POLITECNICO MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Hardware-Accelerated Replication on FPGA-Based SmartNICs

Tesi di Laurea Magistrale in
High Performance Computing Engineering- Ingegneria del Calcolo ad Alte Prestazioni

Author: **Kamil Hanna**

Student ID: 231910
Advisor: Prof. Davide Zoni
Co-advisors: Prof. Gianni Antichi

Academic Year: 2024-2025

Abstract

Replication is a fundamental technique in distributed storage systems, ensuring durability and availability even in the presence of failures. Traditional replication protocols are typically implemented in software and executed on general-purpose CPUs, but as data volumes grow and networks accelerate to 100∼Gb/s and beyond, these approaches struggle with latency overheads and high CPU utilization. Recent trends point toward hardware acceleration, with RDMA and programmable SmartNICs offering new opportunities to push replication closer to the network. In high-performance computing (HPC), where massive datasets are distributed across thousands of nodes, efficient replication is equally critical for fault tolerance and checkpointing, yet conventional software-based methods often fall short at scale.

This thesis explores this direction by designing and implementing a custom replication protocol directly in hardware on FPGA-based SmartNICs. The implementation is built on the AMD Alveo U55C platform using the open-source OpenNIC shell, leveraging QDMA over PCIe for host interaction, HBM for high-speed buffering, and QSFP28 links for inter-node communication. The architecture was deployed on a two-node testbed running a leader–follower replication protocol, where leadership is distributed across keys: some updates are led by the first SmartNIC, while others are led by the second. This setup provides a balanced replication scheme that more closely resembles real-world distributed storage systems.

The design was validated through hardware experiments which confirm correct ordering, delivery, and acknowledgment of replicated updates across nodes. Benchmarking shows that the hardware-based protocol achieves replication at line rate with predictable latency, while avoiding the software overhead seen in conventional implementations. Additional experiments evaluate the impact of integrating replication logic with HBM on resource utilization and end-to-end latency.

Keywords: Replication protocols, High Performance Computing, SmartNICs, HBM

Abstract in lingua italiana

La replicazione è una tecnica fondamentale nei sistemi di archiviazione distribuiti, in quanto garantisce durabilità e disponibilità anche in presenza di guasti. I protocolli di replicazione tradizionali sono tipicamente implementati in software ed eseguiti su CPU general-purpose; tuttavia, con l'aumento dei volumi di dati e l'evoluzione delle reti verso velocità di 100∼Gb/s e oltre, questi approcci incontrano difficoltà dovute alla latenza aggiuntiva e all'elevato utilizzo della CPU. Le tendenze più recenti si orientano verso l'accelerazione hardware, con RDMA e SmartNIC programmabili che offrono nuove opportunità per avvicinare la replicazione alla rete. Nell'ambito del calcolo ad alte prestazioni (HPC), dove dataset di dimensioni massive sono distribuiti su migliaia di nodi, una replicazione efficiente è altrettanto cruciale per la tolleranza ai guasti e le operazioni di checkpoint, ma i metodi convenzionali su software risultano spesso insufficienti su larga scala.

Questa tesi esplora tale direzione progettando e implementando un protocollo di replicazione personalizzato direttamente in hardware su SmartNIC basate su FPGA. L'implementazione è realizzata sulla piattaforma AMD Alveo U55C utilizzando la shell open-source OpenNIC, sfruttando il QDMA su PCIe per l'interazione con l'host, l'HBM per il buffering ad alta velocità e i collegamenti QSFP28 per la comunicazione inter-nodo. L'architettura è stata validata su un testbed a due nodi che esegue un protocollo di replicazione leader–follower, in cui la leadership è distribuita in base alle chiavi: alcuni aggiornamenti sono guidati dalla prima SmartNIC, mentre altri dalla seconda. Questo approccio fornisce uno schema di replicazione bilanciato che riflette più da vicino i sistemi di archiviazione distribuiti reali.

Il progetto è stato validato attraverso esperimenti hardware che hanno confermato il corretto ordinamento, la consegna e l'acknowledgment degli aggiornamenti replicati tra i nodi. I benchmark mostrano che il protocollo implementato in hardware raggiunge la replicazione a velocità di linea con una latenza prevedibile, evitando al contempo il sovraccarico software tipico delle implementazioni convenzionali. Ulteriori esperimenti valutano l'impatto dell'integrazione della logica di replicazione con l'HBM sull'utilizzo delle risorse e sulla latenza end-to-end.

Parole chiave: Protocolli di replicazione, HPC, SmartNIC, HBM

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

1 Introduction **1**

2 Background **3**

2.1 Replication Protocols . . . 3
2.2 Storage and Network-Based Replication . . . 4
2.2.1 Storage Replication . . . 4
2.2.2 Network-Based Replication . . . 5
2.2.3 Transport Protocols . . . 5
2.3 SmartNICs . . . 6
2.4 Field Programmable Gate Arrays (FPGAs) . . . 6
2.5 Storage Class Memories (SCM) . . . 7
2.6 Hardware Platform Overview . . . 9
2.6.1 XCU55C Ultrascale+ FPGA . . . 9
2.6.2 HBM . . . 10
2.6.3 PCIe . . . 11
2.6.4 QSFP . . . 12
2.7 OpenNIC . . . 13
2.7.1 QDMA Subsystem . . . 13
2.7.2 CMAC Subsystem . . . 14
2.7.3 AXI Interconnect . . . 14
2.7.4 User Logic Boxes . . . 14
2.7.5 AXI Transactions . . . 15

3 Related Work **17**
3.0.1 Performance Metrics . . . 17
3.0.2 Feasibility Metrics . . . 18
3.1 Review of Replication Protocols Literature . . . 19
3.1.1 Chaapar . . . 23
3.1.2 Active-Memory . . . 23
3.1.3 Tailwind . . . 24
3.1.4 Hyperloop . . . 25
3.1.5 HERMES . . . 26
3.1.6 SWARM . . . 27
3.1.7 MINOS . . . 28
3.1.8 APUS . . . 29
3.1.9 Mu . . . 30
3.1.10 Derecho . . . 31
3.2 SmartNIC Memory Innovations . . . 33
3.2.1 RDMA Design Guidelines . . . 33
3.2.2 PMNet . . . 34
3.2.3 NIC Flush . . . 35
3.2.4 DFI . . . 36
3.2.5 StRoM . . . 37
3.2.6 SODA . . . 38

4 Architecture **41**
4.1 OpenNIC with HBM . . . 41
4.1.1 HBM . . . 42
4.1.2 Memory Controller . . . 42
4.2 OpenNIC with Replication . . . 43
4.2.1 Key-Value Store . . . 44
4.2.2 Replication Engine . . . 44
4.2.3 Parser and Deparser . . . 44
4.2.4 Packet Structure . . . 45

5 Experimental Results **47**
5.1 Evaluation Metrics . . . 47
5.1.1 Latency . . . 47
5.1.2 Resource Utilization . . . 47
5.2 Experimental Setup . . . 48
5.2.1 Simulation Setup . . . 48

5.2.2 Hardware Setup . . . 49
5.2.3 ILA . . . 50
5.2.4 Benchmark Environment . . . 50
5.3 Simulation Case 1: CMAC - Host Memory Communication . . . 51
5.3.1 Operation Flow . . . 51
5.3.2 Simulation Results . . . 52
5.3.3 More Fine-Grained Results . . . 53
5.4 Simulation Case 2: CMAC - HBM Communication . . . 55
5.4.1 Operation Flow . . . 55
5.4.2 Simulation Results . . . 56
5.4.3 More Fine-Grained Results . . . 57
5.5 Simulation Case 3: HBM - Host Memory Communication . . . 59
5.5.1 Operation Flow . . . 59
5.5.2 Simulation Results . . . 60
5.5.3 More Fine-Grained Results . . . 61
5.6 Prototyping . . . 63
5.6.1 Functional Correctness . . . 63
5.6.2 Prototype Results . . . 63
5.7 Benchmarking . . . 66
5.7.1 Vanilla OpenNIC Benchmark . . . 67
5.7.2 Software replication Benchmark . . . 68

6 Conclusions **71**

Bibliography **73**

List of Figures **77**

List of Tables **79**

Acknowledgements **81**

1 | Introduction

Distributed systems form the backbone of modern data-intensive applications, scaling out computation and storage across many machines to meet the demands of users and workloads. Scaling, however, comes at a cost: extra hardware and the performance overhead of keeping data consistent and available. Replication is central to this problem. By maintaining multiple copies of data across nodes, systems improve availability and fault tolerance, but replication protocols also sit directly on the critical path of performance. They introduce latency from two main sources: coordination among nodes, and persistence to storage.

The coordination cost arises from ordering, acknowledgments, and failure recovery, which traditionally have been handled entirely in software on general-purpose CPUs. The persistence cost comes from ensuring durability, often involving synchronous storage operations that add delays to every write. Both costs become more pronounced as data volumes grow and networks accelerate. Early distributed systems research proposed the "fat client" idea, where clients take on some replication responsibilities instead of relying solely on servers. While not widely adopted then, this idea has found new life in today's programmable SmartNICs, which allow parts of replication logic to move from the CPU into the NIC, closer to the network. In effect, the NIC becomes a modern fat client, offloading tasks such as packet handling, ordering, and consistency checks.

This thesis explores that direction by building a hardware-based replication protocol on an FPGA SmartNIC. Our starting point was the open-source *OpenNIC* shell on AMD's Alveo U55C platform, which integrates High Bandwidth Memory (HBM) and QSFP28 networking. We first added HBM support to OpenNIC and evaluated access paths through PCIe and QSFP. Measurements showed that PCIe traversal introduced significantly more latency than NIC-to-NIC communication over QSFP, leading us to design the architecture around a NIC-first data path. From there, we turned to replication itself, asking what kind of protocol could run efficiently in hardware and what abstractions were needed to handle consistency and coherence.

The resulting design separates concerns into two layers. At the software level, we assume an **application layer** that enforces per-key locks and leader selection: each key is owned by a leader NIC, which coordinates updates with its followers. At the hardware level, a **replication layer** implemented fully on the FPGA processes, forwards, and applies updates at line rate, using HBM as the primary working memory. This split keeps control decisions in software while keeping the data path fast and predictable in hardware. It also reflects a broader trend in SmartNIC design, where large on-board memories and reconfigurable logic make NICs capable of handling far more than packet I/O.

The contributions of this thesis are threefold. First, we present a baseline replication architecture fully implemented in hardware on the U55C SmartNIC, integrated with HBM and operating at 100∼Gb/s. Second, we evaluate our design, by analyzing how adding replication logic changes the cost profile of a vanilla OpenNIC shell, in terms of resources and latency. Third, we compare the hardware approach against a software-based replication model to highlight the efficiency of running replication in hardware.

