PreVV: Eliminating Store Queue via Premature Value Validation for Dataflow Circuit on FPGA

Kuangjie Zou†, Yifan Zhang, Zicheng Zhang, Guoyu Li, Jianli Chen, Kun Wang*, Jun Yu
State Key Lab of Integrated Chips & Systems, and School of Microelectronics, Fudan University, Shanghai, China
Email: †kjzou23@m.fudan.edu.cn, *kun.wang@ieee.org

***Abstract*—Dynamic scheduling in high-level synthesis (HLS) maximizes pipeline performance by enabling out-of-order scheduling of load and store requests at runtime. However, this method introduces unpredictable memory dependencies, leading to data disambiguation challenges. Load-store queues (LSQs), commonly used in superscalar CPUs, offer a potential solution for HLS. However, LSQs in dynamically scheduled HLS implementations often suffer from high resource overhead and scalability limitations. In this paper, we introduce *PreVV*, an architecture based on premature value validation designed to address memory disambiguation with minimal resource overhead. Our approach substitutes LSQ with several *PreVV* components and a straightforward premature queue. We prevent potential deadlocks by incorporating a specific tag that can send 'fake' tokens to prevent the accumulation of outdated data. Furthermore, we demonstrate that our design has scalability potential. We implement our design using several hardware templates and an LLVM pass to generate targeted dataflow circuits with *PreVV*. Experimental results on various benchmarks with data hazards show that, compared to state-of-the-art dynamic HLS, *PreVV*16 (a version with a premature queue depth of 16) reduces LUT usage by 43.91% and FF usage by 33.09%, with minimal impact on timing performance. Meanwhile, *PreVV*64 (a version with a premature queue depth of 64) reduces LUT usage by 27.21% and FF usage by 33.10%, without affecting timing performance.**

***Index Terms*—high-level synthesis, dataflow circuits, load-store queue, FPGA**

I. Introduction

FPGAs demonstrate remarkable efficiency across various domains, highlighting their versatility and applicability. However, for software programmers lacking expertise in *hardware description languages* (HDLs) like Verilog and VHDL, deploying software onto FPGA platforms can be challenging. To address this issue, *High-Level Synthesis* (HLS) has been introduced to improve productivity. HLS compiles and schedules high-level programming languages within dedicated tools, ultimately generating tangible circuits [14]. How the code is scheduled significantly impacts the circuit's overall performance.

Based on whether pipeline cycles are fixed during synthesis, there are two primary scheduling approaches: static and dynamic scheduling [15]. Both methods focus on optimizing loop pipelining to allow simultaneous execution of multiple iterations. The initiation interval (*II*), representing the cycles between consecutive iterations, measures pipeline throughput [24]. HLS tools aim to minimize the *II* while maintaining calculation correctness.

*Corresponding author.

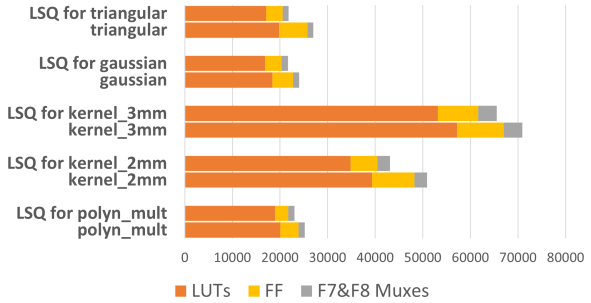

Fig. 1: LSQ resource usage in Dynamatic. More than 80% of the resources (include LUTs, FFs and muxes) are allocated to LSQ while resources for calculation only occupies less than 20%.

Static schedulers, also known as modulo schedulers, use the same scheduling method for all iterations [23]. However, estimating precise hardware behavior at compile-time is challenging, often causing the scheduler to use the worst-case *II* for the pipeline [6]. Dynamic scheduling provides an alternative to overcome the limitations of static scheduling. It determines the *II* at runtime based on data and control flow, without pre-estimating hardware behavior [15]. However, this approach may lead to unpredictable memory accesses, potentially causing inaccuracies. To address this issue in dynamic scheduling, load-store queues (LSQs) are used. LSQs record the original sequence of a program, allowing the processor to recover the correct program order [4] [14].

Although LSQs for FPGAs do not need to be as complex as those in superscalar processors, they still cause significant area and resource overhead and reduce operating frequency [22]. For example, consider the state-of-the-art dynamic scheduling HLS compiler *Dynamatic*. Fig. 1 illustrates resource usage of the circuit generated by *Dynamatic* for various tasks. Nearly all resources are allocated to a single LSQ, while computation uses less than 20%. This resource-intensive LSQ limits the use of dataflow circuits for more complex tasks. The large queue also creates a significant search burden, reducing the circuit's frequency.

