# PoCo: Extending Task-Parallel HLS Programming with Shared Multi-Producer Multi-Consumer Buffer Support

AKHIL RAJ BARANWAL and ZHENMAN FANG, School of Engineering Science, Simon Fraser University, Canada

Advancements in High-Level Synthesis (HLS) tools have enabled task-level parallelism on FPGAs. However, prevailing frameworks predominantly employ single-producer-single-consumer (SPSC) models for task communication, thus limiting application scenarios. Analysis of designs becomes non-trivial with an increasing number of tasks in task-parallel systems. Adding features to existing designs often requires re-profiling of several task interfaces, redesign of the overall inter-task connectivity, and describing a new floorplan. This paper proposes PoCo, a novel framework to design scalable multi-producer-multi-consumer (MPMC) models on task-parallel systems. PoCo introduces a shared-buffer abstraction that facilitates dynamic and high-bandwidth access to shared on-chip memory resources, incorporates latency-insensitive communication, and implements placement-aware design strategies to mitigate routing congestion. The frontend provides convenient APIs to access the buffer memory, while the backend features an optimized and pipelined datapath. Empirical evaluations demonstrate that PoCo achieves up to 50% reduction in on-chip memory utilization on SPSC models without performance degradation. Additionally, three case studies on distinct real-world applications reveal up to 1.5× frequency improvements and simplified dataflow management in heterogeneous FPGA accelerator designs.

CCS Concepts: • **Hardware → On-chip resource management**; **Resource binding and sharing**; **Hardware-software codesign**; • **Computer systems organization → Data flow architectures**; **Reconfigurable computing**; **High-level language**.

Additional Key Words and Phrases: Multi-producer multi-consumer, buffer optimization, floorplan optimization, multi-die FPGA, high-level synthesis

**ACM Reference Format:**
Akhil Raj Baranwal and Zhenman Fang. 2025. PoCo: Extending Task-Parallel HLS Programming with Shared Multi-Producer Multi-Consumer Buffer Support. *ACM Trans. Reconfig. Technol. Syst.* 1, 1, Article 1 (October 2025), 32 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

In recent years, FPGAs have seen surging interest as datacenter accelerators, marked by major industry moves such as AMD's acquisition of Xilinx [2] and Intel's investment in Altera [1].

Hyperscale cloud providers have deployed FPGAs at scale to speed up diverse workloads—from machine learning to network processing—and offer FPGA-based instances to customers [3]. Despite this momentum, significant challenges remain as FPGA development is still quite complex. With larger designs, the underlying hardware design presents non-trivial challenges to handle from a software perspective.

Authors' Contact Information: Akhil Raj Baranwal, akhil_baranwal@sfu.ca; Zhenman Fang, zhenman@sfu.ca, School of Engineering Science, Simon Fraser University, Burnaby, British Columbia, Canada.

**H**igh **L**evel **S**ynthesis (HLS) tools evaluate the accelerator program as a monolithic control-state based on the inferred sequential semantics of consuming and producing data on the off-chip memory - resulting in kernels that ingest data from the off-chip memory, compute it as defined by the designer, and push the output back onto the off-chip memory, while the host runs the drivers to interact with the off-chip memory. In contrast, current state-of-the-art frameworks like TAPA [14, 23] and PASTA [16, 17] offer models for **T**ask **P**arallel **S**ystems (TPS) where the design is compiled as multiple (HLS) tasks running independently, interfacing via FIFO streams and/or buffer channels. These frameworks implement a model wherein producer tasks and consumer tasks are connected by latency-insensitive communication channels. To address timing closure issues in multi-die FPGAs, these tools involve a coarse-grained floorplanning stage to place the tasks in local FPGA regions, followed by a pipelining step to add pipeline registers between tasks and their communication channels.

It is challenging for HLS tools to outperform designs implemented by hardware engineers working directly with RTL. A major reason for this is the opacity of these tools, wherein the user is responsible only for the high-level code. Alternatively, working with RTL flows is quite complex, which limits the scope of applications that can be designed by the front-end programmer. When building FPGA applications, a designer has to work not just on the accelerator kernel itself but also the application-specific data transfer to and from its interfaces. They need to consider the interaction of all tasks, individually optimizing each for the intermediary dataflow. This makes incremental improvement quite tedious and error-prone even at the level of an unoptimized proof-of-concept, because the addition of a new task or feature may require rewriting several other tasks, adding new communication channels between all existing tasks participating with the new one.

While these tools provide commendable automatic optimizations, especially considering the convenient high-level programming abstractions they support, there are specific challenges that show up only during implementation of an application. These challenges span the entire stack of the FPGA design cycle - from the superficial level of behavioral design to the on-fabric placement optimizations towards optimal floorplans. Existing HLS tools hinder the development of complex task arrangements on hardware, mainly due to their rigidity of the underlying channels. They work on an explicit single-producer single-consumer (SPSC) model with point-to-point FIFO or buffer channels, which means that there can be at most one producer and one consumer connected to these channels. For example, if an on-chip memory needs to be accessed by two consumers, they should be wrapped in a parent task boundary so that the memory sees only one consumer. This constraint is obvious to a hardware designer because inputs to an on-chip element cannot have multiple drivers and outputs must have well-defined loads. But, this also means that all inter-task interaction involving memory has to be explicitly programmed on a one-to-one basis. Although this limitation seems rather harmless at first glance, the issue is the design complexity faced by the programmer because no three tasks ever share the same view of on-chip memory. Without a common view of on-chip memory, dataflow can be coordinated only between two tasks, not more. Such a limitation, however obvious, exist on hardware and is opaque to the software designer using a high-level language like C/C++. As such, the overall paradigm of **A**ccelerator-**R**ich **A**rchitectures (ARAs), which contains multiple tasks with complex inter-task communication, depends not only on TPS models for concurrent execution of the compositional tasks, but also on a robust Multi-Producer-Multi-Consumer (MPMC) communication interface to a shared buffer that provides a common view of (on-chip) memory.

To this end, we present the PoCo framework to offer a standardized on-chip memory view and make its connected tasks appear as plug-and-play IPs in an MPMC model connected to a high-performance datapath. PoCo takes a TPS program and buffer configuration as input and generates an MPMC buffer design with floorplan-aware optimizations.

Introducing MPMC models brings several challenges. First, enabling dataflow between multiple producers and consumers introduces complexity of intermediary connectivity and coordination of several producer-consumer relationships at once. Second, a common view of memory is needed to standardize a buffer implementation that supports arrays with different partitioning schemes such as block, cyclic, and complete as these are essential for dataflow optimizations in HLS designs. Third, an increasing complexity of the buffer implementation makes efficient data movement a challenge to and from the on-chip memory. Fourth, the underlying logic of accessing large buffer arrangements may lead to layout and timing issues with fabric-implementations. We discuss all these challenges in detail in Sec 2.

These challenges introduced by MPMC models have been addressed at different levels of the FPGA design stack. First, we provide a shared-buffer abstraction for TPS frameworks which defines a standardized channel on which a producer/consumer shall transact data in an MPMC model. Second, the abstraction is appended with an allocator that assigns indices to memory regions inside the MPMC-buffer. We then leverage the PASTA framework [16] to design the memory backend of the shared-buffer such that it can support different dimensions and partitioning schemes. Third, to saturate data movement to and from dual-port memories, we provide a RTL module that provides better throughput from memory cores (BRAMs/URAMs) in ping-pong setups. Fourth, we modify the floorplanner so that it may generate suitable constraints to alleviate local congestion in the floorplan. Finally, to support different configurations, we provide a buffer generator that automatically stitches the user's tasks with the buffer pipeline. With these solutions, we create a TPS-based internal buffer pipeline to handle memory requests between different user tasks while still keeping them sufficiently dissociated with the underlying challenges of doing so.

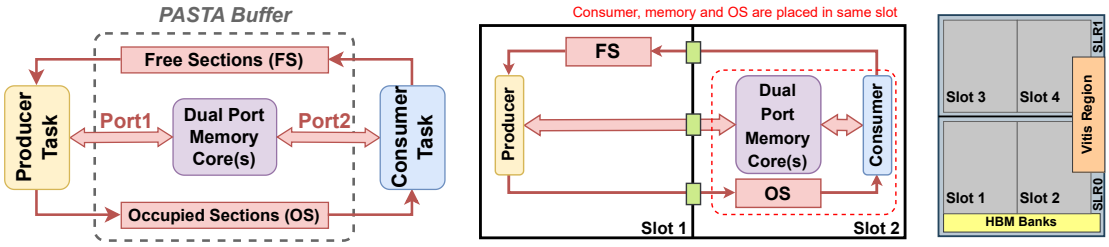
Our frontend consists of a programming interface where users can specify the data-type, buffer depth, partitioning scheme, memory core type, the number of blocks, and (optionally) the number of pages using C++ template types. To provide ease of use in the frontend, PoCo also provides convenient APIs for declaration and usage of MPMC-buffers, followed by a source-to-source transformation to hide the latency effects of the buffer pipeline.

We first test the backend buffer optimizations on a few benchmarks from the Rodinia-HLS [10] suite and show that we achieve similar frequency as PASTA [16] on AMD-Xilinx's Alveo U50 and Alveo U280 boards while using lesser on-chip memory resources. We also show that for memory-bandwidth constrained designs that involve BRAM duplication, our buffer optimizations lead to up to 50% reduction of memory resources without any loss in frequency. Finally, we perform three case studies to demonstrate that designs built with our PoCo framework can achieve up to 1.5× frequency improvement over prior work such as AutoBridge [15] and PASTA [16], while greatly simplifying programming efforts.

In summary, this work makes the following contributions:

1). Analysis and design of an MPMC buffer abstraction with dynamic buffer management for task-parallel systems.
2). Hardware optimization of dual-port memory channels so as to extract maximum available bandwidth.
3). PoCo, a framework to support MPMC models, providing convenient APIs along with frontend and backend optimizations.
4). Experiments and case studies to demonstrate functional and performance enhancements using PoCo.

The rest of the paper is organized as follows. In Sec 2, we first introduce existing work in this area. Next, we use an example design to describe the challenges encountered when implementing MPMC designs and aim to understand the concerns from the perspective of the frontend programmer. In pursuit of an easy-to-use framework, we propose a dynamic shared buffer abstraction for MPMC models in Sec 3. Then, we detail the architecture and discuss its implementation in Sec 4.




(a) Latency insensitive PASTA buffer channel (red) (b) Placement strategy using pipeline registers (green) (c) Alveo U50

Fig. 1. Overview of the PASTA [16, 17] framework - (a) buffer channel, (b) slot placement, (c) schematic device overview

In Sec 5, we describe the overall toolflow and discuss the frontend and backend optimizations. In Sec 6, we conduct experiments through two setups to showcase the efficiency and scalability of our MPMC buffer design.

In Sec 6.4, we conduct three case studies to demonstrate the effectiveness of using MPMC buffers in real-world applications. We discuss related work in Sec 7, and conclude in Sec 8.

## 2 Background and Motivation

### 2.1 Overview of Existing TPS Frameworks

Task-Parallel Systems (TPS) are suitable for spatial architectures with heterogeneous Processing Elements (PEs). They expect an explicit dictation of *tasks*, each having its own separate FSM (Finite State Machine), driven by the availability of data on its interfaces [6]. Frameworks like TAPA [14, 23] and PASTA [16, 17] offer TPS models where the design is compiled as multiple independent tasks, interfacing via latency-insensitive FIFO streams and/or buffer channels. This gives the flexibility of fragmenting the design into multiple sub-components and eases the physical layout of the design.

The stream interfaces described in TAPA and buffer channels described in PASTA are point-to-point connections that exist between a *producer* task and a *consumer* task. Stream interfaces are essentially point-to-point FIFOs. PASTA buffer channels contain dual port memory-core(s) and two FIFOs for tokenization, as shown in Fig. 1a. For the remainder of this paper, we use the term *memory-core* to denote a unit of on-chip memory (composed of BRAM/URAM cells) and the term *buffer* to denote a memory abstraction from the perspective of a user task. The *buffer* may be made of one or more units of one type of *memory-core*. The producer and consumer tasks connects with each memory core via one of its ports. The two FIFOs connect the producer and consumer directly, with the *Free Sections (FS)* FIFO sending tokens from consumer to producer, while the *Occupied Sections (OS)* FIFO sends tokens from producer to consumer. To support ping-pong buffering, each memory-core includes multiple sections, allowing the producer to write the next data tile into one of the free sections while the consumer reads one of the occupied (i.e., previously written) sections. The FS FIFO contains tokens of all the sections that do not contain any valid data and are ready to be written to by the producer task. The OS FIFO contains all sections that contain valid data and are ready to be read by the consumer task. Tokens are unique indices of the sections present in the memory-core.

To perform a memory access, the producer reads a token from the FS FIFO and accesses the corresponding section from the memory cores. After writing the data, the producer writes the token back into the OS FIFO. The consumer reads available tokens from the OS FIFO, and accesses the valid data that the producer has already written. Once a read is done, the token is written back to the FS FIFO, so that it may be rewritten by the producer.

On modern multi-die FPGAs, large monolith designs often suffer from poor timing closure during the Placement and Routing (PNR) phase due to complex state machines and combinational paths that may stretch between the entire

span of the monolith. This issue is more pronounced with die-crossing - connections between two dies or Super Logic Regions (SLRs), since they incur significant timing penalty. Since TPS frameworks implement the accelerator as multiple independent tasks, each can be constrained inside a *slot*, which are Pblocks defined inside an SLR. For example, Fig. 1c shows the schematic layout of the Alveo U50 FPGA, which has 2 SLRs, each divided into 2 slots. Constraining tasks to slots minimizes long combinational nets by preventing the associated cells from overspilling the slot boundary. The only nets between the slots are, therefore, either FIFO channels or PASTA buffer channels. Both of these channels can be modified by adding pipeline registers in between, which improve timing closure at the expense of extra latency. Consequently, the producer and consumer can be placed in different slots with a pipelined communicational channel in between, resulting in high-frequency designs.

When a FIFO channel with depth $D$ is modified by inserting $X$ pipeline registers, the FIFO logic is also modified to assert the *full* signal earlier - when the FIFO has $D - X$ values available. This is necessary since the *full* signal (generated at the FIFO's output) requires $X$ cycles to be registered at the FIFO's input. In a purely streaming design - that is, a design with only FIFO interfaces between the tasks, an increase in latency does not have any significant effect since the throughput remains the same. The buffer channel includes 2 FIFOs and 2 memory ports, so its pipelining is handled differently. Pipelining the path between the producer and memory has no effect on the producer because the data to be written is latched along with the address. However, pipelining the path between the consumer and memory increases the latency of each read operation, since the memory latches the read-data after reading the address. If $X$ registers are inserted on the consumer-memory path, the read operations suffer a latency of $2X$ ($X$ cycles for the address to be sent and $X$ cycles for the read-data to be received). In most cases, the compiler can hide this latency by either overlapping read operations or scheduling them earlier. However, if the task is such that the read-data influences the loop Initiation Interval (II), adding pipeline registers impact the performance of the design negatively. When two large tasks connected via a buffer channel need to be placed in separate slots, PASTA framework packs the consumer and its associated memory-cores together, as shown in Fig. 1b. The FS and OS FIFOs are pipelined normally on the input side. When PASTA detects that the producer may also perform some reads, it packs the producer as well in the same FPGA slot.