Together, these results demonstrate both the feasibility and the benefits of pushing replication into SmartNIC hardware, reducing overheads that otherwise limit distributed storage and even high-performance computing workloads.

The remainder of this thesis is structured as follows. Chapter 2 provides the concepts needed to grasp the work of our thesis such as Storage Replication , SmartNICs, FPGAs, and Storage Class Memories, OpenNIC, and other. Chapter 3 reviews related work in terms of replication protocols and SmartNIC innovations from the literature. Chapter 4 presents the design of our replication architecture. Chapter 5 reports our experimental results, covering both simulation, and replication baseline hardware results, in addition to our benchmarks. Finally, Chapter 6 discusses limitations, outlines future directions, and concludes the thesis.

2 | Background

To fully comprehend the goal of this thesis, it is essential to grasp foundational concepts in both theoretical and technical aspects in the field of replication protocols, SmartNICs, and FPGAs. This chapter is dedicated to introducing key concepts, providing the necessary back-ground knowledge that ensure a clear understanding of the work.

2.1. Replication Protocols

Replication protocols are the mechanisms that keep multiple copies of data in sync across a system. Their purpose is simple, guarantee the durability and availability of data, even when parts of the system fail. The way the later is achieved, depends on the guarantees required. Replication protocols, sit at the heart of distributed databases, cloud storage platforms, and large-scale file systems, shaping how updates are propagated and how quickly a system can recover from a fault.

The most common protocol designs, include **leader-based replication**, where a designated node (the leader) orders all writes and ensures they are propagated to followers; **primary–backup**, which is a simple leader-follower variants ; **quorum-based** approaches such as Paxos or Raft, which rely on a majority vote to agree on state; and lighter **peer-to-peer synchronization**, where nodes exchange updates more loosely to maximize availability. Each style reflects a trade-off between latency, throughput, fault tolerance, and the level of consistency that clients will observe. In practice, leader-based replication is widely used in relational databases such as PostgreSQL or MySQL, where a single node coordinates updates and pushes them to followers. Quorum-based approaches, on the other hand, are found in systems like etcd or Consul, which use Raft to ensure consistency among configuration replicas in cloud platforms.

Historically, replication protocols were built entirely in software, integrated into the storage or the database engine and executed on general-purpose CPUs. Although this provided flexibility, it also consumed significant compute resources for tasks such as message handling, log serialization, and coordination. These tasks required to run the protocol,

were competing with the application workloads. Thus, inducing overhead which turned into bottlenecks and making the application efficiency lower. This has even become worse in the world of big data, where data volumes grew and network became faster. To address this, modern systems increasingly lean on he hardware support. **RDMA** enables direct memory-to-memory transfers between servers with minimal CPU involvement, while **programmable SmartNICs** make it possible to offload packet handling, ordering, or even parts of the replication logic into hardware. These hardware-assisted approaches aim to deliver replication at line rate with predictable latency, freeing the host CPU for higher-level tasks and making replication more efficient at scale.

2.2. Storage and Network-Based Replication

While replication protocols define how updates are coordinated across replicas, their practical implementation depends on the mechanisms used to store, transmit, and acknowledge data. In modern systems, this spans multiple layers: local storage replication techniques that guarantee persistence and consistency, network-based mechanism that propagate updates across servers, and transport protocols that carry replication traffic. Together, these layers determine not only the reliability of replication but also its performance and scalability.

2.2.1. Storage Replication

Storage replication ensures that data remains durable and available even in the presence of hardware failures, crashes, or network partitions. The two dominant modes are **synchronous** replication, where a write is acknowledged only after all replicas confirm it, and **asynchronous** replication, where a write is acknowledged immediately and shipped to replicas in the background. Synchronous replication provides strong consistency guarantees at the cost of higher latency, while asynchronous replication offers lower latency but risks data loss during failures. These trade-offs are often expressed through the *Recovery Point Objective* (RPO), which defines how much data might be lost in a failure, and the *Recovery Time Objective* (RTO), which measures how quickly a system can recover.

Consistency models refine these guarantees further. Strong consistency ensures that every read reflects the latest committed write, while eventual consistency allows temporary divergence with the expectation of later convergence. Between these extremes, models such as causal consistency or read-your-writes provide middle grounds that balance availability and responsiveness. Persistency also plays a role: updates are usually logged or staged in non-volatile storage before acknowledgment, ensuring durability even after crashes.

2.2.2. Network-Based Replication

When replication extends across multiple servers or even data centers, the network becomes a critical part of the storage pipeline. Updates are transmitted as network messages, typically carrying write requests or log records, which must be delivered reliably and in order. Depending on the mode, acknowledgments may be required before a write is considered committed (synchronous) or can be deferred until later (asynchronous). Beyond correctness, practical issues such as congestion, flow control, and packet loss must be managed to avoid bottlenecks.

Failures introduce another challenge: replicas that lag behind must resynchronize with the rest of the system. This often involves replaying change logs or applying snapshots to bring them back into a consistent state. These mechanisms highlight how closely replication performance depends on network bandwidth, latency, and reliability.

2.2.3. Transport Protocols

At the transport layer, most replication traffic relies on **TCP** [21], which guarantees reliable, in-order delivery and is universally supported. While TCP ensures correctness, it also introduces latency through retransmissions, congestion control, and head-of-line blocking. To mitigate this, storage systems often tune TCP—for example, by disabling Nagle's algorithm or adjusting buffer sizes.

Some systems instead leverage **UDP** [20], which provides a lightweight, connectionless transport with minimal overhead. While UDP does not guarantee reliability or ordering on its own, these features can be implemented at the application or NIC level, providing designers more flexibility. Protocols like **QUIC**, which build up on UDP, add congestion control and reliability in user space and are increasingly being explored in modern data services.

Beyond TCP and UDP, technologies such as **RDMA** (RoCE, iWARP) allow zero-copy, kernel-bypass data transfers, reducing both latency and CPU overhead. Emerging protocols like **QUIC**, which builds on UDP, provide flexible congestion control and user-space deployment, though they remain less common in storage replication today. In addition, programmable SmartNICs enable custom transport logic to be implemented directly in hardware, allowing replication traffic to run at line rate with predictable latency and reduced host involvement.

2.3. SmartNICs

SmartNICs, or Smart Network Interface Cards [12], are specialized network adapters that extend beyond traditional packet I/O providing programmable acceleration capabilities directly on the NIC. These special NICs emerged as a response to the growing computational load of specialized network functions, especially in data centers, cloud computing and supercomputers. CPUs in computationally demanding environments are either highly scarce or monetized. This is where SmartNICs really shine, as it offers offloading and accelerating network processing tasks (e.g. filtering, encryption..) from the main CPU. Such, leads to freeing up the host CPU to focus on other critical workloads but also enhances overall system performance and reduces latency. Essentially, SmartNICs act as co-processors for the network, making them a key enabler of modern, scalable, and efficient infrastructure.

SmartNICs can be broadly categorized into three types: software-programmable NICs, which rely on embedded processors; ASIC-based NICs, which achieve high efficiency but lack flexibility; and FPGA-based NICs, which balance programmability and performance by allowing custom logic to be deployed. FPGA-based SmartNICs have recently gained popularity in the research and innovation field, as they enable rapid prototyping of user-specific tasks without restrictions. Furthermore, in the recent years; SmartNICs use in distributed storage and replication systems have soared, due to the need of moving large amounts of data in the network. This trend reflects a broader industry shift toward in-network computing as a way to overcome CPU and I/O bottlenecks in data-intensive infrastructures.

2.4. Field Programmable Gate Arrays (FPGAs)

Field Programmable Gate Arrays (**FPGAs**) are reconfigurable integrated circuits that offer a unique combination of flexibility and performance, making them increasingly important in the era of exponential data growth and cloud computing.

An FPGA is built from a matrix of Configurable Logic Blocks (CLBs), each containing lookup tables, flip-flops, and multiplexers, which can be interconnected through a reconfigurable routing fabric. Additionally, modern FPGAs introduce more specialized resources such as Digital Signal Processing (DSP) slices, embedded Block RAM (BRAM), storage class memories (SCMs) and far more complex hardware resources. FPGAs can repeatedly be reprogrammed by developers using Hardware Description Languages (HDLs) such as VHDL, Verilog, or SystemVerilog, with VHDL being the first HDL in the industry,

introduced by the Department of Defense (DoD) 1985.

FPGAs occupy a middle ground between software-programmable CPUs and fixed-function ASICs, combining much of the performance of hardware specialization with the flexibility of re-programmability. This balance of speed and adaptability has led major cloud providers, such as Microsoft, to deploy FPGAs at scale for accelerating search, AI, and network processing in their data centers. Moreover, FPGAs have been also adopted in domains that require critical workloads such as aerospace, medical imaging, automotive systems and security.

Historically, the FPGA was first introduced in 1985 by Ross Freeman, co-founder of Xilinx, who envisioned a chip that could be programmed after manufacturing. The concept quickly gained attraction with early FPGAs offering hundreds of thousands of logic gates. Over time, new companies such as Altera (Intel today), and Actel helped popularize the technology, while Xilinx established itself as a leader in the space. Today, FPGAs come in multiple form factors ranging from small development kits and embedded SoCs to PCIe accelerator cards for data centers and FPGA-based SmartNICs for networking and storage offload featuring millions of logic elements, hardened memory and DSP blocks, and high-speed transceivers, making them indispensable in both embedded systems and large-scale cloud infrastructures.

2.5. Storage Class Memories (SCM)

Storage Class Memories **(SCM)** [14] represent a new class of non-volatile memory technologies that bridge the gap between volatile main memory and persistent storage. They combine the high bandwidth and low latency of traditional **DRAM** (Dynamic Random Access Memory) with the persistence, cost-effectiveness, and density of **NAND** Flash memories (e.g. SSDs). In other words, SCMs deliver high-speed data access technologies at the price tag of standard storage solutions. The non-volatile nature of SCMs, unlike power-dependent DRAMs for instance, gives it the special feature of persistency. Persistency is an important feature in memories, which ensures data retention during power outages of system crashes.

SCMs were initially designed to address the growing demands of data-intensive systems, which require memory technologies capable of extremely fast data access. They provide a promising solution solution in these environments, where conventional memory hierarchies struggle to keep pace.

SCMs occupy a unique space within the storage spectrum and several technologies fall

under it's umbrella. Intel Optane DC Persistent Memory, based on 3D XPoint, offers byte-addressable persistence, with near-DRAM latency. Phase-Change Memory (PCM), has been adopted as another candidate, leveraging material state changes to store data. High Bandwidth Memory (HBM), although volatile, is also often categorized as an SCM due to its stacked-die architecture and it's parallelism which offer extremely high transfer rates.