To address the issues caused by LSQs, Elakhras *et al.* [10] optimized the LSQ architecture by converting potential memory dependencies into virtual data dependencies and decoupling data and control flow using the Sequentializer (*SEQ*). Although this approach enhances LSQ frequency, it demands additional area and resources. Liu *et al.* [16] proposed an algorithm to determine the most cost-effective LSQ depths in dataflow circuits while preserving optimal circuit throughput. However,

978-3-9826741-0-0/DATE25/© 2025 EDAA

```
for (unsigned i = 0; i < N;
++i) {
   a[b[i]] += A
   b[i] += B;
   }
}
```
(a) Sequential-Update RAW

```
for (unsigned i = 0; i < N;
++i){
   a[b[i] + f(x)] += A //  (1)
   b[i + g(x)] += B; //  (2)
   }
}
```
(b) Function-Dependent RAW

Fig. 2: An example show two different types of RAW data hazard. (a) Data are updated in the same iteration. (b) Data are updated cross an unknown number of iterations.

this approach is a compromise and does not directly solve the issues caused by LSQs. Szafarczyk *et al.* [14] improved LSQ frequency by using an 'age' tag to streamline search algorithms. However, this method does not address the area and resource overhead.

Due to the area overhead, high-performance FPGA accelerators must reserve significant space for LSQs to ensure high-speed computing, making them incompatible with edge devices that have limited resources. Thus, an optimized LSQ architecture or an alternative with low area overhead and high frequency is necessary. To achieve this goal, we make the following contributions:

- **A low-resource overhead approach that replaces the LSQ with a simple premature queue using premature value validation architecture.** To address the area and resource overhead of current LSQs, we introduce premature value validation to verify the correctness of premature operators, eliminating the need for the entire store queue. This approach enables true out-of-order execution for all operations, even those with potential memory dependencies, unlike traditional dataflow circuits where operations must follow LSQ order. We demonstrate how our architecture accurately detects memory dependency violations and why it is more area- and resource-efficient than state-of-the-art LSQs, using design principles.
- **An optimization for scalability and deadlock elimination.** We enhance the scalability of our design by simplifying the dimensions of a complex memory disambiguation problem. By employing this dimension-reducing approach, we can address any memory disambiguation problem similarly to a three-operation scenario. To ensure the stability of our architecture, we add a specific tag to certain operations, prompting them to send 'fake' signals, thereby clearing the pipeline and preventing potential deadlock.
- **Competitive Performance.** We evaluate our work against state-of-the-art dynamic HLS compilers. The experimental results show that our work significantly lowers area and resource overhead. We also demonstrate that our work is more scalable than state-of-the-art LSQs, making it more suitable for future, larger dataflow circuits.

II. BACKGROUND & RELATED WORK

In this section, we discuss the need for dynamic scheduling and dataflow circuits. We also address the memory disambiguation problem in dynamic scheduling and the necessity for load-store queues.

A. Dynamic Scheduling & Dataflow Circuits

Traditional HLS tools utilize conservative static scheduling, leveraging hardware sharing via finite state machines. However, static schedulers often assign a conservative *II* to ensure correctness. To tackle this issue, Josipović et al. [15] proposed the concept of dynamic scheduling in HLS tools, inspired by out-of-order processors [7]. Dynamic scheduling involves making decisions at runtime within the local circuit, rather than at compile-time. A circuit that employs dynamic scheduling is referred to as a dataflow circuit, based on the latency-intensive design concept proposed by Carloni *et al.* [17]. A segment of high-level code without conditional judgments is treated as a unit, known as a basic block (*BB*). In addition to the datapath used in conservative circuits, dataflow circuits incorporate control flow between two related BBs. The overall control flow is represented by a control-flow graph (CFG). To ensure the validity of data transfer between two BBs, a two-way handshake signal is added, indicating whether the source *BB* has completed data preparation and the target *BB* is ready to accept it [12].

B. Memory Disambiguation Problem

Dynamic scheduling borrows concepts from out-of-order processors and faces similar memory disambiguation challenges [14]. Consider the code in Fig. 2 as an example. Each of the code segments has a read-after-write (RAW) hazard. Modern compilers may be able to resolve the memory dependencies in Fig. 2(a) [7], but they still cannot analyze the exact dependencies in Fig. 2(b), as the indices of statements (1) and (2) are determined by functions $f(x)$ and $g(x)$, which can only be known at runtime. We define C_i^n as statement n in iteration i. Suppose that in C_3^2, new data will be stored to $b[10]$ and in C_5^1 this data is loaded. If these operations are allowed to be executed completely out-of-order, C_5^1 may be executed before C_3^2, potentially leading to erroneous memory access. To address this issue, modern processors implement an LSQ to preserve the original instruction sequence. When a memory operation is to be committed—i.e., reading data from RAM or writing data back to RAM—the processor searches the LSQ to determine if any memory dependencies exist. This process is also referred to as associated searching [1]- [3] [19] [20].