The communication model of existing TPS frameworks follow a strictly single-producer single-consumer (SPSC) model, wherein all inter-task communication is explicitly implemented as point-to-point channels. FIFO streams do not connect the tasks to any on-chip resource, so they are by nature an implementation of SPSC communication. However, buffers involve connections with on-chip memory, which often needs to be accessed through multiple producers and consumers depending on the application. When on-chip memory (OCM) needs to be accessed by multiple producers and/or multiple consumers, typical HLS flows and TPS frameworks encounter several key challenges during implementation. Interestingly, most of these challenges show up as poor performance and/or routing failures only during actual implementation of MPMC designs. To illustrate these better, we take the example of the Shuffle workload. In the context of a Map-Reduce application, the Shuffle stage gathers every key–value pair generated by the map tasks, partitions them by different keys, streams each partition to the reducer responsible for the key, and locally orders the data so each reducer receives a contiguous run of values it can process sequentially. This heavy data-movement phase re-aligns scattered fragments into sorted, co-located blocks, and sets both the latency-floor and bandwidth-ceiling for the entire Map-Reduce job. In the following text, we use the shuffle kernel as an example design to describe the challenges encountered when implementing MPMC designs, understand the programmers' expectation from an automation framework that supports MPMC models, and discuss considerations to be made when implementing such a framework.

Table 1. Description of terminology in the context of task-parallel systems with buffer support

| Term | Description |
|---|---|
| SPSC | Single Producer Single Consumer |
| MPMC | Multiple Producer Multiple Consumer |
| Memory-core | A unit of on-chip memory composed either of BRAM cells or URAM cells |
| Buffer | A memory abstraction from the perspective of a user task |
| Slot | A region on FPGA fabric composed of several Pblocks |
| Task | A processing element with an independent FSM, connected to other tasks only via standardised FIFO or PASTA buffer channels |

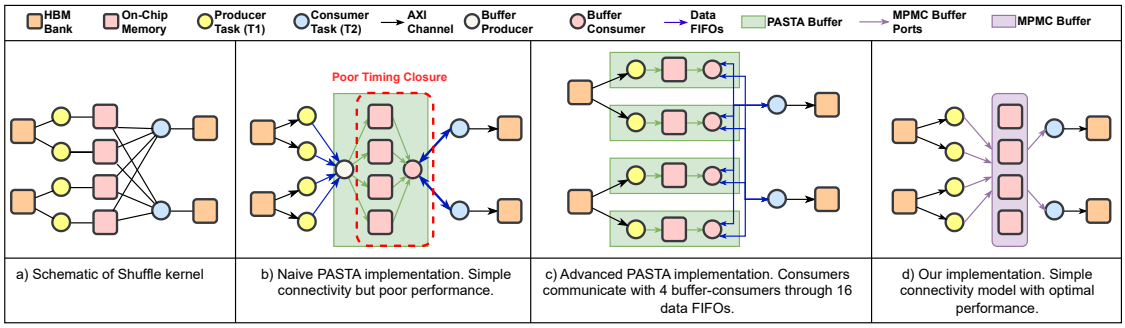



Fig. 2. Shuffle kernel - (a) Workload schematic; (b) A naive PASTA implementation wraps the buffers with one producer and consumer task; (c) Advanced PASTA implementation requires 4 extra buffer consumers and 16 data FIFOs; (d) Our implementation simplifies MPMC communication scheme.

### 2.2 Challenges towards Implementing MPMC Models

Table 1 lists the terminology we use in the context of buffer applications on task-parallel systems. Fig. 2a shows the schematic of a Shuffle kernel, with four instances of task T1, four buffers, and two instances of task T2. T1 (producer task) reads data from the HBM and produces key-value pairs in its respective buffer. T2 (consumer task) accumulates all values of a specific key from each buffer and streams it sequentially into the HBM. Each instance of T1 sinks into its respective buffer, while each instance of T2 requires a connection to all buffers.

*2.2.1* ***Connectivity and Access Scheduling.*** When an on-chip memory (OCM) resource is accessed by multiple consumers, HLS tools insert demultiplexers (deMUXs) between multiple consumers and the shared resource, switching the deMUXs based on the inferred schedule between the consumers. This means that the deMUXs' selection signal is driven by the FSM of *all* consumers accessing this resource, inferring combinational nets between them. Similarly, when a resource is accessed by multiple producers, HLS tools insert multiplexers (MUXs), generating a selection logic based on the inferred schedule. This method of binding multiple tasks together using combinational nets creates monolithic designs that may have large combinational nets. Separating these large tasks across a slot invites timing penalty. Fig. 2b shows the combinational nets inferred to implement sharing of OCM between the 2 instances of T2. The boundary of the inseparable monolith is shown as a dashed red-line.

Existing TPS frameworks offer no method to arbitrate a resource between multiple producers and/or consumers. Consequently, no two producers or consumers sharing the same OCM element can exist as independent FSMs and the compiler must infer combinational paths between them to arbitrate their access. With a naive implementation model (Fig. 2b), the designer will want to combine the memory elements and T2 tasks into a single wrapper task (red dashed line). For such designs, HLS tools will evaluate static dependencies and employ time-division multiplexing

(TDM) to provide access to the buffers because the access-pattern cannot be inferred at compile time. For example, the two user-consumers can make requests to different locations in the same OCM at the same time, for which the access will have to be statically time-multiplexed, possibly inflating the initiation interval (II) for each access. This solution is acceptable only when all consumers access the common memory element at the same rate. In case one consumer accesses the memory more frequently than another, they are still assigned the same bandwidth due to the static TDM. In addition, this strategy constrains the buffer-consumer and all the common on-chip resource within the same task boundary and creates a large unified task instead of several smaller ones, effects of which are observed either as hotspot congestion, over-utilization of resources in a slot, or poor clock-frequency due to die-crossings in later PNR stages. This severely hampers the development of multi-producer multi-consumer (MPMC) models on task parallel systems.

When following the advanced PASTA implementation (Fig. 2c), the designer will first need to create intermediary consumer tasks for each buffer to satisfy the SPSC model, and then connect each T2 task with the buffer-consumer tasks. Furthermore, each buffer-consumer must handle multiple requests from T2 tasks, read from the buffer, and send a response to the appropriate T2 task. Fig. 2c shows the data FIFOs involved for this communication in blue. Note that each arrow is double-headed, meaning that there are a total of 16 new FIFOs (2 between each buffer-T2 pair) that carry requests to the buffer-consumer task and responses back from them.

The performance of the implementation described in Fig. 2c will largely depend on how the buffer-consumer task is designed. If this task simply arbiters over each request, the resulting static schedule will only ever utilize half of the total bandwidth available, since there are two instances of T2. Moreover, different instances of T2 might perform different number of total reads. In such a case, when a task finishes its accesses sooner than the other, the other task should ideally receive a higher bandwidth. However, due to the static schedule, the pending task communicates on its usual routine and cycles are wasted arbitrating access to an idle task that will never perform any more reads. If the buffer-consumer implements a priority of requests, one instance of T2 will be backpressured until the other instance has completed its accesses. For optimal performance, the buffer-consumer has to be carefully designed such that they can recognize the idle state of a connected task and divert the remaining bandwidth to the active tasks, with switching priorities or other advanced logic suited to the application at hand. These optimizations are arduous to implement because adding just one new instance of either T1 or T2 in the design requires the re-profiling of all other tasks and interfaces. This hinders portability, impedes reuse of existing work, and makes incremental work challenging.

With these considerations, the first expectation from the perspective of the designer would be a simple and standardized connectivity between tasks and shared resources. This connectivity should abstract the dynamic scheduling of access requests between different tasks. This is essential not just in the interest of performance, but more importantly for preserving the design philosophy of independent tasks in TPS. For MPMC models, there is not much use to the distinction between the producer and consumer functionality of a task, so it is prospicient to have an MPMC model not differentiate between a producer and a consumer at the interface level, since that will only put more restrictions on the user to explicitly select the nature of the task. Instead, we assume that any task may have the requirement to produce *and* consume. For the remainder of this paper, we will use the term *transactor* to refer to a task that accesses (either produce or consume) a shared resource. For example, the shuffle kernel in Fig. 2 has 6 transactors.

*2.2.2* ***Control Coordination and Resource Reuse***. In many designs, tasks often deviate from their maximum allocated resource budgets. In the shuffle workload, an instance of T1 generates key-value pairs and stores them in its corresponding buffer. The number of such key-value pairs generated will be dependent on input data and the key. Naturally, different instances of T1 have differing workloads and hence differing usages of the buffer, i.e., one instance

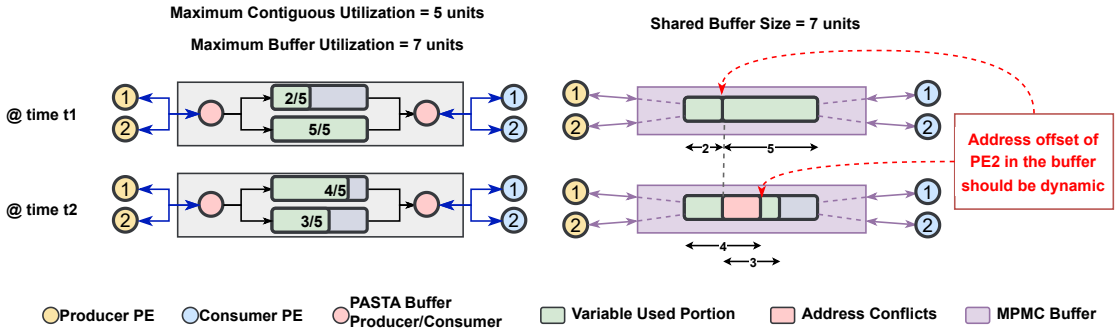



Fig. 3. Separated buffers need to be converted into one shared buffer to provide a common view of memory in MPMC model. When variably-utilized buffers are shared, the producers need dynamic address-offsets based on their buffer usage at that specific time.

of T1 might not make use of the full depth of its corresponding buffer, while another instance might require more depth to store all values of a key. As another example, in applications involving graphs, different parent nodes might refer to different numbers of child nodes at different times in the overall execution, utilizing a variable size in the buffer. Hence, as the number of producers and consumers grows in an MPMC model, it is not advisable to make static allocations of constant-size buffers for all individual tasks because all tasks combined do not utilize the full address-space of the buffer at the same time. Moreover, if buffers are assigned statically to PEs, the buffer-size will have to hold the largest piece of data that a PE may produce during overall execution, which will inflate OCM utilization quickly. With an increasing number of independent tasks, the opportunity of buffer reuse also grows, and a framework supporting MPMC models should also provide each transactor the flexibility to use OCM judiciously and enable buffer reuse. However, sharing and reuse of buffers is not straightforward. Producers in MPMC models should not inadvertently overwrite each other's data, which requires that the producers should be connected to all other producers to track each others' resource usage.

As an example, Fig. 3 shows a sample application with 2 producer PEs and 2 consumer PEs, each accessing two buffers through a PASTA buffer channel. The maximum allocated resource budget for a single PE is 5 units, with both PEs combined using 7 units in total. The utilization of the buffers varies dynamically. When the buffers are merged such that they can be shared, a dynamic offset is required for PE-2 to prevent corrupting data written by PE-1. Hence, there exists a requirement of automatically managing ownership between transactors while still keeping it transparent enough for the designer to drive the reuse of underutilized buffers. The transactors know the size of data only at runtime, so the designer should be able to leverage the benefits of an MPMC model without having to explicitly plan the intricate management of ownership and reuse. To prevent address conflicts, a centralized mechanism of book-keeping the dynamic allocation of resources is required. This is essentially a control-level problem, and should ideally create minimal interference on any ongoing data transfer.

Because a buffer may be produced by as many tasks and consumed by as many tasks as are present in the MPMC model, the target framework should also provide convenient APIs that are able to perform I/O requests and optionally acquire a read/write mutex over a resource, so that it becomes locked either for reading or for writing. As an example, write requests from T1 must be stalled until *all* instances of T2 have finished generating read requests to prevent data hazards. There should exist some mechanism between the individual modules to signal that a resource has been written so that the read can resume and vice versa. To keep the usage simple for the designer, the transactor should only be responsible for signalling the mutex, its maintenance should be opaque. To the transactors, the framework should provide the same global view of shared on-chip memory with unique addresses for all locations available in the global

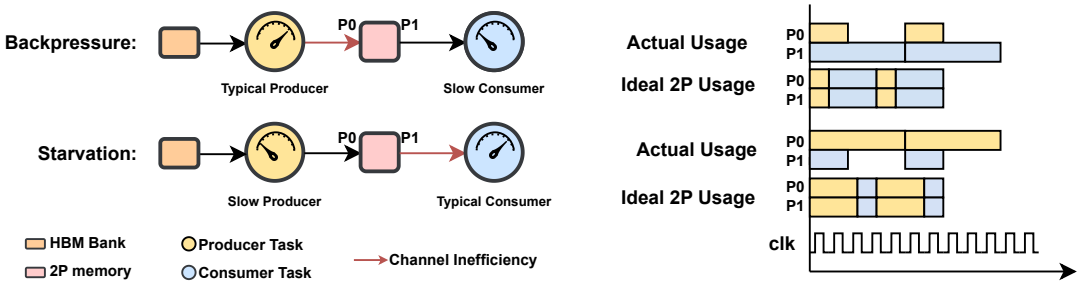



Fig. 4. Imbalanced pipelines do not use full dual-port bandwidth during backpressure or starvation

shared pool. Transactors should be able to request the allocation and deallocation of resources dynamically. With this feature, once a piece of data has been produced, it is immediately available to any other transactor by sharing just the address, much like sharing a pointer in a typical software programming context.

*2.2.3* ***Bandwidth Under-utilization in Imbalanced Pipelines.*** According to the SPSC model, each task must have well-defined interface channels (FIFO channels [14] or buffer channels [16]) and there must only ever be one producer and one consumer connected to this channel.

When splitting the producer-consumer pair connected via buffer channels into two regions, the tool allocates one memory-port each for the producer and consumer. But this is an inefficient implementation. Typically, when a ping-pong is set up for balanced producer and consumer tasks, both buffer ports are fully utilized - the producer writes through the first port, and the consumer reads from the second port. However, for cases like loading of sparse data, the time taken to produce the data is more than the time taken to consume it [24, 25]. For imbalanced pipelines where the consumer (or producer) has finished accessing all the required contents and the second port is not being utilized, the corresponding producer (or consumer) is still accessing data through a single port (Fig. 4). In such cases, the buffer interface often works at 50% of the available bandwidth. When an MPMC model scales the number of transactors, there are many more permutations of how different transactors can produce and consume data at arbitrary events, which increases the probability of having imbalanced pipelines and the complexity of profiling them.