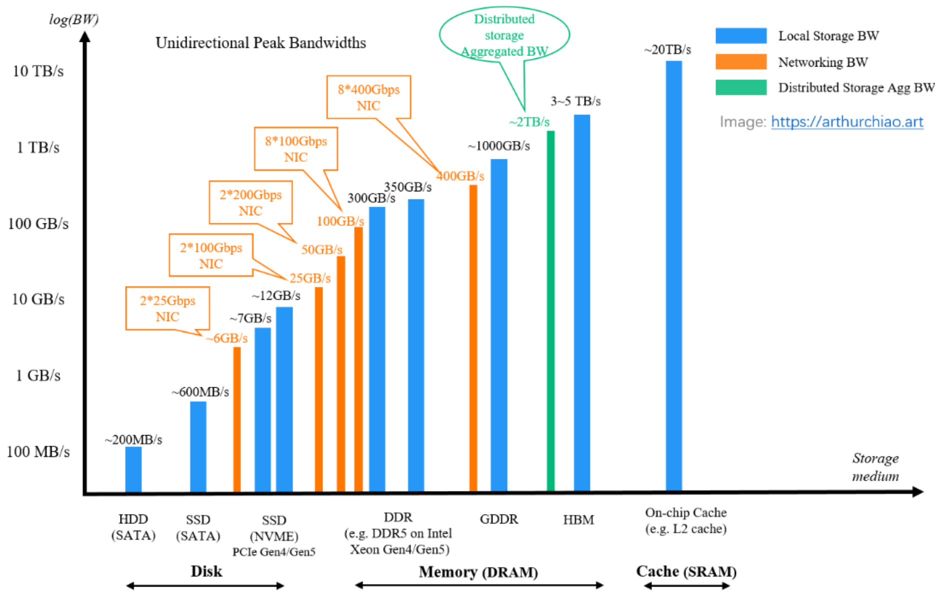

Figure 2.1: Peak bandwidth of storage media, networking, and distributed storage solutions. [5].

In modern FPGAs, SCMs have been widely adopted to meet the demand of system designers, with FPGA-based SmartNICs being a prominent example. The Alveo U55C, for instance, integrates HBM2, which delivers up to 460∼GB/s of aggregate bandwidth through its parallel interfaces, enabling the accelerator to buffer and process large data streams efficiently.

What makes SCMs unique is that they combine the speed of memory with the durability of storage. Some of their most important features include:

- **Persistence**: SCM retains data even when power is removed, eliminating the need for constant power, unlike DRAM.
- **Low Latency**: SCM offers significantly faster read and write speeds compared to NAND flash, often reaching microsecond or even nanosecond access times, approaching DRAM-like performance.

- **High Endurance**: SCM can withstand a far greater number of write cycles compared to NAND flash, extending its lifespan and reducing wear-out concerns.
- **Byte Addressability**: SCM allows data to be accessed and manipulated at the byte level, similar to DRAM, enabling more efficient data management.
- **Direct Memory Access**: SCM can be directly integrated into the memory bus of a system, bypassing traditional I/O bottlenecks and enabling data to be accessed as seamlessly as DRAM.

2.6. Hardware Platform Overview

The experimental hardware platform used in this work is the AMD Alveo U55C, a SmartNIC and accelerator card built on the Xilinx UltraScale+ FPGA architecture. This section introduces the U55C itself and its significant key sub systems namely, the **HBM**, **PCIe**, and **QSFP**, which are central to our replication architecture.

2.6.1. XCU55C Ultrascale+ FPGA

The AMD Alveo U55C Data Center Accelerator card is a high-performance computing solution designed for data-intensive workloads [3]. It is powered by AMD's Xilinx UltraScale+ FPGA, which provides customizable hardware acceleration for applications such as artificial intelligence, machine learning, big data analytics, storage systems, and high-performance computing (HPC).

A key feature of the U55C is its integrated High Bandwidth Memory (HBM2) subsystem, which provides significantly higher memory bandwidth compared to traditional DDR memory architectures by connecting stacked memory directly to the FPGA fabric. This feature makes the U55C ideal for workloads that require large scale data processing, due to its fast data access and reduced memory latency. In addition, the board integrates a PCIe which it uses to connect to a host system. It can operate either a PCIe Gen3x16 or as a Gen4x8 link, both providing comparable peak bandwidth of approximately 16∼GB/s. The operation mode depends on the capabilities of the host system and the deployed FPGA shell, only one mode is active at a time. For network connectivity, the U55C integrates two QSFP28 ports, each capable of operating at up to 100∼Gb/s. These ports can be configured to support a range of Ethernet standards, and can achieve an aggregate throughput of 200∼Gb/s.

The U55C is optimized for cloud deployments and modern data centers, offering flexibility and scalability for various computing environments. Its combination of HBM bandwidth,

PCIe connectivity, QSFP-based networking, and FPGA technology makes it a powerful solution for accelerating complex network tasks, while reducing the overall power consumption and footprint compared to other traditional accelerators.

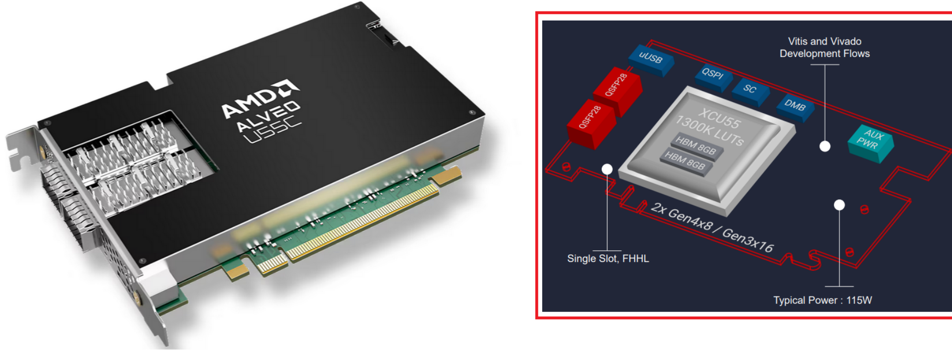

Figure 2.2: The Alveo U55C Board [3].

Category	**Specification**
Compute Resources	Look-up Tables (LUTs): 1304K Registers: 2607K DSP Slices: 9024
Dimensions	Height: Full Height Length: Half Length Width: Single Slot
Memory	HBM Memory Capacity: 16 GB HBM Total Bandwidth: 460 GB/s Internal SRAM Capacity: 43 MB Internal SRAM Total Bandwidth: 35 TB/s
Physical Interfaces	PCI Express: Gen3 x16, 2 x Gen4 x8 Network Interfaces: 2x QSFP28
Tool Support	Vitis Developer Environment: Yes Vitis Platform: Gen3 x16 XDMA Vivado Design Suite: Yes
Power & Thermal	Maximum Total Power: 150W Typical Power: 115W Thermal Cooling: Passive
Target Workloads	Big data analytics and search, financial computing, computational storage, and machine learning

Table 2.1: Compute Resources and Specifications of AMD U55C Board.

2.6.2. HBM

High Bandwidth Memory (**HBM**) [2] is a revolutionary high performance volatile memory technology that falls under the category of Storage Class Memory (**SCM**). It is designed to deliver significantly higher bandwidth compared to traditional memory solutions such as DDR or GDDR. It achieves this by stacking multiple DRAM dies vertically and connecting

them through-silicon vias (TSVs), which enable massive parallelism and reduced signaling distances. HBM has been adopted in a variety of domains including graphics accelerators, FPGAs, networking devices, and supercomputers, where large data volumes must be processed with low latency and high energy efficiency.

The U55C employs **HBM2**, the second generation of the standard. Each HBM2 stack can contain up to eight DRAM dies and exposes a wide 1024-bit interface, delivering data transfer rates of up to 256∼GB/s per stack. On the U55C, the FPGA fabric connects to HBM2 through 32 independent AXI (Advanced eXtensible Interface) channels (16 per stack), each 256 bits wide. This setup enables highly parallel data accessing, efficient data movement, and an aggregate bandwidth of up to 460∼GB/s. Such characteristics make the HBM2 particularly highly attractive and competitive with other memory solutions when it comes to data-intensive applications, such as in-network replication, where throughput and low memory latency are critical.

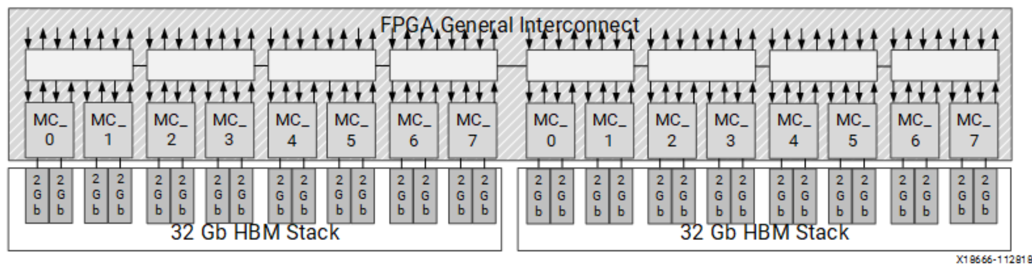

Figure 2.3: 64GB HBM Two Stack Configuration [2].

2.6.3. PCIe

Peripheral Component Interconnect Express (**PCIe**) is the dominant high-speed serial interconnected for connecting accelerators, network cards, storage devices and other peripherals to the host system. It uses a point-to-point topology and a layered protocol stack consisting of the transaction, data link, and physical layers [15]. Communication is packet-based, with headers carrying control and addressing information alongside the payload.

The transaction layer supports two classes of operations: posted and non-posted. Posted operations are said to be fire-and-forget. For instance, memory writes, since they can be assumed complete once a request is sent, without any further interaction needed. Non-posted operations, such as memory reads, require a completion packet to be returned, meaning the requester must track outstanding requests. The later, is famous for introducing stalls; which is why they are avoided whenever possible, especially in performance

critical systems. Reliability is ensured by the data link layer, which provides acknowledgments, CRC checking, and credit-based flow control to guarantee that each endpoint has sufficient buffer space to receive packets. The data link layer is known to add around 8% to 10% overhead to the communication.

An important feature of PCIe is **bus mastering**, which allows peripherals such as FPGAs to directly access host memory using Direct Memory Access (DMA). This capability is central for accelerator cards, enabling data movement between host memory and the device without CPU intervention. This feature also allows PCIe to support message-signaled interrupts, allowing devices to notify the host efficiently by writing to special interrupt registers.

PCIe enforces reordering of packets to preserve correctness, with a few rules [16]. Posted write requests are guaranteed to arrive in the order they were sent, but may be completed before outstanding read requests. Meanwhile, a read request will never bypass a previously issued write. Read completions belonging to the same request are also guaranteed to arrive in order, but beyond these constraints, PCIe transactions can be freely reordered by the interconnect.

