A Dual-Clock Multiple-Queue Shared Buffer

Anastasios Psarras, Michalis Paschou,
Chrysostomos Nicopoulos, and Giorgos Dimitrakopoulos

Abstract—Multiple parallel queues are versatile hardware data structures that are extensively used in modern digital systems. To achieve maximum scalability, the multiple queues are built on top of a dynamically-allocated shared buffer that allocates the buffer space to the various active queues, based on a linked-list organization. This work focuses on dynamically-allocated multiple-queue shared buffers that allow their read and write ports to operate in different clock domains. The proposed dual-clock shared buffer follows a tightly-coupled organization that merges the tasks of signal synchronization across asynchronous clock domains and queueing (buffering), in a common hardware module. When compared to other state-of-the-art dual-clock multiple-queue designs, the new architecture is demonstrated to yield a substantially lower-cost implementation. Specifically, hardware area savings of up to 55 percent are achieved, while still supporting full-throughput operation.

Index Terms—Multiple queues, shared buffering, dual-clock FIFO, clock-domain crossing

1 Introduction

On-chip buffering structures are fundamental memory blocks used in the majority of digital systems. Buffering is often instantiated as either a single queue, or a collection of parallel queues, which are used to temporarily store data in a First-in-First-Out (FIFO) order during processing and/or transfers (e.g., from one hardware module to another). Multiple parallel queues find wide applicability in Networks-on-Chip (NoC) for the implementation of virtual channels [1], and to support multiple outstanding transactions within the network interfaces [2], [3], while they are often found in multi-threaded processors [4].

The microarchitecture of a *single* FIFO queue has been extensively explored in the past. Prior research has covered both fully synchronous implementations (where the read the and write ports belong to the same clock domain), and asynchronous designs (aka dual-clock) [5], [6], which allow the read and write ports to operate in different clock domains. In the latter case, the dual-clock FIFOs provide the required temporary storage space, and they allow for safe data transfers across the two clock domains.

The implementation of *multiple* parallel queues involves more design options. In their most primitive form, these hardware-based queues have a specific statically-allocated maximum size. The static space allocation may lead to inefficient use of memory resources; e.g., at a specific point in time, one queue may be under-utilized, while others may be full and would potentially benefit from more space. This intuitive observation about the innate deficiencies of static buffer allocation has led to the development of more complex parallel-queue structures that employ *buffer-sharing* [7], [8], [9]. In shared buffers supporting multiple parallel queues, the entire pool of available buffer space is common to (and shared by) all queues. Consequently, the utilization of memory space is much more efficient, and the dynamic (run-time) allocation of space to the various active queues can follow any desired policy.

In this paper, we focus on the design of a multiple-queue shared buffer that allows its read and write ports to belong to asynchronous clock domains. Such structures combine the scalability of dynamic buffer allocation with the flexibility offered by the integrated Clock Domain Crossing (CDC). Although shared buffering and CDC can be performed independently using a *loosely-coupled* organization [10], in this paper, we follow a *tightly-coupled* approach, similar to [11], where the tasks of signal synchronization and queueing (buffering) are "merged" and performed using a common hardware module. Although the basic organization resembles the architecture of [11], the state required by the dynamically-allocated shared buffer is distributed in a different way across the read and write clock domains, which allows us to minimize the amount of information that needs to be transferred across the two clock domains in order for the read and write sides to stay in sync. In this way, we remove any restriction in buffer allocation and effectively combine the dynamic shared-buffer allocation with low-latency data transfers across the two clock domains, which, in turn, translates to lower buffering requirements under high-throughput operation.

The proposed multi-queue shared buffer architecture-and its associated hardware implementation-opens up a new design space, which is quantitatively explored in this paper. When compared to state-of-the-art alternatives, the developed new architecture is demonstrated to yield significantly lower-cost design implementations.

The rest of the paper is organized as follows: Section 2 summarizes the organization of dual-clock FIFOs for single and multiple queues. Section 3 introduces the proposed dual-clock multiple-queue shared buffer, while Section 4 presents the experimental results comparing the proposed and current state-of-the-art buffer architectures. Finally, conclusions are drawn in Section 5.

2 State-of-the-Art Dual-Clock FIFOs

2.1 Single Queues

A dual-clock FIFO follows the basic organization depicted in Fig. 1. Incoming data is written using the write clock to the position indexed by the tail pointer, as long as the full signal at the write port is not asserted. New data is read out of the FIFO in the read clock domain by extracting the data stored in the address indexed by the head pointer, as long as the FIFO is not empty. The full and empty conditions are determined independently on the write and read sides, by allowing each side to compare its local pointer to the synchronized version of the pointer of the other side. Incoming data is stored in consecutive positions of the memory array of the dual-clock FIFO, which allows the read/write side to determine independently the number of items stored in the FIFO by merely measuring the distance between the local and remote pointers.

Head/tail pointer synchronization is performed using brute-force synchronizers. This is possible, since (a) the pointers are monotonically increased by one on every read and write operation, and (b) they follow Gray encoding. In this way, any synchronization error that may occur during the pointer transfer across the other clock domain does not disrupt the pointer location by more than one increment. The CDC failure rate of the dual-clock FIFO is determined by the depth of the brute-force synchronizers [6]. The dual-clock FIFO depicted in Fig. 1 uses 2-stage brute-force synchronizers; increasing the depth of the brute-force synchronizers reduces the risk of failure.

- A. Psarras, M. Paschou, and G. Dimitrakopoulos are with the Electrical and Computer Engineering Department, Democritus University of Thrace, Xanthi 671 00, Greece. E-mail: {apsarra, dimitrak}@ee.duth.gr, mich_pasc@yahoo.gr.
- C. Nicopoulos is with the Electrical and Computer Engineering Department, University of Cyprus, Nicosia 1678, Cyprus. E-mail: nicopoulos@ucy.ac.cy.

