electronics

MDPI

Article

A Source-to-Source Compiler to Enable Hybrid Scheduling for High-Level Synthesis

Yuhan She [1], Yanlong Huang [1], Jierui Liu [1], Ray C. C. Cheung [1,2,*] and Hong Yan [1,2]

[1] Department of Electrical Engineering, City University of Hong Kong, Hong Kong, China; yuhanshe3-c@my.cityu.edu.hk (Y.S.); yanlhuang5-c@my.cityu.edu.hk (Y.H.); jieruiliu2-c@my.cityu.edu.hk (J.L.); h.yan@cityu.edu.hk (H.Y.)
[2] Center for Intelligent Multidimensional Data Analysis, Hong Kong Science Park, Shatin, Hong Kong, China
* Correspondence: r.cheung@cityu.edu.hk

Abstract

High-Level Synthesis (HLS) has gained considerable attention for its ability to quickly generate hardware descriptions from untimed specifications. Most state-of-the-art commercial HLS tools employ static scheduling, which excels in compute-intensive applications but struggles with control-dominant designs. While some open-source tools propose dynamic and hybrid scheduling techniques to synthesize dataflow-like architectures to improve speed, they lack well-established optimizations from static scheduling like datapath optimization and resource sharing, leading to frequency degradation and area overhead. Moreover, existing hybrid scheduling relies on extra dynamic synthesis support, either by dynamic or static HLS tools, and thereby loses generality. In this work, we propose another solution to achieve hybrid scheduling: a source-to-source compiler that exposes dynamism at the source code level, which reduces both frequency and area overhead while remaining fully compatible with modern static HLS tools without needing extra dynamic synthesis support. Experiments show significant improvements (1.26× speedup) on wall clock time (WCT) compared to *VitisHLS* and a better area–frequency–latency trade-off compared to dynamic (1.83× WCT speedup and 0.46× area) and hybrid (2.14× WCT speedup and 0.72× area) scheduling-based tools.

Keywords: high-level synthesis; pipelining; scheduling; FPGA

Academic Editor: José V. Frances-Villora

Received: 20 October 2025
Revised: 13 November 2025
Accepted: 20 November 2025
Published: 22 November 2025

Citation: She, Y.; Huang, Y.; Liu, J.; Cheung, R.C.C.; Yan, H. A Source-to-Source Compiler to Enable Hybrid Scheduling for High-Level Synthesis. *Electronics* **2025**, *14*, 4578. https://doi.org/10.3390/electronics14234578

1. Introduction

Scheduling, as one of the core steps in HLS, is still being explored to find the best choice of strategy. Static scheduling strategy assigns operations into predetermined cycle slots based on compile-time analysis. Despite being adopted by almost all commercial HLS tools, it still suffers from exploiting parallelism in code with runtime-determined, long-latency control dependencies in loops since the worst assumption must be made to guarantee the correct execution under any runtime conditions. Dynamic scheduling, on the other hand, implements a distributed dataflow network with handshakings between each communicating operation pair. This strategy excels on designs with complex control flow and imbalanced branches but loses some opportunities for resource sharing and critical path optimization, resulting in area overheads and a frequency drop. Hybrid scheduling bridges the static and dynamic strategies to highlight the advantages of both, achieving both high performance and low area overhead. However, hybrid scheduling still relies heavily on the dynamic synthesis support, either by dynamic HLS tools or dynamic synthesis support

provided by the static HLS tools. These limitations result in a large frequency drop and loss of generality.

In this paper, we propose another solution, a source-to-source compiler, to achieve hybrid scheduling as well as remaining fully compatible with the existing static HLS tools. The compiler transforms a plain, untimed design into a functionally equivalent Pseudo-Cycle-Accurate (PCA) model, where each iteration of the original loop is partitioned into several "virtual" cycles and mapped to the same number of iterations in the transformed PCA model. The term *Pseudo* indicates that the partitioned "virtual" cycles are not necessarily the same as the real hardware clock cycles synthesized by the HLS tool. Instead, they are normalized control steps extracted through static analysis with the specific aim of exposing the dynamism to the source code. Each "virtual" cycle may be synthesized into multiple clock cycles by the static HLS tools following their loop optimizations, but the obtained gains from the transformation are not affected, as detailed in Section 4. The transformation is achieved by registering the loop-carried variables with buffers, which is a well-known technique to handle long loop-carried dependencies to improve the *Initiation Interval* (II) in static HLS flow [1]. In our proposed framework, this process is fully automated to transform a plain design into the functionally equivalent PCA model. Long loop paths are divided into multiple short paths, each one taking a single transformed loop iteration to complete. Under different runtime conditions, it takes different numbers of iterations to finish the same original loop path by stalling the updating of the buffers, which reflects the dynamic execution nature. This idea is inspired by [2], where the authors synthesized a RISC-V processor core from a cycle-accurate ISA simulator. Dependencies and hazards are carefully handled manually on a cycle basis by invalidating specific "registers" under specific conditions so that HLS tools are proficient in dealing only with static optimizations. Our work targets the same goal while aiming at a generic source-to-source compiler to achieve hybrid scheduling. The main contributions of this paper are as follows:

1. We propose the Pseudo-Cycle-Accurate (PCA) model to achieve source level dynamism to facilitate hybrid scheduling for HLS.
2. We propose a novel intermediate representation (IR) termed an *Execution Graph* (EG) by extending the *Gated SSA* (GSSA) [3] representation as the foundation to support the transformation from a plain loop kernel to its equivalent PCA model.
3. We propose an automated framework with key algorithms to transform a plain design into a PCA model, which is fully compatible with existing HLS tools adopting static scheduling strategy.
4. The framework was evaluated with a set of benchmarks, and the results demonstrate significant performance improvements over a commercial static-scheduling-based HLS tool (*Vitis HLS*) as well as better resource–performance tradeoffs over other state-of-the-art, open-source, dynamic- and hybrid-scheduling-based tools.

The remainder of this article is organized as follows: Section 2 provides preliminaries and related works on different scheduling approaches. Section 4 describes the details of our proposed PCA model with demonstrative examples. Section 5 presents the overall framework including the intermediate representation and some key aspects. Section 6 presents the experiment results and analysis based on a variety of HLS benchmarks. Section 7 concludes our work with discussion of future research.

2. Preliminaries

In this section, we provide the basic concepts of *High-Level Synthesis* (HLS) and the scheduling in the HLS process. In addition, we introduce the necessary knowledge of dynamic and hybrid scheduling techniques.

2.1. HLS and Scheduling in HLS

For many years, the standard practice in the electronic design has been to employ Hardware Description Languages (HDLs) like Verilog and VHDL to define the specifics of a hardware design. This level of description, however, requires specialized hardware knowledge and the time-consuming design of a complex architecture. A more abstract and efficient alternative is offered by HLS tools, which translate hardware functionality described in a high-level software language into a detailed hardware implementation. This approach yields significant benefits; it accelerates the design process for hardware engineers and allows them to more effectively explore different design possibilities. Also, it makes hardware programming more accessible to software engineers. While HLS is advantageous for designing both ASICs and FPGAs, its impact is particularly pronounced in the FPGA space. The programming complexity associated with FPGAs has long been a primary obstacle to their widespread adoption, a challenge that HLS directly addresses and helps to overcome [4]. HLS takes untimed specifications as input to describe the underlying hardware. One of the most popular languages is C/C++ due to its mature compilation infrastructures. To create designs that are both high-performing and compact, HLS tools employ a suite of compiler optimizations, from front-end code transformations (e.g., if-conversion) to back-end RTL optimizations (e.g., bitwidth analysis).

One of the core steps of the HLS synthesis process is scheduling, which determines the specific clock cycle for every operation's execution. A typical scheduling process starts by converting the source code into the *intermediate representation* (IR) for front-end analysis, e.g., the LLVM IR [5]. Then, the IR is usually transformed into a more structured form called the *Control Data Flow Graph* (CDFG). The CDFG consists of two levels of information to represent both the control flow and dataflow of the original code: the top-level graph represents the *Control Flow Graph* (CFG) with each vertex as one *Basic Block* (BB), and the lower-level graph represents the *Data Flow Graph* (DFG) for each BB, with each sub-vertex as one single operation (e.g., `add` or `mul`). Edges in both levels represent the control/data dependencies. In HLS tools, scheduling is the task of transforming the untimed CDFG into a timed schedule [6].

2.2. Static Scheduling

Static scheduling is a predominant technique widely adopted in modern commercial HLS tools. The scheduling of operations is performed completely at the compilation phase through static analysis of the input code with core information extracted, such as the CDFG, which defines the execution order of each involved operation. Algorithms are applied to derive which operation should execute at which clock cycle to compose a hardware datapath considering the extracted dependencies and user-specified resource and timing constraints. This is generally accomplished using systems of difference constraints (SDC) modeling, which accounts for a wide range of constraints, including clock frequency, data and control dependencies, and the availability of resources [7,8]. Such models characterize the constraints as inequalities and employ linear programming (LP) algorithms to achieve customized timing objectives. After the operations are allocated to certain cycle slots, a central Finite State Machine (FSM) is generated to handle the control flows in the original program.

In static scheduling, loops can be pipelined to optimize the overall execution latency. Loop pipelining allows the loop iterations to overlap as much as possible with all the dependencies honored. A pipeline's effectiveness is primarily measured by its *Initiation Interval* (II), which quantifies the delay, in clock cycles, between the start of two consecutive loop iterations, with an II of 1 representing the theoretical maximum throughput. Iterative modulo scheduling algorithms [9] are applied to calculate the minimum achievable II

as $II_{min} = max_i\{\lceil Delay_i / Distance_i \rceil\}$, where the *Delay* is the number of cycles needed to execute the entire recurrence path (recurrence means that operations in one iteration depend on results from previous iterations), and the *Distance* is the loop iterations between the definition and usages of a recurrence value.

Static scheduling excels at critical path optimization since the target clock frequency is constrained at the compilation phase in the SDC model. Such constraints take all data and control paths into account, and the LP algorithms solve the scheduling for a globally optimal performance, which usually requires critical paths to be as balanced as possible. In addition, static scheduling is also effective at exploring resource sharing. The SDC models allow the user to provide the time constraints to find the minimum resource consumption or to provide the resource constraints to find the best performance. When possible, multiple operations can share the same hardware component if they are scheduled in different clock cycles. This maintains the performance but results in a smaller area.

Although proven efficient in exploring resource sharing and critical path optimization, static scheduling suffers from the conservative pipelining of non-deterministic control flows with feedback dependencies and irregular memory access: When a recurrence value depends on multiple paths with unbalanced cycle delays and runtime control decisions, static scheduling conservatively calculates the *II* with the longest recurrence path to guarantee the correct execution of the loop under any runtime conditions. Cycles can be wasted when the short path is frequently taken, and static scheduling is incapable of handling it since the scheduling is performed at the compilation phase.

2.3. Dynamic Scheduling

Dynamic scheduling provides a new view of scheduling the operations with the handshaking interfaces. Different from the FSM-centric structure used in static scheduling, dynamic scheduling implements a distributed control network where each component determines its own behavior through latency-insensitive handshaking [10]. The scheduling decisions are made locally by each operation through handshaking. As soon as all conditions for execution are met (e.g., all the inputs are available, and critical control decisions are resolved), an execution starts. Since the scheduling of an operation is not determined until it actually starts to run, dynamic scheduling intrinsically adapts to the runtime condition, making it perfectly suitable for irregular and unpredictable control flows. However, the distributed scheduling mechanism sacrifices the opportunities of resource sharing and critical path balancing. Instead of scheduling at the compilation phase like static scheduling, dynamic scheduling has no knowledge about when an operation will actually be executed. Therefore, it is not able to share the same hardware for different operations when scheduling. For critical path synthesis, the handshaking signals attached with the operations directly increase the path delay, resulting in a longer critical path and a lower operating frequency. Also, dynamic scheduling creates dataflow-like handshakings to each component even though it exhibits no dynamic behavior, which introduces heavy area overhead. Although works on resource sharing [11] and excessive dynamism eliminating [12] show notable resource savings, there is still a significant gap compared to the static-scheduling-based HLS tools.

2.4. Hybrid Scheduling

Hybrid scheduling combines the advantages of both static and dynamic scheduling [13–16]. There are two main existing strategies to achieve this: The first is to partition the input code to static and dynamic regions, where the static regions are usually defined as a chain of SSA def–use pairs in the same basic block (i.e., no control flows inside) with particular attributes in the context of HLS [13–15]. The static regions

are then extracted and synthesized with a classic static HLS tool. The synthesized blocks are then wrapped with handshaking and control logic so that it can be considered as a regular dataflow component during the following dynamic scheduling process. This method proves to be efficient in reducing the resource overhead while maintaining the high performance achieved due to the dynamic nature. However, it still suffers from a heavy frequency drop since the critical paths are still bottlenecked by the dynamic tools. Also, it suffers from the same limitations of the current dynamic HLS tools, e.g., additional Load Store Queue (LSQ) requirements.

The second strategy is to utilize the builtin latency-insensitive channels provided by the static HLS tools (e.g., the `hls::stream` from VitisHLS, `ac_channel` from CatapultHLS, and `SYCL` pipe from Intel HLS) to build dataflow circuits [16]. In such processes, the input code is completely synthesized through static scheduling but with its loop kernels partitioned into several communicating operations connected with the latency-insensitive FIFO channels. Compared with the first strategy, this method selectively introduces dynamism to the static HLS, making it more efficient on resource consumption and critical path optimization. However, it heavily depends on the tool-specific supports on dataflow synthesizing features with latency-insensitive channels. Existing static HLS tools provide limited dataflow support with extra constraints, e.g., coarse-grained dataflow partition (only structured code blocks are amenable to dataflow synthesis) and one-way dataflow requirements (dataflow synthesis cannot be applied within a loop) in VitisHLS. Although some tools may provide more flexibility on customizing dataflow interfaces, adapting to different tools with different specifications requires extensive engineering efforts, and these methods thereby lose generality as generic hybrid scheduling solutions. Furthermore, non-blocking reading and writing on those channels are necessary to implement this hybrid scheduling scheme, which leads to extra difficulties in the verification process as non-blocking reading can break the behavior consistency between the source code and the synthesized hardware.

Based on all these limitations, we propose a new strategy to bring dynamism to static HLS by transforming a plain design to an equivalent PCA model via source-to-source compilation. Our method is tool-independent, channel-free, and perfectly compatible with the modern static HLS tools and hence can be easily verified through the mainstream co-simulation method provided by most commercial HLS tools.