To manage resource availability, the data link layer implements flow control through a credit-based system. Each endpoint advertises its buffer capacity when the link is established, and every transmitted packet consumes credits until it is acknowledged. This mechanism ensures that no side overwhelms the other with more traffic than it can store, while still allowing the link to be utilized efficiently.

Although PCIe provides high throughput, it can become a limiting factor as network interfaces scale. Prior work has shown that even at 40∼Gb/s, PCIe may begin to deteriorate performance, acting as a bottleneck in certain NIC designs [19], even with optimized software stacks.

2.6.4. QSFP

Quad Small Form-Factor Pluggable (**QSFP**) is a standard for compact, hot-swappable transceivers that support high-speed data communication over optical fibers or copper cables. It is widely used in modern data centers and high-performance networks, supporting Ethernet standards such as 40 GbE and 100 GbE. The popularity of QSFP stems from its ability to deliver high throughput in a small form factor, while maintaining flexibility through support for different signaling and physical media options [17].

On the U55C, two QSFP28 ports are provided, each capable of sustaining Ethernet links

up to 100~Gb/s [8]. These ports are connected internally to hardened 100 GbE **CMAC (Configurable Media Access Controller)** blocks, which implement low-level Ethernet functions such as framing, flow control, and error detection directly in FPGA hardware. With an aggregate bandwidth of 200~Gb/s, the combination of QSFP28 interfaces and CMACs enables the U55C to achieve line-rate packet processing. This makes the platform well-suited for storage replication workloads, where large volumes of traffic must be reliably moved across the network with minimal latency.

2.7. OpenNIC

OpenNIC is an open-source FPGA-based SmartNIC (Smart Network Interface Cards) project developed by Xilinx. Unlike fixed-function NICs, OpenNIC is fully customizable. Built on Xilinx's FPGA technology, it provides flexible and programmable networking that allows offloading of computationally expensive tasks. Engineers, can design their own network functions or accelerator hardware and include them in the OpenNIC platform, making their systems faster and more efficient. For instance, developers can design their own IPs for tasks such as packet processing, encryption, and traffic management, and include them in OpenNIC to offload them from the main CPU. This approach allows the NIC to be tailored to specific needs, whether for data centers, cloud computing, or other high-performance applications. OpenNIC's open-source nature also promotes transparency and collaboration, making it an attractive choice for both research and production use cases.

In our work, we employ OpenNIC as the foundation for building our replication hardware baseline on the Alveo U55C. The framework exposes high-speed interfaces to the FPGA fabric, including the **QDMA** engine for PCIe host communication, **CMAC** blocks for 100 GbE networking, and a rich set of **AXI** interconnects for connecting user logic to memory and streaming data paths. These components allow us to integrate custom replication logic into the SmartNIC pipeline while relying on proven infrastructure for host and network I/O.

2.7.1. QDMA Subsystem

The Queue Direct Memory Access (**QDMA**) subsystem in OpenNIC connects the FPGA fabric to the host system over PCIe. It supports both **AXI-Stream** and **AXI Memory-Mapped** interfaces, enabling data to be exchanged either as continuous streams or through direct memory transactions. Standard host-to-card (H2C) and card-to-host (C2H) channels are exposed to user logic, while internal blocks manage traffic flow and

queue assignment. The subsystem can support up to four physical functions, each of which appears as a network device in the host operating system, allowing multiple independent data paths to coexist. This design enables OpenNIC to sustain high-throughput communication between host applications and the SmartNIC pipeline.

2.7.2. CMAC Subsystem

The Configurable Media Access Controller (**CMAC**) subsystem in OpenNIC provides the high-speed Ethernet interface through QSFP28 ports. Up to two CMAC instances can be instantiated, each handling one 100 Gb/s link. The subsystem uses **AXI-Stream** interfaces for packet data, operating on a 322 MHz data path clock. To align with the QDMA side of the pipeline, a packet adapter bridges the clock domains and ensures that packets are fully buffered before transmission, as the CMAC requires complete frames to be delivered without interruption. In the receive direction, the subsystem buffers incoming packets and performs error checking, discarding corrupted or incomplete frames. Together, the CMAC subsystems provide the network-facing endpoints of OpenNIC, enabling reliable, line-rate Ethernet communication directly from the FPGA fabric.

2.7.3. AXI Interconnect

The AMBA Advanced eXtensible Interface (**AXI**) family of protocols forms the internal interconnect within OpenNIC. The design is primarily based on **AXI-Stream**, which provides high-throughput, low-latency channels for packetized data as it flows through the SmartNIC pipeline. For configuration and control, **AXI-Lite** is used, enabling lightweight register access. Memory-mapped transactions are supported via **AXI-MM**, allowing user logic to perform direct reads and writes to memory resources. This combination of AXI protocols gives OpenNIC a modular and scalable structure, making it easier to integrate custom accelerators into the SmartNIC framework.

2.7.4. User Logic Boxes

OpenNIC provides dedicated **user logic boxes** as insertion points for custom hardware plugins. There are two such boxes: one operating at 250MHz between the QDMA subsystem and the packet adapters, and another operating at 322MHz between the CMAC subsystems and the packet adapters. Each box exposes standard **AXI-Stream** interfaces for data and an **AXI-Lite** control interface running at 125MHz, allowing user modules to be integrated without modifying the core subsystems. These boxes give developers the flexibility to implement application-specific functions—such as packet filtering, monitor-

ing, or acceleration—directly within the SmartNIC data path while leveraging OpenNIC's existing infrastructure for host and network I/O.

2.7.5. AXI Transactions

The Advanced eXtensible Interface (**AXI**) protocol, part of the AMBA standard, is the primary interconnect mechanism inside OpenNIC and across many FPGA designs. It defines how components exchange data and control information, providing flexibility while maintaining high throughput. In OpenNIC, different subsystems rely on AXI protocols depending on the nature of the data transfer: **AXI-Stream** for packetized flows and **AXI Memory-Mapped** for direct memory access.

AXI-Stream

AXI-Stream is optimized for high-bandwidth, low-latency data transfer where information is sent as continuous streams of packets without explicit addresses. It is widely used in OpenNIC to connect subsystems such as QDMA, CMAC and user logic. Streaming interfaces ensure that packet data can move efficiently through the pipeline, making it the backbone of the SmartNIC data path.

AXI Memory-Mapped

AXI Memory-Mapped (AXI-MM) enables direct read and write transactions to specific memory locations. Unlike AXI-Stream, each transaction carries an address along with the data, making it well suited for memory subsystems. Multiple versions of the AXI-MM protocol can be found, for instance in OpenNIC, AXI4-MM is used by the QDMA to exchange data with host memory over PCIe, while an HBM memory is accessed via AXI3-MM. AXI-MM allows accelerators to handle both packetized streams and memory-oriented transfers within a unified framework.

3 | Related Work

In this chapter, we analyze the existing literature that is relevant to our work. The papers discussed are evaluated, highlighting their contributions and limitations. The first two sections of this chapter introduce evaluation metrics used to evaluate the performance and feasibility of a solution. The third section targets replication protocols, where an in-depth analysis is presented to highlight critical design tradeoffs. Finally, the fourth section targets SmartNIC memory innovations, where a general analysis of different literature is presented.

3.0.1. Performance Metrics

Throughput

Throughput describes how many operations a system can complete in a given amount of time. In the context of memory persistence and replication protocols, it reflects the system's ability to efficiently process and replicate memory updates. For performance-sensitive applications, high throughput indicates that the system can handle a larger workload. Throughput is commonly expressed as operations per second (op/s), and in high-performance environments it is typically reported in kilo operations per second (**kops**).

$$\text{Throughput (THR)} = \frac{\text{Total Operations}}{\text{Total Time (seconds)}}$$

Latency

The term "latency" describes the amount of time that passes between a request being made and its fulfillment. It encompasses the duration required to maintain data across replicas in replication procedures. For real-time or near-real-time applications, lower latency is essential since it increases system responsiveness. Latency is commonly measured using average latency, which reflects the typical response time, and tail latency (e.g., 99th percentile), which highlights worst-case delays under load or failures.

$$L_{\text{avg}} = \frac{1}{N} \sum_{i=1}^{N} (T_{\text{response},i} - T_{\text{request},i})$$

$$L_{\text{tail}} = \text{Percentile}_{99} \left(\{T_{\text{response},i} - T_{\text{request},i}\}_{i=1}^{N} \right)$$

Memory overhead

The term "Memory overhead" refers the additional memory consumed by a system to support the replication protocol functionality. In other words, it is an extra chunk of data beyond the actual application data that is being stored in the memory. This can include metadata, logs, indexes, versioning information, acknowledgment tracking, coordination buffers, fault tolerance, and others. Memory overhead is commonly measured in bytes, and can be calculated using the following formula.

$$\text{Memory Overhead} = \frac{\text{Auxiliary Memory}}{\text{Data Memory}}$$

where:

- Auxiliary Memory = Memory used for replication metadata (e.g., logs, indexes..).
- Data Memory = Memory required to store the actual data (without replication overhead).

3.0.2. Feasibility Metrics

CPU Utilization

This indicator assesses how well a replication protocol makes use of CPU resources. It aids in evaluating the cost-performance trade-offs. High utilization may be a sign of effective operation, but it also raises the possibility of overloading. Meanwhile, Lower values indicate more efficient offloading of work from CPUs to other subsystems such as NICs or accelerators.

$$\text{CPU Utilization (\%)} = \frac{\text{CPU Time Used}}{\text{Total CPU Time Available}} \times 100$$

3.1. Review of Replication Protocols Literature

Past research has established taxonomies for classifying replication protocols [7]. Building on this foundation and examining recent advancements in replication protocol optimizations, we conducted a structured analysis of the state-of-art. Our goal is to provide a clear, comparative perspective on how different architectural choices, consistency models, hardware utilization influence performance in replication systems. To achieve this, we conducted an in-depth comparison of different replication protocols from the available literature. Furthermore, to make the results interpretable, we clustered related concepts and notions and built a table for each cluster.

In this section, we present three comparative tables. The first table represents the taxonomy of replication protocols and is split into four quadrants as shown below. The quadrants are based on two operational patterns, leader-based (L) vs decentralized (D) and total order (TO) vs per-key order (PKO). This results in four classes of protocols :

- **LTO**: Leader-based Total Order
- **LPKO**: Leader-based Per-Key Order
- **DTO**: Decentralized Total Order
- **DPKO**: Decentralized Per-Key Order

	Total order	**Per key order**
Leader-based	Mu [1], APUS [28], Tailwind [25]	Chaapar [10], Active-Memory [30], Hyperloop [13]
Decentralized (Leaderless)	Derecho [9] , MINOS [22]	SWARM [18], HERMES [11] , Chaapar [10]

Table 3.1: Taxonomy of replication protocols from the literatures

It's important to note that all replication protocols discussed in the literature exhibit strong consistency.