C. Load-Store Queues in Dynamic scheduling

To address the memory disambiguation problem, dataflow circuits also utilize the concept of LSQs in processors, but in a completely different form [14]. The good news is that dataflow circuits do not need to address this problem as complexly as CPUs do. Elakhras *et al.* [8] employ a fast token delivery strategy to allocate tokens for the LSQ without hindering the rest of the circuit. Szafarczyk *et al.* [14] design a fundamentally different LSQ by separating the load queue and store queue from the original load and store content-addressable memory (CAM). The downside is that, in terms of processors, the original program order is known and can be used to construct an LSQ. However, this sequential order is not present in dataflow circuits [7]. Therefore, recovering

the original instruction sequence is the primary challenge for dataflow circuits. Josipović *et al.* [4] utilize on-board ROM to store the original sequence. To reduce resource overhead, they divide the HLS code into several groups, ensuring that there are no conditional judgments, and only store group information instead of all operations in ROM. In Fig. 2(b), C^1 and C^2 form a group and their original sequence is stored in a ROM. Once an iteration begins, a group allocator retrieves the order from ROM and adds the iteration cycle to the LSQ, thereby maintaining the program order. However, these approaches still require the fundamental LSQ structure, which incurs significant resource costs, as demonstrated in Section I.

III. Premature Value Validation

In this section, we present our Premature Value Validation (***PreVV***) architecture to eliminate the LSQ in dataflow circuits. Let us refer back to the example in Fig. 2(b).

Definition 1: Two statements C^m and C^n that may cause memory disambiguation problem with each other are called an ambiguous pair $Am\{C^m, C^n\}$. An ambiguous pair is created by a load and store operation, which operates at the atomic level.

In Fig. 2(b), C^1 and C^2 form an ambiguous pair $Am\{C^1, C^2\}$, because they encounter a RAW hazard. C^1 loads the result from $b[i]$, while C^2 stores the result to $b[i]$. We do not require the operations in $Am\{C^1, C^2\}$ to be executed in the original order, as this is controlled by components such as the LSQ. Instead, we allow these operations to be issued out of order, similar to the remaining part of the dataflow circuit. But we should notice that the correctness of the result at this stage is not guaranteed and the result should not be written back to memory. We refer to this stage as a *premature* stage, and the operation at this stage as a *premature* operation. To store the result of a premature operation for future validation, we use an assembly to save four properties of a operation C^m:

$$P_m = \{iter_m, index_m, value_m, Op_m\}, \tag{1}$$

where $iter_m$ represents the iteration cycle, $index_m$ denotes the target of a operation C^m, $value_m$ indicates the value a load operation reads or a store will write back and Op_m specifies whether the operation is a load or a store.

We use a queue called the premature queue to hold all the P_m. Each premature operation saves its P_m as soon as $value_m$ is known. This action triggers a *validation* stage to determine whether the preceding premature operations are correct. Given a certain P_m, if any existing operation C^n in premature queue meets the following conditions:

$$iter_m < iter_n, \tag{2}$$
$$Op_m \neq Op_n, \tag{3}$$
$$index_m = index_n, \tag{4}$$
$$value_m \neq value_n, \tag{5}$$

we can infer that C^n has incorrectly obtained the data. Therefore, the entire pipeline following it needs to be squashed. Conversely, if none of the existing operations fulfill the conditions, all of these operations are validated as correct and can be committed successfully. Eq. (2) to (5) demonstrate the validation of operations across different iterations. However, if $iter_m = iter_n$, we cannot determine which operation should be executed earlier. To address this issue, we can apply the concept from the group allocator designed by Josipović *et al.* [4]. We can use a tuple to store the original sequence of an ambiguous pair. When $iter_m = iter_n$, we can compare the sequence with the tuple and check Eq. (3) to (5) in the same manner described above.

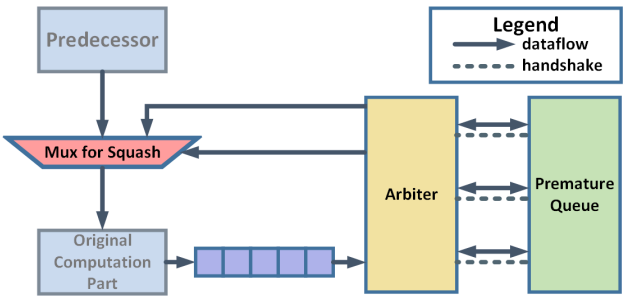

Fig. 3: Our *PreVV* architecture of one ambiguous pair, which contains one arbiter, premature queue and mux.

IV. Physical Design for Premature Value Validation Components

This section describes the premature components utilized to implement the approach outlined in Section III. There are three main types of units: a premature queue to store all premature operations for future validation, an arbiter to validate all premature operations saved in the queue whenever a new premature operation arrives, and a mux to determine whether to continue or squash the pipeline.

A. Design Overview

Fig. 3 provides an overview of our architecture. Predecessor and original computation part in pale blue refer to standard dataflow components [21]. We apply ***PreVV*** to each ambiguous pair. The arbiter retrieves P_m from the ambiguous pair within the original computation section. We use a simple FIFO to cache data before it enters the arbiter, preventing the original pipeline from being stalled and thereby reducing the impact on overall latency. Once the arbiter receives all the data from *premature* stage, it initiates a *validation* stage, searching the entire premature queue from head to tail. If a specific item in the *premature* stage satisfies Eq. (2) to (5), an erroneous premature operation C^{Err} is identified. The arbiter then needs to copy and send $iter_{Err}$ back to the mux, along with a squash signal to trigger the flushing of the pipeline. A ROM is also included to store the sequence of the ambiguous pair.