Manuscript received 6 Feb. 2017; revised 27 Apr. 2017; accepted 11 May 2017. Date of publication 16 May 2017; date of current version 14 Sept. 2017.
(Corresponding author: Anastasios Psarras.)
Recommended for acceptance by P. Faraboschi.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2017.2705141

0018-9340 © 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

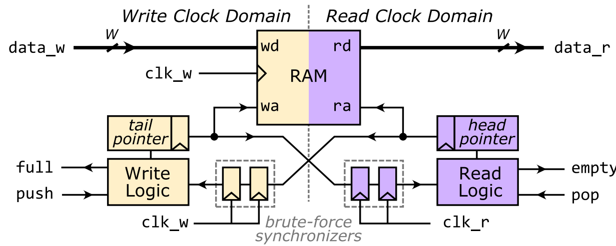

Fig. 1. The organization of a single-queue dual-clock FIFO.

2.2 Multiple Queues

In the case of multiple parallel queues, the read and write ports must be enriched, as shown in Fig. 2, to allow the read and write operations to occur to any of the available queues. The write and read ports provide a single data input and a single push/pop signal, enhanced with a queue identifier that designates the accessed queue. The state of each queue is reflected by the full/empty signals (one per queue) that indicate which queues are eligible for an enqueue/dequeue operation in the corresponding cycle.

Multiple parallel queues that operate between two clock domains can be implemented by placing multiple dual-clock FIFOs in parallel. Actually, since only one queue may be accessed in each cycle, the queues can share a common dual-port memory array, as shown in Fig. 3a, provided that each queue is implemented in a different statically-allocated address range. The management of each queue can still be performed by a set of head and tail pointers per queue. Head and tail pointers move cyclically inside each static partition, and data is placed in consecutive positions within each partition. Thus, Gray encoding of the head/tail pointers and a set of brute-force synchronizers per queue are enough for transferring the pointers across the two clock domains, as shown in Fig. 3b.

Even if multiple synchronization points exist in parallel, only one of them is accessed per-read and per-write clock cycle, since at most one data item can be read or written. Therefore, the failure rate of this dual-clock multiple queue buffer is equivalent to the failure rate of a single-queue dual-clock FIFO, and it is determined by the depth of the brute-force synchronizers used for each queue.

The organization of the statically-allocated multiple-queue buffer shown in Fig. 3b-which splits the head and tail pointers into the read and write sides, respectively-*cannot* be extended to dynamically-allocated shared buffers. Dynamic memory allocation implies that each memory position may host items destined for different queues at different snapshots in time. Once an item is extracted, its memory position is freed and can be re-allocated to a different queue. Therefore, as long as there is no guarantee that each queue can be placed in consecutive positions (unless unused cycles are spent for queue data reorganization), the head/tail pointers cannot simply be the increments of their previous values. Consequently, using Gray encoding and brute-force synchronization when transferring the pointers across the two clock domains is no longer possible.

To avoid this constraint, a *loosely-coupled* solution that decouples synchronization and shared buffering has been presented. In the CrossOver [10] *loosely-coupled* solution, shown in Fig. 4a, synchronization and buffering are decoupled: data is first synchronized on the write side through a dual-clock FIFO, and then stored in a fully-synchronous shared buffer that belongs entirely to the read clock domain. Alternatively, the dual-clock FIFO could be placed on the read side, and the synchronous shared buffer on the write side, leading to the organization of Fig. 4b. In both cases, the failure rate of CrossOver is determined by the failure rate of the dual-clock FIFO between the two clock domains.

Even though the decoupling of synchronization and shared buffering simplifies integration, it incurs additional data latency, which increases the minimum buffering requirements for full-throughput operation. Also, it forces data belonging to different queues to be serialized inside the dual-clock FIFO placed at the interface of the two clock domains. In this way, items of one queue may be blocked inside the dual-clock FIFO behind items destined for other queues. This blocking creates dependencies among the different queues that may cause performance degradation, or even lead to deadlock, which is avoided by the flow control rules of CrossOver.

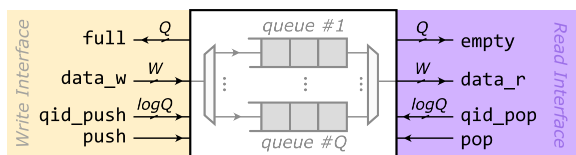

Fig. 2. The multiple-parallel-queue interface provides one data input for writing and reading to/from at most one queue at a time. Write and read interfaces use push/full and empty/pop flow-control signals, accompanied with queue identifiers that indicate the accessed queue.

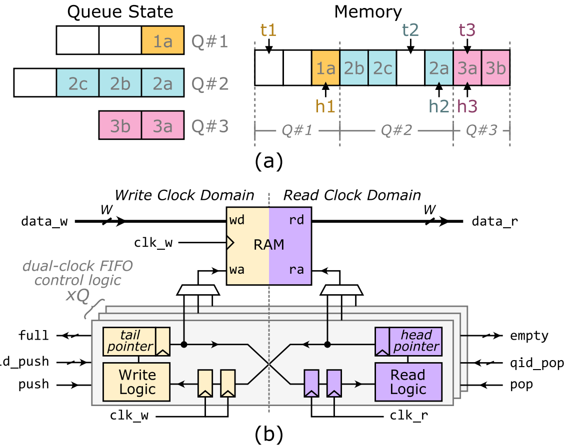

Fig. 3. (a) The multiple-parallel-queue concept with a statically-allocated memory space per queue, which is indexed by separate head/tail pointer pairs (one for each queue); (b) the dual-clock implementation of this concept, which uses multiple copies (one for each queue) of the dual-clock FIFO control logic.