Building on the frame work from the first table, we move to the second table which provides a detailed comparative analysis of recent replication protocol architectures presented in the literature. This table examines five key dimensions, the first being system

architecture, which captures the type of hardware used in the solution. The second being interface, which refers to the systems operational database-replication technology. The method, describes the most important key innovation utilized in the solution, while communication protocol defines the transport mechanism between nodes in the system. Lastly persistency support, describes the durability model or in other words, if the model of the solution provides any persistency and open source tells weather the work has been made open to the public or not. The latter is a crucial consideration for adoption and verification.

Most literature focused on a CPU-based architecture and software-centric solutions, rather than a hardware-based approaches involving SmartNICs. The CPU based solutions, leveraged RDMA NICs from the Mellanox ConnectX Family to improve replication performance, although the proposed solutions are generally applicable to any RDMA-capable NIC. In contrast, SmartNIC-based solutions such as in [10] and [22]) typically utilized the NVIDIA BLueField DPU; however, the focus in these cases remains largely on software configuration and application-level offloading. The BlueField DPU in these papers, was mostly treated as a pre-designed, fixed-function hardware platform with no modification to its underlying architecture. As such, we can say that the majority of these efforts to improve the replication protocols were encapsulated within the software domain.

This highlights the rarity of hardware-based replication solutions in existing research, which is beneficial to our work since it explores relatively an unexplored direction- replication implemented at the hardware level.

Title	System Architecture	Interface Type	Method	Communication Protocol	Persistency Support	Open-Source
Chaapar [10]	SmartNIC	Key-value interface, through SMR with dynamic membership	Cache write requests	Two sided RDMA	Yes via NVMe, with batching to host durable storage	No
Hermes [11]	CPU	Key-value interface, through SMR with, a membership service	Invalidation and timestamps	Two sided RDMA	No persistency, pure in-memory replication	Yes
Tailwind [25]	CPU	Key-value interface, through SMR	Metadata for integrity	One-sided RDMA for replication, & RPCs for control	Yes via battery backed RAM (UPS proteted) & flushing to SSD	No
Active Memory [30]	CPU	Key-value interface, with cross-key transactions with membership service	Undo logging	One sided RDMA	Yes via NVM	No
Hyperloop [13]	CPU	Key-value interface, through SMR	Group primitives	Hybrid One-sided RDMA, with coordi -nated group semantics	Yes via NVM, battery backed DRAM	No
Swarm [18]	CPU	Key-value interface	Speculative-Timestamping	One-sided RDMA	No persistency, pure in-memory replication	Yes
MINOS [22]	SmartNIC	Key-value interface through SMR	Persistency & consistency offloading	Two-sided RPC messaging	Yes via NVM	No
APUS [28]	CPU	Key-value interface, through SMR, with membership service	Persistency offloading	Primarly one sided RDMA for consensus, two sided for failure recovery	Yes via SSD-backed storage, with periodic checkpoints enabling recovery	Yes
Mu [1]	CPU	Any type, generic SMR	Lock-free SMR via RDMA	One-sided RDMA	No persistency, pure in-memory replication	Yes
Derecho [9]	CPU	Key-value interface through SMR, with a membership service	Lock-Free SMR via RDMA	One-sided RDMA by default, but swaps to two-sided RDMA for failure handling	Yes via durable writes to SSD/NVM with recovery guarantees	No

Table 3.2: Analysis of the literature replication protocol architectures, the columns represent : Definition : A brief overview of the protocols design ; System Architecture : Hardware roles in replication ; Interface Type : Compatible storage format with it's replication interface ; Method : The implemented mechanism for replication ; Communication Protocol : Transport mechanism used for data replication ; Support for persistency: Durability of replication ; Open Source : Whether the implementation is publicly available

Finally, the third table wraps up the taxonomy with an in-depth comparison of each solution performance. The metrics presented in the table include latency (average and tail) measured in microseconds (µs), throughput measured in kilo-operations per second (kops) and CPU utilization per NIC. The latter indicates how much the NIC in the proposed solution can offload from the CPU. A lower value means the NIC is more capable of handing tasks independently, reducing CPU workload. With respect to latency, the average and tail latencies are displayed, since they highlight the typical and worst-case performance scenarios. Lastly, throughput, in our case, represents the system's maximum operation processing rate. It is influenced by the interplay of NIC bandwidth (RDMA), CPU and replication protocol efficiency.

Title	Description	Latency (µs)		Through-put (kops)	CPU Util.	Memory overhead (bytes)
		Avg	Tail			
Chaapar [10]	SmartNIC-powered replication protocol that offloads CPU-heavy consistency tasks to the network card.	N/A	N/A	N/A	68%	N/A
Hermes [11]	A low-latency, strongly consistent distributed protocol using RDMA and logical clocks for in-memory data access.	2∼42	15∼69	80∼180	30∼90%	24∼32
Tailwind [25]	Low-latency communication mechanism that operates directly on memory, bypassing CPU and software stacks.	16	28	200∼635	49%	4
Active Memory [30]	RDMA-powered replication protocol that skips backup CPU work by directly writing updates via RDMA.	121	N/A	2400	100%	N/A
Hyperloop [13]	NIC-offloaded protocol that eliminates CPU involvement in distributed transactions by leveraging RDMA and NVM.	10	20∼30	100∼1000	0%	32
Swarm [18])	Leverages RDMA and a novel consensus protocol (In-n-Out) to achieve single-round-trip operations while tolerating failures.	2.4∼3.1	2.8∼4	28000	61.3%	24∼32
MINOS [22]	Optimizes replication protocols by leveraging SmartNICs to offload distributed consistency & persistency protocols.	20∼50	N/A	600∼800	N/A	8∼16
APUS [28]	Uses RDMA to optimize replication protocols while ensuring total ordering through Paxos consensus	8.8	N/A	45	N/A	N/A
Mu [1]	Replaces traditional consensus rounds with direct RDMA log writes, uses RDMA permissions for lock-free synchronization.	1.3	1.6	47000	N/A	N/A
Derecho [9]	Optimizes replication protocols by leveraging RDMA to offload data movemenent accelerate consistency mechanisms.	1.5	N/A	1000	0%	8

Table 3.3: Analysis of Replication Protocols Literature, the columns represent : Description : A brief overview of the protocols design ; Latency (µs) : Average & tail latencies measured in microseconds ; Throughput (kops) : System throughput in thousands of operations per second ; CPU Util : CPU utilization during operation ; Memory overhead : Memory overhead measured in bytes per key/object

3.1.1. Chaapar

In Reliable Replication Protocols on SmartNICs [10], the authors propose a work-in-progress method to improve the offloading of replication protocols to SmartNICs. Their goal is to improve leaderless protocols while minimizing latency. As a matter of fact, their contribution is not a new replication protocol per se, but rather a protocol-agnostic improvement using SmartNICs with attached memories. The core performance gain with this method comes from minimizing communication over PCIe. They avoid this by caching write requests on the SmartNIC's memory, and acknowledging the client immediately, thus removing the PCIe communication between NIC and host from the critical path. This solution is the most similar to our proposed design. Their implementation targets NVIDIA BlueField 3 NICs, as they claim it FPGA based system are more challenging to deploy and face longer development cycles. The resource overhead added by this solution comes mainly from the added memory requirements. In addition, they state that the low power ARM cores available on the BlueField 3 are not adequate for handling a software cache, thus reducing their design's feasibility for this platform. As their work is not fully completed, they provide no evaluations of the design. Because of this, we can not yet compare it with other solutions.

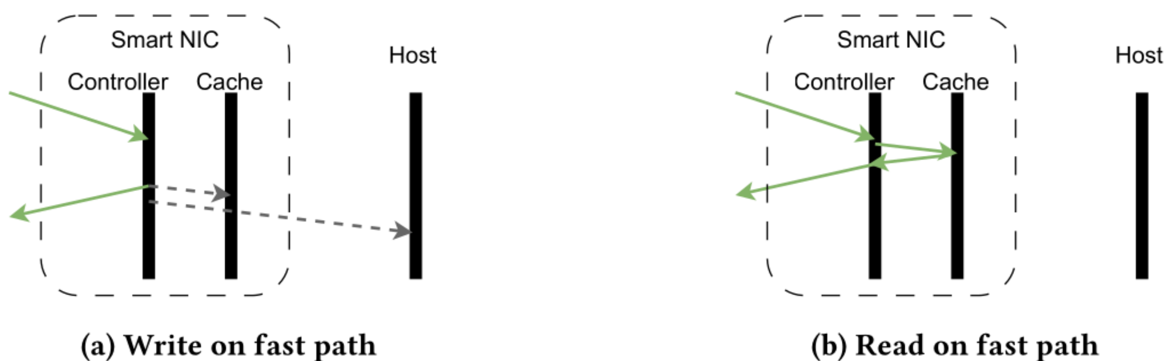

Figure 3.1: How caching is used in Chaapar, as shown in [10].

3.1.2. Active-Memory

In Rethinking Database High Availability with RDMA Networks [30], they propose an alternative primary-backup replication approach that leverages the RDMA capabilities of SmartNICs, with the goal of completely offloading replication operations from the backup nodes CPUs, targeting partitioned and distributed databases. The advantage is that, since the backup node CPU is free, it can process requests for a primary partition, increasing throughput. They argue that existing replication protocols were designed when

the network was the bottleneck, but that nowadays it has shifted to the CPU. Since the goal is not to involve the backup nodes CPUs in the replication, the coordinator must make sure that all changes are replicated on the backup nodes. To achieve this, the primary node sends an undo log before directly updating the memory on the backup nodes with one-sided RDMA operations. So, in case of failure, depending at which step of the replication it happens, the original data can be recovered from the log. The cost of this method is increased network traffic, which the authors claim is not a problem given that newer generation networks shift the bottleneck from the bandwidth to the CPU. As with most other papers shown here, in their evaluation the authors used a Mellanox ConnectX-4 card. While they do achieve 1.2x speedup in latency with respect to log shipping, the more notable improvements come in throughput. They claim to be able to achieve from 1.5x to 2x higher throughput with respect to other state of the art alternatives as the number of machines increases. Also, they notice that increasing the bandwidth does not increase the throughput for the other solutions, because they are bound by the CPU. Overall, the goal of active-memory is similar to that of Tailwind and Hyperloop, but it is difficult to compare with the other two as the focus is not on improving latency as much as the others.