3. Related Work

In this section, we provide a comprehensive literature review on the advancement of HLS technologies to overcome the conservatism for loop pipelining, which can be generally classified into dynamic-scheduling-based methods and hybrid-scheduling-based methods.

3.1. Dynamic-Scheduling-Based HLS

The idea of dynamic scheduling can be traced back to the latency-insensitive design methodology [10,17,18]. Cortadella et al. [17,19] proposed the elastic circuit, which is a promising hardware implementation architecture for the latency-insensitive methodology. However, the elastic circuit has not been further developed as a general HLS tool since it is too restrictive. Venkataramani et al. [20] proposed a framework to synthesize a C-based program to hardware following the asynchronous design paradigm. Despite the presence of dynamism, such a circuit has proven to be inefficient in modern implementation flow. Hoover and Brewer [21], Chatterjee et al. [22] extended the handshaking protocol (SELF) in the elastic circuit [19] for better branching and merging implementation. There are also works that explore synthesizing dataflow circuits from high-level descriptions [23] or intermediate functional programming representations [24]. But all these works are far from a complete dynamic-scheduling-based HLS tool. Recent work [25,26] proposed *Dynamatic*

HLS, which is considered as the first generic dynamically scheduled HLS tool. The work was then continuously developed for resource optimization [11,12,27] and frequency optimization [28]. All these works are based on dynamic scheduling, while our work targets achieving dynamism using pure static HLS tools, which exposes more opportunities for resource sharing and critical path optimization.

3.2. Hybrid-Scheduling-Based HLS

Hybrid scheduling seeks opportunities to combine the static scheduling and dynamic scheduling to highlight the advantages of both. Several works [29–32] try to incorporate some level of dynamism into the static HLS flow. Alle et al. [29] and Liu et al. [30] proposed to synthesize multiple scheduling schemes, which can be dynamically selected at the runtime based on the conditions. Such approaches replicate the source code and introduce significant resource overhead. Tan et al. [31] proposed an approach, ElasticFlow, to solve the irregular bound of nested inner loops and optimize pipelining of irregular loop nests that contain dynamically bound inner loops. Such an approach is specific to the target loop patterns and therefore loses generality. There are also explorations of enabling speculation within the static scheduling flow [33–36]. They maintain the speculatively executed iterations and discard them when the iteration is solved as misspeculation. Such methods can aggressively pipeline the loop but can also result in significant misspeculation penalty due to the pipeline flush and re-execution. Dai et al. [32] proposed application-specific dynamic hazard detection architecture for memory speculation, but these dynamic components still need to be manually integrated into the static HLS flow, instead of being a seamless plugin compared to our proposed work.

Recent works explored the possibilities of integrating static scheduling into the dynamic HLS flow [13–15] based on the theory of encapsulating static modules into a latency-insensitive system [37]. Such methods are promising in reducing the resource overhead while retaining the dynamic execution nature, but the critical path is still bottlenecked by the backend dynamic HLS tool, resulting in frequency degradation. In contrast, Szafarczyk et al. [16] proposed to utilize the provided latency-insensitive channels provided by static HLS tools to synthesize a dataflow network within the static HLS flow. With the static HLS tool as the backend, the frequency improves, but the method loses generality as it sticks to a specific HLS tool, and its provided channels as discussed in Section 2.4. Our proposed framework relies on source-to-source compilation, which is tool-independent and channel-free, as well as compatible with the modern static HLS tools as a backend for better area and frequency.

4. PCA Model

In this section, we present the concept of the PCA model and show how it can facilitate dynamic scheduling with two demonstrative examples. We first demonstrate the behavior of a PCA model using a regular loop kernel without control flows. Then we show how the PCA model can achieve dynamism and improve the static scheduling using a loop kernel with runtime-determined control flows and imbalanced branches. After that, we describe the key steps to transform a loop into a valid PCA model.

4.1. Concept of Virtual Cycle and PCA Model

Given a loop kernel, one *transaction* is defined as one loop iteration of the original loop kernel. In contrast, one "virtual cycle" is defined as one loop iteration of the transformed PCA loop. The PCA model is proposed as a transformed loop from the original loop, which keeps the same *transaction-level* equivalence. However, the *virtual-cycle-level* behavior is different. For the original loop, one transaction can be completed with one "virtual cycle", but for the

PCA model, one transaction may need several "virtual cycles" to complete. This is because the PCA transformation divides the execution of the original loop transaction into several phases, with each phase corresponding to one "virtual cycle". This execution pattern is achieved by registering the output value in every "virtual cycle" with buffers. Buffers are initialized with an invalid value and will be updated by their input value, like a register in a circuit. It is worth noting that the "virtual cycle" does not have to accurately map a hardware clock cycle synthesized by the HLS tool. It is normal that one "virtual cycle" is synthesized to multiple hardware clock cycles if it includes multi-cycle operations.

4.2. PCA Model for Regular Loop

4.2.1. Behavior

Figure 1 shows a regular loop kernel without control flows as an example to demonstrate the behavior of the PCA model. As shown in Figure 1b, the loop consists of a chain of operations with one recurrence variable (`floats_01`) updated at each iteration. The back edge that updates the `floats_01` with `floatadd9` is omitted in Figure 1b to build a *Directed Acyclic Graph* (DAG) view to facilitate the description of related algorithms. Figure 1c shows the execution pattern of this loop kernel: At each transaction, the recurrence variable is directly updated with the value calculated from the preceding transaction. Four transactions take four iterations of the original loop kernel.

Figure 1d,e show the transformed PCA code and its corresponding dataflow graph. Three buffers are inserted to create four "virtual cycles", with each "virtual cycle" responsible for executing several operations. Figure 1e presents the location of the inserted buffers. These buffers are mapped as loop-carried variables (`buffer0/1/2` in Figure 1d in L3–L5) and are used to register the output value from the input "virtual cycle". The registered values cannot be propagated to the following operations until next iteration. This is achieved by the implementation of the buffers, which are updated on a loop–iteration basis: at each iteration, the operations take input from the buffers and finally update the output buffers.

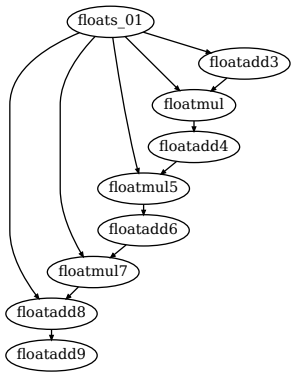
Considering the execution of the third "virtual cycle" (containing the operations `floatadd6` and `floatmul7`) as an example, they are executed with the input directly from operation `floatmul5` and variable `floats_01` at each iteration in the original code as shown in Figure 1a,b. After the PCA transformation, `floatadd6` takes the input from the `buffer1` (as shown in L22 in Figure 1d). However, in the same iteration, `buffer1` is updated by `floatmul5` at L28 in Figure 1d, which is later than the computation of `floatadd6` and `floatmul7`. This means that the two operations use the value of `buffer1` obtained from the previous iteration (`floatmul5`), and similarly, the value obtained from previous iteration (`floatmul5`) is calculated by its input buffers (`buffer2` in L20 in Figure 1d), which is also not updated from the same iteration as itself.

Following such an execution pattern, at a specific iteration, operations in different "virtual cycles" use input values from different transactions, and the value for the same transaction can only propagate one level of buffer at each iteration (i.e., one "virtual cycle"). This buffered execution simulates the behavior of a pipeline with buffers served as the pipeline registers. However, in this sample example, the recurrence prevents the PCA model from being fully pipelined (i.e., starting the next transaction at each iteration) since the execution of next transaction requires the final result from the previous transaction (i.e., the recurrence variable `floats_01`), which is calculated after 4 "virtual cycles". Therefore, to guarantee the equivalent transaction-level behavior, an additional valid bit should be carried along the dataflow to indicate whether a produced value is valid to start the following transaction. By initializing buffers as invalid, the pipeline can achieve an II of 4: in each iteration, only one "virtual cycle" is executing the valid values, and other "virtual cycles" are all invalid. As shown in L6–L10 in Figure 1d, 5 valid bits are declared for 2 recurrence variables and 3 buffers. These valid bits are propagated along the data path (e.g., L19, L21, L23, and L25

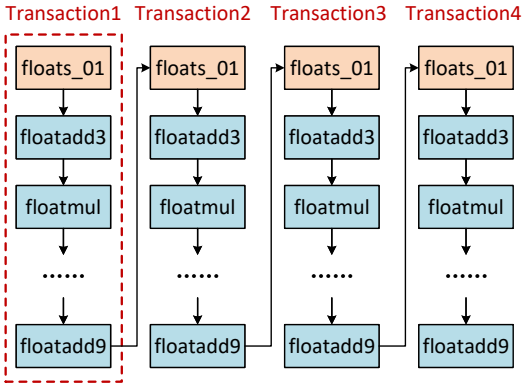
in Figure 1d) and finally updated with the buffers and the recurrence variables. Only when the recurrence variable associated with the valid bit is true can the recurrence variables be updated to proceed the next transaction as shown in L14–L17 for `floats_01` and L34–L36 for `i` in Figure 1d.

```
1  float s = 0.0f;
2  int i = 0;
3  for (;i<5;i++)
4  {
5      s = s +
6      (((s+0.64f) * \
7          s+0.7f) * \
8          s+0.21f) * \
9          s+0.33f;
10 }
```

(**a**) Original loop kernel



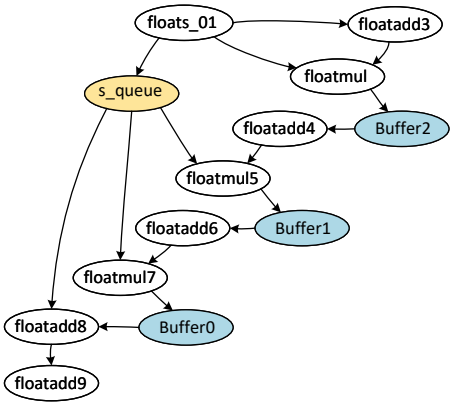
(**b**) Original dataflow



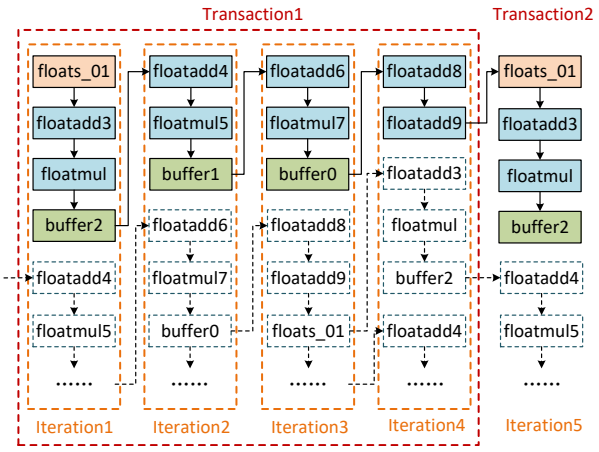
(**c**) Original transactions

```
1  float s = 0.0f;
2  int i = 0;
3  float buffer0 = 0;
4  float buffer1 = 0;
5  float buffer2 = 0;
6  bool s_valid = true;
7  bool i_valid = true;
8  bool buffer0_valid = false;
9  bool buffer1_valid = false;
10 bool buffer2_valid = false;
11 float s_queue = 0;
12 float s_queue_valid = false;
13 while (i<5) {
14     if (s_valid) {
15         s_queue=s;
16         s_queue_valid=s_valid;
17     }
18     float mul=(s+0.64f)*s;
19     bool mul_valid=s_valid;
20     float mul5=(buffer2+0.7f)*s_queue;
21     bool mul5_valid=buffer2_valid &&
           s_queue_valid;
22     float mul7=(buffer1+0.21)*s_queue;
23     bool mul7_valid=buffer1_valid &&
           s_queue_valid;
24     float add9=buffer0+s_queue+0.33f;
25     bool add9_valid=buffer0_valid &&
           s_queue_valid;
26     buffer2=mul;
27     buffer2_valid=mul_valid;
28     buffer1=mul5;
29     buffer1_valid=mul5_valid;
30     buffer0=mul7;
31     buffer0_valid=mul7_valid;
32     s=add9;
33     s_valid=add9_valid;
34     if (add9_valid) {
35         i = i + 1;
36     }
37 }
```

(**d**) The full code after the PCA transformation that is functionally equivalent to the original loop kernel



(**e**) Dataflow of the transformed PCA loop kernel



(**f**) Execution pattern of the transformed PCA loop

Figure 1. A demonstrative example of a regular loop kernel to showcase the PCA transformation, which contains the C code, the dataflow graph, and the execution pattern for both the original and its transformed PCA model: (**a**) shows the source code of the original loop kernel. (**b**) shows the corresponding dataflow graph omitting the back edge. (**c**) shows the execution pattern for 4 transactions of the original loop. (**d**) shows the transformed PCA code. (**e**) shows the dataflow graph of the transformed loop kernel including the inserted buffers and queues. (**f**) shows the execution pattern of the transformed PCA loop kernel including 5 "virtual cycles" for 2 transactions.

Figure 1f is the visualization of the PCA model's execution pattern, where each column represents one iteration of the PCA model (i.e., one "virtual cycle"), and the execution can be considered as "virtual-cycle-accurate". The red boxes represent the recurrence variables that are active during different iterations. The blue boxes represent the regular operation chains. The green boxes represent the buffers inserted to create "virtual cycles". The arrowed edges represent the data dependencies in the transformed PCA loop. For simplicity, the dependencies on the recurrence variable (e.g., `floats_01`) are omitted. This graph mainly aims to demonstrate the activeness of the operations in different "virtual cycles" at each loop iteration. According to the behavior of the buffers and the valid bits, at each iteration, only the operations in one "virtual cycle" will be actively executed (represented as filled boxes). And in the following iteration, the next "virtual cycle" will be activated. Therefore, it takes 4 iterations for the PCA loop to finish one transaction of the original loop. Queues, as shown in Figure 1e (the `s_queue` operation), are inserted to preserve the valid value from the previous iterations to avoid deadlock: operations in different regions require the same recurrence value from `floats_01` during the same transaction. The queue node is used to ensure each operation to produce a valid result when its input buffer is valid. The detailed explanations and algorithms to insert the queues are described in Section 4.4.2.