The framework should provide buffer channels that are able to utilize the full bandwidth of the underlying dual-port memory at all times - providing single-port bandwidth to each if both the producer and consumer are accessing the buffer, while also providing dual-port bandwidth in case only one of them is active. This optimization lies at the crux of how dual-port memory channels are interfaced at the RTL level, and therefore must be kept opaque to the designer.

*2.2.4* ***Layout and Floorplanning.*** There are several considerations for layout and floorplanning when targeting modern multi-die FPGAs, which contain multiple Super Logic Regions (SLRs) that are connected via silicon interposers. For example, AMD-Xilinx Alveo U50 consists of two SLRs connected via LAGUNA crossing registers, which can connect combinational nets between two SLRs, albeit at a higher timing penalty. When implementing the shuffle design in Fig. 2b, PASTA wraps the buffer-consumer and the buffers as a single monolith, packing them all in the same SLR. This will result in a layout as shown in Fig. 5a. When either the consumers or buffers are scaled up, the monolith scales up, and the floorplanning algorithm infers a large task that may easily overspill a slot. Assume that in the shuffle design, scaling up the buffers leads to over-utilization of an FPGA slot. For such a case, the floorplanner will map some cells into a different slot, creating a legal floorplan, but sharply degrading frequency. Even with plenty of resources at disposal in the second slot, it becomes difficult to achieve high-frequency designs owing to long net delays on a few die-crossings.

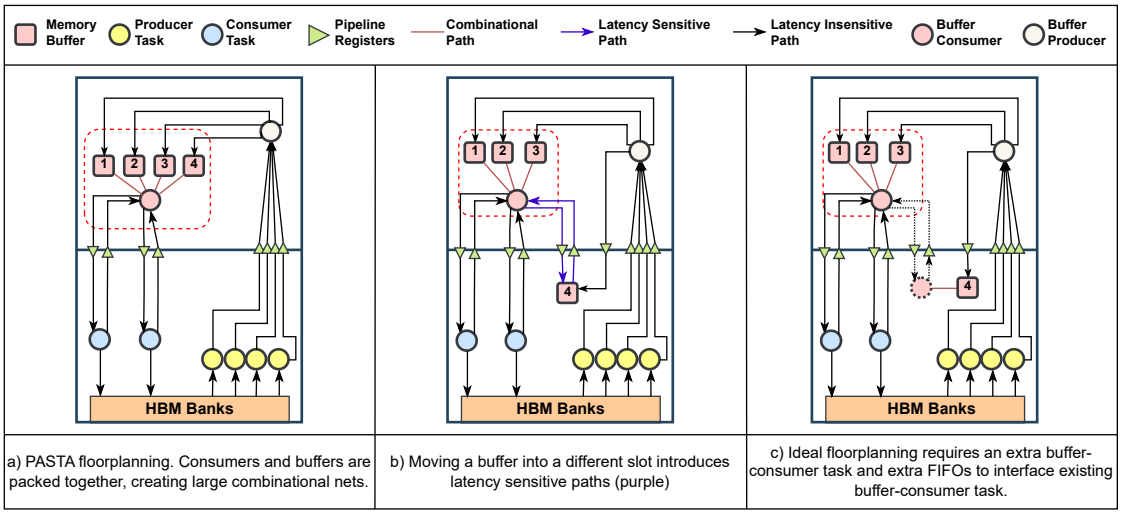



Fig. 5. MPMC systems face several layout constraints. Coarse-grained floorplanning may introduce latency-sensitive paths (purple). Ideal floorplanning requires addition of extra buffer-consumer and FIFOs (dotted figures). Moreover, original buffer-consumer needs relevant changes to perform access for buffer 4 specifically through the extra FIFOs.

Manual floorplanning efforts may try to move a buffer (or consumer) to a different slot. However, it comes with quirks of introducing latency sensitive paths. As described in Fig. 5b, moving even one buffer outside the parent wrapper introduces latency sensitive paths, analysis of which becomes complex when considering the previous considerations of automatic control coordination and access scheduling. The expert designer will plan the floorplan ahead of implementation, introduce an extra PASTA buffer channel in the design, and depend on the movement of the buffer along with its respective consumer to alleviate over-utilization. However, this also requires a redesign of the existing buffer-consumer task, since requests to the moved buffer will require forwarding to separate FIFO channels, including compensating for the introduced latency.

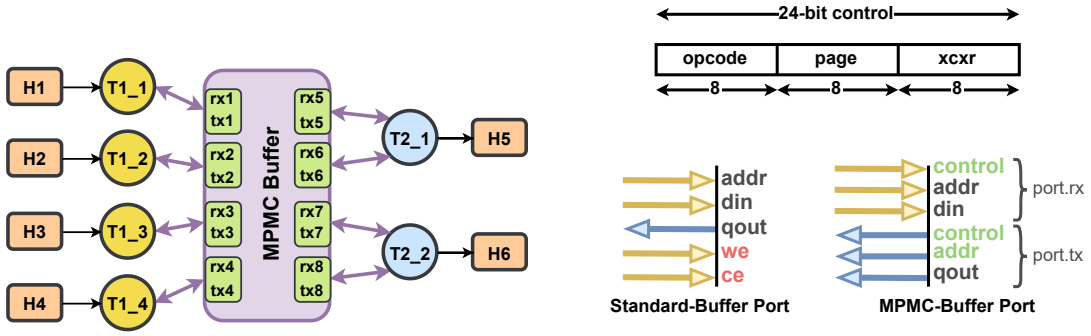
As such, there is no programmatic way of distributing shared resources among different slots. Floorplan effects are only seen in the backend PNR phase and caused primarily due to the nature of the underlying MPMC buffer. Therefore, the framework should devise a way to distribute buffers on the entire available fabric and keep these floorplanning optimizations completely opaque to the frontend designer.

*2.2.5* ***Summary.*** To summarize, there are four key challenges in pursuit of MPMC models.

1). Complexity of connectivity and orchestration of large task-level parallel systems
2). Preventing address conflicts and managing buffer reuse in shared scenarios
3). Under-utilization of buffer bandwidths in imbalanced pipelines
4). Layout and timing issues with fabric implementations

The expected resolutions to these challenges span a wide stack and, to the frontend programmer, should provide:

1). Templates and convenient APIs that reduce the complexity of orchestrating an MPMC TPS
2). Automatic inference and management of the underlying control coordination
3). Deeply optimized data movement at the RTL level
4). Automatic floorplanning optimizations to provide high clock-frequency implementations




(a) Shuffle kernel schematic using the MPMC buffer abstraction. (b) Port interface of standard buffer vs MPMC buffer

Fig. 6. Interface abstraction for MPMC buffers showing (a) a much simpler connectivity model; (b) detailed port implementation

## 3 MPMC Buffer Abstraction

In this section, we introduce the MPMC buffer abstraction according to the expectations of a designer described in the previous section. Gradually, we build the shuffle kernel application as shown in Fig. 6a. The application includes four instances of task *T1* that read data vectors from the HBM and write it into the MPMC buffer using one MPMC-buffer port (Fig. 6b) each, which are then read by 2 instances of task *T2* using 2 MPMC-buffer ports each, processed accordingly, and written out to HBM. The kernel includes four design-specific communication FIFOs, used to share only the index of the allocated page to the corresponding consumer, which have been omitted in the figure.

### 3.1 Reducing Orchestration Complexity

Based on the considerations discussed in Sec 2.2.1, each transactor shall connect to an MPMC buffer through a defined number of MPMC-buffer ports. Shown in Fig. 6b, each MPMC-buffer port is a pair of FIFOs that transfer control and data requests to and from the MPMC buffer. The *control* field is at maximum 24-bit wide and contains three fields.

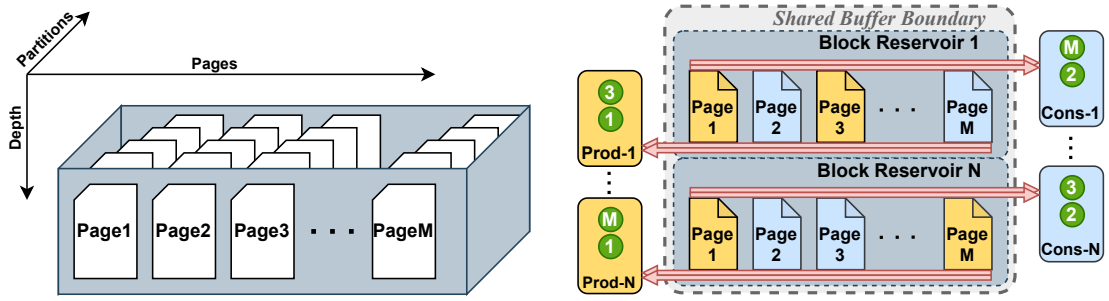
1). *xcxr*: 8-bit index to denote the target accessor-pair (and block-reservoir).

2). *page*: 8-bit index to denote the target page within the block-reservoir.

3). *opcode*: 4-bit command bits + 4-bit status bits to denote different types of requests and responses.

The actual sizes of the fields ultimately depend on the user's buffer configuration, but an upper bound of 24-bits stems from the fixed size of internal registers and counters. The *xcxr* and *page* fields together create a (maximum) 16-bit global address, which can index more memory-cores (units of on-chip memory composed of BRAM/URAM cells) than what are available in total on modern FPGAs. The index of the individual words and the data to be written or read is stored in the *addr* and *data* fields of the MPMC buffer port.

Essentially, the port is composed of two FIFOs - one for transmitting requests and another for receiving responses. The number of MPMC-buffer ports define the maximum bandwidth that can be allotted to a transactor while exposing the entire pool of shared memory in the buffer. This keeps the transactors simple and independent. Addition of more transactors to the MPMC model or increasing the size of the MPMC buffer do not interfere with the existing transactors and require no extra coordination or data FIFOs from the user. The only additional connections required are the optional, design-specific, point-to-point connections between the transactors used to share a specific address.




(a) Hierarchy of a block-reservoir of buffer pages. (b) Access hierarchy model for the MPMC buffer

Fig. 7. Proposed on-chip-memory backend of the MPMC buffer

### 3.2 Automatic Management of Coordination

The problem of automatically managing coordination can be split into three sub-problems. First, a well-defined hierarchy of access should be introduced that will serve as the global memory layout visible to each transactor. Second, a mutex scheme should be implemented based on the proposed access-hierarchy such that the transactors may express a read/write lock over a resource. This shall help the transactors prevent dependence and anti-dependence data hazards, by consuming only after the data is produced and producing new data only after the previous data is consumed. Third, a centralized book-keeping of dynamic de/allocations must be maintained. This shall help the transactors prevent inadvertent writes over each other's address when sharing a resource, without having to track each transactor's accesses.

*3.2.1 Memory Layout.* The MPMC buffer is arranged as multiple *block-reservoirs*, each containing multiple *pages*. Each *page* has one or more partitions and contains a number of words (*depth*) defined in the buffer declaration. This layout of a block-reservoir (or simply, block) is described in Fig. 7a. The MPMC buffer has four dimensions - blocks ($N$), pages ($M$), partitions ($P$), and depth ($D$). Each page has a unique index to generate the address of the individual words in the global context. Each block is interfaced using a pair of producer-consumer tasks shown in Fig. 7b. For the remainder of this paper, these producer-consumer tasks of a block will be termed *accessors* for distinction. All write requests to the block are handled by the producer, while all the read requests are handled by the consumer. Essentially, the accessors become the backend of the MPMC buffer architecture which serve the requests of the transactors.

*3.2.2 Access Hierarchy and Mutexes.* Since there are separate accessors for reads and writes, the responsibility of preventing data-hazards (dependence and anti-dependence) falls on the accessors of a block. Fig. 7b shows the schematic organization of the hierarchy inside the MPMC buffer. Each block along with its accessor-pair (producer and consumer) is composed of as many PASTA (SPPC) buffers as the number of pages within the block. To create a block of $M$ pages, $M$ PASTA buffers are used, with $M$ buffer-producers wrapped in a parent producer, and $M$ buffer-consumers wrapped in a parent consumer. These parent wrappers together are referred to as the *accessor-pair* of a block. Since there are two FIFOs in each buffer channel, there are $2M$ coordination FIFOs installed between the accessor-pair to spin the tokens for each underlying PASTA buffer. Therefore, each page is accessed based on an independent SPSC model, and can leverage the PASTA framework to create the backend for a single page, since it already encapsulates the control scheme required to coordinate the producer-consumer of a single resource. The parent wrappers have some extra logic to maintain the tokens for multiple pages, which is covered later in Sec 4.1.

Table 2. Description of new terminology introduced in context of the MPMC buffer on task-parallel systems

| Term | Description |
|---|---|
| Transactor | A user's task (either producer or consumer) connected to the MPMC buffer |
| Accessor | Any of the producer or consumer tasks of the SPSC backend of the MPMC buffer |
| Page | The smallest contiguous address-space that may be locked for writing or reading |
| Block-Reservoir | A collection of pages connected to the same pair of producer-consumer accessors |

For each page, we keep the number of sections (the number of unique tokens) to 1. This ensures that at any given time a page can only either be written, or read. Due to this organization, the maximum bandwidth of the MPMC buffer is directly dependent on the number of blocks. Each block can perform a simultaneous read and write (albeit from different pages) using the accessor-pair. Each accessor holds the token (mutex) for a page based on the *control.page* field in the request received from the transactor. Only when the transactor signals that the mutex not be retained after performing the current request does the accessor spin the token. This keeps the transactors disjoint from the mutex operation, while still providing control over it.

*3.2.3 Dynamic Memory Management.* With SPSC models, transferring data tiles on FPGA memory is essentially a copy operation, requiring memory duplication. In an MPMC model sharing the same global view of memory, indexing data becomes simple. Once a tile of data has been produced, it is immediately available to all other transactors, owing to the shared memory model. As discussed in Sec 2.2.2, MPMC buffer frameworks should leverage buffer reuse by dynamically allocating memory elements through convenient APIs. The granularity of this allocation is of concern, since it affects the latency incurred for this management. A survey over typical accelerator designs [10] shows that workloads rarely use data tiles smaller than 4KiB. This advises that buffers used to coordinate dataflow rarely demand fine-grained allocations. Hence, we proceed with providing static, coarse-grained allocations at the level of a *page*, which is the smallest allocable unit inside a block over which a transactor may express ownership. This functionality makes it possible for a transactor to request the allocation of a resource and share its address directly with other transactors, as if exchanging a pointer or reference. Technically, all on-chip memory used on the FPGA has to be statically allocated. The dynamic allocation is proposed as the dynamic allocation of indexes in a statically allocated pool of memory.