3 A Tightly-Coupled Dual-Clock Shared Buffer

The design of the proposed dual-clock multi-queue shared buffer aims at a different-*tightly-coupled*-approach, similar to the one in [11], which unifies signal synchronization and shared buffering in a single low-latency and cost-efficient hardware structure. A high-level overview of the proposed dual-clock shared buffer is illustrated in Fig. 5.

For the dual-clock multi-queue shared buffer to operate correctly, the write clock domain must be able to (a) detect the queues' full state, and (b) determine where incoming data should be stored, by keeping track of the memory availability. On the other side, the read logic must (a) be aware of the empty status of the queues, and (b) be able to read data in the order of arrival. Note that it is not

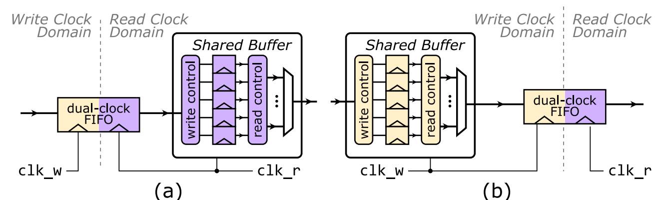

Fig. 4. (a) CrossOver implements parallel queues over a CDC point by using a dual-clock FIFO for data synchronization, along with a fully-synchronous shared buffer located entirely in the read clock domain. (b) An equivalent symmetric organization that places the shared buffer entirely in the write clock domain.

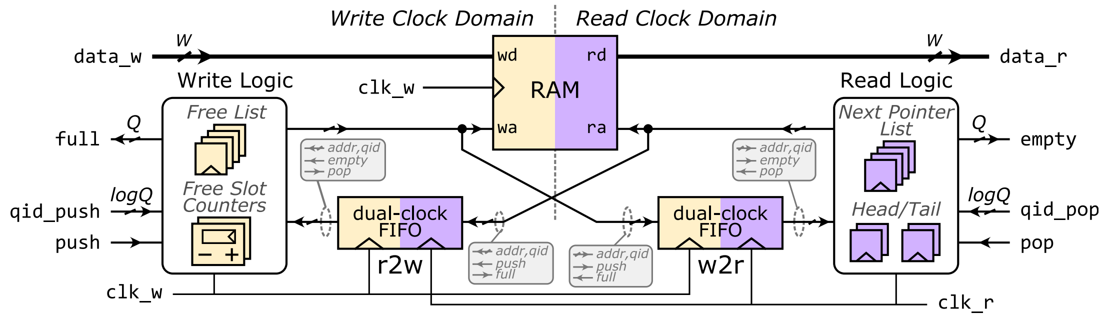

Fig. 5. The organization of the proposed dual-clock multi-queue shared buffer. The state of the write and read logic blocks is synchronized via the w2r and r2w dual-clock FIFOs, which carry the write and read addresses-along with the queue IDs of the most recent enqueue and dequeue events-in the write and read clock domains, respectively.

necessary for the read side to be aware of the memory availability (it will never store data), in the same way that the write side does not need to be able to sequence data in different queues (it will never read data).

3.1 Write/Read Logic Split Across Clock Domains

In the write clock domain, a Free List is used to generate write addresses and track memory availability. Each record of the list is a single-bit element that indicates whether the corresponding address is occupied, or free to be allocated. Each incoming data item is written to the first available address provided by the Free List, *irrespective of the queue to which it belongs*. With this selected strategy, the write logic is oblivious of the state of each queue and queue management occurs completely on the read side. If the Free List runs out of free slots, all queues are designated as full instantaneously. Producing the full signals per queue requires additional Free Slot Counters on the write side, which implement the shared buffering flow control protocol described in Section 3.3.

In the read clock domain, the Next Pointer List and a set of Head and Tail Pointers are used to manage the data items that belong to different queues. One head pointer per queue is used to index the top item for that queue, while the next in-order item can be located through the Next Pointer List. Each record of the list points to the address of the next in-order item. When a pop occurs, the next head pointer is the address pointed to by the Next Pointer List element of the current head. A null element indicates that no other item follows that address and, thus, the queue is empty.

The tail pointers indicate the last element of each queue and are used to populate the Next Pointer List. When a new item is enqueued, the tail of the accessed queue is updated and points to the memory address of the incoming data word, effectively linking it with the previous item of that queue. Thus, the update of the tail pointer needs information that belongs to both the write domain (i.e., the address where the new item was stored), and to the read domain (i.e., the Next Pointer List). Thus, irrespective of which domain the tail pointer is located, some form of information

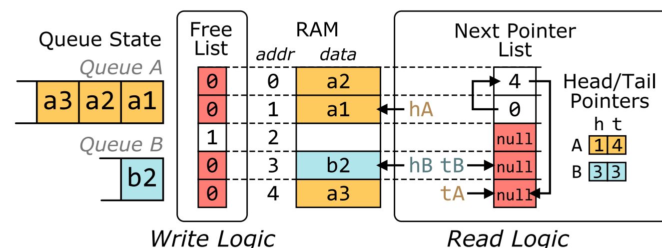

Fig. 6. A snapshot of the distributed state of the proposed dual-clock shared buffer across the two clock domains, when supporting 2 queues. Data is stored in the memory and is sequenced by the read logic, through the use of head and tail pointers and a next pointer list. The write logic only tracks memory availability and generates write addresses for the incoming items.

transfer across the two domains is inevitable when updating the tail pointer of each queue. Since we selected to handle queue management completely in the read domain, we placed the tail pointers into the read domain, too. Even though this decision contradicts the traditional approach of locating the tail pointers in the write domain, it simplifies the overall queue management process, since it minimizes the amount of information that needs to be transferred across the two domains to ensure that the distributed state of the shared buffer stays in sync.

In the proposed design, if the tail pointers were placed in the write clock domain, each time a new data item would be enqueued in the buffer, the Next Pointer List would have to be accessed to correctly update the corresponding tail pointer. Accessing the Next Pointer List of the read domain from the write domain would require two pointer transfers: one pointer-get from the write to the read domain, and one pointer-reply from the read to the write clock domain, thus significantly increasing the latency of internal-state updates.