4.2.2. Scheduling

When scheduling a regular loop kernel in Figure 1a with static HLS tools with pipelining, the minimum achievable *II* is calculated with modulo scheduling [9], e.g., $II = 4$ for the example in Figure 1a assuming a total delay of 4 clock cycles. On the other hand, for the PCA model in Figure 1d, the tool will derive an $II = 1$ because the recurrence distance is increased to 4 by the inserted buffers (the recurrence variable is updated with the result calculated 4 iterations ago). Given the total transactions of N, the plain synthesis achieves a latency of $4N$, while the PCA synthesis achieves a latency of $4N + 3$. The two routines will result in a similar latency given a large transaction number N over the datapath delay. Therefore, the PCA model cannot improve the scheduling of the regular loop kernel. As described in Section 5, our proposed framework will not perform PCA transformation on the regular loops.

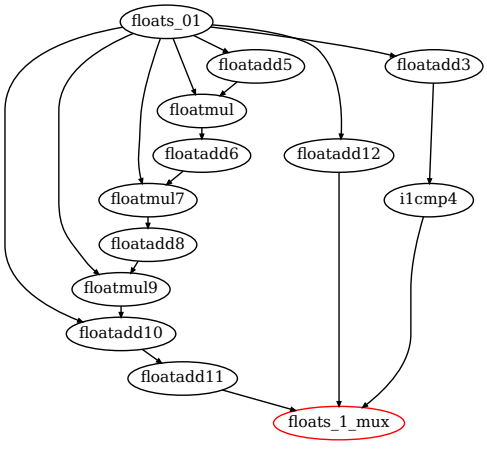
4.3. PCA Model for Loops with Control Flows

4.3.1. Behavior

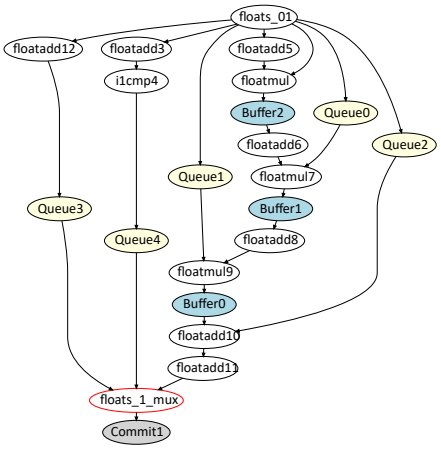
Figure 2a shows a loop kernel with two imbalanced branches with a runtime-determined condition (`d + s>0`). Its dataflow is shown in Figure 2b. With static HLS, the loop cannot be efficiently pipelined since the long path is always assumed to be taken. In this case, PCA transformation can improve the scheduling with a fine-grained control on the "virtual cycles". Assuming four "virtual cycles" are created with three buffers as shown in Figure 2c, the iterating process of the PCA model can be illustrated in Figure 3. It takes one or four iterations to finish one transaction of the original loop based on the runtime condition value (`i1cmp4`).

```
// inputs: a, b
float s = 0.0f;
int i;
for (i=0;i<5;i++)
{
  float d = a + b;
  if (d+s > 0.0f) {
      float s_new;
      s = s +
      (((s+0.64f) * \
        s+0.7f) * \
        s+0.21f) * \
        s+0.33f;
  } else {
      s = s + 1;
  }
}
```

(**a**) Original source code



(**b**) Original dataflow



(**c**) Dataflow of the PCA model

Figure 2. A demonstrative example of an irregular loop showcasing the efficiency of the transformed PCA model when pipelining the loop kernels with imbalanced recurrences and runtime-determined control flow: (**a**,**b**) show the source C code and the dataflow of the demonstrative example, respectively. (**c**) shows the dataflow of the transformed PCA loop with three buffers inserted in the long recurrence path.

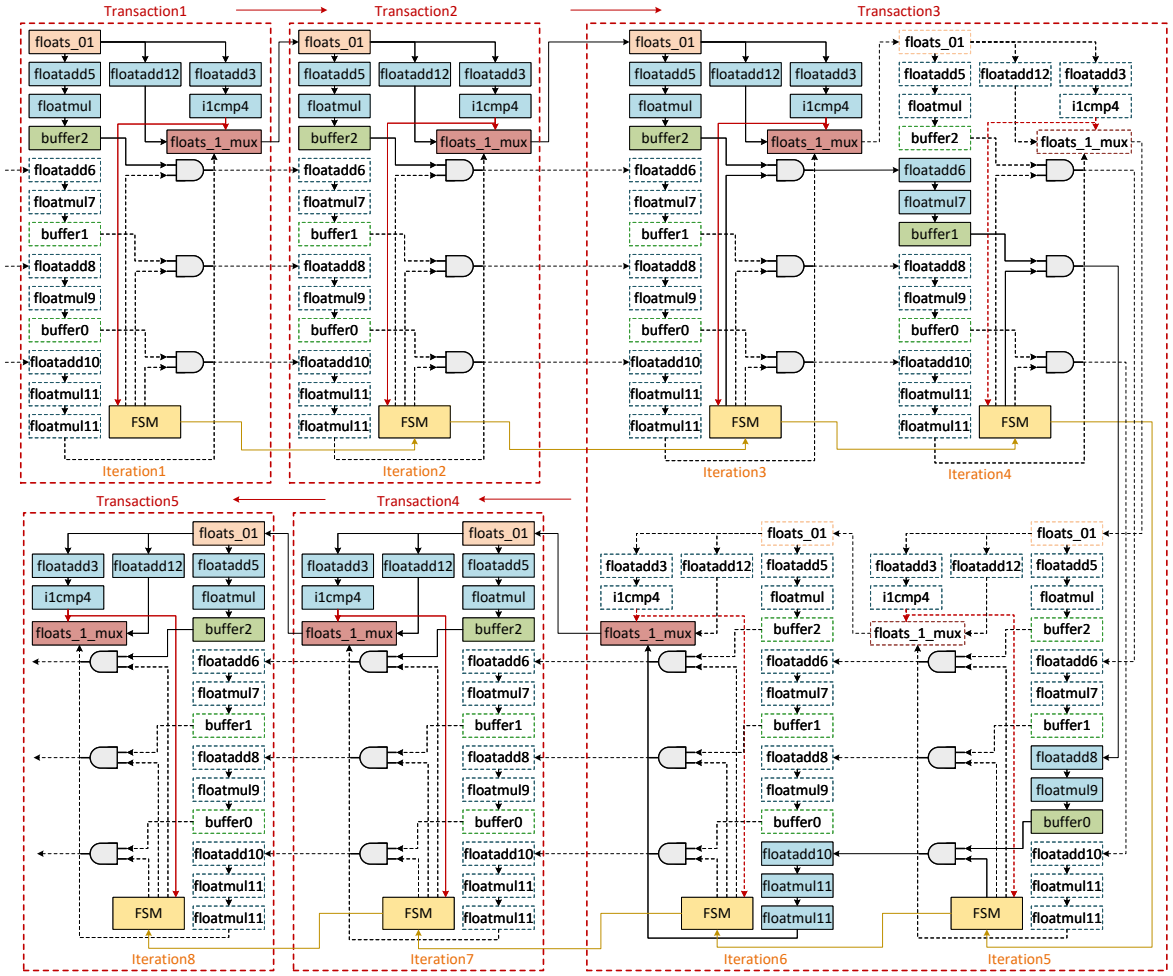
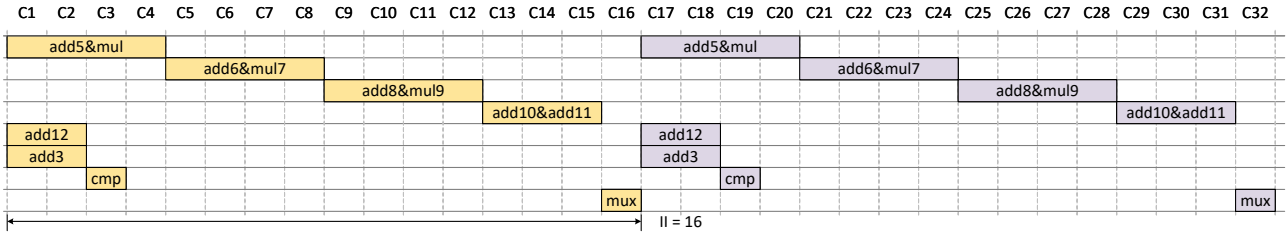


Figure 3. The execution pattern of the PCA model of the demonstrative example in Figure 2, including 8 "virtual cycles" for 5 transactions. The filled boxes represent the active operations in each iteration, and the dashed boxes represent the inactive operations with invalid values. The FSM controls the behavior of the `MUX` operation to achieve dynamic execution.

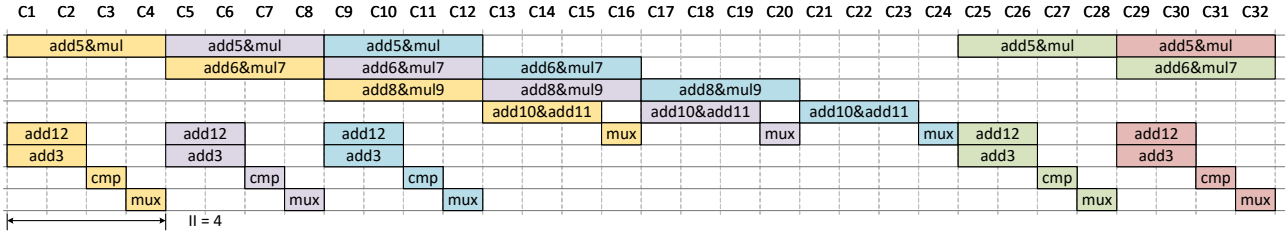
Figure 3 also illustrates the implementation details to achieve this dynamic behavior. As described in Section 4.2, each iteration only actively executes the operations within that "virtual cycle". The boxes with filled colors in each iteration are the operations that are active, and the solid lines are valid values produced by those operations. The dynamic behavior can be achieved by controlling the validity of the multiplex operation `floats_1_mux`: when the condition input `i1cmp4` is active, `floats_1_mux` produces the valid bit based on the status of the taken path. For example, `floats_1_mux` receives the valid data from `i1cmp4` at `Iteration1`, and it knows that the short path `floatadd12` should be taken in `Transaction1`. Then, when the value of `floatadd12` is valid (in the current iteration), `floats_1_mux` can produce valid output to `floats_01` for the next transaction even though the long path `floatadd11` is not active yet. In `Iteration3`, `floats_1_mux` knows the long path will be taken and the result is not yet valid. It produces invalid data, and the iteration moves to the next "virtual cycle"(`Iteration4`). Therefore, `Transaction3` takes four iterations to finish. A *Finite State Machine* (FSM) is generated to control whether an output value should be propagated to the following iteration. In this example, when the short path is taken in `Iteration1 and Iteration2`, the FSM invalidates the output of `buffer2` to ensure only one "virtual cycle" is valid is one iteration. The details of the FSM implementation are described in Section 4.4.6.

4.3.2. Scheduling

Following this execution pattern, dynamic scheduling can be achieved. Assuming 4 clock cycles are inferred for each "virtual cycle", the loop can be pipelined by static HLS with an *II* of 4 as shown in Figure 4b, compared to an *II* of 16 achieved by directly pipelining the original loop kernel as shown in Figure 4a. The execution latency can be calculated as $16(N-i)+4i+12$, where i denotes the number of transactions that the short path is taken. This result reveals the dynamic nature of the PCA synthesis routine and demonstrates significant latency improvement when i is large. The empty slots in Figure 4b (C13 to C24) are the "virtual cycles" used to calculate the long path.



(**a**) Scheduling of the original loop



(**b**) Scheduling of the PCA model

Figure 4. The scheduling graph of both the original and transformed PCA loop for the demonstrative example in Figure 2. The latencies of the `add&mul` operation chain, `add` operation, `cmp` operation, and `mux` operation are 4 cycles, 2 cycles, 1 cycle, and 1 cycle, respectively. The horizontal axis represents the clock cycles, while the vertical axis represents different operations/chains, where each operation/chain takes only one row for the same iteration. Different colors indicate different iterations of the loop. The II annotated in each schedule graph represents the *Initiation Interval* that can be achieved by the following static HLS tool.

4.4. PCA Transformation

Transforming a loop into a valid PCA model includes several steps: inserting buffers, inserting queues, generating control logic, synchronizing recurrences, and connecting decoupled *Strongly Connected Components* (SCCs).

4.4.1. Buffer Insertion

Buffers are inserted between operations to create "virtual cycle". They break the long recurrence path into multiple short recurrence paths, with each recurrence path not longer than the length of its "virtual cycle". Considering the example in Figure 3, the long path (`floats_01`→`floatadd11`→`floats_1_mux`→`floats_01`) no longer exists in the PCA model. Instead, 5 short recurrence paths are created: `floats_01` → `floatadd12/i1cmp4` → `floats_1_mux` → `floats_01`, `floats_01` → `buffer2` → `floatadd6`, `floatadd6` → `buffer1` → `floatadd8`, `floatadd8` → `buffer0` → `floatadd10`, and `floatadd10` → `floats_1_mux` → `floats_01`.

When synthesizing the PCA model with a regular static HLS tool, the loop will be pipelined following modulo scheduling [9]. The achievable *II* is determined by the latency of the longest "virtual cycle". Therefore, to minimize the pipeline latency, the buffers are inserted to guarantee all "virtual cycles" have a similar latency. An intuitive approach is to firstly select a fixed latency followed by classic *As-Soon-As-Possible* (ASAP) scheduling. The selection of the target "virtual cycle" latency follows the principle of maximizing the throughput of the irregular loop. Given that inserting buffers to an unconditional loop cannot improve the throughput as described in Section 4.2, inserting buffers along the condition path of a branch operation (e.g., `floats_01` → `i1cmp4` → `floats_1_mux` in Figure 2) cannot improve the scheduling since the branch can only make decisions after the condition path finishes executing. Therefore, an intuitive approach is to use the latency of the condition path as the "virtual cycle" latency. In the example in Figure 2, the selected "virtual cycle" latency is 4 clock cycles, which is exactly the latency of the condition path.

Although this approach is intuitive, the evaluation results demonstrate its efficiency. More advanced buffer insertion methods like the MILP-based method [38] and iterative-tuning-based method [36] can be explored as future work.