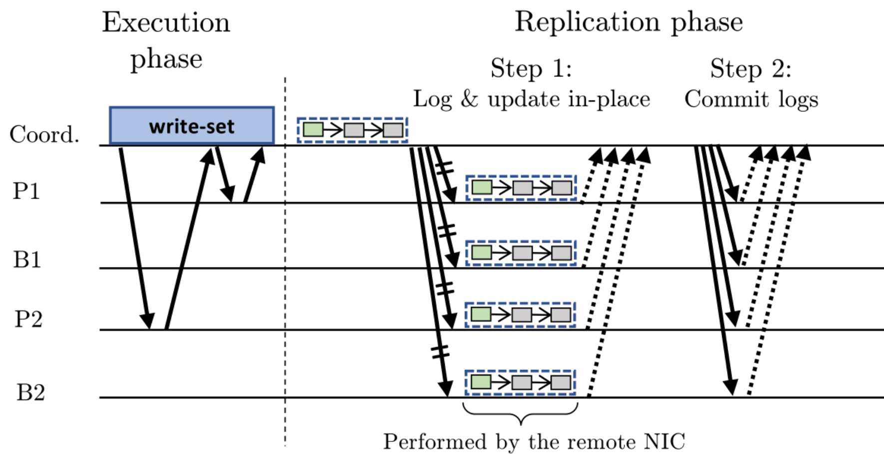

Figure 3.2: The active-memory replication protocol, as shown in [30].

3.1.3. Tailwind

In Tailwind: Fast and Atomic RDMA-based Replication [25], similarly to the previous paper, they propose a primary-backup replication approach where the backup servers are completely passive. They target in-memory key-value stores; specifically, they implement their solution for RAMCloud. According to the authors, the main challenge in using one-sided RDMA operations in primary-backup replication is that if the primary crashes,

the backup needs to be able to realize the write was left incomplete. To do this, Tailwind appends a piece of metadata after every update. Backups use this metadata to verify integrity and locate valid objects during recovery. The backup servers flush the NIC's buffers to SSDs periodically, through a remote procedure call triggered by the source. While backup nodes still have to process this metadata, it still is a great improvement in CPU utilization over two-sided RDMA operations. Also, the tradeoff incurred with this method comes in the form of memory overhead. The authors claim this is negligible, as only 4 bytes are used per each object. Their experiments show that Tailwind reduces RAMCloud's median write latency by 2x and tail latency by 3x; also, it increases throughput for write heavy workloads by 70%. The NIC used in their experiments is a Mellanox ConnectX-3. Compared to its most similar alternative, Active-Memory, this method promises better performance improvements and less network overhead, but the field of application is more limited.

Figure 3.3: A replication operation with Tailwind. In red are the appended pieces of metadata. The receiver CPU stays idle.

3.1.4. Hyperloop

Hyperloop [13], is a replication approach that addresses the challenge of high tail latency in replicated transactional storage systems. It aims to remove the backup CPUs from the critical path of replicated transactional operations. While the authors focus on chain replication, they claim Hyperloop can be used to accelerate other replication protocols. The introduced solution, offloads the entire replication pipeline to RDMA NICs (Mellanox ConnectX-3) using NVM-backed storage, eliminating CPU involvement in the critical path. Hyperloop introduces group-based NIC offloading primitives (gWRITE, gCAS, gMEMCPY, gFLUSH) which facilitates the protocol implementation. These primitives execute ACID- compliant transactions across replicas and leverage RDMA wait and remote work queue manipulation. Such approach, reduces the 99th percentile latency by up to 800x (for instance 14µs vs 11886µs for gCAS). It also reduces replica CPU usage from 100% to 0% while maintaining the throughput rate of 56Gbps on RDMA lines. The literature also explores experiments with RocksDB ad MongoDB under YCSB workloads,

showing 79% lower average latency and 81% higher tail latency. Hyperloop face some drawbacks due to its reliance on RDMA/NVM hardware and its limited fault recovery. However, these drawbacks are minor and trade for high-performance networking and scalability.

Primitive	**Purpose**
gFLUSH	Flushes the NICs caches to durable storage
gWRITE	Replicates data across nodes in a replication group
gCAS	Locks a memory region across replicas with compare-and-swap
gMEMCPY	Copies memory from log region to a persistent region

Table 3.4: Group network primitives introduced by Hyperloop.

3.1.5. HERMES

In Hermes [11], a reliable broadcast-based replication (membership-based) protocol for key-value store applications is introduced. The protocol is intended for in-memory data stores, guaranteeing strong consistency, high throughput, and low latency. Key innovations of HERMES include fault tolerance, local reads , logical timestamps for conflict-free writes and cache coherence invalidations. These features ensure consistency without centralized coordination. HERMES was built for the RDMA-enabled data centers using Mellanox ConnectX-6 NICs and Intel Xeon E5-2630V4 CPU. It achieves 985 MReq/s throughput for reads and 72 MReq/s for writes, with sub 1-µs median read latency and 69µs write tail latency. Moreover, Hermes outperforms the state-of-art protocols such as rCRAQ & rZAB by achieving a higher throughput by 3.4x (20% writes) and reducing tail latency by 3.6x (5% writes). The literature, also finds that HERMES faces some scalability limitations due to drawbacks such as write overheads (1.5 RRTTs per write), network bandwidth issues and failure handling fault recovery. On the positive side, HERMES still scales better (linearly) when compared to other technologies. Lastly, due to its reliance on RDMA hardware, HERMES faces limited hardware compatibility. In conclusion, Hermes makes deliberate trade-offs—such as hardware dependencies and limited scalability—in exchange for its high-performance, strongly consistent replication model.

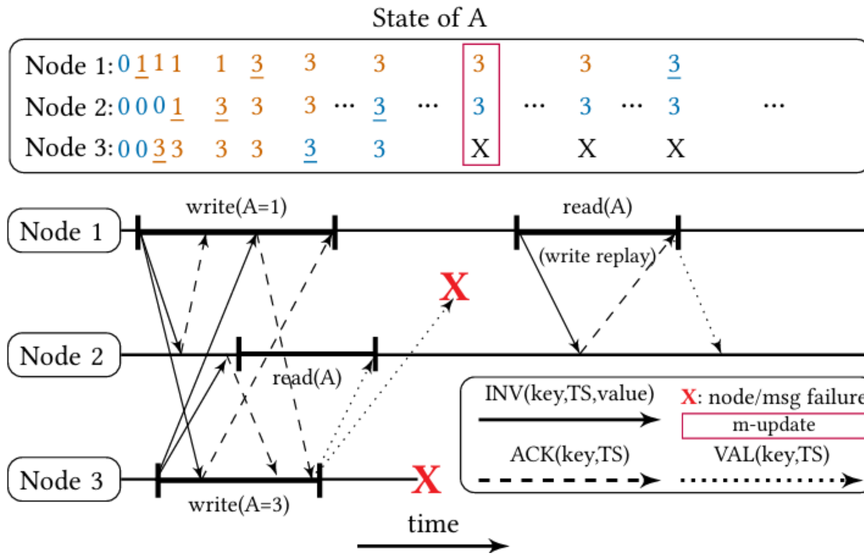

Figure 3.4: Example of HERMES Reconfiguration and Recovery. Adapted from A. Katsarakis et al.(2021), Hermes: a fast, fault-tolerant and linearizable replication protocol [11]

3.1.6. SWARM

In Swarm [18], a novel replication protocol designed for memory-disaggregated architectures is introduced. **SWARM** enables low-latency, strongly consistent access to shared data. The proposal combines two key innovations, *Safe Guess*, a wait-free protocol with single-roundtrip reads and writes while ensuring consistency and fault tolerance. *In-n-Out*, a technique for atomic conditional updates and retrievals of large memory buffers in one roundtrip without compute at memory nodes (RDMA). The authors implement **"SWARM-KV"**, a key-value store leveraging SWARM. SWARM-KV, achieves near-unreplicated latency (2.4µs gets, 3.1µs updates). It also outperforms state-of-the-art systems like FUSEE (2-3.4x faster) by minimizing the number of round-trips. SWARM-KV is built on the RDMA/CXL hardware (100Gbps RDMA NICS, atomic CAS). The protocol scales linearly with clients at the cost of high memory usage (x2 FUSEE memory usage) to ensure lower CPU overhead. For example, it achieves 28.3M ops/sec for 64 clients. Some key limitations in SWARM-KV include its reliance on clock synchrony for timestamp guessing and its growing metadata overhead with an increasing number of clients. In conclusion, the literature find that SWARM sets a new benchmark for low-latency replicated disaggregated memory making it the best technology for microsecond-scale applications.

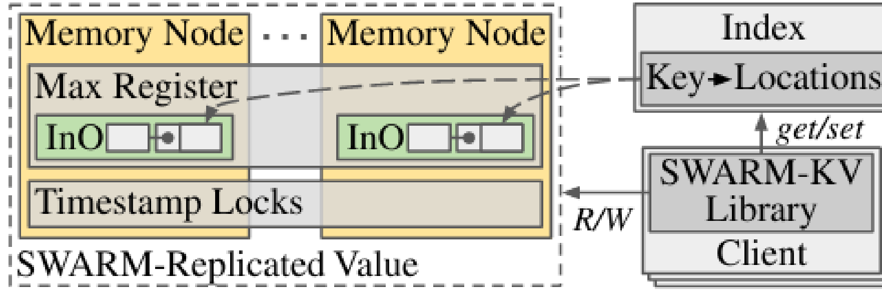

Figure 3.5: Architecture of SWARM-KV with a single key. Adapted from: A. Murat et al. (2024), Swarm: Replicating shared disaggregated-memory data in no time [18]

3.1.7. MINOS

The paper [22] introduces a high-performance framework to enhance the performance of distributed data persistency models (DPP) in leaderless distributed systems by offloading consistency and persistency protocols to a custom SmartNIC architecture. The solution proposed by the author comprises two key components, MINOS-Basline (MINOS-B) and MINOS-Offload (MINOS-O). MINOS-B provides detailed distributed algorithms for Linearizable consistency paired with five types of persistency models (Synchronous, Strict, Read-Enforced, Eventual, and Scope). MINOS-O, an optimized redesigned version of MINOS-B that offloads consistency and persistency protocols to a custom Mellanox BlueField SmartNIC architecture. Optimizations offered in MINOS-O include selective hardware coherence, message batching/broadcasting, and lock free FIFO queues (vFIFO/dFIFO) to accelerate updates. MINOS-O achieves 2-3x lower latency (reduces write latency by 35%) and 2-3x higher workloads throughput when compared to MINOS-B. Scalability wise, MINOS-O maintains low latency even as node counts increase. Moreover, MINOS-O reduces microservice latency by 35% in benchmarks like DeathStar while maintaining strong consistency and durability guarantees. When considering a system without MINOS-O, higher communication overhead is obtained and scalability is limited due to PCIe bottlenecks. The fact that MINOS-O requires custom SmartNIC support for coherence and multicast introduces hardware complexities and tradeoffs in persistency models (Stronger models will still incur higher latency than weak models) and batching overhead. The work highlights the potential of SmartNICs to scale leaderless systems but underscores the need for balanced design choices in coherence, persistency, and offloading.