B. Premature Queue

The premature queue stores the properties of premature operations and sends the elements to the arbiter for validation. The queue contains four labels, corresponding to the elements in P_m. There are two pointers: a head pointer and a tail pointer. The head pointer points to the earliest operation stored in the queue. The tail pointer points to the most recently stored operation in the queue. Each time an operation in the queue is

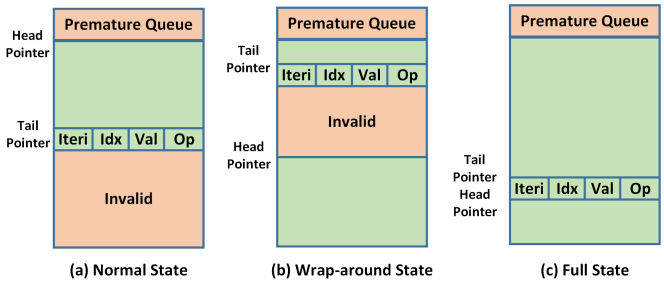

Fig. 4: The premature queue design. (a) Normal state, $P_{head} > P_{tail}$, data are stored between head and tail(painted in green). (b) Wrap-around state, $P_{head} < P_{tail}$, the queue stores data in cycle. (c) Full State, $P_{head} = P_{tail}$, the queue is full and have to stall the arbiter from sending premature operations to it.

validated, the head pointer moves one position forward. Each time a new operation is stored in the queue, the tail pointer moves one position forward. The queue utilizes a circular structure, resetting to the front whenever a pointer reaches the end as Fig. 4 shows. We define a parameter $Depth_q$ to represent the depth of the premature queue. A smaller $Depth_q$ requires fewer resources but is more likely to cause the pipeline to stall when the queue is full.

C. Arbiter

The arbiter serves as the interface between ambiguous pairs and the premature queue. Additionally, the arbiter initiates the *validation* stage and determines whether the pipeline needs to be squashed. The design of arbiter is show in Fig. 5. Ports connected to ambiguous pairs and the premature queue are similar, except that signals from premature operations need to be transmitted to the premature queue, while signals from the premature queue do not need to be sent back. Each entry in the arbiter contains the following fields: the iteration number, the index number, the data value, and the operation type. To ensure that all the aforementioned signals enter the core comparison logic simultaneously, we use merge—a typical dataflow component—to collect all the data of an operation before it is used for validation, thereby reducing race hazards. To transmit the operation type signal, data is not retrieved from premature operations. Instead, this signal is included in the merge. Based on the transmitted operation type signal, we design an *LMerge* (load merge) and an *SMerge* (store merge). These two distinct merges are used to minimize the impact on the original computation component.

Once a merge has collected all the signals from the premature operations, it transmits a packet of data to two destinations: one to be saved in the premature queue and the other to enter the validation logic. If both LMerge and SMerge are ready to send their tokens, which is situation corresponds to the condition in Eq. (2), we need to exact sequence from ROM to determine which operation should execute first. When all signals are ready for validation—i.e., when the mux successfully receives tokens from the premature operations and the premature queue—the *validation* stage will be triggered. Both signals enter the comparator and generate a squash token to flush the pipeline if the results are not equal.

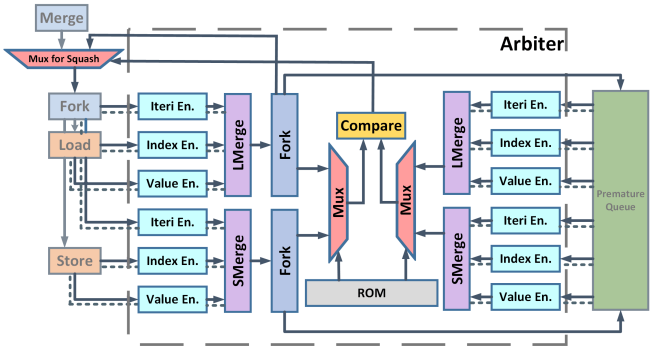

Fig. 5: The detailed design of arbiter. LMerge and SMerge pack the data for the premature queue and trigger the *validation* stage. The ROM stores the sequence of the original program.

V. BUILDING CIRCUITS WITH PREVV & OPTIMIZATION

In the previous section, we used Fig. 2(b) as an example to design a series of components for ***PreVV*** of an ambiguous pair. This design allows us to replace the resource-demanding LSQ with a simpler premature queue and an arbiter to validate all premature operations. However, in real-world tasks, the memory disambiguation problem can be even more challenging. In this section, we enhance the scalability of our approach and optimize our design for condition judgement.

A. Build an Ambiguous Pair with *PreVV*

Using Polyhedral analysis [25], we can easily find the ambiguous pairs. The only thing we need to know is $depth_q$. This parameter significantly influences the performance of the overall circuit both in resource and time. Neither a too small $depth_q$ nor a too large one can bring benefit to the circuit. Therefore, finding an appropriate $depth_q$ does matter.