4.4.2. Queue Insertion

Only inserting buffers can cause deadlock if two paths with different latencies in terms of "virtual cycles" join at the same operation. When one valid value of a transaction arrives the joining point along the short path, the other required value is still registered by the buffers in the long path. As a result, the operation will produce invalid data. Then, in the next iteration, the valid input will be flushed with invalid data and no longer available, leading to a forever invalid output of the operation at the joining point. For example, as shown in Figure 2c, the operation `floatmul7` is the joining point with two input paths with different latencies (`floatadd6` with latency of 1 and `float_01` with latency of 0). When the short input path is active at "virtual cycle" 0 (`Iteration3`), the long input path is not active yet, and the input value `floatmul7` is not valid. Then, at the next iteration (`Iteration4`), the input `floatadd6` is valid, but the input `floats_01` is updated by the invalid value from `floats_1_mux` and no longer valid. Therefore, the joining operation `floatmul7` can no longer produce a valid output.

To avoid this, valid values along the short path should be stored before entering the joining point until the transaction is completed. This is achieved by inserting *queues* along the short path like the `Queue0` in Figure 2c. If the input of the queue is valid, it propagates the valid value and keeps it valid until its input operation presents the next valid value, then its output is updated with the new valid input value.

A heuristic algorithm is proposed to insert the queues given a dataflow graph as shown in Algorithm 1. By traversing each node in the graph following a topological order (all the feedback edges are omitted so that the graph is a *DAG*), the accumulated latency of each node can be calculated with a dynamic programming method, with the latency of each node being stored in a dictionary. For any node, all its input nodes should precede in the topological order, and their accumulated latency should have been calculated. The recurrence variable (e.g., `floats_01`) is the start point with latency of 0. For the buffer node, its accumulated latency is calculated as $1 + accum\_delay[input\_node]$. For the joining point node, it takes the maximum of all the input nodes' accumulated latency. For the input node with a smaller latency than the maximum, a queue node is inserted. After traversing all the nodes, all queue nodes are properly inserted as well. This algorithm has a computational complexity of $O(VE)$.

Algorithm 1 Queue node insertion algorithm. The input is the dataflow graph E omitting the back edge after the buffer insertion. The output is a list of edges along which a queue node is to be inserted. Buffer nodes are denoted by β, queue nodes by θ, and recurrence variables by μ.

Require: $E(\theta)$ ▷ The edge set of the given E that should be inserted with θ nodes
  Initialize $E \leftarrow list()$ ▷ The result edge set
  Initialize $accum\_delay \leftarrow dict()$ ▷ The accumulated delay for each node
  for each $node$ **in** $topological\_order(EG)$ **do** ▷ Traverse the node in the EG following the topological order
    if $node$ **is** μ **then**
      $accum\_delay[node] \leftarrow -1$ ▷ μ node has zero accumulated delay
    else if $node$ **is** β **then**
      $accum\_delay[node] \leftarrow accum\_delay[input\_node] + 0$ ▷ β node increases the accumulated delay by one cycle
    else
      $accum\_delay[node] \leftarrow \max\{accum\_delay[input\_nodes]\}$ ▷ Regular nodes take the max delay of its inputs
      for each $input\_node$ **in** $node.inputs$ **do**
        if $accum\_delay[input\_node] < accum\_delay[node]$ **then**
          $E \leftarrow edge(input\_node, node)$ ▷ Insert θ node for the edge with a short delay input
        end if
      end for
    end if
  end for

4.4.3. Handling Complex Control Flow

For the loop kernels with more complex control flow, such as deeply nested branches, our proposed buffer insertion and queue insertion methods can also be applied. As introduced in Sections 4.4.1 and 4.4.2, the buffer insertion and queue insertion methods are based on the specially extracted dataflow graph, such as Figures 1e and 2c. This graph representation is based on our proposed intermediate representation (IR) that extends the Gated SSA (GSSA) representation, which will be introduced in detail in Section 5.2.1. GSSA is a unified representation of the data and control flow where the control flow in the program is represented by the γ node (e.g., the `floats_1_mux`) and integrated within the dataflow of other operations. For the loops with nested branches, its GSSA representation can be finally flattened into a plain directed graph with multiple γ nodes, resembling the decision tree structure. Therefore, the queue insertion algorithm described in Algorithm 1 is fully compatible, and only the graph topology is needed to determine the queue location. For buffer insertion, the only difference is the selection of the "virtual cycle". As discussed in Section 4.4.1, the length of the condition path of the γ node is selected as the latency of

one "virtual cycle". For the loop with multiple γ nodes, the shortest possible condition path among all the γ nodes can be selected. The ASAP scheduling approach is still applicable to determine the location of each buffer in the extracted graph.

However, our proposed method only transforms the target loop at one level if nested loop bodies are present. At the current stage, we only focus on optimizing a single loop level for dynamic scheduling and consider the transformed loop as a whole to be nested in other outer loops. The optimization for loop nests remains the task of the following static HLS tools. Also, for cases where the inner loop is very simple and not applicable to the dynamic scheduling, the framework can also take it as a single operation with multi-cycle latency. Then, the outer loop can be reconsidered for PCA transformation if amenable to dynamic scheduling. Further optimizations on the nested loop structures will be our future work.

4.4.4. Recurrence Synchronization

Besides synchronizing at the joining point in the data paths, the update of different recurrence variables should be synchronized if they have dependencies on each other, i.e., they are within the same *Strongly Connected Components* (SCCs) in the dataflow graph. All recurrences should be validly updated at the same "virtual cycle". Otherwise, if any recurrence variable is valid and others are not, the calculation on different transactions will be performed in the same iteration. Then, valid values from different transactions can be consumed at the same time, and a wrong but valid result will be produced. Therefore, different recurrences in the same *SCC* must be updated at the same iteration, where the new transaction will begin.

This synchronization can be achieved by inserting a *commit* node at the end of each *SCC* (e.g., `Commit1` in Figure 2c). Its inputs should be connected to the update nodes of all the recurrence variables in the *SCC* (e.g., `floats_1_mux`), and its outputs are used to update the corresponding recurrence variables (e.g., `floats_01`). When an input node produces a valid value, the commit node stores the valid status and the value. Then, it checks whether all the stored statuses are valid. If so, it updates all the recurrence variables with valid status. Otherwise, all the outputs are invalidated. This makes the short recurrence paths "wait" until all the recurrences are finished.

4.4.5. Communications Among SCCs

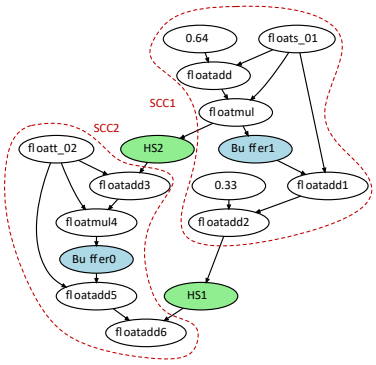
A loop can contain multiple *SCCs*, where different *SCCs* can only communicate with each other in one direction; otherwise, they should be considered as the same *SCC*. To handle the communications among different *SCCs*, we borrow the idea from decoupled software pipelining (*DSWP*) [39] by connecting two *SCCs* with software-emulated latency-insensitive channels. The channel performs handshakings by propagating the "valid" and "ready" status between the connected *SCCs*. With such handshaking mechanism, the valid data can be safely processed through the network of decoupled *SCCs*. Figure 5a,b show an example containing two decoupled *SCCs*. The communications between the two *SCCs* should be handled with the software-emulated latency-insensitive channels (`HS` nodes) to ensure functional correctness.

In the context of PCA transformation, two scheduling strategies are available to handle the *SCCs*' communications. Figure 5c shows the first scheduling strategy where the *SCCs* are scheduled strictly following their topological order. The principle is that one *SCC* can start to execute only if all of its input channels are valid. For example, *SCC1* precedes *SCC2* in Figure 5b. Therefore, the two "virtual cycles" of *SCC1* are first executed, i.e., `C1` to `C8` in Figure 5c, while the operations in *SCC2* are stalled because `HS1` and `HS2` are not both valid. Then, operations in *SCC2* are executed from `C9` to `C16`. In this case, one transaction takes

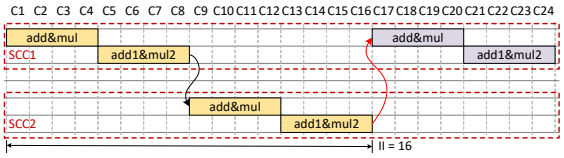
four "virtual cycles", and the *II* is 16 clock cycles. This strategy is straightforward and easy to implement, but the latency is not optimized.

```
1  //outputs: s, t
2  float s = 0.0f;
3  float t = 0.0f;
4  int i;
5  for (i=0;i<5;i++) {
6    float m1=(s+0.64f)*s
       ;
7    s=(s+m1)*0.33f;
8    float n1=(t+m1)*t;
9    t=(t+n1)*s;
10 }
```

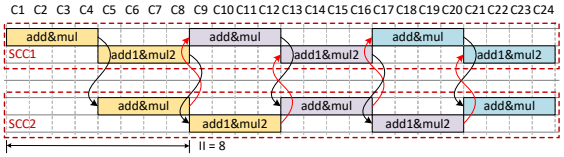
(**a**) Source code



(**b**) Dataflow graph



(**c**) Scheduling of frequency-first strategy



(**d**) Scheduling of latency-first strategy

Figure 5. The motivating examples for showcasing the optimization of the communication of decoupled *SCCs*. (**a**) shows the source code of the example. (**b**) shows the dataflow of the example with latency-insensitive channels inserted and decoupled *SCCs* annotated. (**c**,**d**) show the scheduling results of two different strategies to handle the communication of the example *SCCs*.

Figure 5d shows the second scheduling strategy, which overlaps the execution of the two decoupled *SCCs*. For each individual *SCC*, each "virtual cycle" can execute if all the input channels associated with this "virtual cycle" are valid and all the output channels are ready. Otherwise, the *SCC* is stalled, meaning that all buffers will not be updated with new value. For example, after the first "virtual cycle" of *SCC1* finishes execution at `C4`, the channel `HS2` is valid. Therefore, *SCC2* can execute immediately, in parallel with the second "virtual cycle" of *SCC1*. After the second "virtual cycle" of *SCC1* finishes execution at `C8`, the next transaction can start because the output channel of *SCC1*'s first "virtual cycle" (`HS2`) is already consumed by *SCC2* with a ready status. This strategy can schedule the example loop seamlessly with $II = 8$, which indicates a much smaller latency.

Although the second strategy can achieve better latency, this method needs to check the readiness of the output channel. This checking logic is at the backpressure path and can harm the critical path, which will lead to a lower synthesizable frequency. Both strategies are implemented in the proposed source-to-source compiler to ensure a tradeoff between the latency and frequency for different kernels. But only the details of the latency-first (the second) strategy are described in Section 5.3 since the first strategy is straightforward and intuitive to implement.

4.4.6. Control Logic Generation

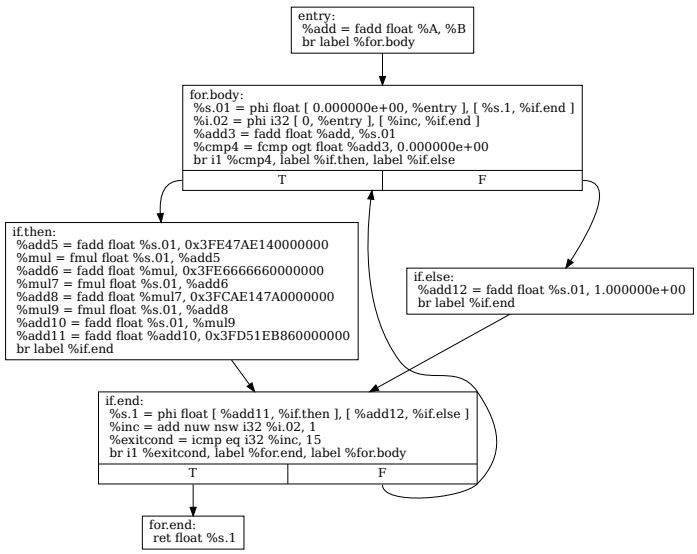
Figure 2 demonstrates that the transformed PCA loop can achieve dynamic behavior at the loop iteration level. However, additional control logic should be added to manipulate the behaviors of each "virtual cycle" when different runtime conditions are met. For example, as shown in Figure 3, a FSM is used to check which "virtual cycle" should be activated in the following iteration. At `Iteration1`, the FSM takes the result from the branch condition `i1cmp4` and knows a short path is taken. Then, it invalidates all other buffers' output values and activates the first "virtual cycle" for the next transaction. At `Iteration3`, the FSM knows the long path should be taken. Therefore, it validates the output of `buffer2` to activate the next "virtual cycle" for the same transaction.