The design of [11], which employs a similar overall architecture, places the tail pointers in the write domain. Each time an enqueue operation occurs, the write domain *pre-allocates* a memory address for the next in-order item of the accessed queue (i.e., the *tail pointer*) and stores it into the main memory, along with the enqueued data. Once the change on the empty state is synchronized to the read clock domain, the read side can locate both the top item for that queue and the address of the next in-order item. In this way, since the next write address is decided in the write clock domain, the location must be reserved (pre-allocated) and cannot be used by any other queue, effectively limiting the available buffer space that can be utilized. On the contrary, the design proposed in this work does not require any pre-allocated and reserved buffer slots.

Fig. 6 illustrates an example of the state inside the Read and Write Logic of the proposed dual-clock shared buffer when implementing 2 queues. In this example, Queue A contains items $a1$, $a2$, and $a3$, and its top item is found in the address pointed to by the head pointer. The next item in order, $a2$, is found in the address pointed to by element 1 in the Next Pointer List, and, subsequently, $a3$ is found in the address pointed to by $a2$ in the Next Pointer List. For Queue B, a null pointer is found in the list record corresponding to its top item, indicating no other item follows. The Read Logic has all the state it needs to sequence items from the two queues. The write side is oblivious to the data order, or to which queue the items belong to; the write side only keeps track of the memory availability through the Free List, to report the queues' full state and to provide new write addresses to the memory.

3.2 Write-Read Logic Synchronization

For the two sides to stay in sync, all push events must be synchronized to the read side, and all pop events must be synchronized to the write side. In case of an enqueue, the read domain only needs to know (a) which queue was accessed, in order to update the

proper head/tail pair, and (b) the address where the new data was stored, in order to update the Next Pointer List. When a dequeue occurs on the read side, the Write Logic only needs to know the address that was emptied, in order to mark it as available in the Free List. This state-synchronization method exploits the fact that, in each cycle, only one queue may be accessed in each clock domain. In a write-clock cycle, *at most one* item can be enqueued, and, in a read-clock cycle, *at most one* item can be removed.

In order for the two sides to exchange that information, they transfer to each other messages through the "r2w" and "w2r" dual-clock FIFOs shown in Fig. 5. The data exchanged between the two clock domains is not required to follow a specific encoding, since the synchronization safety is inherently guaranteed by the dual-clock FIFO.

When an enqueue operation occurs, the Write Logic-in addition to updating its state-pushes an item into the "w2r" dual-clock FIFO, containing the write address (as selected by the Free List), and the queue ID (as selected by the `qid_push` input). In every read-clock cycle, the Read Logic tries to pop the items of the "w2r" FIFO, in order to understand the push events that occurred on the write side and to appropriately update its local state.

Similarly, when an item is dequeued from the shared buffer, the Read Logic pushes the item's address and queue ID into the "r2w" dual-clock FIFO. In every write-clock cycle, the Write Logic tries to pop the synchronized version of that data from the "r2w" FIFO. The address that it receives corresponds to the de-allocated address of the memory, and it is used to free the corresponding entry of the Free List. The received queue ID is used to select the Free Slot Counter that needs to be updated.

The synchronization latency of the "w2r" and "r2w" FIFOs inevitably causes the read and write logic to be out-of-sync for a certain time window. After a data push, the memory and the state of the write-side are updated immediately, but the Read Logic is notified of the new item only after several read-clock cycles. During that time, the Read Logic uses stale pointers and is unaware of the new memory state. A similar effect can be observed in the Write Logic, when a dequeue occurs on the read side. Nevertheless, this notification delay does *not* break the correct operation of the shared buffer, nor does it affect the throughput of the transmission; it only incurs additional latency.

The CDC failure rate of the proposed dual-clock shared buffer is determined by the CDC failure rate of the "w2r" and "r2w" dual-clock FIFOs [12], [13] and thus is the same with the state-of-the-art alternatives presented in Section 2.

3.3 Flow Control and Internal Backpressure

The proposed dual-clock shared buffer follows the push/full flow control at the write interface. If the Write Logic relied only on the state of the Free List to determine the full state of every queue, information about *individual* queues would be lost. Either all of the queues would be considered to be full simultaneously (in the absence of an available item in the Free List), or no queue would be considered full (when the Free List has at least one available item). To control the full-state of each queue *separately*, two requirements are necessary: per-queue Free-Slot Counters (see Fig. 5), and the adoption of a flow-control strategy similar to the one presented in [9] for the case of virtual channels in a NoC. The flow-control policy can be extended to more complex shared-buffer management schemes [14].

Each one of the Q supported queues can have P private slots that cannot be allocated to other queues, plus S slots that can be freely allocated to any queue (total number of buffer slots equals $Q \times P + S$). At least one private slot for each supported queue is typically necessary in shared-buffer implementations, in order to prevent queue starvation (which could also lead to protocol-level deadlocks).

Therefore, the maximum capacity for a single queue is the sum of its private and the globally shared slots (total $P + S$). The Write Logic must be able to control the full-state of each queue independently, by keeping track of the state of both the shared space, and of every queue separately. A queue is full when it has exceeded its private quota (P slots), and none of the S shared slots are available. The Free Slot Counters in the Write Logic keep track of the remaining slots for each queue (out of the total $P + S$), while an additional global shared counter tracks the number of free shared slots, out of S in total. To provide this functionality, the Write Logic should be aware of the ID of the queue in which the push and pop events occur, in order to increment and decrement the proper Free Slot Counter. For an enqueue event, this is directly provided at the interface, along with the push signal. For a dequeue event, this information is transferred via the "r2w" dual-clock FIFO.