### 3.3 Example API Usage

```
using my_buffer =
  mpmcbuffer<
     uint32_t [D],   // word type and block depth
    partition<     // partitioning
      cyclic<P>>,  // strategy and factor
    memcore<BRAM>, // memcore type (BRAM/URAM)
    blocks<N>,     // number of block-reservoirs
    pages<M>       // optional override
  >;

void top(mmap<32> vec_H1 , mmap<32> vec_H2 , mmap<32> vec_H3 ,
         mmap<32> vec_H4 , mmap<32> vec_H5 , mmap<32> vec_H6 ) {
  my_buffer mbuf;
  task().invoke(T1_1 , vec_H1 , mpmc<my_buffer , 1>{mbuf}); // connect 1 MPMC buffer port
        .invoke(T1_2 , vec_H2 , mpmc<my_buffer , 1>{mbuf});
        .invoke(T1_3 , vec_H3 , mpmc<my_buffer , 1>{mbuf});
        .invoke(T1_4 , vec_H4 , mpmc<my_buffer , 1>{mbuf});
        .invoke(T2_1 , vec_H5 , mpmc<my_buffer , 2>{mbuf}); // connect 2 MPMC buffer ports
        .invoke(T2_2 , vec_H6 , mpmc<my_buffer , 2>{mbuf});
}
```

Listing 1. MPMC buffer declaration example

Table 2 lists the new terminology introduced by the MPMC buffer abstraction. Superficially, the MPMC buffer is analogous to a book with multiple chapters (blocks), each containing a number of pages. Each chapter is interfaced

via two accessors, allowing one read and one write access simultaneously, albeit on different pages. Transactors are provided allocations at the granularity of pages, which at a given time can only either be read or written to. We introduce 4 convenient APIs to interface the MPMC buffer. *do_read*, *do_write* provide features of memory I/O, while *allocate* and *free* provide a method to allocate and de-allocate resources dynamically.

Listing 1 describes the declaration of an MPMC buffer for the example in Fig. 6a. All *T1* tasks connect to the MPMC buffer using 1 port each, while *T2* connects using 2 ports. A user can configure the following options in the MPMC buffer declaration (lines 2-9).

1). *Data Type (W)* sets the type declaration of each element of an array in standard C++ syntax.
2). *Number of Partitions (P) & Scheme* selects the number of partitions per page and its partitioning strategy (block/-cyclic/complete), inferred the same way as what they mean in Vitis HLS' array partitioning.
3). *Memory-Core Type* selects whether to use BRAMs or URAMs for the implementation of pages.
4). *Number of Blocks (N)* configures the number of ports on the MPMC buffer.
5). *Block Depth (D)* configures the number of elements in a block. The optimal number of pages is calculated automatically by the tool based on $W$, $D$, and the memory-core type, described later in Sec 4.5. However, the user can enforce a fixed number of pages to fine tune the implementation.

```
1  void T1_1(mmap<32> vec,
2            mpmc<my_buffer, 1> mbuf,  // connect 1 port
3            stream< uint32_t > to_T2) {
4    key_t key = KEY_A;  uint32_t  D = vec[0];
5    uint32_t  start_addr = mbuf[0].allocate(D);
6    to_T2.write(start_addr);
7    to_T2.write(D);
8    for(int i = 0; i < D; i++) {
9      value = map(key, vec[i+1]);
10     mbuf[0]. do_write (
11       start_addr + i, // index of data
12       value,          // value to be written
13       (i!=D-1));      // whether the write mutex should be held after serving this request
14   }
15 }
```

Listing 2. MPMC buffer interface example for an instance of T1.

Listing 2 and 3 describe an example usage of the MPMC buffer for the example in Fig. 6a. T1 requests the allocation of a page, reads data from the HBM, and produces key-value pairs in its respective buffer. Then, T1 shares the index of the allocated page to T2, which accumulates all values written by T1 and streams it sequentially into the HBM. Ultimately, T2 requests deallocation of the pages. The frontend parser captures the buffer configuration and generates 1) MPMC-buffer related tasks in C++, and 2) custom backend buffers in RTL that are later merged with the synthesized RTL. Note that in the shuffle example, we make allocations in the *T1* function, and deallocate in *T2*.

Using the MPMC buffer abstraction, we propose PoCo - a framework that extends task parallel HLS programs to support multi-producer multi-consumer designs by using a shared MPMC buffer.

## 4 MPMC Buffer Implementation

A top-level schematic of PoCo is shown in Fig. 8, which is composed of multiple free-running tasks that help create the internal datapath. The fundamental tasks required to implement the MPMC-buffer abstraction are detailed further. Throughout all diagrams, the arrows in yellow represent the write datapath, those in blue represent the read datapath, green represents the control path, and red represents the SPSC buffer channels. The variables $N$ and $T$ refer to the number of blocks and transactors respectively.

```
void T2_1(mmap<32> vec,
          mpmc<my_buffer, 2> mbuf,  // connect 2 ports
          stream< uint32_t , 2> from_T1 ){
  // init variables
  key_t keys[NUM_KEYS] = {...};
   uint32_t  keys_counter[NUM_KEYS] = {0};
  // wait for T1 to share allocated indexes and size
   uint32_t  start_addr_a =  from_T1 [0].read();  uint32_t  start_addr_b =  from_T1 [1].read();
   uint32_t  D_a =  from_T1 [0].read();  uint32_t  D_b =  from_T1 [1].read();

  for(int i = addr; i < min(D_a, D_b); i++) {
    // get the outputs written by mapper T1_1
    keyvalue_a = mbuf[0]. do_read (
      start_addr_a + i,  // index
      (i!=D_a-1));       // whether the read mutex should be held
    // get the outputs written by mapper T1_2
    keyvalue_b = mbuf[1]. do_read (
      start_addr_b + i,  // index
      (i!=D_b-1));       // whether the read mutex should be held

    key_a = get_key(keyvalue_a);
    key_b = get_key(keyvalue_b);
    for(int k = 0; k < NUM_KEYS; k++) { // compare against all local keys and aggregate
      #pragma HLS unroll
      if(key_a == key_b) {              // merge both values if the keys are the same
        if(key_a == keys[k]) keys_counter[k] += get_value(keyvalue_a) + get_value(keyvalue_b);
      } else {                          // update separately
        if(key_a == keys[k]) keys_counter[k] += get_value(keyvalue_a);
        if(key_b == keys[k]) keys_counter[k] += get_value(keyvalue_b);
      }
    }
  }

  for(int i = addr; i < (D_a - min(D_a, D_b)); i++) { // read any remaining mappings by T1_1
    keyvalue_a = mbuf[0]. do_read (start_addr_a + i, (i!=D_a-1));
    key_a = get_key(keyvalue_a);
    for(int k = 0; k < NUM_KEYS; k++) {
      #pragma HLS unroll
      if(key_a == keys[k])
        keys_counter[k] += get_value(keyvalue_a);
    }
  }

  ... // read any remaining mappings by T1_2

  mbuf[0].free(start_addr_a);               // free pages
  mbuf[1].free(start_addr_b);
  for(int k = 0; k < NUM_KEYS; k++) {       // write to HBM
    vec[k] = keys_counter[k];
  }
}
```

Listing 3. MPMC buffer interface example for an instance of T2.

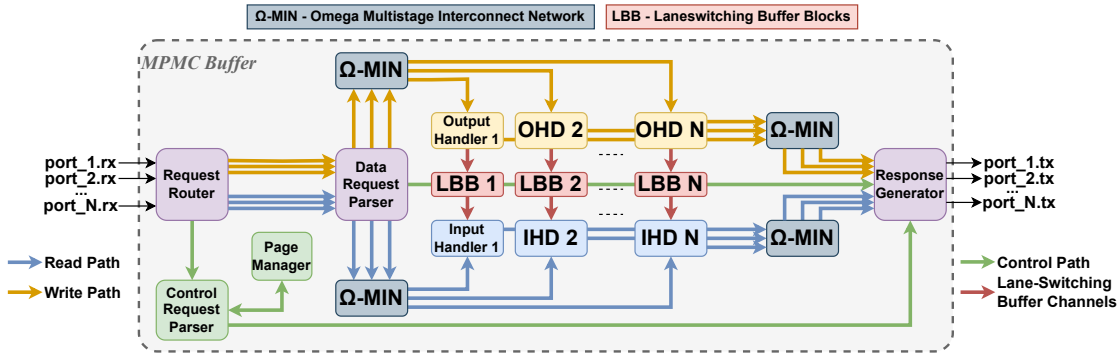



Fig. 8. MPMC buffer architecture in the PoCo framework to support general MPMC models

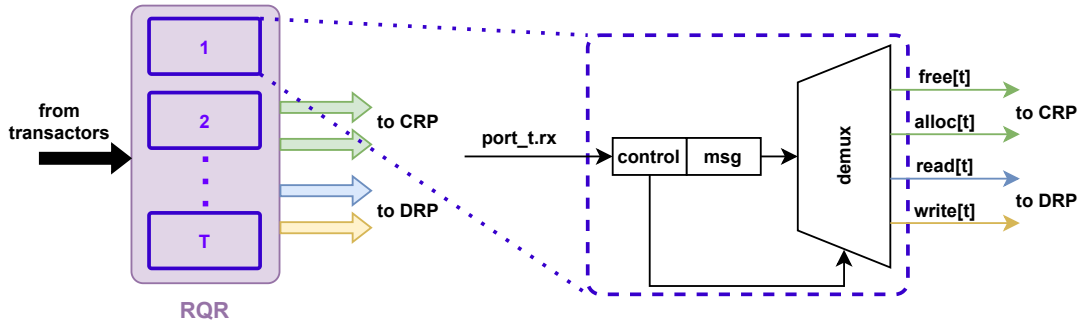



Fig. 9. Request Router (RQR) separates data requests from control requests and routes them to appropriate units

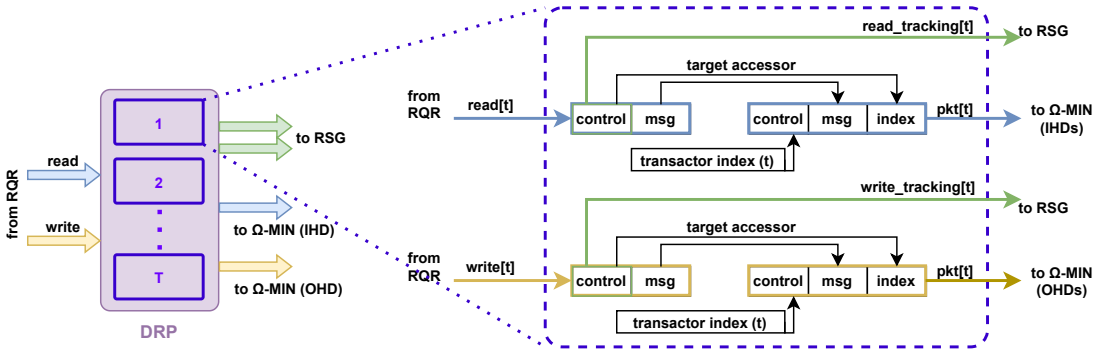



Fig. 10. Data Request Parser (DRP) parses the data request (read/write) and prepares the packet for internal routing.

### 4.1 Request Arbitration and Scheduling

*4.1.1 RQR (ReQuest Router).* All incoming requests are handled by the RQR (Fig. 9). Each request/response is composed of two parts - the *control*, and the *message*, which contains the address and data. RQR parses the requests from each port based on the *control* field, forwarding all control requests (de/alloc) to the Control Request Parser (CRP) and data requests (read/write) to the Data Request Parser (DRP). RQR is made up of $T$ identical units of disjoint tasks that route their requests independently.

*4.1.2 DRP (Data Request Parser).* The DRP (Fig. 10) parses the incoming request-type (read or write) and forwards them towards the accessors. It extracts the index of the target accessor and copies it into the routing tags required by the Ω-MINs. It also copies the current transactor's index into the control field so that the accessor may forward the response back to the correct transactor. For each data request, the DRP also sends some metadata to the RSG to maintain the order of requests received.

*4.1.3 RSG (Response Generator).* The RSG (Fig. 11) cleanly closes all transactions in order by collecting responses from the accessors. For read requests, it sends back the data from the Input Handler (IHD) to the transactor. For write requests, it forwards the acknowledgment status from the Output Handler (OHD). The RSG completes each type of request in order. However, it process reads, writes, free, and alloc responses in decreasing order of priority. All pending reads are responded first, followed by the writes, followed by deallocations, and then allocations. Some designs may want to keep the read and write requests strictly in order, for which the `read_tracking[t]` and `write_tracking[t]`

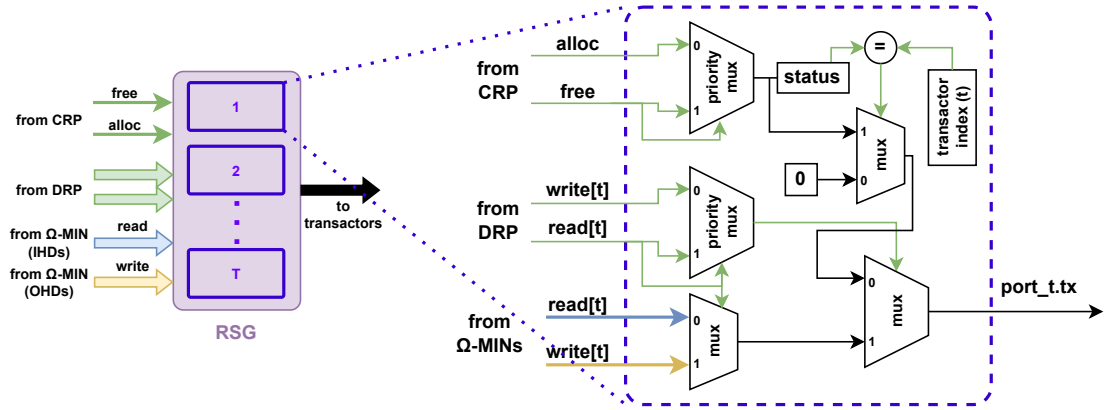


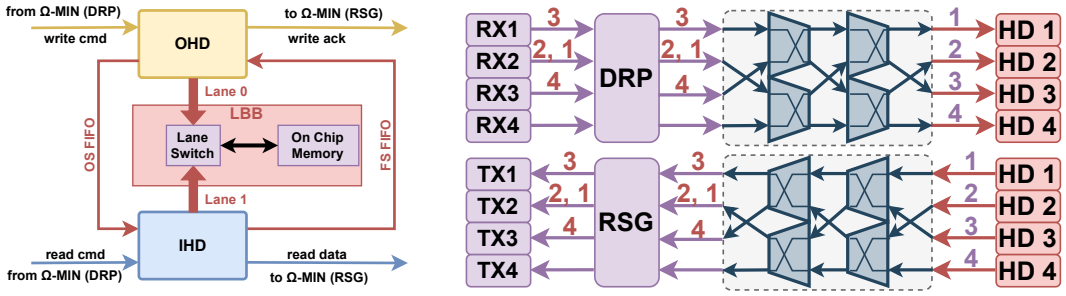

Fig. 11. Response Generator (RSG) cleanly closes all requests by sending responses to the corresponding transactor

FIFOs are merged as one FIFO in the DRP as well as the RSG. This is a compile time decision and can be changed by the designer as a configuration option. The involvement of the alloc and free queues will be explained in Sec 4.3.