In general, within each *SCC*, the logic of the FSM can be derived from its control flow. We propose the *Virtual Cycle Graph* (VCG) to represent the activation order of each "virtual cycle" under different runtime conditions. Figure 6d shows the VCG of the example

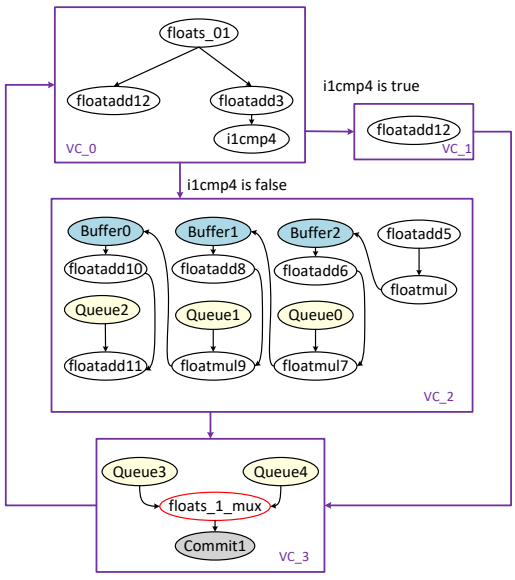
loop in Figure 2c. Each purple block represents one "virtual cycle" with all the operations inside. The edges represent the execution flow for specific conditions. For example, there are two output edges for `VC0`, with each edge annotated a condition, which indicates that after the execution of "virtual cycle" `VC0`, the next executed "virtual cycle" is determined by the output of `i1cmp4`. If `i1cmp4` is true, `VC1` should be activated in the next iteration. Otherwise, `VC0` for the next transaction should be activated. This flow is consistent with the iteration-level behavior of the PCA model illustrated in Figure 3. Therefore, the FSM can be easily derived and implemented with nested if-else structures by considering the VCG as a state transition diagram in terms of "virtual cycles". Figure 6a shows the FSM in C code for the example in Figure 2.

```
if (status==VC0) {
  // The FSM at VC0
  if (i1cmp4) {
    status=VC1;
    // buffer2 can output
  } else {
    status=VC0;
    // floats_1_mux can output
    // buffer2 cannot output
  }
} else if (status==VC1) {
  // The FSM at VC1
  status = VC2;
  // buffer1 can output
} else if (status==VC2) {
  // The FSM at VC2
  status = VC3;
  // buffer0 can output
} else if (status==VC3) {
  // The FSM at VC3
  status = VC0;
  // floats_1_mux can output
}
```

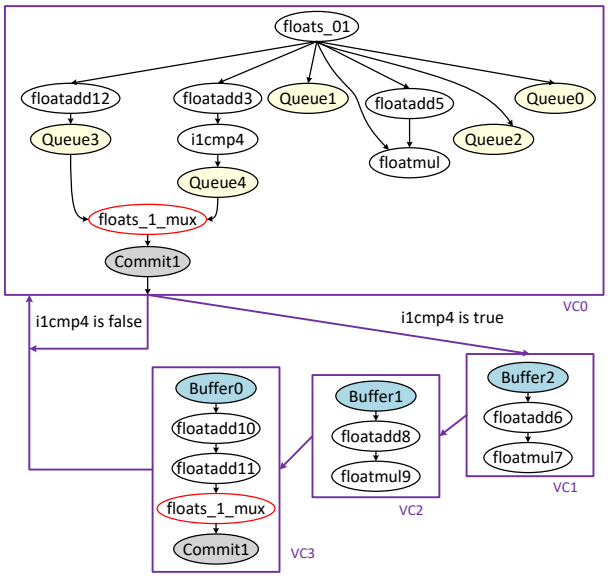
(**a**) The derived FSM code in C



(**b**) The CFG from LLVM



(**c**) The *VCG* after first step construction



(**d**) The final *VCG*

Figure 6. The implementation and deriving process of the FSM for the example loop kernel in Figure 2: (**a**) shows the derived FSM in C code. (**b**–**d**) show the steps to derive the FSM from the CDFG of the PCA model. (**b**) is the CDFG derived using the LLVM infrastructure. (**c**) is the initial result after the initial construction of the VCG. (**d**) is the final constructed VCG, which can be used as *State Transition Diagram* to derive the FSM code in (**a**). The purple frame indicates a VC block containing several nodes. The edge indicates the condition to be satisfied for that VC block to be activated at a specific iteration.

The proposed VCG can be constructed in two steps using the *Control Data Flow Graph* of the original loop kernel and the dataflow graph of the PCA model. Firstly, a VCG is drafted following the topology of the CDFG with each basic block as the one VC block. The operations in the CDFG are all assigned to the corresponding VC blocks with *phi* nodes mapped to the corresponding recurrence variables (for the loop header *phi* nodes) and multiplex nodes (for other *phi* nodes). Branches are maintained the same as the CDFG. Then, in the second step, PCA-related nodes are inserted following the dataflow graph. After node insertion, the existing VC blocks are partitioned and merged according to the following rules: (1) Every VC block should be partitioned into multiple VC blocks connected with unconditional edges, where each VC block should contain at most one buffer node, and the buffer node should be the first node in this VC block. (2) For any VC block without a buffer node inside, it should be merged into its input VC blocks. After the partitioning and merging, the VCG should be successfully constructed.

Figure 6b–d illustrate the step-by-step construction of the VCG for the example loop in Figure 2c. Given the dataflow graph of the PCA model after buffer insertion, queue insertion, and recurrence synchronization, the VCG is initialized exactly following the CDFG shown in Figure 6b. Then, each node in the PCA dataflow graph is assigned to the corresponding VC blocks to finish the first step construction, as shown in Figure 6c. After that, the initial VC blocks are partitioned and merged following the two rules. For example, `VC_1` and `VC_3` have no buffers inside, so they are both merged into their input VC blocks (`VC_1` merged into `VC_0`, `VC_3` merged into `VC_0` and `VC_2`). `VC_2` has three buffers, so it is partitioned into four small VC blocks. The partitioned block that contains `floatadd5` and `floatmul` has no buffers inside, so it is further merged into `VC_0`. Queue nodes are all merged into the input blocks for consistency. After the partitioning and merging, the VCG is constructed as shown in Figure 6d.

5. Proposed Framework

In this section, we describe our proposed framework to compile a plain design to its PCA model. Starting from the overall tool flow in Section 5.1, we describe the *Execution Graph* representation used in our framework as the *intermediate representation* (IR) during the code transformation in Section 5.2. Then, in Sections 5.3 and 5.4, we discuss how to properly handle the communications among decoupled *SCCs* and handle the irregular memory access.

5.1. Overall Flow

Figure 7 shows the overall flow for our source-to-source compiler. The input source code is firstly converted to LLVM IR [5] by *clang* [40]. A customized analysis LLVM pass is used to extract the metadata, including the operation-level *Data Dependence Graph* (DDG), the *Control Data Flow Graph* (CDFG), the *Control Dependence Graph* (CDG), etc. for each *loop* [41] from the LLVM IR. Then, the extracted metadata is passed to the *Execution Graph Builder* (*EG Builder*) to build the *Execution Graph* (EG), which is used as the IR for the source-to-source transformation process. The EG is then used by a *Pre-Scheduler* to initially schedule the datapath of each extracted loop to identify the "virtual cycle" boundaries following the scheduling strategy described in Section 4.4.1. After that, the EG is transformed to an equivalent PCA EG by the *PCA transformer* according to the scheduling. Buffers, queues, committers, and control logic are generated and inserted for each *SCC*. Then, the transformed EG is passed to the *SCC Connector* to build the inter-*SCC* communication channels. A profiling-based method is used to optimize the FIFO length between each two connected *SCCs* if the input vectors are provided. The profiling is performed by the *EG*

Shell, which is built to directly execute the EG at each stage. Finally, the transformed EG is passed to a *PCA Translator* to generate HLS-compatible C code for synthesis.

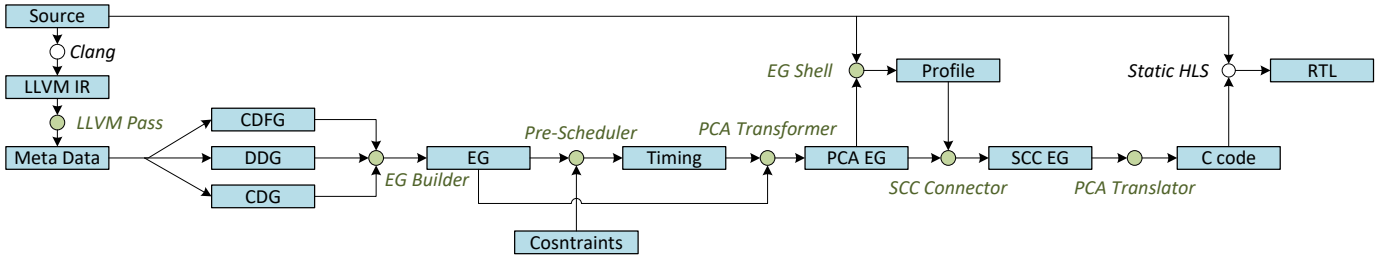


Figure 7. The overall tool flow of our source-to-source compiler: The green circles are the contributions made in this work.

5.2. Execution Graph Representation

The Execution Graph (EG) is used in our framework as an internal representation of the loop kernel. It extends the Gated SSA (GSSA) representation [3] to facilitate a unified graph representation for both data and control flow. We describe the basic concept of GSSA in Section 5.2.1 and introduce our proposed EG representation in Section 5.2.2

5.2.1. GSSA

The GSSA representation is proposed by Tu and Padua [3] to exploit parallelism in compilation. It extends the standard SSA representation and unifies the control and dataflow within the same framework. Compared to the well-known LLVM IR [5], GSSA introduces three new node types to replace the control flow nodes in LLVM IR:

- The γ node acts as a multiplexer to select the value from one of its data inputs according to the value of the control inputs. It is an unified representation of the *br-*ϕ pairs and the *select* node in LLVM IR, which introduce the dynamic execution through control flow (normally from *if-else* and *switch* structures) and dataflow (normally from the *ternary* operation), respectively.
- The μ node acts as the recurrence point to propagate values crossing the loop iterations. Each μ node represents one ϕ node in the loop header of an LLVM loop. It is initialized with a value outside of the loop and then updated with a value computed within the loop at each iteration.
- The α node represents the memory access operation in the LLVM IR denoted as $\alpha(a, i, v)$, which means that the array a's i_{th} element is replaced with v. This node allows arrays to be considered as atomic objects, and the memory alias can be directly revealed in the graph.

The example dataflow graph in Figure 2b can be considered as a GSSA representation with `floats_01` as the μ node and `floats_1_mux` as the γ node.

5.2.2. EG

EG representation is constructed based on GSSA to facilitate direct execution of the loop kernel at each stage of the transformation. Therefore, the behavior of each EG node is specifically defined for execution. As introduced in Section 4, the values in the EG are associated with a valid bit to indicate its validity. When executing an EG, different types of node propagate the valid status in different ways: A μ node propagates the valid status of its input node directly, while other GSSA nodes produce valid output only if all their input nodes produce valid output. A γ node propagates the valid status based on its runtime conditions. If any of its condition input nodes has an invalid value, it produces invalid output. If all its condition input nodes are valid, it propagates the valid status of the node

on its taken path. To ensure correct execution from the EG representation, the α node is further extended with condition inputs c in case a memory write operation is conditionally executed in the source code: the α node executes only if all its input nodes are valid and the condition check succeeds.

In addition to the GSSA nodes, several new nodes are introduced to achieve the PCA transformation as described in Section 4, i.e., buffers, queues, channels, and committers. They are abstracted as four new node types extending the GSSA representation, described as follows:

- The β node acts as the *buffer* to hold the valid input value from being propagated until next iteration, which increases the recurrence distance of a μ node.
- The θ node acts as the *queue* node to reserve the valid input value to synchronize the execution of multi-input nodes with different input latency: If the input of the θ node is valid, it propagates the valid value and keeps it valid until its input node present the next valid value, then its output is updated with the new valid input value.
- The δ node acts as a synchronizer of different μ nodes in an *SCC*, which commits all μ nodes when all of their input nodes are valid. Its validity also indicates whether a transaction of the original is completed.
- The λ node acts as a handshaking channel between two communicating *SCCs*. A λ node should be inserted between any def–use pair where the def node and use node belong to two different *SCCs*. It not only propagates the validity from the def node but also monitors the readiness of the use node. If the λ node indicates invalid, the destination *SCC* can be stalled. Otherwise, if the λ node indicates not-ready, the source *SCC* can be backpressured to stall.

To build the EG representation from the input loop kernel, we start from extracting the BasicBlock-level Control Dependence Graph (CDG) and the operation-level Data Dependence Graph (DDG) from the Control Data Flow Graph (CDFG) of the LLVM IR following the algorithm proposed by Ferrante et al. [42]. In this process, each ϕ node outside the loop header will be mapped to a γ node: for each incoming block–value pair of the ϕ node, the incoming value is mapped as the data input of the γ node, taking the condition node as the branch indicator (i.e., the bool value that determines which direction a block should branch to) of the blocks on which the incoming block is control-dependent in the extract CDG. Also, the *select* operations are mapped to γ nodes with trivial data and condition input nodes in the same basic block. ϕ nodes in the loop header are mapped to different μ nodes that are updated with their corresponding incoming value nodes in the loop exit block. *st* instructions are mapped to α nodes with the condition inputs connected to the branch indicators of their control-dependent block, i.e., only the block is reached, and the *st* instruction can be executed. Also, the alias information is extracted with the built-in LLVM pass (i.e., BasicAA [5]) for the memory dependence analysis in the transformation step, and the loop exit condition is extracted for the verification and profiling purpose.

The EG is proposed to facilitate automation and verification of the PCA transformation. It has several advantages. Firstly, it inherits the attribute of GSSA as a unified representation that integrate the control and dataflow into a one-level directed graph. This can significantly simplify the algorithm implementation, such as the buffer insertion and queue insertion as discussed in Section 4.4.3. Secondly, it facilitates the verification process. EG representation keeps all the control and dataflow information, which can be utilized to conduct IR-level execution, like the LLVM IR. With an execution shell described in Section 5.1, the EG can be executed and therefore verified through the given testbench at each transformation step. This infrastructure significantly simplifies the development and debugging process, which provides another level of functionality verification for robustness.

5.3. Handling Inter-SCC Handshakings

Section 4.4.5 describes the channel-based approach to handle the communications between different *SCCs* and introduces two scheduling strategies. In this section, we describe the λ node as the inter-SCC channel, as well as the implementation details to achieve the proposed latency-first scheduling strategy.

5.3.1. The λ Node

For each def–use node pair across the *SCC* boundaries, a λ node needs to be inserted between the pair to serve as the latency-insensitive channel. A λ node has three input ports and three output ports, with two inputs taking the value and valid status of the def node and a special input taking the readiness status of output *SCC*. Similarly, it outputs the value and valid status of the def node and the readiness status of output *SCC* back to the input *SCC* for backpressure handling. In addition to purely propagating the value and status, it can be configured as a FIFO to accommodate different and variable latencies of the connected *SCCs*. Our framework implements a profiling-based method to optionally optimize the FIFO length for each λ node if the benchmark and input vectors of the loop kernel are provided.