Definition 2: if the average execution time of an ambiguous pair with ***PreVV*** equals to that of its predecessor, we call this pair a matched one. If certain pair is matched, the pipeline will have a minimum possibility to stall. The average execution time of an ambiguous pair t_p is calculated in the formula below:

$$t_p = t_{org}(1 + P_s) + t_{org} = t_{org}(2 + P_s), \tag{6}$$

where t_{org} is the execution time of the calculation part of the original dataflow circuit, and P_s is the possibility of pipeline squashing of this ambiguous pair.

The average execution time of its predecessor can be estimated as the waiting time for its successor to start, which can be written as:

$$t_w = \frac{t_{token}}{depth_q}, \tag{7}$$

where t_{token} accounts for the average stall time for a live out token ready for the premature queue. To make certain pair a matched one, we should guarantee that $t_p = t_w$. Note that we suppose none of the other ambiguous pair influence the estimation above, i.e., there is only one ambiguous pair in the estimation of $t_p = t_w$. To represent this condition, we give the below definition.

Definition 3: Two ambiguous pairs are overlapped if they share certain component(s). Pairs without overlap are called **independent pair**, which follows the below constraint:

$$\frac{d_{mn}}{ClockPeriod} \geq \frac{S_m + S_n}{ClockPeriod} = t_w = \frac{t_{token}}{depth_q}, \tag{8}$$

where d_{mn} is the distance between two ambiguous pairs m and n, i.e., the maximum number of components from the beginning of m to the end of n, and S_m and S_n are the span of m and n, representing the maximum number of components of all path in m and n, respectively. d_{mn} and S_m can be calculated as below:

$$d_{mn} = \max_{\substack{b \in m \\ e \in n}} \sum_{b_m \to e_n}^{C} C_{mn}, \tag{9}$$

$$S_m = \max_{p_{m \to n}} \sum_{C \in p} C_m^p, \tag{10}$$

where b_m and e_n represent the start point and end point of m and n, respectively. C_{mn} is the component from b_m to e_n. C_m^p is the component in path p inside m. In practice, we do not want this overlap happens because this overlap can always increase the complexity of the circuit as well as the resource overhead. We will explain and provide a solution to this problem in the next subsection.

B. Scalability For Large Scale Design

If two pairs overlap, it means that an operation belongs to multiple ambiguous pairs. Simply adding two separate premature queues and two separate arbiters would result in one operation being validated twice across both arbiters. This approach would also double resource usage. If an operation belongs to n ambiguous pairs, the complexity of the circuit Com_n and the frequency of the circuit frq_n will be

$$Com_n = 2^n Com_1, \tag{11}$$

$$frq_n = log_2(frq_1). \tag{12}$$

This approach is impractical for large HLS designs and contradicts our original intention of reducing resource usage. We propose a method to reduce the complexity of the problem. Consider a sequence of operations that have memory disambiguation issues with one another. Dependencies can be simplified, as a series of consecutive loads or stores do not form ambiguous pairs. Validating only one operation is sufficient to ensure overall correctness within each consecutive type. This approach significantly reduces complexity.

C. Optimization For Deadlock

An ambiguous pair may contain an element within an if condition, as shown in Fig. 6. However, the problem changes when considering premature value validation. If we simply use the arbiter described above and the condition evaluates to false, the arbiter will not receive tokens from the other branch in the same iteration. As a result, only the premature operation within the condition will be stored in the premature queue, preventing any validation from occurring. Once the queue overflows, the entire pipeline will stall, resulting in a deadlock. To address this issue, we tag operations that have data dependencies with those inside a conditional judgment. When the actual result of the conditional judgment does not select the tagged path, a pack of fake signals will be sent to the arbiter. When the arbiter receives the fake signal, it is convinced that the ambiguous pair does not take effect in the current iteration. The arbiter will then validate all operations stored in the queue with an iteration number lower than that of the fake signal, thus eliminating the potential for a deadlock.

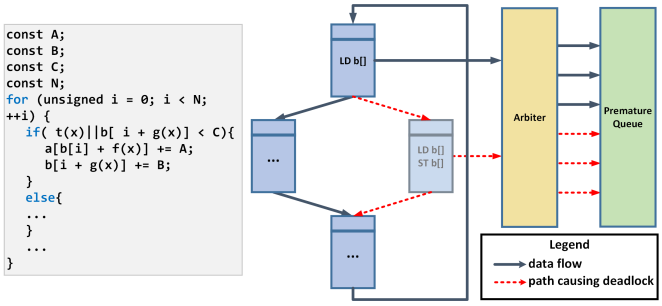

Fig. 6: Potential deadlock caused by *PreVV*. The red dashed lines are the cause of this deadlock.

VI. Experiment

This section describes the effectiveness of our strategy in eliminating resource overhead and its impact on timing.