*4.1.4 I/OHD (Input/Output HanDler).* The IHD and OHD are the accessor pair to a common reservoir block which is the *L*ane-switching *B*uffer *B*lock (LBB). LBBs are the backend SPSC buffers described later in Sec 4.4. The function of IHD and OHD are chiefly similar, except that one processes writes and the other processes reads. The I/OHD (Fig. 12a) parses the request for the address and acquires a mutex on the specific page decoded from the address. When the design is initialized after reset, the FS FIFO contains only one token. Upon a write request, the OHD reads this token and holds it until the request (from the transactor) signals a release of the mutex, after which, the token is written into the OS FIFO. Similarly, the IHD acquires mutex for read requests by reading the token from the OS FIFO and writing it out to the FS FIFO.

Hence, to process a data request, the I/OHD first selects the target page and then waits for the mutex to be acquired. This wait period depends on whether the corresponding accessor had held the token, or if the token was available for immediate consumption. Once the acquisition of the mutex is successful, it reads/writes the memory from/to a specific index. Similar to DRP, it extracts the transactor index from the control field (which was modified by DRP) into the routing tags required by the Ω-MINs. It also copies the current accessor's index into the control field so that the transactor may recognize the response. The IHD, OHD, and the corresponding block-reservoir are composed of several PASTA buffers, in that each page is composed of multiple memory-cores to support partitioning, with each memory-core providing one section of dual-port memory. Each buffer has their own coordination FIFOs spanning between the accessors.

The datapath between the DRP, I/OHDs, and RSG includes four omega-Multistage Interconnect Networks (Ω-MIN) [18]. Each Ω-MIN is composed of $\log_2(N)$ stages of $(N/2)$ 2×2-switches. The Ω-MIN routes the requests from each transactor to the relevant accessor based on the routing tags appended by DRP and I/OHDs, as shown in Fig. 12b. This network achieves a perfect shuffle [26] between adjacent stages, leading to efficient utilization of available bandwidth [22]. This rids the designer from worrying about the complexity of connections to and from different blocks. However, the primary inflation in latency is caused by the Ω-MIN, which leads to a latency of $\lceil(log_2(N)\rceil$ for each network, where $N$ is the total number of blocks. RQR, DRP, and RSG add one cycle each to the latency. The end-to-end access latency between sending a write/read request and receiving a response is called the datapath latency `DP_LATENCY` and




(a) I/OHD pair with its corresponding block (b) Internal datapath for request and response arbitration

Fig. 12. MPMC buffer backend based on SPSC accessors and the intermediary datapath using the Ω-MIN networks (gray)

depends only on the number of transactors ($T$) and accessors ($N$) in the buffer.

$$DP\_LATENCY = 3 + 2\lceil(log_2(max(T, N)))\rceil \tag{1}$$

However, this latency is incurred only for the first response, after which all responses are received back-to-back in a pipelined fashion.

This concludes the implementation of the basic datapath (RQR → DRP → Ω-MIN → I/OHDs → Ω-MIN → RSG), which provides the functionality of automatically coordinating and routing data requests, while keeping the transactors sufficiently separated from the underlying complexity.

### 4.2 Deadlock Prevention

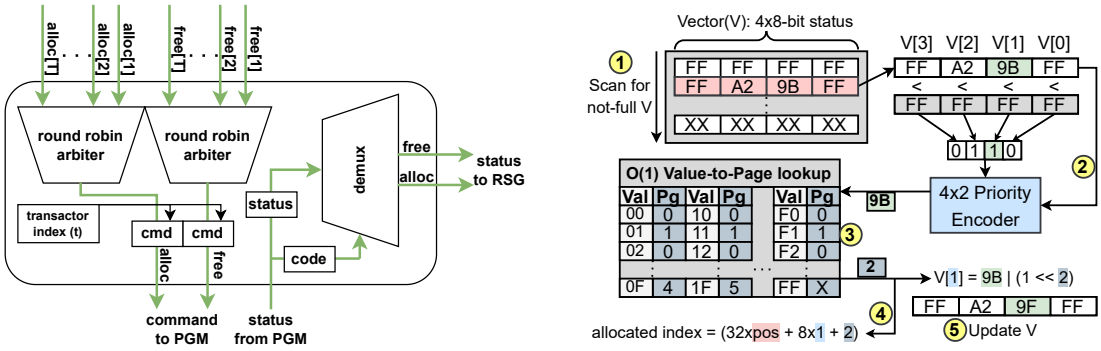
Since all read/write requests are handled automatically by the internal datapath, it is worthwhile to discuss possible deadlock scenarios when multiple transactors access multiple shared pages. For a system deadlock to occur, all of the following Coffman conditions [9] must be met simultaneously.

1). Mutual Exclusion: There is at least one resource (page) that cannot be accessed by two transactors simultaneously. Note in our design, the mutual exclusion happens at the page granularity.
2). Hold and Wait: At least one transactor holds one or more pages and is waiting for additional pages.
3). No Preemption: Only the transactor holding the page can release it.
4). Circular Wait: There exists a cycle of wait-dependencies, wherein one transactor waits for a page held by the other transactor in the dependency chain.

To avoid a deadlock, the user programmer should ensure that after accessing a page (and thus acquiring the lock) for a specific request type, the transactor (task) should release the mutex (i.e., releasing the lock) immediately before accessing any other pages. The release of the mutex shall be signalled by the transactor that plans to make the last write/read request to a page before accessing another: our `do_read()` and `do_write()` APIs have an argument to tell whether it should release the mutex. In this way, we break condition 2 (hold and wait) as well as condition 4 (circular wait), and thus avoid deadlocks.

### 4.3 Dynamic Memory Management

To enable dynamic buffer allocations and management, two modules are added.




(a) Control Request Parser

(b) Dynamic page allocation flow in PGM.

Fig. 13. Components of the control path - (a) CRP and (b) PGM, that manage the dynamic de/allocation of addresses

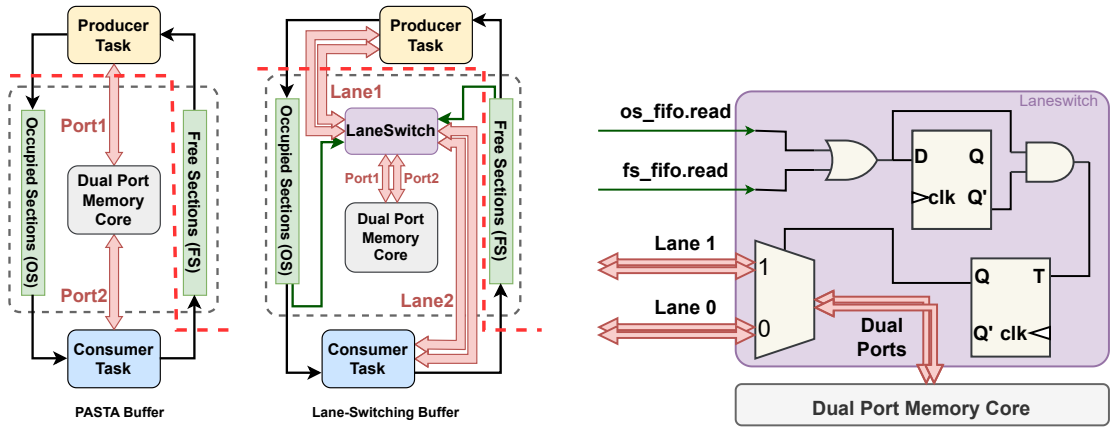
*4.3.1 CRP (Control Request Parser).* The CRP (Fig. 13a) processes two queues to manage de/allocation requests from the transactors. It always prioritizes deallocation requests over allocations. These requests are communicated to the Page Manager (PGM), the response from which is parsed and sent back to RSG, which notifies the relevant transactor about the reference. As mentioned already, the RSG processes any pending data requests on a transactor before responding with the control requests.

*4.3.2 PGM (PaGe Manager).* The PGM implements the book-keeping for de/allocating pages. As shown in Fig. 13b, allocation status of $P$ pages is maintained as a $(\lceil (P/32) \rceil)$-deep array of 32-bit vectors with a worst-case allocation latency of $(\lceil (P/32) \rceil + 2)$ cycles. Upon receiving a request, (1) the vectors are scanned to find which one is not full by (2) checking whether the 4-bit comparator input to the priority encoder is non-zero. (3) The byte-value corresponding to the output from the encoder is indexed in a 256-entry lookup table, which contains the position of the least significant zero bit in that value. This finds the next allocable page index in each 8-page cluster in O(1) time. (4) The bit position is then used to find the allocation index, which also (5) updates the original vector value. If all pages are already allocated, PGM waits for a de-allocation request before sending a response.

This functionality enables the transactors to request unique indexes dynamically. The CRP arbitrates control requests in a round-robin fashion from all transactors into two queues - one for all allocations and another for all deallocations. Therefore, the deallocations can be requested by any task, irrespective of who requested the allocation. This allows a producer task to allocate an index and leave it to the consumers to deallocate it at their own pace, after however many reads are required. The worst-case end-to-end latency for the control-path (RQR → CRP → PGM → CRP → RSG) depends only on the number of pages ($P$), and is given by

$$CP\_LATENCY = 5 + \lceil (P/32) \rceil \tag{2}$$

Typically, a design will see frequent de/allocations either when the global pool is being overutilized, or when the size of data tiles in the application is too variable. No direct solution can be offered for the former case, as the tasks must wait for pages to free up before accessing them. For the latter, the frequency of allocations can be reduced by tuning the page size. Multiple strategies of performing allocations are available in literature. PoCo implements a simple scheme since the allocations are a fixed constant in size. Should the user wish to change this scheme, changes only to the PGM module will be required, keeping all other modules as is. Moreover, if an MPMC design works primarily on deterministic indexes, the CRP and PGM can be entirely removed.




(a) Difference between PASTA buffers and Laneswitching buffers. (b) Laneswitch module that drives access to BRAM/URAM

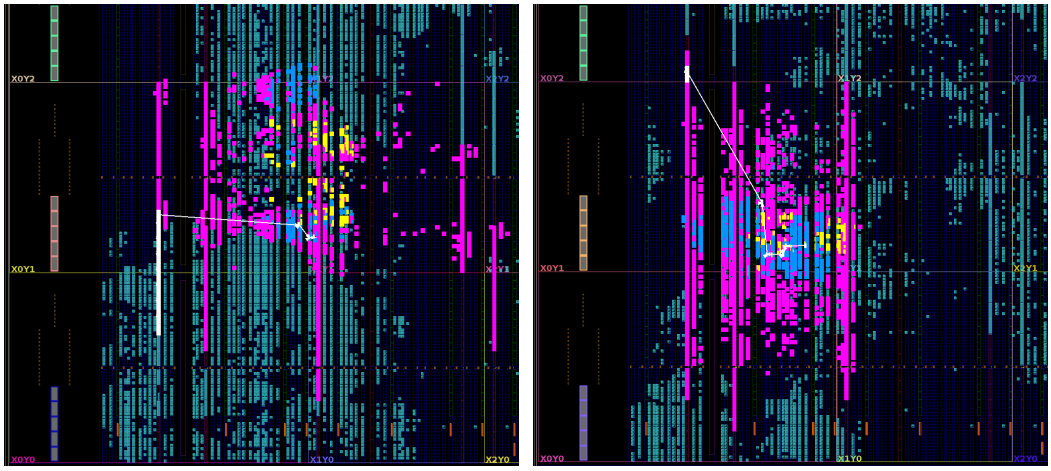
Fig. 14. (a) Comparison between PASTA [16] and our Lane-switching Buffers. Slot boundary is shown as dashed red line for cases when buffers are split in two slots; (b) Implementation of Laneswitch module for optimizing bandwidth in imbalanced pipelines.

### 4.4 Automatic Lane-Switching for Buffer Under-Utilization

When dual port memories are used with a producer and consumer, existing tools allocate one port for each, connecting the two with FIFOs that spin access-tokens to enable mutual coordination, shown in Fig. 14a. In Sec 2.2.3, we discussed how standard SPSC buffers are underutilized in imbalanced pipelines. To provide maximum utilization of both ports, the new abstraction should provide a dynamic switching of the ports to the producer and the consumer.

The feature of switching the ports between the producer and consumer is equivalent to switching traffic on two lanes. A *lane* is defined as the two ports of a dual-port memory that connect to the producer or the consumer. As shown in Fig. 14a, we design a RTL module called Lane-Switch (LS), which manages two diversions of a lane across a fork and acts as a driver for the underlying BRAM/URAM memory-core. The producer and the consumer lanes each connect to the LS module, which dynamically decides which lane to connect the memory ports to. At a given time, both ports of the memory must be diverted either to the producer or to the consumer. So, the laneswitched buffer spins just 1 token in the Free Sections (FS) and Occupied Sections (OS) FIFOs. The LS module monitors the producer/consumer activity from respective FS/OS FIFOs to preemptively check which task will hold the access-token and switches the memory ports to the corresponding lane for the next access. First, the read activity on both FIFOs is OR-ed and consequently AND-ed with its delayed, inverted signal to capture trigger events as a pulse. Next, this pulse drives a toggle flip-flop that selects the demultiplexer's output. The latency between capturing the read on either of the OS/FS FIFOs and switching the lane is 1 cycle. Since the producer/consumer tasks start accessing data one cycle after issuing the read, the LS module switches the memory ports to the appropriate lane just in time.

PoCo toolflow stitches this RTL block automatically between the tasks and the memories. When requiring cores with multiple partitions, each partition is stitched with its own LS module, wherein all LS modules inside a buffer are switched at the same clock cycle. The result is a memory-core abstraction, called the **L**ane-switching **B**uffer (LB) that enables optimal bandwidth utilization of SPSC buffers in imbalanced scenarios (discussed in Sec 2.2.3), while also ensuring tokenized accesses. We use a LB to implement a *page* in the MPMC buffer. Multiples of these pages are contained in a block (Fig. 7a), called the **L**ane-switching **B**uffer **B**lock (LBB), which the I/OHDs interface.



(a) 2 pages, 8192 elements each (237 MHz, 17.8% LUTs) (b) 8 pages, 2048 elements each (305 MHz, 19.5% LUTs)

Fig. 15. Deeper pages with a higher cascade height lead to a worse critical path (white)

### 4.5 Floorplanning Congeniality

In Sec 2.2.4, we discussed how typical monolith designs create long net delays on fabric, which degrades frequency even with plenty of fabric resources at disposal. Such issues are observed primarily as a limitation of the underlying fabric, but also caused by design decisions which disregard the underlying fabric. Therefore, PoCo implements two features to mitigate its impact on the frequency of the overall design, including the user's transactors.