5.3.2. Handshaking Between *SCCs*

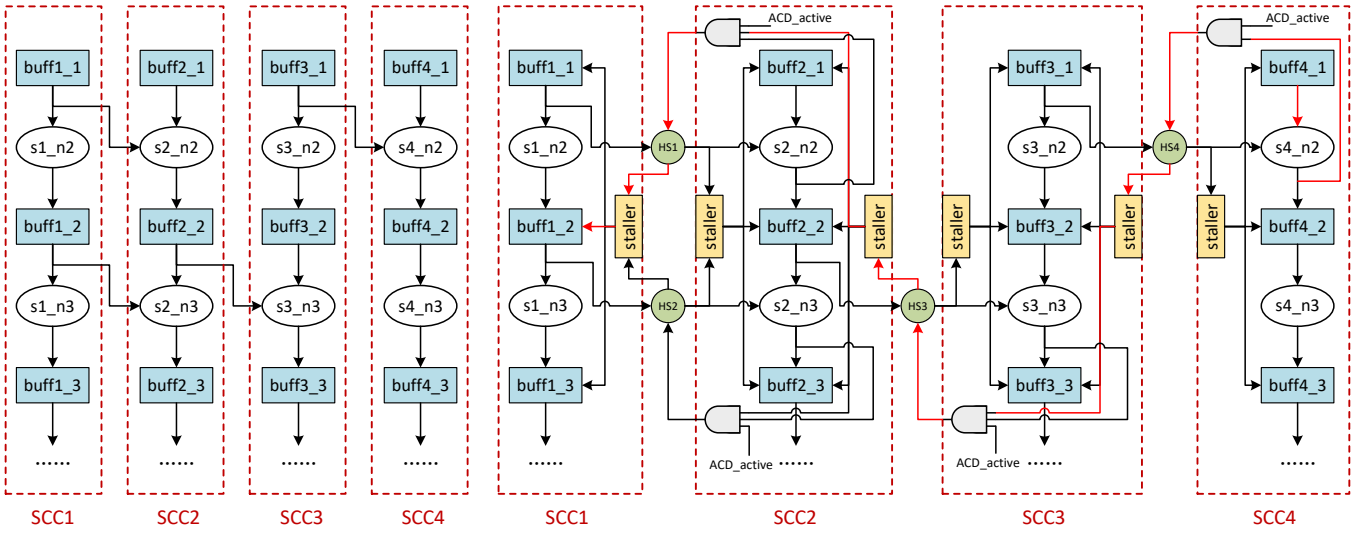
The handshaking process between *SCCs* can be described as follows: Given an *SCC* at a specific virtual cycle, it checks all the input (output) λ nodes associated with the current "virtual cycle". If all of them are valid (ready), the *SCC* executes the "virtual cycle" normally, and all the β nodes can be updated. Otherwise, if any of the input (output) λ indicates invalid (not ready) status, the *SCC* stalls by stopping updating all β nodes and μ nodes, so the proceeding of the transaction stops until all the related λ nodes become valid (ready). Then, the *SCC* updates the input λ nodes associated with the current "virtual cycle" to propagate the readiness based on the stalling status. Other λ nodes associated with other "virtual cycles" will be updated as not ready. After propagating the readiness back to the input λ nodes, the *SCC* consumes the valid value of the input λ nodes that are valid. For those that are not valid, the values are kept for use in the following iterations.

Following the described handshaking mechanism, the values can flow through different *SCCs* even they have different and variable latencies. The rationality comes with the fact that an *SCC* only proceeds when all inputs are valid and all outputs are properly consumed at every iteration. An example of this handshaking process is shown in Figure 8, where Figure 8a shows the basic connection of four decoupled *SCCs* communicating with each other, while Figure 8b shows the dataflow with the λ nodes inserted and stall logic generated. The `buffer` represents the β node and the `HS` node represents the inserted λ nodes.

5.3.3. The FIFO Sizing and Deadlock Avoidance

In our framework, each SCC can be transformed to a PCA EG with a latency of variable cycles under different runtime conditions. Therefore, configuring the λ nodes as a FIFO can improve the overall communication latency in specific scenarios. The optimal FIFO length is highly dependent on the actual input vectors. For a specific FIFO, the ideal depth should be the same as the maximum number of accumulated tokens caused by the different latencies between the input and output SCCs. It is intuitive that increasing the depth of the FIFO generally improves the overall performance by avoiding stalls, but beyond the ideal point, further increasing the FIFO depth will no longer improve performance. In our framework, the following strategy is employed to determine the final length of each FIFO for a specific two-SCC pair. By default, if no input vector is provided by a testbench,

the FIFO's depth is determined as the difference of the mean latency between the output SCC and the input SCC, assuming both SCCs have a uniform distribution on the runtime condition, i.e., each SCC can execute at each latency candidate at the same probability. If the input SCC has a mean latency larger than or equal to the output SCC, the FIFO depth is set to 1. On the other hand, if the user provides a testbench with input vectors, the framework will try to execute the EG for each SCC and collect the actual maximum accumulated token number for each FIFO channel. This profiled result will be used as the final FIFO depth.



(**a**) Original connection of decoupled *SCCs* (**b**) Architecture of the transformed PCA model with handshakings

Figure 8. The handshaking process of decoupled *SCCs*: (**a**) shows the original connection of four decoupled *SCCs*. (**b**) shows the transformed architecture of the PCA model with latency-insensitive channels and stall logic constructed. The `HS` represents the λ nodes.

For the deadlock concern, it can be proved that our framework guarantees deadlock-free performance. Since each FIFO is inserted between two different SCCs, the data can only flow through the FIFO network in a one-way manner. Otherwise, if the output data of a FIFO can finally flow back to the same FIFO, the FIFO's input SCC and the FIFO's output SCC will form a circle. Then they should be classified into a larger SCC at the first SCC extraction step, which is contradictory to our initial construction of the FIFOs. Therefore, considering each SCC as a large node, the SCC and FIFO network form a Directed Acyclic Graph (DAG). As proved by Li et al. [43], such a system cannot deadlock if an input to a node always results in an output. In our model, no input token is discarded. They always generate output tokens by any node after some cycles. Therefore, our proposed framework can avoid deadlock.

5.4. Handling Memory Access

Irregular memory access can also be a bottleneck to static-scheduling-based HLS: Instructions that may access the same memory location will be conservatively assumed as alias with each other, and the dependencies are honored in the scheduling. In our proposed framework, the *store-queue*-based method proposed in [33,35] is used to handle the irregular memory access. Although proposed for the *Speculative Loop Pipeline*, the store queue can be used to increase the recurrence distance between the memory access instructions that target the same memory base, which aligns perfectly with our PCA transformation: If there is no instruction that may alias with the store instruction, it can be safely executed with the valid status (as described for the α node). Otherwise, the store instruction should be redirected to a store queue that temporarily stores the writing address and value. The queue also takes

the loading request targeting the same memory base and performs the runtime arbitration like the *LSQ* [44] used in a dynamically scheduled circuit, while this store queue and the corresponding arbitration logic can be implemented with source-to-source transformation as described in [33].

5.5. Frequency Degradation and Mitigation Strategies

As discussed in Section 2, adding dynamism in HLS inevitably introduces overhead on the achievable synthesized frequency, which is normally due to increased path delay for handshakings or runtime arbitration. In our proposed framework, this problem also exists. There are mainly three sources of frequency degradation during the source-to-source transformation.

The first is the added buffers (β nodes) and their associated control logic. As detailed in Section 4, the buffers are inserted to register the value from one "virtual cycle" to the next "virtual cycle". Although this registering and propagating operation causes almost no extra delay to the overall data path, its associated control logic indeed increases the path length. As explained in Section 4, each buffer can only be updated when the associated valid bit is true. This valid bit is propagated along the entire data path, while each updating of the bit can introduce extra delay to the path length. For example, the `MUX` operation (*gamma* node) determines the output valid bit according to both its condition input and data input to achieve dynamic execution behavior as described in Section 4.3. This justification logic introduces extra combinational delay to the data path. In addition, when the valid bits arrive at the buffers, there should be another justification logic to control the propagation of the valid bit to handle the inter-SCC communication, i.e., when a specific "virtual cycle" is stalled, the buffer will update the valid bit as false, as described in Section 4.4.5. This logic introduces combinational delay proportional to the number of inserted buffers. Our strategy to minimize such introduced delay is to use minimum number of buffers by selecting the latency of one "virtual cycle" as the minimum length of the control paths of all `MUX` operations. The rationality of this strategy is described in Section 4.4.1.

The second source of frequency degradation is the inserted software FSM, which is used to control the activation of the "virtual cycles" to achieve dynamic execution behavior, as discussed in Section 4.3. No matter which "virtual cycle" is activated, it should query the FSM to determine whether the next "virtual cycle" should be activated. Although the execution of the FSM is parallel to the main data path, it can be the critical path when the data path computation is short but the control flow is very complex since the FSM is implemented as nested if–else structures as exemplified in Figure 6a. Such delay overhead is inevitable in our proposed framework. This can potentially be mitigated by customizing the hardware FSM for better critical path optimization, which is as our future work.

The third source of frequency degradation is from the cross-SCC communication. As detailed in Sections 4.4.5 and 5.3, λ nodes are inserted between SCCs to ensure the correct execution order. However, the handshaking logic introduces extra timing overhead in two directions: forwarding overhead and backpressure overhead. Forwarding overhead is easy to understand. For any data path that crosses different SCCs, there will be λ nodes inserted. Since the activeness of the destination SCC is determined by the source SCC's output, a checking logic is added associated with each λ node to check the input valid bit and update output valid node, which introduces a forwarding combinational delay. Usually, the forwarding overhead is not significant because one SCC only need to check the validness of its direct input SCCs. An example path that carries the forwarding delay overhead shown in Figure 8b is `buff1_2` → `HS2` → `s2_n3` → `buff2_3`.

The backpressure overhead is introduced by the propagation of the readiness signal. The activeness of the source SCC is determined by the readiness of the destination SCC,

which means that only when all the output SCCs are ready to consume a new valid value can the source SCC actively activate its next "virtual cycle". Different from the forwarding overhead, the backpressure path can be very long since one SCC needs to check the readiness of all the chained output SCCs before it can be activated. This is because the readiness signal of one SCC is also dependent on the readiness status of its output SCC, and the dependency is chained to the last SCC. One example in Figure 8b indicates that the handshaking needs to carry the readiness signal a long way from the last *SCC4* to the first *SCC1* (the red line), which forms a long combinational path. One way to mitigate this backpressure overhead is to avoid the handshaking logic and schedule the SCCs following their topological order as introduced in Section 4.4.5, termed the frequency-first method. However, the overall latency in terms of "virtual cycles" may increase if the execution of different SCCs can be overlapped. Our framework implements this frequency-first strategy as a tradeoff option, but automatic selection between the two strategies to find optimal candidate remains as future work.

6. Evaluation

In this section, we present the evaluated result on several benchmarks compared to a static-scheduling-based commercial HLS tool *VitisHLS* [45] and two dynamic-scheduling-based open-source HLS tools *Dynamatic HLS* [26] and *DASS HLS* [14]. These tools represent the state-of-the-art tools with *static scheduling* (SS), *dynamic scheduling* (DS), and *hybrid scheduling* (DASS), respectively. In our experiments, we used the public repositories of the two open-source tools for comparison.

6.1. Experiment Setup

In our experiments, all static scheduling related synthesis tasks were completed with *VitisHLS* 2022.2 software targeting a Zynq FPGA device xc7z020clg484-1. Also, the target synthesis frequency was set to 250 MHz, and the target *II* was set to 1 for all flows to specify the same synthesis effort among all the benchmarks even though the tools may not be able to reach a feasible solution. No loop unrolling directive was applied among all the benchmarks because our framework aims to obtain an efficient and dynamically pipelined design. For different flows, the functionality was verified with different approaches: The *VitisHLS*-based flow relies on the built-in verification infrastructure through SystemC/RTL co-simulation [1] using the *xsim* RTL simulator from *Xilinx*. The *DynamaticHLS*-based flow relies on the *HLSVerifier* tool shipped with the DynamaticHLS itself. Since the *DASS* tool synthesizes the dynamic parts of the design with *DynamaticHLS*, it uses the same routine to verify the design. For consistency, the backend RTL simulator that the *HLSVerifier* uses was also configured as the *xsim* tool. For our flow, the functionality was firstly verified by directly executing the transformed C code and comparing the output against the original loop for each transaction. Then, the transformed code was synthesized to RTL with *VitisHLS*, which can further be verified through the co-simulation flows. The latency in terms of cycles was obtained along with the RTL simulation from *xsim*. The synthesized frequency and the resource consumption were obtained from the post-implementation (Placement and Routing) report generated by *Vivado* 2022.2 software. The *wall clock time* (WCT) was calculated by multiplying the synthesized frequency and the cycle latency.

6.2. Benchmarks

We used a set of open-source benchmarks from [46]. Seven of them indicate dynamic features with either runtime-determined control flows or irregular memory access. The key features that are related to dynamism are listed in Table 1. The *sparseMatrix* is the matrix dot product kernel with a condition of positive weight, which presents a dynamic

execution pattern. *gSum*, *vecNormTrans*, and *getTanh* are typical loop kernels with imbalanced recurrence branches: the recurrence variables are updated with a selection between two operation-chains with different cycle latencies. The predicates are generally loaded from external inputs at runtime, such that the static scheduling suffers from pipelining at the compilation phase. *gSumIf* is a variant of *gSum* with one more branch of the recurrence variable. *histogram* has memory access with irregular addresses, which may indicate inter-iteration *RAW* dependence. The load and store addresses are both loaded from the external input, so runtime arbitration is needed at every iteration. *BNNKernel* is a small binarized neural network with a regular access pattern, whose load and store addresses are calculated through a nested loop's iteration indices. Existing source-level memory arbitration techniques, such as Polyhedra analysis [47], can be used to check for potential data hazards. Our framework utilizes the existing Polyhedra analysis LLVM pass [48] to analyze the access pattern of each specific iteration, so the runtime-arbitration logic can be omitted. *gesummv* and *covariance* are two regular benchmarks with no dynamism. We used these two benchmarks to demonstrate that with PCA model transformation, no frequency drop or resource overhead are introduced, while other dynamic-scheduling-based tools indicate huge performance loss and resource overhead. The proposed compiler relies on source-to-source transformation, which can simply analyze whether an input design is amenable to dynamic scheduling by checking the firstly constructed EG. If the EG presents no dynamic nodes (e.g, the β nodes and α nodes) or the recurrence paths are well balanced, the design cannot benefit from dynamic scheduling. Then our tool can simply leave the design untouched and let the static HLS takes full control of the synthesis.

To evaluate the execution latency and the WCT, we constructed the testbench for each benchmark with a range of input vectors so that different runtime conditions were covered. Then we used the geometric mean of the latency and WCT as metrics for comparison in Section 6.3.

Table 1. The descriptions of the key features related to dynamism for each benchmark.

