-based system to show that the incremental strategy recompiles only necessary operators just like software compilation. In most of the benchmarks, only one operator is incrementally refined in the NoC-based system except for Digit Recognition in which 10 identical operators are tuned together throughout the DSE. The reason why the number is not always 1 is that the first compilation runs multiple compile runs in parallel for all operators. If new operators are generated with a new design point (e.g. parallelization factor in Rendering), these new operators need to be compiled together. If the page assignment changes because the newly compiled operator consumes more resources than before, all operators with the new PR pages need to be placed and routed.

9 DISCUSSION

Our experimental results show that our fast bottleneck identification (1) guides the users or the automation through the impactful design points that decrease application latency for both NoC-based and monolithic design flows, and (2) our incremental compilation reduces DSE time by recompiling only changed operators. Initial working designs are available in minutes, and improved designs become available every few minutes. Except for Optical Flow, the NoC-based incremental compilation produces lower latency designs than the purely monolithic flow for any compile time budget (the incremental curves are under the monolithic curves). Despite the limitations of the NoC platform, the final performance of the monolithic designs accelerated by the NoC-based flow in the early iterations is comparable to the performance of the final, monolithic-only optimization.

In Optical Flow, optimizations to repair NoC-bandwidth limitations eliminate the compile time benefits of the incremental-compilation scheme. Future work will explore both (1) how to better use the FIFO counter feedback to more quickly resolve NoC bottlenecks and (2) more optimization knobs to relieve NoC bottlenecks.

Our DSE case studies are simple and have limited design space for illustrative purposes. It will be valuable to extend the benchmark set and the tuning parameters available to each design.

Although we showcase that our incremental strategy can be integrated with performance/resource models in CNN benchmarks the idea can be generalized. We can categorize (1) applications that have to be evaluated in runtime with real data because of data-dependent functions (e.g. Rendering) and (2) applications that can have reasonably good starting points from model-based methods (CNNs). In (2) case, as shown in CNN benchmarks, model-based methods can reduce the design space so that the fully mapped, NoC-based system can start from a realistic design.

10 CONCLUSIONS

FPGA development is different from software programming because of the lack of visibility on the inner state of the design and slow compilation. While there exist previous works on hardware profiling and fast FPGA compilation, both efforts need to be integrated to support a software-like development experience for FPGAs. Our integrated stream FIFO counters automatically identify bottlenecks that limit performance. Our case studies show that our incremental refinement strategy using these lightweight counters and fast incremental compilation iterates initial yet important design points in 2–3 minutes and achieves 1.3–2.7× faster DSE time compared to the monolithic compilation while improving the kernel execution latency by 2.2–12.7×.

ACKNOWLEDGEMENTS

AMD donated Vivado and Vitis tools for use in this work.

REFERENCES

[1] Syed Asad Alam, David Gregg, Giulio Gambardella, Thomas Preusser, and Michaela Blott. 2022. On the RTL Implementation of FINN Matrix Vector Unit. *ACM Transactions on Embedded Computing Systems.* (2022).

[2] AMD. 2022. Vitis Embedded Platform Source Repository. https://github.com/Xilinx/Vitis_Embedded_Platform_Source/tree/2022.1.

[3] AMD 2023. *Vitis High-Level Synthesis User Guide.* AMD, 2100 Logic Drive, San Jose, CA 95124.

[4] Michaela Blott, Thomas B. Preußer, Nicholas J. Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. 2018. FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks. *ACM Transactions on Reconfigurable Technology and Systems* 11, 3 (2018).

[5] Young-kyu Choi and Jason Cong. 2018. HLS-Based Optimization and Design Space Exploration for Applications with Variable Loop Bounds. In *Proceedings of the International Conference on Computer-Aided Design.* 1–8.

[6] Young-kyu Choi, Peng Zhang, Peng Li, and Jason Cong. 2017. HLscope+: Fast and Accurate Performance Estimation for FPGA HLS. In *Proceedings of the International Conference on Computer-Aided Design.* IEEE Press, 691–698.