*4.5.1 Independent Block Floorplanning.* For an MPMC buffer with $T$ ports (i.e., blocks), the RQR, DRP, and RSG are schematic abstractions that are made out of $T$ identical unit tasks with no combinational nets between the units. So each $t$-th unit can be placed anywhere on fabric irrespective of where the other units are placed. This offers high sparsity to the MPMC buffer due to the nature of its individual compositional tasks and allows reduction of PoCo specific constraints in the final placement phase. All inter-task connections except the LBB-to-IHD interfaces are latency-insensitive and can be interlaced with pipeline registers to meet timing at the cost of increased latency, making the MPMC buffer compatible with coarse-grained floorplanning. Each accessor, block-reservoir, and compositional task can be moved independent of the transactors, which in turn allows user-tasks and their associated buffers to scale independently of each other. For large buffers, the independent blocks with interface channels make it possible to move the design into different slots without impacting throughput. Even for small buffers which occupy only as much BRAMs as are available in a slot, the flexibility to move some nets into a different region helps with reducing the interconnect density. For example, a MPMC buffer design with 4 transactors and 4 accessors using 60% BRAM resources in a slot routes at 290 MHz (for a 333 MHz target) when the tasks are constrained to be co-located in the same slot. When the tasks are allowed to be placed independently of each other, the achieved frequency is 305 MHz.

*4.5.2 Timing Considerations.* The page depth (D) has an effect on the overall frequency of the MPMC buffer. Typical FPGA architectures place SRAM cells in separate columns, When using BRAMs, increasing the cascade-height parameter starts impacting the combinational read-paths to the corresponding IHD. PoCo can accommodate the same buffer size with shallower pages at the expense of marginal LUT overhead. This is equivalent to reducing the height of the page while instantiating more of them inside a block, leading to a higher number of shallow pages. Fig. 15 shows a design

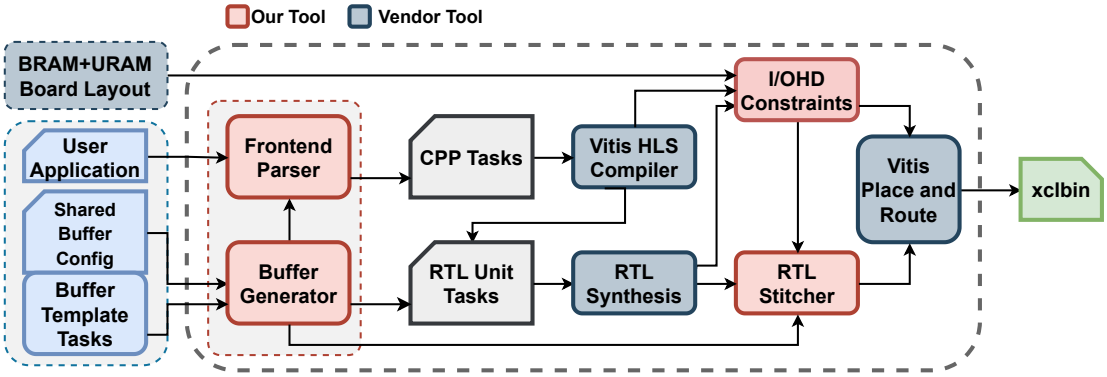



Fig. 16. Overview of PoCo toolflow.

with 2 partitions and 4 blocks, with 128 bit data using 40% BRAMs in a slot, which can only achieve 237 MHz. The second configuration (15b) with 8 shallow pages routes at 305 MHz with 1.7% more LUTs than the first configuration (15a) with 2 deep pages. This is further evaluated in Sec 6 (see Fig. 18b). Pages implemented with URAM cores are deeper, but incur a higher latency than those with BRAM cores. For a cascade-height of $H$, PoCo configures the latency for URAM access to $H/2+2$. This allows a pipeline on every second URAM, resulting in 1) good timing and 2) placement flexibility, since not all URAMs need to be placed in the same cascade column [27].

## 5 Overview of PoCo Framework

### 5.1 PoCo Toolflow Overview

Fig. 16 shows our complete automation flow which is built based on the PASTA toolflow. The user writes their program in a task-parallel HLS model. First, the frontend parser extracts all metadata related to the tasks and their interfaces (FIFOs, SPSC and MPMC buffers) and constructs a task graph. Template tasks and Lane-switching buffers are created according to the captured MPMC-buffer configuration. For user tasks that participate in the MPMC model, the definitions are modified to interface with the RQR and RSG modules using the port's TX and RX FIFOs respectively. Then, for all tasks, the definitions are transformed to generate the appropriate FIFO/buffer interfaces. Second, all tasks are compiled to RTL via Vitis HLS and further synthesized in Vivado to get accurate resource utilization reports. Third, the reports are used by a coarse-grained floorplanner (built on top of AutoBridge [15]) to find a mapping of tasks onto FPGA slots based on the board layout. Interfaces crossing the slots are pipelined in this step. Then, using the resource utilization reports, a set of constraints for the I/OHD modules and corresponding LBBs are generated. Fourth, the backend memory employed for the MPMC-buffer is merged with the tasks (in RTL). The final design is packed as a XO file. Finally, Vivado performs placement and routing on the XO file using the constraints to generate the FPGA bitstream.

### 5.2 Frontend Optimizations

Listing 1, 2, and 3 already describe the declaration and APIs of an MPMC buffer. Due to the buffer's internal pipeline, the response for the reads arrive a few cycles after the request is sent. Listing 4 shows an example call to the `do_read` API inside a loop. In each iteration, a distinct output is created using `value` after it is read. To send and receive the requests and responses respectively on the individual TX and RX FIFOs of the MPMC buffer-port, a source-to-source transformation is needed. A naive implementation of the transform is shown in Listing 5. Once the request is sent, the

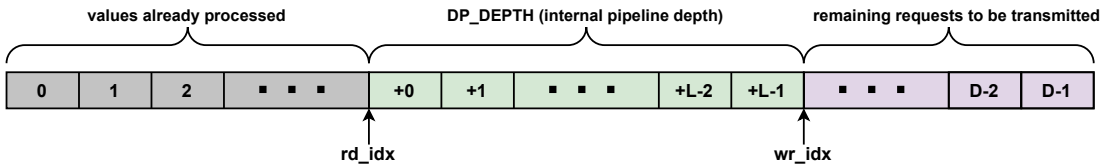



Fig. 17. Each MPMC buffer request traverses through a pipe of latency $L = DP\_DEPTH$ (Eq.1) before becoming a response.

variable `value` is updated only when the response is received from the RX FIFO of the MPMC buffer port, which suffers the latency of transferring data through the internal pipeline the MPMC buffer.

```
for(int i = addr; i < D; i++) {
  // blocking request to read one value from buffer
  sb_rsp_t value = mbuf[0].do_read(i, (i!=D-1));
  uint32_t result = process(value);
  outputstream.write(result);
}
```

Listing 4. Sample read loop

```
sb_req_t gen_read_req(uint32_t addr, bool mutex) {
    sb_req_t req = {0};
    req.control = gen_control(addr, mutex); // get the control field based on the address and mutex
    req.addr = gen_addr(addr);              // get the address to be accessed inside the page
    return req;
}

for(int i = addr; i < D; i++) {
  mbuf[0].tx.write(gen_read_req(i, (i!=D-1)));   // TX FIFO request
  value = mbuf[0].rx.read();                     // RX FIFO response
  result = process(value);
  outputstream.write(result);
}
```

Listing 5. Naive transformation incurs a fixed latency for each successive response due to the latency of the internal pipeline

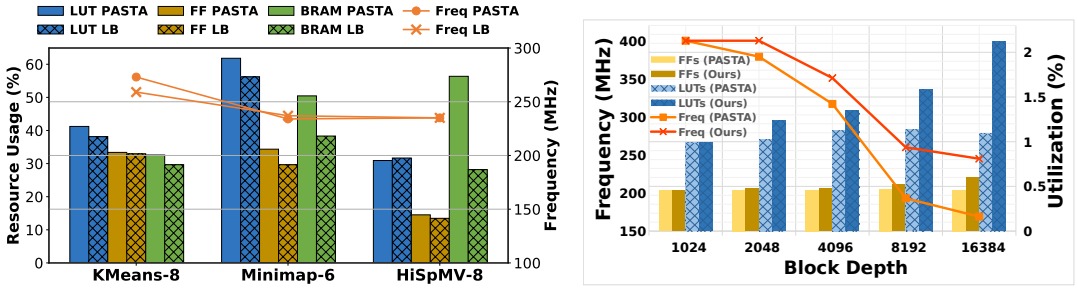
In the interest of performance, tasks cannot afford to wait for the response for each read request before issuing the next one. Therefore, a source to source transformation is applied to the loops involving `do_read` APIs such that the task shall not wait for the response of each request, which converts the code in Listing 4 to that in Listing 6. The transformation first captures the induction variable and bounds of the loop. Then, it makes two concurrent loops - one for the TX FIFO and another for RX FIFO with separate induction variables `write_idx` and `read_idx` respectively, bound with the same loop bound $D$. These two loops are independent of each other and synthesize to operate in parallel. Although the two loops are semantically independent, the TX FIFO loop depends functionally on the RX FIFO loops through the depth of the internal datapth of the MPMC buffer. If the distance between the `write_idx` and `read_idx` becomes greater than the depth of the internal datapath (1), it simply means that a backpressure is being observed at the RX port, in which case, the TX FIFO will assert its full signal at some point and backpressure the requests. Fig. 17 shows how $D$ requests and responses are handled in a pipelined fashion after applying the source-to-source transformations.

### 5.3 Backend Placement Optimizations

To improve the timing closure for task-parallel programs using the MPMC buffer model, we extend the PASTA [17] backend and discuss only the optimizations specific to the MPMC buffer. As shown in Fig. 16, based on the data extracted by the fronted, the task-utilization, SLR-crossing information, and BRAM/URAM column arrangement on the board layout are used to generate constraints for the I/OHD modules. First, custom Pblocks divide the entire fabric into several partitions with similar BRAMs/URAMs. Second, the synthesis reports are used to find the resource requirements of the

```
int wr_idx=0, rd_idx=0;
for(int wr_idx=0; wr_idx < D;) {      // maintain write pointer for transmitted requests
  if(!mbuf[0].tx.full()) {
    mbuf[0].tx.write(gen_read_req(wr_idx, (wr_idx!=TILE-1)));
    wr_idx++;
  }
}
for(int rd_idx=0; (rd_idx < D); ) {  // maintain read pointer for received responses
  if(!mbuf[0].rx.empty()) {             // if there are valid values in the pipe
    value = mbuf[0].rx.read();
    outputstream.write(process(value)); // process the loop normally
    rd_idx++;
  }
}
```

Listing 6. PoCo's source-to-source transform of Listing 4




(a) Evaluation of lane-switching buffers vs. PASTA [17]. (b) Impact of scaling block-depth in an MPMC buffer.

Fig. 18. Experiments involving PoCo's backend optimizations

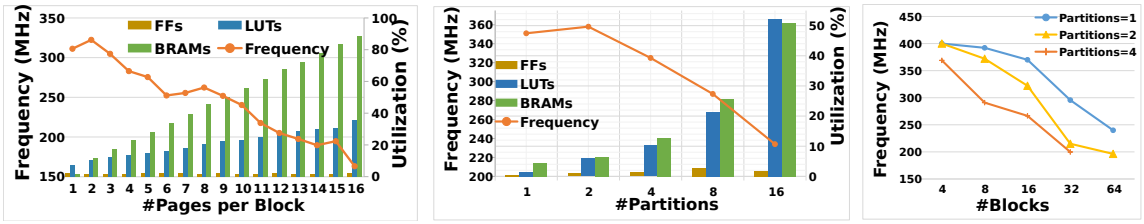
IHD along with its block-reservoir, since they are the only necessary combinational unions - disjoint modules that have combinational nets in between them. Third, the tool calculates how many Pblocks are required by a combinational union, and maps one or more unions to one or more Pblocks based on their resource usage. Finally, constraints are created based on this mapping, effectively distributing the unions across the entire fabric.

The MPMC buffer is generic, since each module can be fine tuned without impacting the overall buffer abstraction. For example, the LBB can be swapped out for a different abstraction of memory. If the application requires broadcasting, the RSG can be updated to notify each transactor with the data read from a single accessor. If the dynamic allocations are not needed, the CRP and PGM can be entirely removed.

## 6 Experiments and Evaluation

### 6.1 Lane-Switching Buffer

We use Vitis and Vitis HLS 2023.2, XRT 2023.2, and open-source PASTA version 0.0.20240104.2 for all our experiments. First, we test the lane-switching buffer (LB) alone to highlight its effectiveness in common SPSC workloads. Three benchmarks are implemented using both PASTA buffer and our lane-switching buffer: two workloads (KMeans, Minimap) from the Rodinia-HLS benchmark suite [10] and a HiSpMV accelerator [24] with 8 PEs on Alveo U280 FPGA, summarized in Fig. 18a. As discussed in Sec 2.2.3, bandwidth constrained designs perform BRAM duplications to provide the full dual-port bandwidth to the consumer task. For KMeans and Minimap, compared to PASTA, the LB implementation reduces the BRAM utilization by 3% and 13% respectively. HiSpMV uses buffers only for loading dense vectors from the HBM. Therefore, all its duplicated PASTA buffers can be replaced with LBs, offering a 50% reduction in BRAM utilization compared to PASTA while achieving similar performance. The resource overhead of the LS modules is proportional to




(a) Scaling number of pages in a block; 4 blocks, 8 partitions per page

(b) Scaling number of partitions; 4 blocks, 4 pages per block. X axis: $log_2$ scale

(c) Scaling number of blocks; 1 page per block. X axis: $log_2$ scale

Fig. 19. Scaling different dimensions of the MPMC buffer. Data-width = 128 bits. Post-route target frequency = 400 MHz.

the dimensions of the LB, but still negligible. For example, a LB with 8 partitions of 64-bit wide dual-port memories (1024-bit wide write and read) uses LS blocks worth only 98 LUTs and 33 FFs.

### 6.2 MPMC Buffer Scalability in PoCo

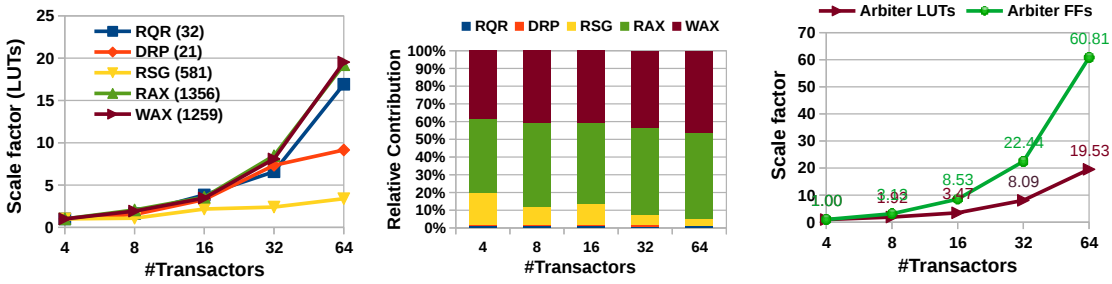
To verify the utility of the MPMC buffer, we implement a simple kernel which consists of two tasks to test the MPMC buffer bandwidth (Fig. 22). *Task1* reads values from one HBM channel and writes them into the MPMC buffer on all ports in parallel, *Task2* reads all values and writes them out to the HBM, where the host verifies them. This arrangement ensures minimal impact of non-buffer tasks on utilization. All metrics are post-route on Alveo U50 FPGA. The data-type of each element is fixed to 128 bits for all tests.