The use of "w2r" and "r2w" FIFOs to transfer the write and read addresses across the two clock domains implies the possibility of back-pressure occurring internally in the dual-clock shared buffer. If the dual-clock FIFOs are not deep enough, it is possible that they become full, blocking the Write or Read Logic from sending notifications to the other side. If both of them become full simultaneously, then a cyclic dependency is formed, with both sides waiting for the other to pop from the dual-clock FIFO. However, this blocking is *guaranteed* to be temporary, since both sides will always be able to consume a FIFO item. Consuming a "w2r" item on the read side only requires having enough space in the Next Pointer List and a head/tail pointer pair that corresponds to that queue; both of these conditions are always true. The item's consumption is independent of any other actions at the read interface (e.g., it does not depend on whether a pop may occur). Similarly, on the write side, an "r2w" item can be consumed unconditionally, since there is always a Free List record that corresponds to the incoming read address. Thus, within a finite amount of time, the blocking will *definitely* be released, breaking the dependency cycle and eliminating any possibility of a deadlock.

Nevertheless, this blocking affects the flow control of the newly proposed dual-clock multi-queue shared buffer. If the "w2r" FIFO is full, then no push operation can occur to any of the queues. Otherwise, the address and the queue ID of the enqueued item will be lost. Likewise, if the "r2w" FIFO becomes full, no item can be popped from the read interface. Therefore, the flow-control rules must be enhanced: if the "w2r" FIFO is full, all queues must show up as full. Similarly, if the "r2w" FIFO is full, all queues must appear as empty.

It is also possible to completely avoid this blocking by appropriately sizing the FIFOs. Making them as deep as the memory would eliminate any blocking possibility. For instance, if the "w2r" FIFO becomes full, this would imply that the memory is also full, so no other push can anyway occur. The same applies to the "r2w" FIFO; if it gets full, then all memory positions have been freed and all queues are empty. Thus, no other pop is anyway possible, and no extra rules are required. This approach sizes the "w2r" and "r2w" FIFOs pessimistically as the memory depth is increased. As will be shown in Section 4, much shallower FIFOs still suffice to avoid internal back-pressure and to achieve full-throughput operation. In any case, the area overhead imposed by the "w2r" and "r2w" FIFOs is minimal and independent of the buffer's data width. Those FIFOs store addresses and queue identifiers and, thus, their width scales logarithmically with buffer depth and the number of queues supported by the buffer.

3.4 Baseline and Latency-Optimized State Update

The addresses and queue identifiers that are read out of the "w2r" and "r2w" FIFOs are used to update the state of the Read and Write Logic, respectively. The state update can occur either with one-cycle latency, or with zero latency. In the first case, shown in

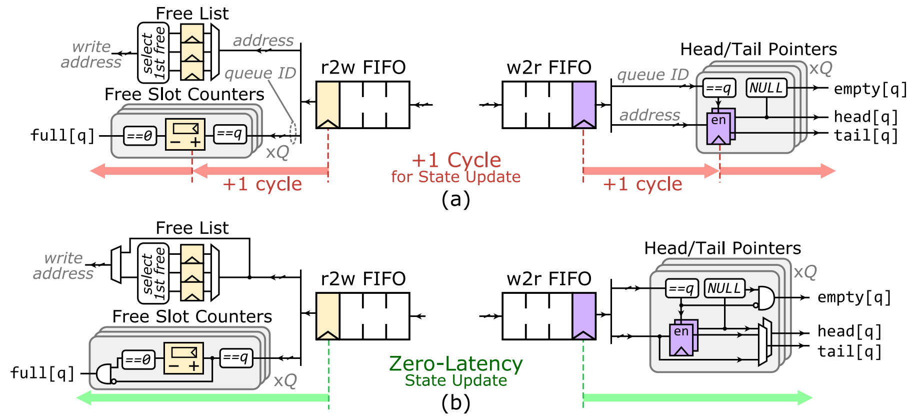

Fig. 7. State-update alternative approaches: (a) Pipelined state update with 1-cycle latency; (b) Zero-latency state-update that produces full/empty conditions in parallel to the state-register update, through bypass paths.

Fig. 7a, the addresses first update the head and tail pointers, or the Free List, and, in the following cycle, this update is reflected in the empty/full signals. In this case, the head/tail pointers and the registers of the Free List act as pipeline registers separating the read paths of the dual-clock FIFOs from the rest of the queue management operations.

In the second case (i.e., zero-latency update), shown in Fig. 7b, bypass paths can be added on both read/write sides, allowing the state to be concurrently reflected at the interface, as soon as the FIFO item is received. On the read side, the empty state of the queue in which the push operation occurred must be bypassed whenever a FIFO item exists for that queue, causing the empty generation to be modified as depicted in Fig. 7b. If the accessed queue was previously empty, the head pointer must also be bypassed to point to the incoming read address. Similar bypass paths are added to the Write Logic, as seen in Fig. 7b, so that the queues do not appear as full if an "r2w" item is detected. In that case, the Free List is bypassed to treat the incoming read address as available, so that a push can occur, even if the current state of the list indicates a full memory.

4 Analysis & Experimental Results

To evaluate the proposed dual-clock multi-queue shared buffer and compare it against the current state-of-the-art alternatives, we implemented all designs in SystemVerilog. Latency and throughput measurements are derived from simulation, while hardware cost evaluation is conducted after synthesizing the hardware models in a 45 nm standard-cell library and performing placement-and-routing of the resulting designs using the Cadence backend flow. For all architectures, the data memories were implemented using Standard-Cell-based Memories (SCM) using flip-flops, assuming 64-bit wide data interfaces. The same dual-clock FIFO design-employing 2-stage brute-force synchronizers-is used in all cases. Simulations were performed on the test setup shown in Fig. 8. In all examined scenarios, the transmitter (tx) that operates under the write clock domain connects to the multiple-queue buffer's write interface, enqueuing at most one data item per write-clock cycle, while the receiver (rx) is connected to the buffer's read interface, extracting at most one item in each read-clock cycle.