A. Methodology

Our methodology is implemented as several hardware templates and an LLVM pass within Dynamatic, a state-of-the-art dynamic scheduling compiler [15]. The workflow is detailed below. First, the design from Section IV is realized as hardware templates and registered in Dynamatic. Six different kernels are also realized for evaluation in HLS form. Next, polyhedral analysis employed in Dynamatic is applied to identify all the ambiguous pairs within these kernels. Then, our LLVM pass is used to replace the LSQ in Dynamatic with ***PreVV*** components and generate the VHDL netlists. Finally, the design is evaluated against circuits generated by (1) plain Dynamatic without any plugins and (2) Dynamatic with a fast load-store queue allocation plugin [8]. Both Dynamatic and our design are synthesized using Vivado 2022.2 on xc7k160tfbg484-2, with a clock period constraint of 4 ns. To verify the correctness of the design, simulations are conducted in ModelSim 10.5c by comparing the results of the C++ kernel and the VHDL output. Two main benchmarks are focused on: (1) Overall resource usage, including LUTs, registers, and muxes. The use of DSP is not evaluated, as neither LSQ nor ***PreVV*** utilizes DSP. (2) The timing performance, which is represented in the number of total cycles, clock period (CP) and execution time.

We use several computational kernels as benchmarks. These kernels are all loop-nested in which memory dependencies are common in both the inner and outer loops. Therefore, LSQ or ***PreVV*** must be used to generate these circuits. The benchmarks we use are described below.

Kernel 2mm and **Kernel 3mm** refer to two and three matrix multiplications, respectively. They evaluate the performance of multidimensional array operations by combining two matrix

TABLE I: Resource usage in circuits generated by [15], [8] and our work.

Benchmark	LUT						FF					
	[15]	[8]	***PreVV*16**	***PreVV*64**	***PreVV*16** vs. [8]	***PreVV*64** vs. [8]	[15]	[8]	***PreVV*16**	***PreVV*64**	***PreVV*16** vs. [8]	***PreVV*64** vs. [8]
polyn_mult	20086	21567	14564	17859	**-32.47%**	**-17.19%**	2009	2101	1251	1785	**-40.46%**	**-15.04%**
2mm	39330	22190	10487	14518	**-52.74%**	**-34.57%**	8918	8715	4014	4687	**-53.94%**	**-46.22%**
3mm	57212	39742	24157	27842	**-39.21%**	**-29.94%**	9771	7661	3847	4494	**-49.78%**	**-41.34%**
gaussian	18383	19665	10687	13697	**-45.65%**	**-30.35%**	4339	4620	2451	2845	**-46.95%**	**-38.42%**
triangular	19830	20581	9814	15648	**-52.32%**	**-23.97%**	5921	6078	3951	4589	**-35.00%**	**-38.42%**
goemean					**-43.75%**	**-26.45%**					**-44.70%**	**-33.54%**

TABLE II: Timing performance in circuits generated by [15], [8] and our work.

Benchmark	Cycle Count				Clock Period (ns)				Exection Time (μs)					
	[15]	[8]	***PreVV*16**	***PreVV*64**	[15]	[8]	***PreVV*16**	***PreVV*64**	[15]	[8]	***PreVV*16**	***PreVV*64**	***PreVV*16** vs. [8]	***PreVV*64** vs. [8]
polyn_mult	2701	2401	2512	2314	7.26	7.24	7.2	7.2	19.61	17.38	18.09	16.66	+4.05%	**-4.16%**
2mm	3231	2498	2789	2471	7.80	7.77	7.68	7.63	25.20	19.41	21.42	18.85	+10.36%	**-2.86%**
3mm	4382	2498	2789	2471	8.29	7.78	7.7	7.72	36.33	19.43	21.48	19.08	+10.50%	**-1.84%**
gaussian	7651	6871	8754	6681	8.16	8.16	8.06	8.06	62.43	56.07	70.56	53.85	+25.84%	**-3.96%**
triangular	9895	9892	9912	9812	9.18	7.36	7.31	7.31	90.84	72.81	72.46	71.73	-0.48%	**-1.48%**
goemean														**-2.64%**

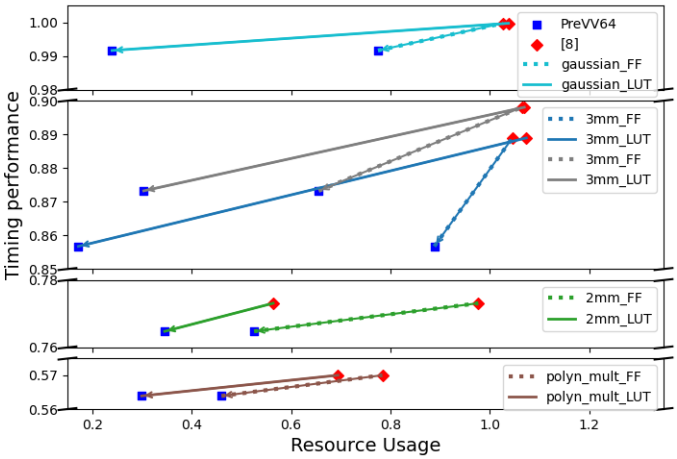

Fig. 7: Resource usage (LUT in solid and FF in dashed lines) in circuits generated by [8] and our work. The results are normalized to those of Dynamatic [15].

multiplications with matrix addition. These experiments involve complicated loop structures, resulting in numerous ambiguous pairs.