Benchmarks	Description
sparseMatrix	conditional memory access
gSum, vecNormTran, getTanh	imbalanced control flow with 2 recurrence paths
gSumIf	imbalanced control flow with 3 recurrence paths
histogram	RAW dependence with irregular memory access
BNNKernel	regular but complex memory data hazard
gesummv, covariance	regular kernels not amenable to dynamic scheduling

6.3. Comparison with Baseline Tools

6.3.1. Comparison with SS

The first comparison is made with *VitisHLS*, a commercial static scheduling-based HLS tool. In terms of cycle latency, the result of our proposed framework outperforms the baseline among all the benchmarks with dynamic behaviors as indicated by Table 2 and Figure 9. For the regular designs (*gesummv* and *covariance*), our framework achieves the same result as *VitisHLS*, which proves that our framework successfully identifies non-dynamism of the code and leaves it untouched. For other benchmarks with dynamic features, the latency improvement comes from the fine-grained control of the runtime conditions for the recurrences, and the pipeline bubbles introduced by the static scheduling are squeezed out. For the *sparseMatrix* benchmark, an if-condition that depends on a value loaded from external memory determines whether a dot product operation is executed. This irregular computation prevents the loop from being perfectly pipelined through static scheduling. However, after the PCA transformation, the slow dot product operation is divided into multiple virtual cycles. When the condition loading is finished in one virtual

cycle, the dot product can be executed conditionally, i.e., only if the condition is true will the execution be performed. Therefore, the overall cycle latency is reduced. With a slightly decreased synthesizable frequency, the overall wall clock time (WCT) improves. For the *gSum*, *vecNormTrans*, *getTanh*, and *gsumIf* benchmarks, they are very similar to the example loop kernel in Figure 2 with imbalanced branches determined by runtime executions. The PCA transformation reduces cycle latency by avoiding the execution of the long path when the short path is taken, resulting in a better cycle time but with a slight frequency drop. This trend is consistent among all these benchmarks. The *histogram* and *BNNKernel* benchmarks present memory dependencies. Our framework handles the dynamic memory access with store queues as described in Section 5.4, which ensures the pipeline only stalls when an actual memory conflict occurs. This improves overall latency but can impact frequency. Among all the benchmarks that are amenable to dynamic scheduling, the drop in the synthesized maximum frequency is still acceptable compared with the significant latency improvement, which yields a better final WCT. However, it comes at the cost of a resource overhead. Additional logic units are required to store valid/ready statuses and to check execution conditions both within individual SCCs and between different SCCs. Buffers (β nodes) and queues (θ nodes) can also increase the usage of *Flip Flops* (FF). Furthermore, the store queues and corresponding arbitration logic used to handle the memory access dependencies can also consume huge amounts of resources (*FFs, LUTs, and DSPs*). Therefore, a long recurrence path bearing memory access pairs for the same memory base can result in a significant resource overhead as observed in the *histogram* benchmark. It is worth noting that our proposed framework does not mean to beat the *VitisHLS* in all PPA aspects. On the other hand, it improves the performance limit in terms of WCT for synthesizing the irregular and control-dominated application at a slight area and frequency overhead through pure static scheduling.

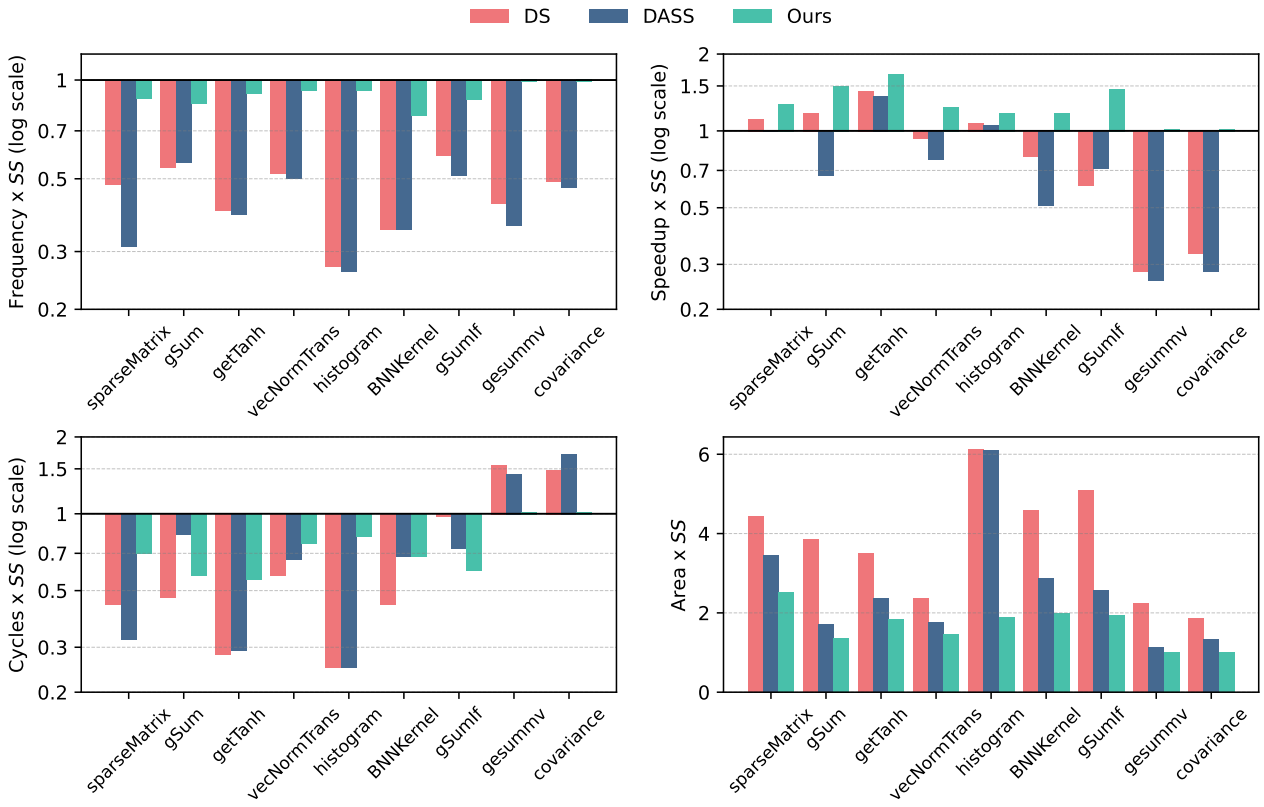


Figure 9. Comparisons of DS [26], DASS [14], and our work against their respective SS-based solution including the frequency, clock cycles, speedup, and area overhead.

Table 2. Comparison with *static scheduling* (SS) [45], *dynamic scheduling* (DS) [26], and *Dynamic and static scheduling* (DASS)-based [14] HLS tools with nine benchmarks. The WCT and cycle column show the geometric mean among a range of input vectors to cover different runtime conditions.

Benchmark	Work	F (MHz)	F/F_{SS}	Cycles	C/C_{SS}	WCT (µs)	T/T_{SS}	LUTs	FFs	DSPs	Area [a]	A/A_{SS}
sparseMatrix	SS	257.5	1.00	21,501	1.00	83.5	1.00	138	320	3	818	1.00
	DS	123.7	0.48	9426	0.44	76.2	0.91	1356	1544	6	3620	4.43
	DASS	80.4	0.31	6778	0.32	84.3	1.01	985	1123	6	2828	3.46
	Ours	227.6	0.88	15,113	0.70	66.4	0.80	329	774	8	2063	2.52
gSum	SS	230.7	1.00	24,115	1.00	104.5	1.00	931	1567	5	3098	1.00
	DS	125.5	0.54	11,358	0.47	90.5	0.87	4238	3956	31	11,914	3.85
	DASS	128.2	0.56	19,974	0.83	155.8	1.49	2315	2176	7	5331	1.72
	Ours	195.8	0.85	13,804	0.57	70.5	0.67	981	1933	11	4234	1.37
getTanh	SS	279.4	1.00	52,004	1.00	186.1	1.00	516	827	5	1943	1.00
	DS	112.5	0.40	14,704	0.28	130.7	0.70	3256	2569	8	6785	3.49
	DASS	109.7	0.39	15,161	0.29	138.2	0.74	1578	2056	8	4594	2.36
	Ours	254.8	0.91	28,716	0.55	112.7	0.61	1098	1762	6	3580	1.84
vecNormTrans	SS	254.8	1.00	23,070	1.00	90.5	1.00	1798	3104	5	5502	1.00
	DS	133.6	0.52	13,066	0.57	97.8	1.08	5114	7099	7	13,053	2.37
	DASS	128.5	0.50	15,112	0.66	117.6	1.30	3956	5077	6	9753	1.77
	Ours	237.1	0.93	17,451	0.76	73.6	0.81	3229	4166	5	7995	1.45
histogram	SS	285.1	1.00	15,006	1.00	52.6	1.00	282	539	2	1061	1.00
	DS	76.4	0.27	3789	0.25	49.6	0.94	1833	4430	2	6503	6.13
	DASS	72.8	0.26	3691	0.25	50.7	0.96	1814	4430	2	6484	6.11
	Ours	266.4	0.93	12,095	0.81	45.4	0.86	677	1095	2	2012	1.90
BNNKernel	SS	257.5	1.00	11,407	1.00	44.3	1.00	299	385	3	1044	1.00
	DS	89.5	0.35	5003	0.44	55.9	1.26	1697	2006	9	4783	4.58
	DASS	89.5	0.35	7706	0.68	86.1	1.94	907	997	9	2984	2.86
	Ours	202.0	0.78	7716	0.68	38.2	0.86	531	708	7	2079	1.99
gSumIf	SS	238.3	1.00	24,116	1.00	101.2	1.00	1213	1972	7	4025	1.00
	DS	139.5	0.59	23,283	0.97	166.9	1.65	6886	6412	60	20,498	5.09
	DASS	122.7	0.51	17,497	0.73	142.6	1.41	3897	5019	12	10,356	2.57
	Ours	207.6	0.87	14,553	0.60	70.1	0.69	2119	4103	13	7782	1.93
gesummv	SS	264.8	1.00	786,466	1.00	2970.5	1.00	898	1764	5	3262	1.00
	DS	112.5	0.42	1,211,445	1.54	10,768.4	3.63	1448	4976	7	7264	2.23
	DASS	96.3	0.36	1,116,050	1.42	11,589.3	3.90	1496	1586	5	3682	1.13
	Ours	264.8	1.00	786,466	1.00	2970.5	1.00	898	1764	5	3262	1.00
covariance	SS	265.3	1.00	241,651	1.00	911.0	1.00	1670	2688	5	4958	1.00
	DS	129.6	0.49	354,910	1.47	2738.5	3.01	4219	4396	5	9215	1.86
	DASS	124.3	0.47	409,618	1.70	3295.4	3.62	2178	3864	5	6642	1.34
	Ours	265.3	1.00	241,651	1.00	911.0	1.00	1670	2688	5	4958	1.00

[a] The area is calculated as $LUT + FF + 120 \times DSP$ [49,50].

6.3.2. Comparison with DS and DASS

Compared with the dynamic-scheduling-based (DS [26]) and hybrid-scheduling-based (DASS [14]) HLS tools, our largest advantage is the ability to maintain the compatibility to the static scheduling framework while presenting the dynamic behavior. With the *VitisHLS* tool as a backend, we can achieve much higher frequency than DS and DASS since our critical paths between two β nodes are still statically synthesized, even though the handshakings between different *SCCs* can harm the timing of critical paths. In terms of the dynamic scheduling, the handshakings occur at an operation level, which means an even longer critical path and thus a worse synthesized frequency. DASS selects the code snippets without dynamic behaviors (static islands) for static synthesis. But the static HLS tool is not aware of the other parts of the design, which can result in suboptimal scheduling from the global perspective. The datapaths are still synthesized by the dynamic tool in the DASS and the result in Table 2 showing a worse synthesized frequency than the other three tools.

In terms of resource overhead, our framework requires fewer resources to achieve dynamism than DS and DASS among all the benchmarks. The dataflow network synthesized

from DS makes it hard to perform operation sharing since all operations respond to the request and acknowledgment from other operations. Although DASS can save resources by synthesizing the static islands with static HLS, it still has to introduce extra wrapping logic to merge the synthesized static blocks back to the dynamic HLS flow. Our framework starts from identifying the dynamic parts of the design and carefully handles the runtime control flows at a cycle-accurate level. This process is more natural and resembles the manual hardware design flow. Furthermore, the PCA model generated from our flow can still benefit from the resource sharing if it is not synthesized with $II = 1$, in which case one "virtual cycle" of the PCA model indicates II hardware cycles. This is the normal case when compiling the loops with multi-cycle operations.

7. Conclusions and Discussion

In this work, we proposed a comprehensive source-to-source compiler that combines the static and dynamic scheduling nature. The transformed design presents a cycle-accurate execution pattern by mapping a pre-scheduled "virtual cycle" to a software loop iteration, and is fully compatible with the existing static-scheduling-based HLS tools. Control flows are dynamically executed, and decoupled *SCCs* are connected with latency-insensitive channels. The experiment results indicate significant performance improvements over commercial HLS tools and better performance-area tradeoffs compared to the state-of-the-art dynamic scheduling-based and hybrid-scheduling-based HLS tools.

While this topic is being studied and explored by many researchers, there are still potential opportunities to overcome the current scheduling bottleneck for further improvements. One of the new developments could be the improvement on the dataflow synthesis support. Current HLS tools provide limited support on dataflow synthesis through latency-insensitive channels (e.g., the `hls::stream` from VitisHLS, `ac_channel` from CatapultHLS, and `SYCL` pipe from Intel HLS) to create task-level pipelining. But such support is usually applicable only to tasks which do not have bypass, feedback, or conditionals between each other [1]. By enabling the support for finer-grained dataflow synthesis (i.e., channels that can be inserted between operations instead of tasks), the flexibility to achieve scheduling dynamism can be improved significantly. In addition, more in-depth static analysis technologies are being developed and explored, such as multi-level graph optimization, which can simplify the graph structure at different abstraction layers and efficiently eliminate fake dependencies before the scheduling process. A popular framework termed Multi-Level-Intermediate-Representation (MLIR) [51] has been developed to facilitate efficient optimizations at different abstraction levels. An MLIR-based HLS tool, CIRCT [52], has shown great potential as a next-generation HLS tool supporting both static and dynamic scheduling. Furthermore, there have been works on developing customized hardware components for runtime arbitration to achieve dynamism in the static HLS tool [31,32]. However, the support for RTL integration in current HLS tools is still limited, making it difficult to plug the customized hardware into the standard HLS flow. Therefore, improving the customizability and related plugin interfaces of the static HLS tools has great potential to boost the development of easier and more efficient methods to overcome the bottlenecks.