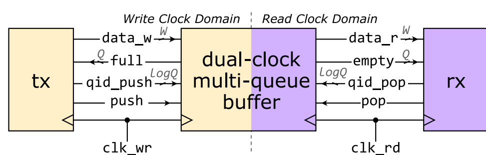

Fig. 8. The simulation setup used to measure latency and throughput of all dual-clock multiple-queue buffer architectures under comparison.

4.1 Notification Latency

The latency required for synchronizing the enqueue and dequeue events across the two clock domains involves both local state-update operations, and the transfer of required information through the "w2r" and "r2w" dual-clock FIFOs. For either an enqueue, or a dequeue, the notification latency consists of three components:

Local State Update: One cycle is spent in the domain where the event occurred, to update the local state and to push a message to the "w2r" or "r2w" dual-clock FIFO.

Synchronization: The transfer of addresses and the IDs of the accessed queues through the "w2r" and "r2w" dual-clock FIFOs requires a certain number of cycles, which depend on the depth of the brute-force synchronizers of each FIFO.

Remote State Update: The update of the Next Pointer List and the Head/Tail Pointers (Read Logic), as well as the Free List and Free Slot Counters (Write Logic) costs either one-cycle latency, or zero latency, according to the selected state-update policy shown in Fig. 7.

The notification latency incurred for the enqueue (L_{enq}) and dequeue (L_{deq}) operations, when the local and remote state-update costs 1 cycle each (Fig. 7a) and the "w2r" and "r2w" FIFOs insert a b-cycle CDC latency, is equal to

$$L_{enq} = 1\, clk_{wr} + (b+1)\, clk_{rd} \tag{1}$$

$$L_{deq} = (b+1)\, clk_{wr} + 1\, clk_{rd}. \tag{2}$$

Note that clk_{rd} and clk_{wr} refer to read and write clock cycles, respectively. Alternatively, when employing the zero-latency remote state-update shown in Fig. 7b, $L_{enq} = 1\, clk_{wr} + b\, clk_{rd}$, and $L_{deq} = b\, clk_{wr} + 1\, clk_{rd}$.

The proposed shared buffer compares favorably in terms of latency to the state-of-the-art architectures presented in Section 2. The notification latencies of all four alternatives are presented in Table 1, assuming a $b = 2$-cycle CDC latency for the dual-clock FIFOs. The "Static" design refers to the parallel dual-clock FIFOs that share a common statically allocated memory space, as shown in Fig. 3; "CrossOver" is the loosely-coupled architecture shown in Fig. 4a, which performs synchronization and shared buffering independently, while "Shared" and "Shared-LO" represent the proposed dual-clock multi-queue shared buffer with 1-cycle and 0-cycle remote state-update latencies, respectively.

"Static" requires one local-clock cycle for updating the local state (e.g., increasing the head or tail pointer), and 2 remote-clock cycles in the brute-force synchronizers, incurring the same latency as the Shared-LO architecture. CrossOver spends one more cycle to either transfer the incoming data word from the dual-clock FIFO to the synchronous shared buffer for an enqueue, or to update the remote state in the case of a dequeue.

TABLE 1
Notification Latencies of the Dual-Clock Multiple-Queue Buffer Designs Under Comparison

	Enqueue		Dequeue	
	clk_{wr}	clk_{rd}	clk_{wr}	clk_{rd}
Static	1	2	2	1
CrossOver	1	3	3	1
Shared	1	3	3	1
Shared-LO	1	2	2	1

4.2 Buffering Requirements

The number of buffers supported for each queue determines the data transfer throughput across the two clock domains. For full-throughput data transfers, the buffer must have enough slots to cover the delay of a *full notification loop* (aka Round-Trip-Time, RTT). The notification loop is the delay elapsed from the time the transmitter writes a word at the write interface, until it is notified back that the receiver has dequeued this particular word. The notification loop is, effectively, the sum of the latencies of an enqueue and a dequeue event. For dual-clock buffer architectures, the notification loop consists of both write- and read-clock cycles. E.g., a full notification loop for the "Shared-LO" architecture consists of 3 cycles in each clock domain (see Table 1).

In a dual-clock environment, the throughput of data transfers is measured differently on the read and write sides, due to the (possibly) different clock frequencies in the two domains. In this paper, we measure throughput from the perspective of the slow clock domain that determines the highest possible achievable throughput across the CDC boundary. Thus, we always want to check if the slow clock domain is fully utilized, even if-at the same time-the fast clock domain inevitably experiences idle cycles imposed by the slower rate of the other domain.

The number of buffer slots that guarantee full-throughput data transfers, as derived from simulations of a 2-queue system for different write/read clock-frequency ratios (f_{wr}/f_{rd}), is illustrated in Fig. 9. The worst-case buffering requirement is detected when the same clock frequency is used by both write and read clock domains, and the requirement relaxes as the frequency ratio is increased, regardless of which side is faster. For CrossOver and the proposed Shared and Shared-LO designs, we assume that each queue uses one private buffer slot $P = 1$ and the remaining S buffers can be shared arbitrarily by both queues. Both the private and shared buffers can be used by a queue to achieve full throughput data transfers.

Shared-LO outperforms all three architectures under all clock-frequency ratios, with a worst-case buffering requirement of 7 slots, under equal write and read clock frequencies. The Shared design experiences an increased notification latency, which leads to increased buffering requirements. Even though CrossOver also employs shared buffering, it requires up to ×2 more buffer space, due to its loosely coupled organization. CrossOver has to pay a

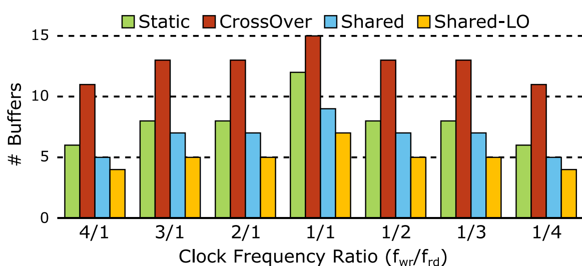

Fig. 9. The minimum number of buffers required by all architectures under comparison to enable full-throughput operation under various write/read clock-frequency ratios.