Polyn_mult is a kernel used for polynomial multiplication, where two polynomials are represented as arrays. Requiring a large number of operations, the kernel is computate-bounded. The kernel also features double nested loops, which generate ambiguous pairs. Unlike 2mm and 3mm, **polyn_mult** limits data reuse, thus better demonstrating the performance of LSQ.

Gaussian Elimination and **Triangular Matrix Multiplication** are typically used for solving systems of linear equations or performing matrix operations, such as finding the inverse of a matrix or conducting LU decomposition. Similar to the benchmarks above, these kernels are also loop-nested.

B. Experiment Results

Table II shows the results from [15], [8] and our work. ***PreVV*16** and ***PreVV*64** represent ***PreVV*** with a premature queue depth of 16 or 64, respectively. ***PreVV*16** significantly reduces resource usage, with an average decrease of 45.21% in LUTs and 33.09% in FFs. However, the total number of cycles increases by 10.79%. ***PreVV*16** achieves a smaller reduction in resource usage, with an average decrease of 27.2% in LUTs and 33.1% in FFs, along with a 2.64% decrease in execution time.

The observed reduction in resource overhead aligns with our expectations. This is due to replacing the entire LSQ with a simpler premature queue. Additionally, introducing an arbiter for value validation adds some extra overhead. As a result, overall resource usage decreased by 40%, rather than the expected 50%, when replacing the original LQ and SQ with a single queue. However, the design requires a 10.79% increase in clock cycles at $Depth_q = 16$ due to pipeline flushing caused by incorrect premature operations, resulting in additional time consumption. Meanwhile, both ***PreVV*16** and ***PreVV*64** reduce the CP because our ***PreVV*** architecture does not need complex LSQ searching logic. Increasing the depth of the premature queue resulted a decreasing 2.64% in execution time compared to state-of-the-art LSQ designs. This suggests that when the premature queue depth is too small, it fills up quickly, causing backpressure to the arbiter and leading to pipeline stalls. The reduction of total execution time of ***PreVV*64** suggests that by carefully choosing an appropriate $depth_q$, reducing resource overhead without incurring any timing cost is possible.

VII. Conclusion

Dataflow circuits can achieve high performance due to their out-of-order execution but incur significant resource overhead. This work presents a premature value validation architecture that replaces the resource-intensive LSQ with an arbiter and a simple premature queue. Experiments demonstrate that our design can significantly reduce resource overhead (up to 56.14% in one case) without any extra timing cost (and even improvement when parameter is carefully chosen). Adjusting the depth of the premature queue allows for a tradeoff between timing performance and resource overhead.

Acknowledgment

This work was partly supported by Project of Shanghai EC (24KXZNA12). We thank CFFF platform of Fudan University for access to high performance computing hardware.

REFERENCES

[1] H. W. Cain and M. H. Lipasti, "Memory Ordering: A Value-Based Approach," SIGARCH Comput. Archit. News, vol. 32, no. 2, p. 90, Mar. 2004, doi: 10.1145/1028176.1006709.

[2] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "Correction: InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy".

[3] S. Subramaniam and G. Loh, "Fire-and-Forget: Load/Store Scheduling with No Store Queue at All," in 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06), Orlando, FL, USA: IEEE, Dec. 2006, pp. 273–284. doi: 10.1109/micro.2006.26.

[4] L. Josipovic, P. Brisk, and P. Ienne, "An Out-of-Order Load-Store Queue for Spatial Computing," ACM Trans. Embed. Comput. Syst., vol. 16, no. 5s, pp. 1–19, Oct. 2017, doi: 10.1145/3126525.

[5] L. Josipovic, A. Guerrieri, and P. Ienne, "Speculative Dataflow Circuits," in Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside CA USA: ACM, Feb. 2019, pp. 162–171. doi: 10.1145/3289602.3293914.

[6] L. Josipovic, A. Bhattacharyya, A. Guerrieri, and P. Ienne, "Shrink It or Shed It! Minimize the Use of LSQs in Dataflow Designs," in 2019 International Conference on Field-Programmable Technology (ICFPT), Tianjin, China: IEEE, Dec. 2019, pp. 197–205. doi: 10.1109/ICFPT47387.2019.00031.

[7] L. Josipovic, A. Guerrieri, and P. Ienne, "Synthesizing General-Purpose Code Into Dynamically Scheduled Circuits," IEEE Circuits Syst. Mag., vol. 21, no. 2, pp. 97–118, 2021, doi: 10.1109/MCAS.2021.3071631.

[8] A. Elakhras, R. Sawhney, A. Guerrieri, L. Josipovic, and P. Ienne, "Straight to the Queue: Fast Load-Store Queue Allocation in Dataflow Circuits," in Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Monterey CA USA: ACM, Feb. 2023, pp. 39–45. doi: 10.1145/3543622.3573050.