[7] Steve Dai, Yuan Zhou, Hang Zhang, Ecenur Ustun, Evangeline F.Y. Young, and Zhiru Zhang. 2018. Fast and Accurate Estimation of Quality of Results in High-Level Synthesis with Machine Learning. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines.* 129–132.

[8] André DeHon. 2000. Compact, Multilayer Layout for Butterfly Fat-Tree. In *ACM Symposium on Parallel Algorithms and Architectures.* ACM, 206–215.

[9] André DeHon, Yury Markovsky, Eylon Caspi, Michael Chu, Randy Huang, Stylianos Perissakis, Laura Pozzi, Joseph Yeh, and John Wawrzynek. 2006. Stream Computations Organized for Reconfigurable Execution. *Journal of Microprocessors and Microsystems* 30, 6 (September 2006), 334–354.

[10] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and Z. Wu. 2018. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation* 13, 07 (jul 2018).

[11] Hans Giessen. 2023. *Accelerating HLS Autotuning of Large Highly-Parameterized Reconfigurable SoC Mappings.* Ph. D. Dissertation. University of Pennsylvania.

[12] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. 1982. gprof: a Call Graph Execution Profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction.* ACM SIGPLAN, ACM, 120–126. SIGPLAN Notices, Volume 17, Number 6.

[13] Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Eddie Hung, Wuxi Li, Jason Lau, Weikang Qiao, Yuze Chi, Linghao Song, Yuanlong Xiao, Alireza Kaviani, Zhiru Zhang, and Jason Cong. 2023. RapidStream 2.0: Automated Parallel Implementation of Latency–Insensitive FPGA Designs Through Partial Reconfiguration. *ACM Transactions on Reconfigurable Technology and Systems* 16, 4 (2023), 30 pages.

[14] Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Jie Wang, Yuze Chi, Weikang Qiao, Alireza Kaviani, Zhiru Zhang, and Jason Cong. 2022. RapidStream: Parallel Physical Implementation of FPGA HLS Designs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA). 1–12.

[15] Intel 2018. *AN 797: Partially Reconfiguring a Design on Intel Arria 10 GX FPGA Development Board.* Intel. https://www.altera.com/documentation/ihj1482170009390.html

[16] 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, Article 46 (jun 2023), 30 pages.

[17] Gilles Kahn. 1974. The Semantics of a Simple Language for Parallel Programming. In *Proceedings of the IFIP CONGRESS 74.* North-Holland Publishing Company, 471–475.

[18] Nachiket Kapre. 2017. Deflection-routed butterfly fat trees on FPGAs. In *Proceedings of the International Conference on Field-Programmable Logic and Applications.* 1–8.

[19] George Karypis and Vipin Kumar. 2009. MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0. http://www.cs.umn.edu/~metis.

[20] B. S. Landman and R. L. Russo. 1971. On Pin Versus Block Relationship for Partitions of Logic Circuits. 20 (1971).

[21] Chris Lavin and Alireza Kaviani. 2018. RapidWright: Enabling Custom Crafted Implementations for FPGAs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines.* 133–140.

[22] Charles E. Leiserson. 1989. *VLSI Theory and Parallel Supercomputing.* MIT/LCS/TM 402. MIT, 545 Technology Sq., Cambridge, MA 02139.

[23] Sihao Liu, Jian Weng, Dylan Kupsh, Atefeh Sohrabizadeh, Zhengrong Wang, Licheng Guo, Jiuyang Liu, Maxim Zhulin, Rishabh Mani, Lucheng Zhang, Jason Cong, and Tony Nowatzki. 2022. OverGen: Improving FPGA Usability through Domain-specific Overlay Generation. In *IEEE/ACM International Symposium on Microarchitecture.* 35–56.