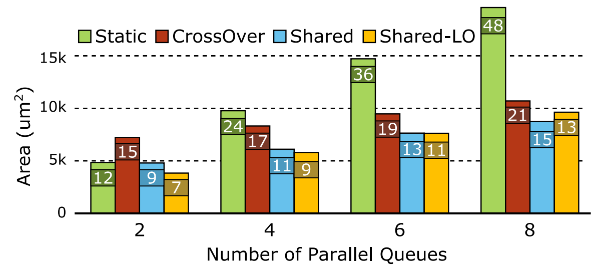

Fig. 10. The hardware area occupied by the buffers under comparison when they support 2, 4, 6, and 8 queues. The depth of each buffer, annotated inside each bar, corresponds to the minimum amount of buffering required to support full throughput under possibly equal write and read clock frequencies (i.e., worst-case scenario).

fixed overhead for the dual-clock data FIFO that is used to separate synchronization from buffering. On top of that, its increased notification latency, as presented in Table 1, leads to even more increased buffering requirements. On the other hand, Static suffers from the obvious drawbacks of static memory allocation. In this case, not all buffer slots can be used by all queues. Thus, although Static requires 6 slots to achieve full throughput, it requires such budget *per queue*, leading to a total memory depth of 12. This effect becomes more prominent as the number of parallel queues supported by the dual-clock buffer is increased.

In the following experiment, we compare the actual hardware area occupied by each architecture when it implements 2, 4, 6, and 8 queues, focusing on the worst-case scenario, where the write and read clock frequencies can become equal ($f_{wr}/f_{rd} = 1$). For every case, the depth is set to the minimum that can provide full-throughput operation. Fig. 10 depicts the area of 64-bit buffers optimized for 1 GHz operation at 45 nm/0.8 V. The number inside each bar indicates the number of buffers used by each architecture to achieve 100 percent throughput. In all cases, the proposed dual-clock shared buffer achieves considerable area savings in delivering 100 percent throughput, even if the depth of the internal "w2r" and "r2w" dual-clock FIFOs is selected pessimistically and set equal to the memory depth. The proposed architecture requires up to 55 and 33 percent less area than the Static and CrossOver designs, respectively. For 2 and 4 queues, Shared-LO presents the most area-efficient architecture, with 46 percent area savings compared to Static and 20 percent compared to CrossOver. As the number of queues increases, the area gap with CrossOver narrows down to 10 percent, since Shared-LO must meet tighter timing constraints to support the bypass paths; this is the price paid for offering reduced notification latency. These additional timing constraints make the area of Shared-LO larger than the area of Shared for the case of 6 and 8 queues, even if Shared-LO requires fewer total buffers.

4.3 Depth of the "w2r" and "r2w" Dual-Clock FIFOs

As previously mentioned, when the "w2r" and "r2w" dual-clock FIFOs are as deep as the buffer memory, data transmission can never be interrupted by internal back-pressure. Nevertheless, shallower "w2r" and "r2w" FIFOs can be used even if internal back-pressure, indeed, occurs. Since the transmission will always be limited by the slowest clock, as long as the internal blocking does not degrade throughput to less than the processing rate of the slowest domain, performance will be unaffected.

To test the effect of shallower "w2r" and "r2w" FIFOs, we measured the throughput of Shared-LO supporting 2 queues in a 16-slot shared memory for various "w2r" and "r2w" FIFO depths. Fig. 11 plots the results for data-burst lengths of 1 to 16 words, for a clock-frequency ratio of $f_{wr}/f_{rd} = 2/1$. The reported throughput is normalized to the throughput of the best-case scenario, where the dual-clock FIFOs are as deep as the memory (16 slots). It can be

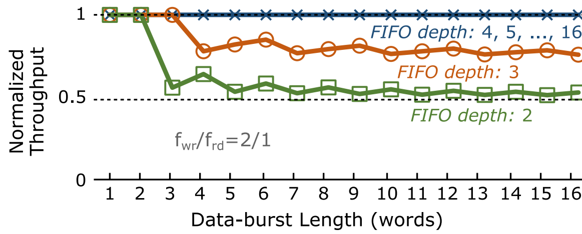

Fig. 11. The throughput of the proposed dual-clock multi-queue shared buffer for various data-burst lengths, when the "w2r" and "r2w" FIFOs are made shallower than the buffer's memory depth (16 slots).

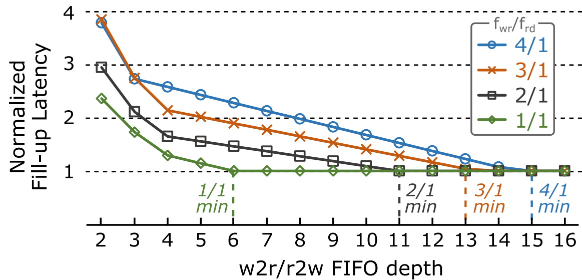

Fig. 12. The *fill-up* (or *flush*) *latency* of the proposed dual-clock shared buffer with 16 memory slots, as the "w2r" (or "r2w") FIFO depth is decreased under different write/read clock-frequency ratios (*write* clock is faster). Values are normalized to the best-case scenario where the "w2r" and "r2w" FIFOs are 16-slot deep.

observed that, even for 4-slot deep FIFOs, the performance is indistinguishable from the best-case scenario. The throughput starts degrading when the FIFOs are equipped with only 2 or 3 slots. Hence, using shallower "w2r" and "r2w" FIFOs is possible, since throughput will always be limited by the slowest of the two clocks. For instance, when the transmitter is twice as fast as the receiver, the maximum throughput will be limited by the receiver's rate of extracting words. From the transmitter's perspective, this is 1 word per 2 transmitter cycles. As long as the "w2r" and "r2w" FIFOs are deep enough to allow this rate, then the overall performance will not be affected. In the case of Fig. 11, although the "w2r" FIFO occasionally gets full and blocks the transmission, the stall rate does not degrade the transmission rate to a value lower than the receiver's maximum rate of 1 word per 2 write-clock cycles.