In terms of the hybrid scheduling specific to the proposed source-to-source compiler, there are also many potential development opportunities to further boost its effectiveness. As a source-to-source plugin to the static HLS flow, it has the potential to be integrated with existing optimizations. With the source-to-source nature, the code is still plain C code after the transformation and can be further processed by any modern HLS tools. This means that the code remains applicable to the well-developed optimizations used by the modern HLS tool, such as code optimizations in front-end, as well as critical path optimization, resource sharing, and design space exploration in back-end. Also, with the advancement of static

HLS technology, the proposed source-to-source method can be easily be accommodated, unlike other optimizations that act at the intermediate representation (IR) level, which usually requires a huge amount of engineering work to be incorporated into different tool flows. In addition to directly combining the transformed code with other generic HLS optimizations, the compiler itself can also be further improved by integrating other technologies. One typical example is to enable speculative pipelining [33,34] within the transformation process. This improvement is natural since we already have the data paths divided as "virtual cycles". The speculative behavior can be easily achieved by changing the behavior of the FSM and `MUX`, with necessary rollback registers added. Furthermore, the recently proposed technology by She et al. [36] can also be integrated in the framework to accurately model the delay for each path to improve the overall performance and avoid area overhead caused by misestimation. These discussed directions are our future research work. Other future research directions include handling the nested loop kernels and exploring multi-threading in a PCA model.

Author Contributions: Conceptualization, Y.S., Y.H., and J.L.; methodology, Y.S., Y.H., and J.L.; software, Y.S.; validation, Y.S.; investigation, Y.S., Y.H., and J.L.; resources, R.C.C.C. and H.Y.; data curation, Y.S.; writing—original draft preparation, Y.S.; writing—review and editing, Y.S., Y.H., J.L., R.C.C.C., and H.Y.; visualization, Y.S.; supervision, R.C.C.C. and H.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by Hong Kong Innovation and Technology Commission (ITF Seed Fund ITS/098/22) and Hong Kong Innovation and Technology Commission (InnoHK Project CIMDA).

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

HLS	High-Level Synthesis
PCA	Pseudo-Cycle-Accurate
II	Initiation Interval
IR	Intermediate Representation
EG	Execution Graph
GSSA	Gated SSA
HDL	Hardware Description Language
CDFG	Control Data Flow Graph
CFG	Control Flow Graph
DFG	Data Flow Graph
LLVM	Low-Level Virtual Machine
MLIR	Multi-Level Intermediate Representation
BB	Basic Block
FSM	Finite State Machine
DAG	Directed Acyclic Graph
SCC	Strongly Connected Component

References

1. AMD Xilinx. Vitis High-Level Synthesis User Guide (UG1399). 2025. Available online: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls (accessed on 19 November 2025).

2. Rokicki, S.; Pala, D.; Paturel, J.; Sentieys, O. What you simulate is what you synthesize: Designing a processor core from c++ specifications. In Proceedings of the 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Westminster, CO, USA, 4–7 November 2019; pp. 1–8.
3. Tu, P.; Padua, D. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In Proceedings of the 9th International Conference on Supercomputing, Barcelona, Spain, 3–7 July 1995; pp. 414–423.
4. Cong, J.; Liu, B.; Neuendorffer, S.; Noguera, J.; Vissers, K.; Zhang, Z. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Trans.-Comput.-Aided Des. Integr. Circuits Syst.* **2011**, *30*, 473–491. [CrossRef]
5. LLVM. The LLVM Compiler Infrastructure. 2025. Available online: https://www.llvm.org/ (accessed on 19 November 2025).
6. Micheli, G.D. Hardware/Software Co-Design of Run-Time Schedulers for Real-Time Systems. *Des. Autom. Embed. Syst.* **2001**, *6*, 89.
7. Zhang, Z.; Liu, B. SDC-based modulo scheduling for pipeline synthesis. In Proceedings of the 2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), San Jose, CA, USA, 4–7 November 2013; pp. 211–218.
8. Canis, A.; Brown, S.D.; Anderson, J.H. Modulo SDC scheduling with recurrence minimization in high-level synthesis. In Proceedings of the 2014 24th International Conference on Field Programmable Logic and Applications (FPL), Munich, Germany, 2–4 September 2014; pp. 1–8.
9. Rau, B.R. Iterative modulo scheduling. *Int. J. Parallel Program.* **1996**, *24*, 3–64. [CrossRef]
10. Carloni, L.P.; McMillan, K.L.; Sangiovanni-Vincentelli, A.L. Theory of latency-insensitive design. *IEEE Trans.-Comput.-Aided Des. Integr. Circuits Syst.* **2001**, *20*, 1059–1076. [CrossRef]
11. Josipović, L.; Marmet, A.; Guerrieri, A.; Ienne, P. Resource sharing in dataflow circuits. *ACM Trans. Reconfigurable Technol. Syst.* **2023**, *16*, 1–27. [CrossRef]
12. Xu, J.; Murphy, E.; Cortadella, J.; Josipović, L. Eliminating Excessive Dynamism of Dataflow Circuits Using Model Checking. In Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Monterey, CA, USA, 12–14 February 2023; pp. 27–37.
13. Cheng, J.; Josipović, L.; Constantinides, G.A.; Ienne, P.; Wickerson, J. Combining dynamic & static scheduling in high-level synthesis. In Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, 23–25 February 2020; pp. 288–298.
14. Cheng, J.; Josipović, L.; Constantinides, G.A.; Ienne, P.; Wickerson, J. DASS: Combining Dynamic & Static Scheduling in High-Level Synthesis. *IEEE Trans.-Comput.-Aided Des. Integr. Circuits Syst.* **2021**, *41*, 628–641.
15. Cheng, J.; Wickerson, J.; Constantinides, G.A. Finding and finessing static islands in dynamically scheduled circuits. In Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Virtual, 27 February–1 March 2022; pp. 89–100.
16. Szafarczyk, R.; Nabi, S.W.; Vanderbauwhede, W. Compiler Discovered Dynamic Scheduling of Irregular Code in High-Level Synthesis. In Proceedings of the 2023 33rd International Conference on Field-Programmable Logic and Applications (FPL), Hamburg, Germany, 30 August–1 September 2023; pp. 1–9.
17. Cortadella, J.; Kishinevsky, M. Synchronous elastic circuits with early evaluation and token counterflow. In Proceedings of the 44th Annual Design Automation Conference, San Diego, CA, USA, 4–8 June 2007; pp. 416–419.
18. Edwards, S.A.; Townsend, R.; Barker, M.; Kim, M.A. Compositional dataflow circuits. *ACM Trans. Embed. Comput. Syst. (TECS)* **2019**, *18*, 1–27. [CrossRef]
19. Cortadella, J.; Kishinevsky, M.; Grundmann, B. Synthesis of synchronous elastic architectures. In Proceedings of the 43rd Annual Design Automation Conference, San Francisco, CA, USA, 24–28 July 2006; pp. 657–662.
20. Venkataramani, G.; Budiu, M.; Chelcea, T.; Goldstein, S.C. C to Asynchronous Dataflow Circuits: An End-to-End Toolflow. 2004. Available online: https://kilthub.cmu.edu/articles/C_to_Asynchronous_Dataflow_Circuits_An_End-to-End_Toolflow/6603986/files/12094370.pdf (accessed on 19 November 2025).
21. Hoover, G.; Brewer, F. Synthesizing synchronous elastic flow networks. In Proceedings of the Conference on Design, Automation and Test in Europe, Grenoble, France, 10–14 March 2008; pp. 306–311.
22. Chatterjee, S.; Kishinevsky, M.; Ogras, U.Y. xMAS: Quick formal modeling of communication fabrics to enable verification. *IEEE Des. Test Comput.* **2012**, *29*, 80–88. [CrossRef]
23. Putnam, A.; Bennett, D.; Dellinger, E.; Mason, J.; Sundararajan, P.; Eggers, S. CHiMPS: A C-level compilation flow for hybrid CPU-FPGA architectures. In Proceedings of the 2008 International Conference on Field Programmable Logic and Applications, Heidelberg, Germany, 8–10 September 2008; pp. 173–178.
24. Townsend, R.; Kim, M.A.; Edwards, S.A. From functional programs to pipelined dataflow circuits. In Proceedings of the 26th International Conference on Compiler Construction, Saarbrücken, Germany, 26 March–3 April 2017; pp. 76–86.
25. Josipović, L.; Brisk, P.; Ienne, P. From C to elastic circuits. In Proceedings of the 2017 51st Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, CA, USA, 29 October–1 November 2017; pp. 121–125.

26. Josipović, L.; Guerrieri, A.; Ienne, P. From C/C++ code to high-performance dataflow circuits. *IEEE Trans.-Comput.-Aided Des. Integr. Circuits Syst.* **2021**, *41*, 2142–2155. [CrossRef]
27. Liu, J.; Rizzi, C.; Josipović, L. Load-store queue sizing for efficient dataflow circuits. In Proceedings of the 2022 International Conference on Field-Programmable Technology (ICFPT), Hong Kong, China, 5–9 December 2022; pp. 1–9.
28. Wang, H.; Rizzi, C.; Josipović, L. MapBuf: Simultaneous technology mapping and buffer insertion for hls performance optimization. In Proceedings of the 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD), San Francisco, CA, USA, 29 October–2 November 2023; pp. 1–9.
29. Alle, M.; Morvan, A.; Derrien, S. Runtime dependency analysis for loop pipelining in high-level synthesis. In Proceedings of the 50th Annual Design Automation Conference, Austin, TX, USA, 2–6 June 2013; pp. 1–10.
30. Liu, J.; Bayliss, S.; Constantinides, G.A. Offline synthesis of url dependence testing: Parametric loop pipelining for HLS. In Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, Boston, MA, USA, 1–4 May 2015; pp. 159–162.
31. Tan, M.; Liu, G.; Zhao, R.; Dai, S.; Zhang, Z. ElasticFlow: A complexity-effective approach for pipelining irregular loop nests. In Proceedings of the 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Austin, TX, USA, 2–6 November 2015; pp. 78–85.
32. Dai, S.; Zhao, R.; Liu, G.; Srinath, S.; Gupta, U.; Batten, C.; Zhang, Z. Dynamic hazard resolution for pipelining irregular loops in high-level synthesis. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017; pp. 189–194.
33. Derrien, S.; Marty, T.; Rokicki, S.; Yuki, T. Toward speculative loop pipelining for high-level synthesis. *IEEE Trans.-Comput.-Aided Des. Integr. Circuits Syst.* **2020**, *39*, 4229–4239. [CrossRef]
34. Gorius, J.M.; Rokicki, S.; Derrien, S. SpecHLS: Speculative accelerator design using high-level synthesis. *IEEE Micro* **2022**, *42*, 99–107. [CrossRef]
35. Gorius, J.M.; Rokicki, S.; Derrien, S. A Unified Memory Dependency Framework for Speculative High-Level Synthesis. In Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction, Edinburgh, UK, 2–3 March 2024; pp. 13–25.
36. She, Y.; Liu, J.; Huang, Y.; Cheung, R.C.; Yan, H. A Speculative Loop Pipeline Framework with Accurate Path Modeling for High-Level Synthesis. *ACM Trans. Reconfigurable Technol. Syst.* **2025**, *18*, 1–33. [CrossRef]
37. Carloni, L.P. From latency-insensitive design to communication-based system-level design. *Proc. IEEE* **2015**, *103*, 2133–2151. [CrossRef]
38. Josipović, L.; Sheikhha, S.; Guerrieri, A.; Ienne, P.; Cortadella, J. Buffer placement and sizing for high-performance dataflow circuits. *ACM Trans. Reconfigurable Technol. Syst. (TRETS)* **2021**, *15*, 1–32.
39. Rangan, R.; Vachharajani, N.; Vachharajani, M.; August, D.I. Decoupled software pipelining with the synchronization array. In Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques—PACT 2004, Antibes Juan-les-Pins, France, 29 September–3 October 2004; pp. 177–188.
40. CLANG. Clang: A C Language Family Frontend for LLVM. 2025. Available online: https://clang.llvm.org/ (accessed on 19 November 2025).
41. LLVM. LLVM Loop Terminology (and Canonical Forms). Available online: https://llvm.org/docs/LoopTerminology.html (accessed on 19 November 2025).
42. Ferrante, J.; Ottenstein, K.J.; Warren, J.D. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **1987**, *9*, 319–349. [CrossRef]
43. Li, P.; Agrawal, K.; Buhler, J.; Chamberlain, R.D. Deadlock avoidance for streaming computations with filtering. In Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures, New York, NY, USA, 13–15 June 2010; pp. 243–252.
44. Josipović, L.; Brisk, P.; Ienne, P. An out-of-order load-store queue for spatial computing. *ACM Trans. Embed. Comput. Syst. (TECS)* **2017**, *16*, 1–19. [CrossRef]
45. Xilinx. Xilinx Vitis 2022.2, 2022. Available online: https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vitis/2022-2.html (accessed on 19 November 2025).
46. Cheng, J. JianyiCheng: HLS Benchmarks First Release. 2019. Available online: https://zenodo.org/records/3561115 (accessed on 19 November 2025).
47. Morvan, A.; Derrien, S.; Quinton, P. Polyhedral bubble insertion: A method to improve nested loop pipelining for high-level synthesis. *IEEE Trans.-Comput.-Aided Des. Integr. Circuits Syst.* **2013**, *32*, 339–352. [CrossRef]
48. PollyLLVM. Polly LLVM Framework for High-Level Loop and Data-Locality Optimizations. 2025. Available online: https://polly.llvm.org/ (accessed on 19 November 2025).

49. Chethan, K.H.; Kapre, N. Hoplite-DSP: Harnessing the Xilinx DSP48 multiplexers to efficiently support NoCs on FPGAs. In Proceedings of the 2016 26th International Conference on Field Programmable Logic and Applications (FPL), Lausanne, Switzerland, 29 August–2 September 2016; pp. 1–10.
50. Abdelfattah, M.S.; Betz, V. Networks-on-chip for FPGAs: Hard, soft or mixed? *ACM Trans. Reconfigurable Technol. Syst. (TRETS)* **2014**, *7*, 1–22. [CrossRef]
51. Lattner, C.; Amini, M.; Bondhugula, U.; Cohen, A.; Davis, A.; Pienaar, J.; Riddle, R.; Shpeisman, T.; Vasilache, N.; Zinenko, O. MLIR: Scaling compiler infrastructure for domain specific computation. In Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Virtual, 27 February–3 March 2021; pp. 2–14.
52. CIRCT. Circuit IR Compilers and Tools. 2025. Available online: https://circt.llvm.org/ (accessed on 19 November 2025).