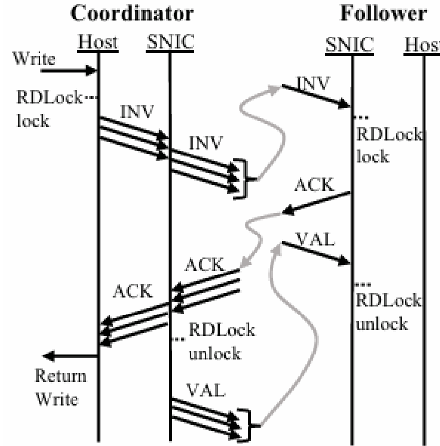

Figure 3.6: Diagram of MINOS-O SmartNIC offloading. Adapted from: Psistakis et al. (2024), Minos: Distributed consistency [22]

3.1.8. APUS

In [28] a high-performance paxos protocol design for state machine replication is introduced. The proposal leverages RDMA to reduce consensus latency, ensure strong consistency , support unmodified server programs and improve scaling. The solution employed an multi-paxos variant with a leader backup model and was implemented on the 40Gbps Mellanox ConnectX-3 NIC. One of the main performance gains in this paper is reducing the consensus latency to just 8.2 µs via one-sided RDMA writes and RDMA kernel TCP/IP stack bypass. When the authors compare their solution with other TCP/IP-based Paxos systems such as Zookeeper/libPaxos, they demonstrate a speedup of 4.9-32.3x. Moreover, when also compared with other RDMA-based protocols such as DARE, the proposed implementation excels in high concurrency delivering lower latency by 4.9x under heavy update workloads. Regarding scalability, APUS scales efficiently, when tested with multiple replicas, it followed a linear scaling and maintained a nearly constant increase in latency, about 7. 3% from 3 to 9 replicas. This has been achieved thanks to the parallel log replication an quorum-based acknowledgment applied in the literature. On drawbacks, the solution faces some challenges in terms of RDMA dependency and does not optimize read only requests, which introduce additional synchronization overhead for consistency. In conclusion, the literature proposal prove how RDMA low-latency one-side operations can revolutionize Paxos performance, making it an ideal solution for low-latency, high-availability systems.

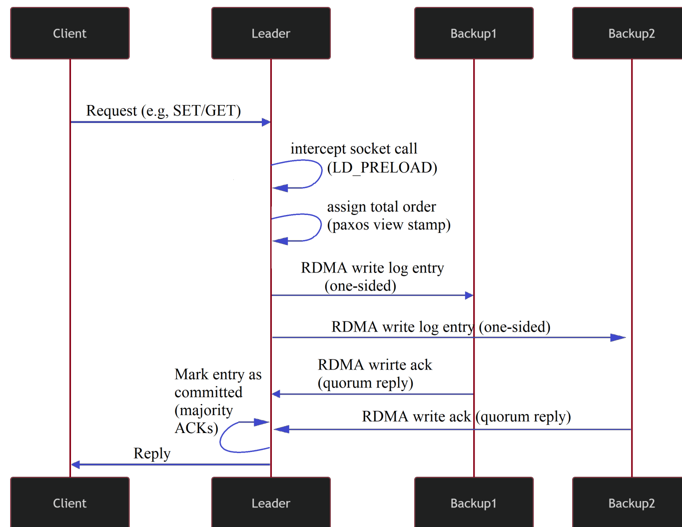

Figure 3.7: APUS RDMA powered paxos workflow pipeline

3.1.9. Mu

The paper[1] introduces Mu, a state machine replication system leveraging RDMA to achieve ultra low latency replication. The key innovation in this literature is a one-sided RDMA-based consensus protocol, where the leader replicates requests by directly wtiting to followers logs via RDMA, eliminating multi-round communication. Some important dimensions of the solution involve the hardware used, such as the RDMA Mellanox Connect-X4 100Gbps NIC, in addition to the Intel Xeon E5-2640 v4 CPU running at 2.40Ghz. Software wise, the solution proposed by the author follows a leader election mechanism and involves one-sided RDMA writes in addition to logging for consistency. When it comes to latency, a system with Mu achieves 1.3µs median replication latency. Whereas a system without Mu achieves 2.7x slower replication latency, such as HERMES 4.55µs and DARE 6.8µs. The authors also discuss the importance of Mu when it comes to Fail-over time, where it also peaks in terms of latency reduction leading to average of 873µs, much lower when compared to a system without Mu 10ms. Furthermore, Mu scales to 47M ops/sec with batching and ensures a total-order, linearizable consistency across its replicas, supporting a 3-way replication for fault tolerance. To mention drawbacks, Mu's heavy reliance on RDMA limits it to LANs and its in-memory design lacks persistence. Also, Mu faces sensitivity when it comes to OS scheduling and the need for modifications to capture/inject requests. Finally, despite all the discussed tradeoffs, Mu sets a standard for micro-second scale fault tolerance applications such as financial trading applications, by combining low latency, high throughput and strong consistency

in an novel RDMA-optimized design.

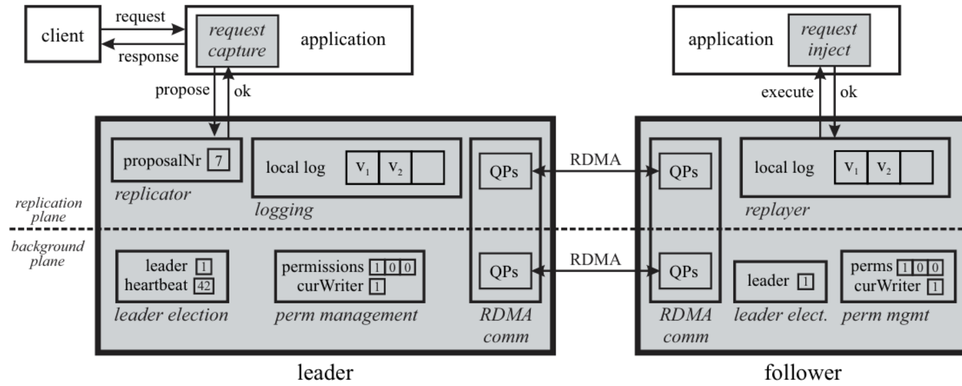

Figure 3.8: Diagram demonstrating Mu's architecture. Adapted from : M. K. Aguilera et al. (2020), Microsecond consensus for microsecond applications [1]

3.1.10. Derecho

Derecho [9], is a high performance library for state machine replication in cloud services, designed to enable strong consistency with low latency and high throughput. Some key innovations of Derecho are the non-blocking, pipelined protocols that leverages RDMA for efficient data movement, non-blocking data movement, in addition to shared state tables enabling asynchronous progress detection. Moreover, temporal snapshot isolation is employed for consistent, lock-free queries. The proposed solution was setup on an RDMA Mellanox ConnectX-4 100Gbps NIC. The RDMA-leveraging employment in the system achieves up to 16GB/s throughput and 1.5µs latency in optimal conditions. Moreover, performance benchmarks show how Derecho significantly outperforms traditional solutions like LibPaxos and Zookeeper, with 100x higher throughput over TCP and a 4x additional improvement with RDMA. Without Derecho, traditional paxos libraries suffer from blocking two-phase commits, limiting throughput and scalability. The authors have evaluated the proposed solution on 128 servers and have seen an efficient scaling phenomena. When it comes to drawbacks, the authors state that the synchronization overhead is minimal, meanwhile, they mention a huge trade-off favoring consistency over availability. In addition to those, the system face a high performance drop when tested on non-RDMA networks, implying its high RDMA dependency. On the positive side, the system is persistent and offers recovery guarantees when it comes to persistence via logs. In summary, the analysis highlights Derecho's breakthroughs in scalable, consistent replication for modern cloud applications, by demonstrating empirical results to back its claims.

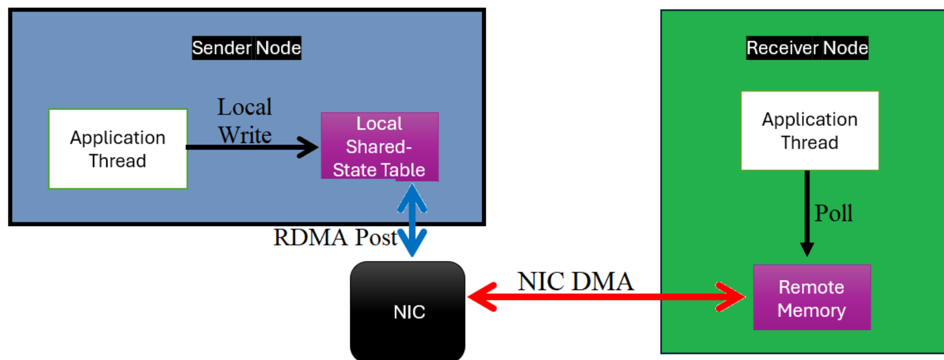

Figure 3.9: Diagram demonstrating Derecho's RDMA-Powered Zero-Copyy SST Update Flow

3.2. SmartNIC Memory Innovations

In this section, we review papers exploring how SmartNIC memory innovations contribute to application improvements, often in conjunction with RDMA.

Title	Goal	Contribution	Open Source
RDMA Design [31]	Improve concurrent accesses performance with RDMA	Suggestions for pessimistic and optimistic synchronization with RDMA	No
PMNet [23]	Improve latency of write requests in servers	Log write requests to the NIC's memory, then send them to server	Yes
NIC Flush [6]	Remove problems caused by NIC's cache	New RDMA primitive to flush the NIC's cache to persistent memory	Yes
DFI [26]	Ease the development of RDMA systems	Efficient abstraction of RDMA operations with low code complexity	Yes
StRoM [24]	Have RDMA on an open source NIC	An implementation of a NIC with RDMA capabilities for FPGAs	Yes
SODA [27]	Optimize disaggregated memory performance	Enables offloading network attached memory tasks onto SmartNIC	Yes

Table 3.5: Analysis of papers regarding SmartNIC memory innovations, the Columns represent : Goal : A brief overview of the solution goal ; Contribution : The implemented mechanism capability ; Open Source : Whether the implementation is publicly available

3.2.1. RDMA Design Guidelines

Concurrent accesses to remote data structures require synchronization. In Guidelines for Correct, Efficient, and Scalable Synchronization using One-Sided RDMA [31], the authors analyze two categories of synchronization techniques, pessimistic and optimistic. In particular, they focus on implementations that leverage RDMA, arguing that current proposed implementations can be improved, and that some do not even ensure data consistency. Pessimistic approaches make use of RDMA atomic operations to mirror

latching. A diagram of the traditional mechanism and the proposed improvements are shown in Figure 3.10. On the other hand, optimistic approaches implement a write unlatch optimization with versioning, FaRM and CRC. The problem with the former is that with RDMA, read operations are not guaranteed to be ordered due to the PCIe bus. Meanwhile, the problem with the latter is that, besides the added computation overhead, it is still a probabilistic method. The authors use multiple types of Mellanox ConnecX NICs to evaluate their designs and attain a good performance improvement in their experiments where uncontended atomics scale to 51.2Mops, while contended ones drop to 2.32Mops (due to NIC serialization). Moreover, the literature indicates that for lower amounts of workers, pessimistic approaches perform better. Meanwhile, optimistic concurrency scales much better. In summary, the paper underscores that no single solution can fit all workload characteristics (contention and data size) and that hardware choice is of high importance.