Similar results are derived for various clock-frequency ratios, regardless of which clock domain is faster. When the write and read clock frequencies are equal, 6 slots suffice in supporting any burst length with 100 percent throughput.

In order to assess more clearly the impact of the depth of the "w2r" and "r2w" dual-clock FIFOs, we define and measure the values of two buffer properties: (a) the *fill-up latency*, which is the time required for a fast transmitter to fill up an empty buffer, and (b) the *flush latency*, which is the time required for a fast receiver to read all data words from a full buffer. These situations present two extremes of traffic behavior, whereby one of the two sides remains idle for a long time, and suddenly becomes fully active.

Fig. 12 shows the *fill-up latency* as the "w2r" FIFO depth is decreased, normalized to the latency of the best-case configuration (16-slot deep "w2r" FIFO). The values are reported for various clock-frequency ratios, when the write clock domain is faster than the read clock domain, up to the point where the write/read clock frequencies are the same. For same-frequency clocks (ratio 1/1), 6 slots are enough to never interrupt the fill-up process. As the clock ratio is increased, the minimum requirement increases to 11 slots for a 2/1 ratio, 14 for 3/1, and 15 for 4/1. Note that the *flush latency* values are exactly the same as the *fill-up latency* values, but for the inverse clock ratios.

5 CONCLUSION

The efficiency and scalability of multiple-queue hardware data structures that can operate across arbitrary clock domains are judged by their inherent read/write notification latency, and the buffering space required to achieve high-throughput operation. The proposed dual-clock multi-queue shared buffer significantly reduces the hardware area cost, under equal notification latency, when compared to state-of-the-art architectures. The reaped gain is a direct consequence of the employed tightly-coupled organization, the reorganization of the write and read logic across the two clock domains, and the proper selection of the messages that need to be transferred across the two clock domains, in order for the state of the shared buffer to remain in sync.

REFERENCES

[1] W. J. Dally, "Virtual-channel flow control," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, no. 2, pp. 194–205, Mar. 1992.
[2] M. Daneshtalab, M. Ebrahimi, P. Liljeberg, J. Plosila, and H. Tenhunen, "Memory-efficient on-chip network with adaptive interfaces," *IEEE Trans. Comput.-Aided Des. Integrated Circuits Syst.*, vol. 31, no. 1, pp. 146–159, Jan. 2012.
[3] A. D. Tune, A. B. Laughton, D. A. Sara, S. J. Salisbury, and P. A. Riocreux, "Transaction response modification within interconnect circuitry," U.S. Patent 0 103 776, Apr. 14, 2015.
[4] H. Wang, I. Koren, and C. M. Krishna, "An adaptive resource partitioning algorithm for SMT processors," in *Proc. IEEE/ACM Int. Conf. Parallel Archit. Compilation Techn.*, 2008, pp. 230–239.
[5] C. E. Cummings, "Simulation and synthesis techniques for asynchronous FIFO design," presented at the Synopsys Users Group Conf., San Jose, CA, USA, 2002.
[6] R. Ginosar, "Metastability and synchronizers: A tutorial," *IEEE Des. Test Comput.*, vol. 28, no. 5, pp. 23–35, Sep./Oct. 2011.
[7] Y. Tamir and G. L. Frazier, "Dynamically-allocated multi-queue buffers for VLSI communication switches," *IEEE Trans. Comput.*, vol. 41, no. 6, pp. 725–737, Jun. 1992.
[8] C. A. Nicopoulos, D. Park, J. Kim, N. Vijaykrishnan, M. S. Yousif, and C. R. Das, "ViChaR: A dynamic virtual channel regulator for network-on-chip routers," in *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2006, pp. 333–346.
[9] I. Seitanidis, A. Psarras, K. Chrysanthou, C. Nicopoulos, and G. Dimitrakopoulos, "ElastiStore: Flexible elastic buffering for virtual-channel-based networks on chip," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 23, no. 12, pp. 3015–3028, Dec. 2015.
[10] M. Paschou, A. Psarras, C. Nicopoulos, and G. Dimitrakopoulos, "CrossOver: Clock domain crossing under virtual-channel flow control," in *Proc. IEEE Des. Autom. Test Europe Conf. Exhib.*, 2016, pp. 1183–1188.
[11] D. A. Sherlock, "System with multiple dynamically-sized logical FIFOs sharing single memory and with read/write pointers independently selectable and simultaneously responsive to respective read/write FIFO selections," U.S. Patent 6,269,413, Jul. 31, 2001.
[12] R. W. Apperson, Z. Yu, M. J. Meeuwsen, T. Mohsenin, and B. M. Baas, "A scalable dual-clock FIFO for data transfers between arbitrary and haltable clock domains," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 15, no. 10, pp. 1125–1134, Oct. 2007.
[13] S. Beer and R. Ginosar, "Eleven ways to boost your synchronizer," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 23, no. 6, pp. 1040–1049, Jun. 2015.
[14] D. U. Becker, N. Jiang, G. Michelogiannakis, and W. J. Dally, "Adaptive backpressure: Efficient buffer management for on-chip networks," in *Proc. IEEE 30th Int. Conf. Comput. Des.*, 2012, pp. 419–426.