[9] J. Xu and L. Josipović, "Suppressing Spurious Dynamism of Dataflow Circuits via Latency and Occupancy Balancing," in Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Monterey CA USA: ACM, Apr. 2024, pp. 188–198. doi: 10.1145/3626202.3637570.

[10] A. Elakhras, A. Guerrieri, L. Josipovic, and P. Ienne, "Survival of the Fastest: Enabling More Out-of-Order Execution in Dataflow Circuits," in Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Monterey CA USA: ACM, Apr. 2024, pp. 44–54. doi: 10.1145/3626202.3637556.

[11] J.-M. Gorius, S. Rokicki, and S. Derrien, "A Unified Memory Dependency Framework for Speculative High-Level Synthesis," in Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction, Edinburgh United Kingdom: ACM, Feb. 2024, pp. 13–25. doi: 10.1145/3640537.3641581.

[12] S. Derrien, T. Marty, S. Rokicki, and T. Yuki, "Toward Speculative Loop Pipelining for High-Level Synthesis," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol. 39, no. 11, pp. 4229–4239, Nov. 2020, doi: 10.1109/TCAD.2020.3012866.

[13] J.-M. Gorius, S. Rokicki, and S. Derrien, "SpecHLS: Speculative Accelerator Design Using High-Level Synthesis," IEEE Micro, vol. 42, no. 5, pp. 99–107, Sep. 2022, doi: 10.1109/MM.2022.3188136.

[14] R. Szafarczyk, S. W. Nabi, and W. Vanderbauwhede, "A High-Frequency Load-Store Queue with Speculative Allocations for High-Level Synthesis," Nov. 14, 2023, arXiv: arXiv:2311.08198. Accessed: Sep. 18, 2024. [Online]. Available: http://arxiv.org/abs/2311.08198

[15] L. Josipović, A. Guerrieri, and P. Ienne, "Invited Tutorial: Dynamatic: From C/C++ to Dynamically Scheduled Circuits," in Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside CA USA: ACM, Feb. 2020, pp. 1–10. doi: 10.1145/3373087.3375391.

[16] J. Liu, C. Rizzi, and L. Josipovic, "Load-Store Queue Sizing for Efficient Dataflow Circuits," in 2022 International Conference on Field-Programmable Technology (ICFPT), Hong Kong: IEEE, Dec. 2022, pp. 1–9. doi: 10.1109/ICFPT56656.2022.9974425.

[17] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol. 20, no. 9, pp. 1059–1076, Sep. 2001, doi: 10.1109/43.945302.

[18] N. Gädke-Lütjens, Dynamic Scheduling in High-Level Compilation for Adaptive Computers, Ph.D. dissertation, Technischen Universität Braunschweig, Braunschweig, Germany, Apr. 2011.

[19] M. Budiu, P. V. Artigas, and S. C. Goldstein, "Dataflow: A complement to superscalar," in Proc. IEEE Int. Symp. Performance Analysis of Systems and Software (ISPASS), Austin, TX, USA, Mar. 2005, pp. 177-186.

[20] Arvind and R. S. Nikhil, "Executing a program on the MIT Tagged-Token dataflow architecture," IEEE Trans. Computers, vol. 39, no. 3, pp. 300–318, Mar. 1990.

[21] J. Cheng, G. A. Constantinides, and J. Wickerson, "DASS: Combining Dynamic & Static Scheduling in High-Level Synthesis," vol. 41, no. 3, 2022.

[22] A. Elakhras, A. Guerrieri, L. Josipovic, and P. Ienne, "Unleashing Parallelism in Elastic Circuits with Faster Token Delivery," in 2022 32nd International Conference on Field-Programmable Logic and Applications (FPL), Belfast, United Kingdom: IEEE, Aug. 2022, pp. 253–261. doi: 10.1109/FPL57034.2022.00046.

[23] B. R. Rau, "Iterative modulo scheduling," International Journal of Parallel Programming, vol. 24, no. 1, pp. 3–64, Feb. 1996.

[24] J. H. Patel and E. S. Davidson, "Improving the throughput of a pipeline by insertion of delays," in Proc. 3rd Annu. Symp. Comput. Archit., 1976, pp. 159–164.

[25] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly-polyhedral optimization in LLVM," in Proc. 1st Int. Workshop Polyhedral Compilation Techniques (IMPACT), Chamonix, France, Apr. 2011, pp. 1–6.

[26] Y. Wang, Y. Li, H. Zhang, et al., "Moth: A hardware accelerator for neural radiance field inference on FPGA," in Proc. 2023 IEEE 31st Annu. Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM), IEEE, 2023, pp. 227–227.

[27] K. Qian, Z. Liu, Y. Liu, et al., "AutoHammer: Breaking the compilation wall between deep neural network and overlay-based FPGA accelerator," in Proc. 2024 ACM/SIGDA Int. Symp. Field Program. Gate Arrays, 2024, pp. 185–185.