[24] Charles Lo and Paul Chow. 2018. Multi-Fidelity Optimization for High-Level-Synthesis Directives. In *Proceedings of the International Conference on Field-Programmable Logic and Applications.*

[25] Charles Lo and Paul Chow. 2020. Hierarchical Modelling of Generators in Design-Space Exploration. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines.* 186–194.

[26] Alexander Montgomerie-Corcoran, Zhewen Yu, and Christos-Savvas Bouganis. 2022. SAMO: Optimised Mapping of Convolutional Neural Networks to Streaming Architectures. In *Proceedings of the International Conference on Field-Programmable Logic and Applications.*

[27] Tan Nguyen, Zachary Blair, Stephen Neuendorffer, and John Wawrzynek. 2023. SPADES: A Productive Design Flow for Versal Programmable Logic. In *Proceedings of the International Conference on Field-Programmable Logic and Applications.*

[28] Dongjoon Park, Yuanlong Xiao, and André DeHon. 2022. Fast and Flexible FPGA Development using Hierarchical Partial Reconfiguration. In *Proceedings of the International Conference on Field-Programmable Technology.*

[29] Dongjoon Park, Yuanlong Xiao, Nevo Magnezi, and André DeHon. 2018. Case for Fast FPGA Compilation using Partial Reconfiguration. In *Proceedings of the International Conference on Field-Programmable Logic and Applications.*

[30] Benjamin Carrion Schafer and Zi Wang. 2020. High-Level Synthesis Design Space Exploration: Past, Present, and Future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2020), 2628–2639.

[31] Lesley Shannon and Paul Chow. 2004. Maximizing system performance: using reconfigurability to monitor system communications. In *Proceedings of the International Conference on Field-Programmable Technology.*

[32] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2022. AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators. *ACM Transactions on Reconfigurable Technology and Systems* (2022).

[33] James Thomas, Chris Lavin, and Alireza Kaviani. 2021. Software-like Compilation for Data Center FPGA Accelerators. In *Proceedings of the International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies.*

[34] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays.*

[35] Stylianos I. Venieris and Christos-Savvas Bouganis. 2019. fpgaConvNet: Mapping Regular and Irregular Convolutional Neural Networks on FPGAs. *IEEE Transactions on Neural Networks and Learning Systems* 30, 2 (2019), 326–342.

[36] Yuanlong Xiao, Syed Ahmed, and André DeHon. 2020. Fast Linking of Separately Compiled FPGA Blocks without a NoC. In *Proceedings of the International Conference on Field-Programmable Technology.*

[37] Yuanlong Xiao, Eric Micallef, Andrew Butt, Matthew Hofmann, Marc Alston, Matthew Goldsmith, Andrew Merczynski-Hait, and André DeHon. 2022. PLD: Fast FPGA Compilation to Make Reconfigurable Acceleration Compatible with Modern Incremental Refinement Software Development. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland). 933–945.

[38] Yuanlong Xiao, Dongjoon Park, Andrew Butt, Hans Giesen, Zhaoyang Han, Rui Ding, Nevo Magnezi, and André DeHon. 2019. Reducing FPGA Compile Time with Separate Compilation for FPGA Building Blocks. In *Proceedings of the International Conference on Field-Programmable Technology.*

[39] Xilinx, Inc. 2021. *UG909: Vivado Design Suite User Guide: Dynamic Function eXchange.* Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug909-vivado-partial-reconfiguration.pdf

[40] Mang Yu, Sitao Huang, and Deming Chen. 2021. Chimera: A Hybrid Machine Learning-Driven Multi-Objective Design Space Exploration Tool for FPGA High-Level Synthesis. In *Intelligent Data Engineering and Automated Learning.* Springer-Verlag, Berlin, Heidelberg.

[41] J. Zhao, T. Liang, S. Sinha, and W. Zhang. 2019. Machine Learning Based Routing Congestion Prediction in FPGA High-Level Synthesis. In *Proceedings of the Conference and Exhibition on Design, Automation and Test in Europe.* 1130–1135.

[42] 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. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays.* 269–278.