*6.2.1* ***Scaling Number of Pages (*** *M* ***).*** This configuration uses 4 blocks, 8 partitions per page, summarized in Fig. 19a. Each page is implemented using a fixed arrangement of 2 cascaded 36Kb-BRAM blocks. With a linear increase in the number of pages, we observe a linear increase in the accessors' LUT footprint and BRAM usage, and a graceful degradation in the frequency. The FF utilization averages 1.86% for the entire sweep.

*6.2.2* ***Scaling Number of Partitions (*** *P* ***).*** This configuration uses 4 blocks, 4 pages per block, summarized in Fig. 19b. As expected, the BRAM and LUT utilization scales linearly with $P$. By default, the floorplanner divides each SLR into two segments, inserting pipeline registers for logic between the two segments as it precludes SLR-wide combinational paths. When $P$ increases, the density of BRAM and the possibility of conflicted nets increases. When $P$ is large, it creates the floorplan without segmenting the SLRs. This saves many registers that would otherwise be used in pipelining the channels in between the segments, which is observed as a drop in FF utilization for P=16.

*6.2.3* ***Scaling Number of Blocks (*** *N* ***).*** This configuration uses 1 page per block. Scaling blocks also scales the number of ports (concurrency). The buffer-depth ($D$) is fixed to 1024 to mitigate its impact on resource utilization or routing effects. We perform three sweeps with different $P$ (Fig. 19c). When trying to use many ports with an increased partition size, the bottleneck to the MPMC model is the limited LAGUNA registers present between the slots. Consider the implementation with N=64 and P=4. The control signals (*opcode*, *xcxr*, *page*) are $8 + \log_2(64) + \log_2(0)$=14-bits wide, the address is $\log_2(D)$ =10-bits wide, and the data is 128-bits wide for each port. The width of each port is therefore $2 \times (128 + 24) \times P$ =1216-bits. When accounting for N=64, there are not enough SLR crossings available.

*6.2.4* ***Scaling Buffer Depth (*** *D* ***).*** This configuration uses 4 blocks, 1 partition per page, summarized in Fig. 18b. This sweep aims to check how the frequency degrades with an increasing number of RAMs. PoCo automatically chooses the optimal dimensions of blocks based on the cascade-height calculated from the buffer depth. For example, when




(a) Breakdown of normalized LUT-scaling (b) Breakdown of relative LUT footprint (c) LUT&FF scaling of all arbiters combined

Fig. 20. Resource-usage breakdown of LUTs and FFs used by the buffer's compositional tasks. RAX = Read Arbiter Switch-fabric, WAX = Write Arbiter Switch-fabric. Each arbiter switch-fabric includes 2 Ω-MINs on the corresponding path (refer to Fig. 8).

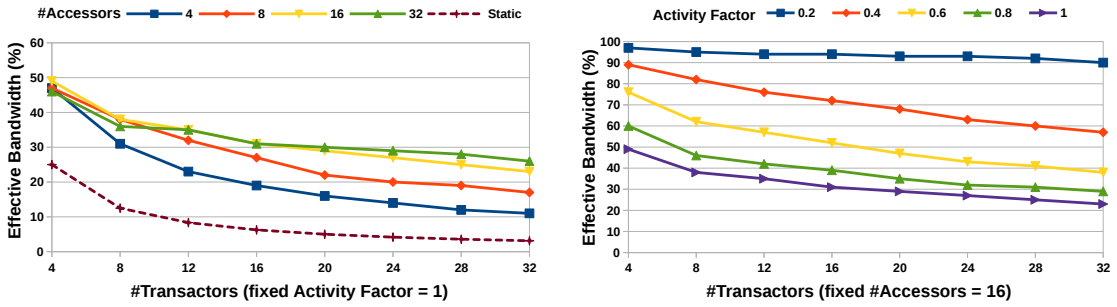
using 4096 elements, 8×36Kb-BRAMs need to be in cascade. PoCo reconfigures the block-reservoirs into a 4×2 matrix (4 pages with 2 BRAMs in cascade) at a fourth of the depth at a marginal cost of LUT resources. This provides ≃1.3× improvement in frequency compared to default cascading.

*6.2.5* ***Scalability of Buffer Tasks***. Fig. 20a shows the breakdown of LUT resource usage for the compositional modules of the datapath. The modules on the control path (CRP and PGM) do not scale with the increase in transactors, and are hence omitted from this analysis. The trend of the LUT utilization increases linearly (X axis: $log_2$ scale) for all modules, although the slope differs. The legend also mentions the number of LUTs at the normalized value (4 transactors and 4 blocks). For example, DRP's LUT utilization at 64 transactors is ≃ 9× that at 4 transactors. Fig. 20b shows the relative footprint of each module. RQR and DRP are small modules since they simply parse a field and perform some bit-manipulations within the request before forwarding it. The priority selection of RSG involves more complexity in dynamically selecting which input queue (alloc, free, read, write) gets forwarded to the output (refer to Fig. 11), which inserts several *sparsemux* instances that are dependent on the width of the data bus. The read and write arbiter switch fabrics (RAX, WAX) dominate the LUT usage due to the completely generic connectivity they offer between transactors and accessors. Fig. 20c shows the trend for increasing LUTs and FFs for the entire arbiter fabric combined. The FF utilization of the RQR, DRP, and RSG are negligible and hence omitted from this analysis. By increasing the number of partitions and the width of the data bus, the baseline FF utilization increases. Even though the FF utilization scales much more aggressively than the LUT utilization, it sits at a modest ≃ 1% utilization of the total FF resources. The real limitation to the scalability of the MPMC buffer on current FPGA architectures is not the FF utilization, rather the limited die-crossing registers between the SLRs.

### 6.3 Latency Overhead and Effective Bandwidth under Heavy Workloads

We measure the latency for different request as the number of cycles between latching of the request on the input of the RQR and that of receiving the response on the output of the RSG.

*6.3.1* *Latency.* The minimum datapath-latency is given by Eq. 1, wherein each Ω-MIN contributes $\lceil(log_2(max(T, N)))\rceil$ cycles on the path. In actual implementation, the minimum datapath latency may be higher depending on where the tasks are placed in fabric and how many extra pipeline registers have been added during the coarse-grained floorplanning stage. Assuming that free pages are available, the worst-case control-path latency for allocations is given by Eq. 2. In actual implementations, it is dependent on how many control requests as well as data requests are in flight.




(a) EB for fixed activity over variable accessors.

(b) EB for 16 accessors over variable activity rates.

Fig. 21. Decay of Effective Bandwidth (EB) for different accessors and activity factors.

Moreover, unlike the datapath, the control path is affected by data requests too because the RSG prioritizes responding to pending data requests of a transactor before the control requests.

The overall latency for each packet is quite dynamic, depends on the number of requests in flight, and may overlap with waiting and arbitration of other requests, making it complex to profile exactly in active scenarios.

*6.3.2 Effective Bandwidth.* To analyze the effective bandwidth that the fabric can provide, we conduct a new test, wherein our setup includes a variable number of transactors ($T$), each generating 1024 requests containing random addresses to the buffer. The setup assumes that all pages are already allocated to prevent control overhead. We introduce a new factor for scaling the input activity ($A$), which decides the probability that a transactor will send a packet at a given cycle. For example, with $A = 1$, each transactor on the network will issue one packet every cycle (i.e., with a probability of 1), injecting $T \times 1024$ packets. With $A = 0.5$, each transactor on the network will issue one packet per cycle with a probability of 0.5, while still injecting $T \times 1024$ packets in total. The effective bandwidth is then calculated based on the average number of cycles wasted in arbitrating conflicts for each packet. A 50% effective bandwidth means that during the total transfer, enough conflicts were faced so as to reduce the bandwidth to $T/2$ packets for each cycle that a new packet was injected. The setup is done in two sweeps - first, over the number of accessor-pairs ($N$ blocks) considering $A = 1$, and second, over $A$ considering $N = 16$. The Ω-MIN provides only one path between a pair of end-points (transactor/accessor). Therefore, although it is easier to route, situations where one path (one transactor-accessor route) is excessively used is bound to see conflicts and limit the performance of other ongoing transactions. Therefore, $A = 1$ is the worst situation for the fabric, as the general recommendation for such dense access would be to have a dedicated SPSC channel between that transactor-accessor pair. Fig. 21a shows that even in the worst case, the effective bandwidth is ≃20% better than static scheduling for the same number of transactors. Fig. 21b shows that for a fixed number of accessors, decreasing the activity factor improves the effective bandwidth roofline of the overall network, since the probability of arbitration conflicts reduce. To summarize, Ω-MINs achieve best performance with moderate activity on all paths, making them a good use-case for dynamic and uncharacterizable traffic.

### 6.4 Case Studies Using PoCo

We discuss three case studies, highlighting benefits of our PoCo framework. Task-graphs and results for all case studies are summarized in Fig. 22 and Table 3. PoCo's dynamic allocation of resources is proposed as the dynamic allocation of indexes in a statically allocated pool of memory. The reported BRAM usage is the static allocation size that is reserved while implementing a specific MPMC buffer design on fabric. We chose applications that would require the same

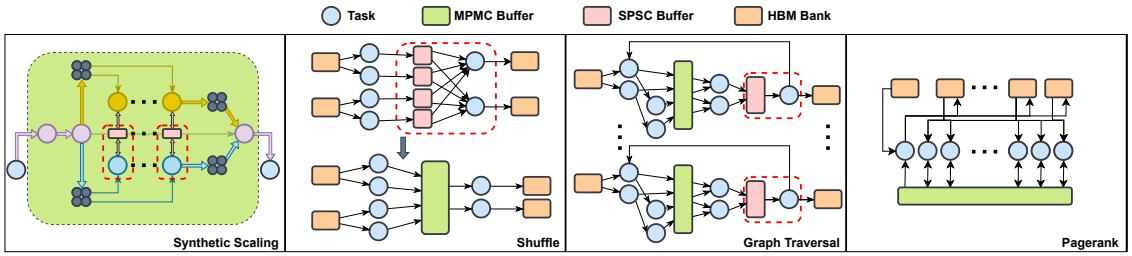



Fig. 22. Task graphs for different evaluations and case studies.

Table 3. Post-implementation results for case studies comparing PoCo framework and other tools.

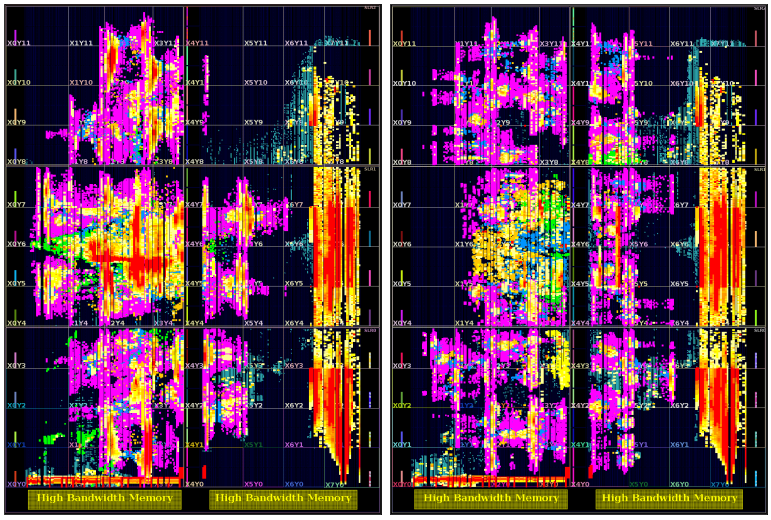
| Application (FPGA) | Tool | LUTs (%) | FFs (%) | BRAM (%) | URAM (%) | DSP (%) | Freq (MHz) |
|---|---|---|---|---|---|---|---|
| Shuffle (U50) | Vitis 2023.2* | 25.81 | 16.76 | 63.43 | 0 | 15.75 | failed |
| | TAPA [14] | 26.03 | 18.96 | 63.50 | 0 | 15.75 | 52 |
| | PASTA [17] | 26.31 | 19.38 | 63.50 | 0 | 15.75 | 144 |
| | PoCo | 29.10 | 20.04 | 63.50 | 0 | 15.75 | 218 |
| GraphTraversal (U280) | AutoBridge [15] | 25.83 | 13.07 | 64.49 | 2.08 | ~0 | 168 |
| | PoCo | 26.16 | 14.87 | 64.49 | 2.08 | ~0 | 232 |
| Pagerank (U280) | ReGraph [7] | 44.76 | 39.30 | 29.22 | 93.33 | ~0 | 214 |
| | PoCo | 57.92 | 45.66 | 32.40 | 93.33 | ~0 | 191 |

* Post-synthesis results.

amount of static memory so that a fair comparison can be made and the frequency improvements for PoCo are not influenced by a favourable reduction of on-chip memory usage. All case studies run on the actual FPGA boards and all metrics are post-route.

*6.4.1* ***Shuffle.*** Shown in Fig. 22, the input contains a set of text strings $S$, partitioned as subsets of strings $S_i$ into the $i$-th HBM bank connected to the mappers. $K$ map tasks for each subset $S_i$ are implemented. Each mapper works on a unique $key_k$ where $k \in K$. For each string $s_j \in S_i$, the $k$-th mapper calculates a hash based on the number of occurrences of $key_k$ in the string, storing the key-value pairs in the shared buffer. We set the depth of the buffers to 16,384, which uses 63.5% of available BRAMs on Alveo U50 FPGA, i.e., 127% BRAM utilization of one SLR. The sorter tasks read the key-value pairs from the shared buffer, and if the key matches, accumulate and store it in the HBM. Vitis fails to generate a physical layout for this design. An implementation with TAPA only achieves a frequency of 52 MHz. Without our coarse-grained constraints for MPMC buffers, PASTA (with our MPMC frontend support) only achieves a frequency of 144 MHz. PoCo distributes the MPMC buffer-blocks to different pblocks, achieving 218 MHz frequency, 4.2× better than TAPA and 1.5× better than PASTA. The actual speedup compared to PASTA is 1.66× (higher than the 1.5× frequency improvement), due to the dual-port bandwidth, consuming the buffers faster. Note our PASTA implementation includes buffer producers that are dynamically scheduled and handle arbitration as well, since we wanted to keep the designs as behaviourally close as possible.

*6.4.2* ***Graph Traversal.*** This application performs feature extraction from graph data. It involves multiple PEs that visit each neighbor of a node sequentially, extracting the data stored in the node structure. When the neighbor-list of a node exceeds the size that can fit in a page, the PE dynamically requests a new one, and frees it when all nodes on that page have been processed. The MPMC model has 16 blocks, with 256 pages each, using 64% of BRAMs on Alveo U280. The routing directive was set to AggressiveExplore and the placement directive was set to EarlyBlockPlacement. The baseline backend Autobridge [15] does a good job at placement. However, it packs the middle-left slot while trying to




(a) AutoBridge: 169 MHz (b) PoCo: 232 MHz

Fig. 23. Device view implementation of interconnect density for Graph-traversal application on Alveo U280 using ~64% BRAMs. Comparison between (a) Autobridge [15] and (b) PoCo's extra constraints, which move some I/OHDs to the top-right. LBBs: pink; IHDs: blue; OHDs: yellow; control path: green.

minimize SLR crossings, increasing the interconnect density (Fig. 23a), leading to a frequency of 169MHz. Our PoCo framework knows that the accessors are latency insensitive, and the tool is able to move 4 groups of I/OHDs to the top-right slot without changing any design functionality, albeit increasing the number of slot crossings. This achieves a frequency of 232MHz and leads to a significant reduction in interconnect density, producing no hotspots (Fig. 23b).

*6.4.3* ***Page Rank.*** Finally, we use the MPMC buffer to implement similar pipelines as described in ReGraph [7] on Alveo U280. The data movement between the *scatter-gather* modules feature a similar design requirement that the MPMC buffer provides. The *scatter* PEs are implemented as the transactors that connect to the shared-buffer, while the data-router and gather PEs are implemented through the DRP-to-OHD datapath. For this application, we omit the CRP and PGM blocks since the transactors have a fixed location to write to. The transactors send the update tuples to the OHDs based on the vertices of the graph. The OHDs accumulate it into the connected LBBs. The accumulated data is then read out into a separate HBM bank. The final implementation utilizes 93% URAMs. The frequency achieved is 193 MHz, 21 MHz below ReGraph [7]. Since the buffer holds a limited number of values at once, the pipeline has to be looped several times. The latency of transferring ownership of the page can be as long as 12 cycles based on where the scatter and merger tasks are placed on the fabric. This latency is incurred for every mutex-switch which adds up over many repetitions of the kernel. However, ReGraph is specialized for graph processing, while PoCo is a generic MPMC framework for all applications.

### 6.5 Measurement of Programming Convenience

In Table 4, we share the number of lines of code (LOC) that PoCo uses compared to PASTA, although the exact value depends on the type of transformations and the succinctness of the original program. We also report another metric, the number of interfaces that the design needs to create, which more strongly quantifies programmer productivity. The

Table 4. Comparison of programmer convenience measured through lines of code (LOC) and number of interfaces.

| Application | PASTA LOC | PoCo LOC | PASTA #Interfaces | PoCo #Interfaces |
|---|---|---|---|---|
| Bandwidth Testing | 1097 | 160 | 12 | 2 |
| Shuffle | 1571 | 649 | 22 | 14 |
| GraphTraversal | 1364 | 454 | 47 | 15 |

reduction is due to 1 MPMC port providing access to $N$ accessors. The higher the value of $N$, the more the reduction. Note that vectorized channels (stream/buffer) of the same data type are counted as one interface.

## 7 Related Work

**Standardized Memory Models:** A global view of memory has been discussed in CoRAM [8], which suggests that the architecture should present to the user a common, virtualized appearance of the FPGA memory. The edge-memory is assumed as one global pool of addressable region that different users may access through coordinated threads. These threads are managed by a centralized scheduler that directly controls the amount of requests that can be made at a given time. Extending the scope beyond FPGA fabric, LEAP [13] is an FPGA operating system built around latency-insensitive communication channels which are inferred at compile time. LEAP provides standardized interfaces and automatic management of the underlying on-chip memory resources. LEAP scratchpads [4] provide an expandable memory abstraction which presents a BRAM interface, transparently using the on-chip memory as private caches.

These interesting paradigms expose to the user a unified view of the off-chip memory, which may be able to support complex multi-producer multi-consumer models. However, the sharing of information between connected users in these prior studies suffers the latency of off-chip access and the arbitration latency of the centralized scratchpad controller. To the best of our knowledge, PoCo is the first work that supports MPMC models with on-chip memories for modern HLS programming models.

**Dynamic Memory Allocations:** Contributions like HiDMM [21] feature support for dynamic memory allocations. The tool takes a cross-stack approach, profiling different allocations from source-code into separated *heaps*. Compared to PoCo, HiDMM provides a cleaner programming model by using the standard *malloc* and *free* calls for the allocations and deallocations. However, the allocations are performed by profiling the access pattern at compile time, assuming a static schedule for access to these pointers. Therefore, consumers in MPMC models get mapped to the same heap, resulting in similar fabric limitations of connectivity in designs as discussed in Sec 2.2.1. Further, the static scheduling may not be useful when handling sparse data or when expecting latency insensitiveness.

**Frequency Optimizations on Multi-die FPGAs:** In addition to TAPA and PASTA, Elastic-DF [5] accelerates dataflow applications by partitioning the design into different dies of an FPGA. FADO [11, 12] extends this idea into non-dataflow applications as well. It iteratively co-optimizes the directives and floorplan of HLS designs implemented on multi-die FPGAs and drastically reduces the search time required for a legal physical layout. However, these tools do not have enough information of the underlying design so as to optimize them at a coarser granularity. Given that PoCo has a well-defined architecture, there are architecture-specific coarse-grained optimizations that can be leveraged. The MPMC-buffer appears as a set of movable blocks that can potentially benefit from the integration of novel floorplanners like AMF-Placer 2.0 [20].

**Increasing Programmer Productivity:** HeteroCL [19] is a Python-embedded DSL plus compilation flow for programming heterogenous accelerators. It decouples the user-algorithm from three orthogonal hardware customizations – customization of compute, memory access, and data representations., However, unlike TAPA/PASTA/PoCo, HeteroCL does not have explicit support for task parallelism and does not support coarse-grained floorplanning optimizations.

OpenCL-based HLS and recent Vitis HLS have some limited support for task parallelism, where tasks can communicate with each other via streaming (i.e., FIFO) channels. However, they do not support MPMC buffer channels which PoCo does, and do not support coarse-grained floorplanning optimizations. CPU and GPU architectures are built with a fixed memory hierarchy in mind, often with a cache-coherent shared memory model, especially for CPUs. However, FPGA designers expect excessive customization of compute and memory access. Hardware managed cache-coherency is often not necessary in most workloads. However, for certain applications with irregular memory accesses, we may consider a hybrid cache-and-buffer approach.

## 8 Conclusion

This paper introduces PoCo, an end-to-end automation framework for constructing scalable datapaths for task-parallel programming models with MPMC buffer support. It offers a generalized on-chip memory view for task-parallel systems, allowing them to dynamically allocate and exchange memory references. In the frontend, we first design a easy-to-use shared buffer abstraction that simplifies dataflow management for end users. In the backend, we optimize dual-port memory accesses for imbalanced pipelines and automate the coarse-grained placement of MPMC buffer blocks. Compared to state-of-the-art framework PASTA, we achieve an average of 22% (up to 50%) reduction in BRAM utilization for SPSC buffers. Moreover, three case studies using PoCo on Alveo U50 and U280 FPGAs showcases its effectiveness in allowing rapid prototyping of MPMC designs while offering considerable frequency improvements (1.5×) for latency-insensitive dataflows compared to current state-of-the-art tools.

## 9 Acknowledgments

This work is partly supported by NSERC Discovery Grant RGPIN-2019-04613, DGECR-2019-00120, and CFI John R. Evans Leaders Fund.

## References

[1] 2022. *Altera is being realistic about FPGA compute in the datacenter.* https://www.nextplatform.com/2024/09/26/altera-is-being-realistic-about-fpga-compute-in-the-datacenter/

[2] 2022. *AMD Completes Xilinx Acquisition And The Obvious Synergies Spell Great Potential.* https://www.forbes.com/sites/davealtavilla/2022/02/14/amd-completes-xilinx-acquisition-and-the-obvious-synergies-spell-great-potential/

[3] 2024. *Accelerated chip verification using AWS EC2 F1 and VeriFire.* https://aws.amazon.com/blogs/industries/awsec2fiandverifire/

[4] Michael Adler, Kermin E Fleming, Angshuman Parashar, Michael Pellauer, Joel Emer, et al. 2010. LEAP Scratchpads: Automatic Memory and Cache Management for Reconfigurable Logic [Extended Version]. (2010).

[5] Tobias Alonso, Lucian Petrica, Mario Ruiz, Jakoba Petri-Koenig, Yaman Umuroglu, Ioannis Stamelos, Elias Koromilas, Michaela Blott, and Kees Vissers. 2021. Elastic-DF: Scaling Performance of DNN Inference in FPGA Clouds through Automatic Partitioning. *ACM Trans. Reconfigurable Technol. Syst.* 15, 2, Article 15 (Dec. 2021), 34 pages. doi:10.1145/3470567

[6] AMD-Xilinx. 2023. *Vitis HLS: High-Performance Design Using Task-level Parallelism (WP554).* https://docs.amd.com/r/en-US/wp554-high-performance-design/

[7] Xinyu Chen, Yao Chen, Feng Cheng, Hongshi Tan, Bingsheng He, and Weng-Fai Wong. 2022. ReGraph: Scaling Graph Processing on HBM-enabled FPGAs with Heterogeneous Pipelines. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO).* 1342–1358. doi:10.1109/MICRO56248.2022.00092

[8] Eric S. Chung, James C. Hoe, and Ken Mai. 2011. CoRAM: an in-fabric memory architecture for FPGA-based computing. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '11).* Association for Computing Machinery, New York, NY, USA, 97–106. doi:10.1145/1950413.1950435

[9] E. G. Coffman, M. Elphick, and A. Shoshani. 1971. System Deadlocks. *ACM Comput. Surv.* 3, 2 (June 1971), 67–78. doi:10.1145/356586.356588

[10] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. 2018. Understanding Performance Differences of FPGAs and GPUs. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).* 93–96. doi:10.1109/FCCM.2018.00023

[11] Linfeng Du, Tingyuan Liang, Sharad Sinha, Zhiyao Xie, and Wei Zhang. 2023. FADO: Floorplan-Aware Directive Optimization for High-Level Synthesis Designs on Multi-Die FPGAs. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '23)*. Association for Computing Machinery, New York, NY, USA, 15–25. doi:10.1145/3543622.3573188

[12] Linfeng Du, Tingyuan Liang, Xiaofeng Zhou, Jinming Ge, Shangkun Li, Sharad Sinha, Jieru Zhao, Zhiyao Xie, and Wei Zhang. 2024. FADO: Floorplan-Aware Directive Optimization Based on Synthesis and Analytical Models for High-Level Synthesis Designs on Multi-Die FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 17, 3, Article 47 (Sept. 2024), 33 pages. doi:10.1145/3653458

[13] Kermin Fleming, Hsin-Jung Yang, Michael Adler, and Joel Emer. 2014. The LEAP FPGA operating system. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8. doi:10.1109/FPL.2014.6927488

[14] Licheng Guo, Yuze Chi, Jason Lau, Linghao Song, Xingyu Tian, Moazin Khatti, Weikang Qiao, Jie Wang, Ecenur Ustun, Zhenman Fang, Zhiru Zhang, and Jason Cong. 2023. TAPA: A Scalable Task-parallel Dataflow Programming Framework for Modern FPGAs with Co-optimization of HLS and Physical Design. *ACM Trans. Reconfigurable Technol. Syst.* 16, 4, Article 63 (dec 2023), 31 pages.

[15] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. 2021. AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 81–92. doi:10.1145/3431920.3439289

[16] Moazin Khatti, Xingyu Tian, Yuze Chi, Licheng Guo, Jason Cong, and Zhenman Fang. 2023. PASTA: Programming and Automation Support for Scalable Task-Parallel HLS Programs on Modern Multi-Die FPGAs. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 12–22. doi:10.1109/FCCM57271.2023.00011

[17] Moazin Khatti, Xingyu Tian, Ahmad Sedigh Baroughi, Akhil Raj Baranwal, Yuze Chi, Licheng Guo, Jason Cong, and Zhenman Fang. 2024. PASTA: Programming and Automation Support for Scalable Task-Parallel HLS Programs on Modern Multi-Die FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 17, 3, Article 42 (Sept. 2024), 31 pages. doi:10.1145/3676849

[18] Takeshi Kumagai and Kazuo Ikegaya. 1986. Organization of the two-dimensional omega network. *Systems and Computers in Japan* 17, 11 (1986), 1–10. doi:10.1002/scj.4690171101 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/scj.4690171101

[19] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 242–251. doi:10.1145/3289602.3293910

[20] Tingyuan Liang, Gengjie Chen, Jieru Zhao, Sharad Sinha, and Wei Zhang. 2024. AMF-Placer 2.0: Open-Source Timing-Driven Analytical Mixed-Size Placer for Large-Scale Heterogeneous FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 9 (2024), 2769–2782. doi:10.1109/TCAD.2024.3373357

[21] Tingyuan Liang, Jieru Zhao, Liang Feng, Sharad Sinha, and Wei Zhang. 2018. Hi-DMM: High-Performance Dynamic Memory Management in High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2555–2566. doi:10.1109/TCAD.2018.2857040

[22] D. Mitra and R. A. Cieslak. 1987. Randomized parallel communications on an extension of the omega network. *J. ACM* 34, 4 (Oct. 1987), 802–824. doi:10.1145/31846.42226

[23] Neha Prakriya, Yuze Chi, Suhail Basalama, Linghao Song, and Jason Cong. 2024. TAPA-CS: Enabling Scalable Accelerator Design on Distributed HBM-FPGAs. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 966–980. doi:10.1145/3620666.3651347

[24] Manoj B. Rajashekar, Xingyu Tian, and Zhenman Fang. 2024. HiSpMV: Hybrid Row Distribution and Vector Buffering for Imbalanced SpMV Acceleration on FPGAs. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '24)*. Association for Computing Machinery, New York, NY, USA, 154–164. doi:10.1145/3626202.3637557

[25] Linghao Song, Yuze Chi, Licheng Guo, and Jason Cong. 2022. Serpens: a high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 211–216. doi:10.1145/3489517.3530420

[26] H. S. Stone. 1971. Parallel Processing with the Perfect Shuffle. *IEEE Trans. Comput.* 20, 2 (Feb. 1971), 153–161. doi:10.1109/T-C.1971.223205

[27] Alveo Xilinx. 2019. *Alveo U280 Data Center Accelerator Card User Guide (UG1314)*. https://www.mouser.com/pdfDocs/u280userguide.pdf