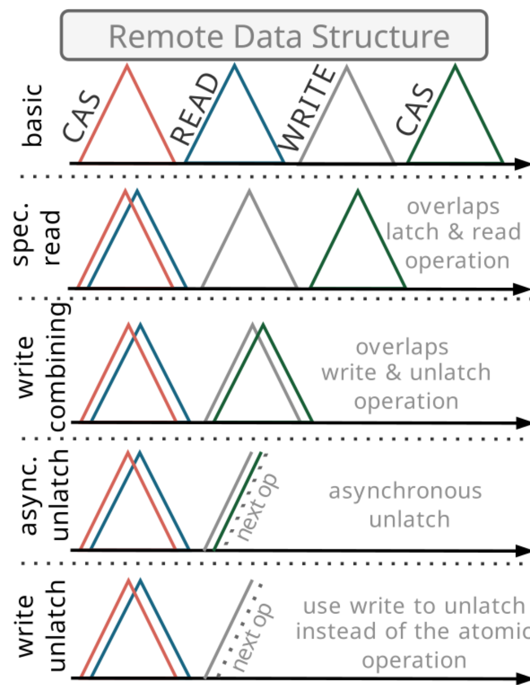

Figure 3.10: Improvements of pessimistic latches, as shown in [31].

3.2.2. PMNet

In PMNet [23], the authors focus on leveraging persistent memory in NICs to reduce the latency of update requests in distributed storage systems. The authors target mainly network-switches and aim to remove the server's processing of update requests from the critical path. The Literature proposed solution is logging update request in the NIC's

memory and acknowledge clients before server processing (i.e. as soon as the update enters the persistent domain). This approach achieves sub-RTT latency. In their solution, they also address system failures and memory insufficiency. In the case of a failure, logged updates are resent in order when the system restarts. Moreover, in an "out of memory" case, requests bypass the NIC. The literature also involves caching systems to speed up read requests. The mentioned solution, was implemented with an FPGA-based 10Gbps NIC on a Xilinx UltraScale+ VCU118 FPGA. Regarding performance benefits, in terms of replication protocol, the solution achieves 5.88x lower latency when compared to a baseline setup on a Mellanox ConnetX-3. Furthermore, the literature discusses performance penalties such as the synchronization overhead caused by the proposed solution in failure scenarios. In conclusion, with PMNet enables an 4.31x throughput improvement of update requests and a 3.23x tail latency reduction across various workloads including real-world scenarios.

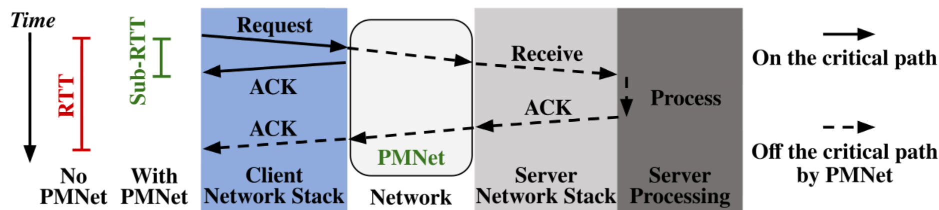

Figure 3.11: Round trip time of a request with PMNet, as shown in [23].

3.2.3. NIC Flush

In Hardware Supported Remote Persistence [6], they argue that the volatile cache of RDMA NICs is a potential threat to data persistency. The threat occurs when after the conclusion of an RDMA write, where a completion message is sent to the writer, while data might still be in the NIC's cache and not persisted to the memory. For this reason, they propose RDMA primitives and durable RPCs to ensure remote data persistence in distributed memory systems. The RDMA primitives include SFlush, WFlush (sender initiated), and Rflush (receiver initiated). These primitives force the receiver of the NIC to flush its cache to the PM. Moreover, to optimize the data persistence process and account for failure and recovery, remote procedure calls are designed by leveraging the designed primitives. The authors use the Mellanox ConnectX-4 InfiniBand RNIC (40/56 GbE) and state that the advantage of their solution relies mainly on the decoupling of the data persistence and the RPC processing. Such, advantage led to an improvement of RPC's throughput by up to 90% and tail latency reduction by up to 49%. The literature also mentions a 3.1x faster failure recovery for write-heavy workloads when compared to

traditional RPC's. With regard to drawbacks, the solution faces a scalability limit along with network (1 RRT operation added for sender initiated primitives) and CPU overhead.

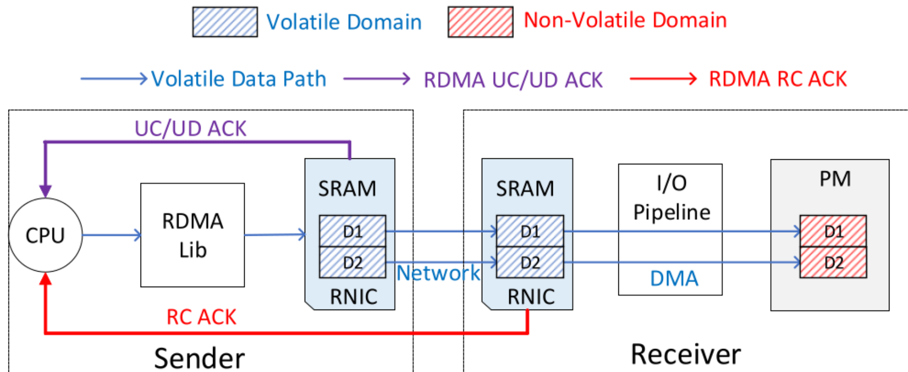

Figure 3.12: An RDMA write. Note that the sender's CPU has no way of knowing when data is in persistent memory.

3.2.4. DFI

DFI authors [26], highlight the difficulty of using RDMA due to its low-level verbs and the rarity of its data-centric application interfaces. In the literature, a new high-level abstraction is designed for RDMA with a focus on replication protocols. The proposal discusses an idea of creating what they call flows, with one or multiple sources and targets. They also provide optimizations for different requirements, such as maximizing bandwidth utilization or minimizing latency. To handle bottleneck, the authors make use of RDMA multi-cast. The solution was implemented on a Mellanox ConnectX-5 100 Gbps NIC. Regarding performance benefits, DFI achieves a throughput increase to reach 1.5 million reqs/s versus 700K reqs/s of the baseline "DARE". Moreover, results related to synchronization overhead show negligible overhead when compared to a raw RDMA setup. However this overhead scales with thread counts. In summary, the literature proves its advantages in performance, especially in terms of throughput and scalability.

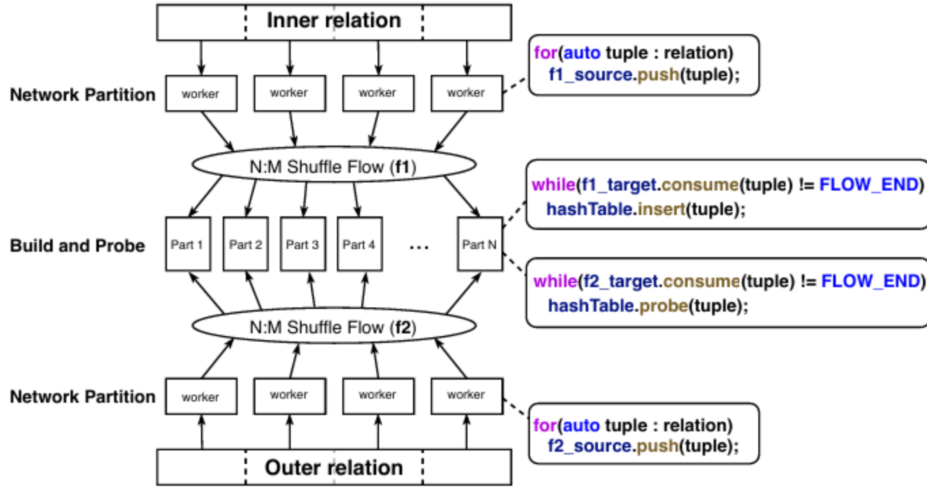

Figure 3.13: Distributed Radix Hash Join with DFI flows. Adapted from : L. Thostrup et al. (2021), DFI: The data flow interface for high-speed netwoks [26]

3.2.5. StRoM

In StRoM [24], the authors present, an optimized FPGA-based RoCE v2 NIC that claims to be the first open-source RDMA NIC implementation. The platform is implemented on two FPGA platforms:a Xilinx Virtex-7 XC7VX690T (10G) and an UltraScale+ XCVU9P (100G) FPGA and is intended for accelerating distributed systems by minimizing data movement and CPU load. StRoM offers the mentioned solution by introducing several key optimizations, including RDMA semantics (two verbs are introduced RDMA_RPC & RDMA_RPC_WRITE), stream processing operations (e.g., filtering aggregation, partitioning), and offloading application-level kernels (e.g., pointer chasing, data shuffling, consistency checks) directly to the NIC while bypassing the CPU. The system achieves 9.4 Gbits on the 10G FPGA, with latency as low as 5μ for small RDMA operations and a message rate of 8M msg/s for 64B payloads. Regarding the 100G version, the throughput achieved is 100 Gbit/s (saturates at 2KB payloads), with the latency reduced by 2 due to higher clock speed (322 Mhz) and a message rate of 40M msg/s for 64B payloads. StRoM significantly improves performance, for instance linked list traversal is reduced from N round trips (RDMA) to 1 round trip (cutting latency by 5µs per hop), 50% lower latency for hash table lookups (from 20µs to 10µs for 4kb Values) and only 1μs overhead versus 40% CPU overhead in software for consistency checks. Moreover, line-rate stream processing is enabled (e.g, HyperLogLog cardinality estimation at 100G), matching raw RDMA performance. To highlight the mentioned benefits, without StRoM traditional RDMA requires multiple rounds (e.g, 2 for hash table lookups), high latency for linked list traversal and CPU involvement for consistency checks. Some of the challenges faced by

StRoM include PCIe bandwidth limitations at 100G and strict kernel design constraints. Synchronization overhead also adds a 1µs latency for kernel offloading due to PCIe round trips. Finally, Strom shares a core feature with OpenNIC: the ability to integrate custom designs within its plugin architecture.

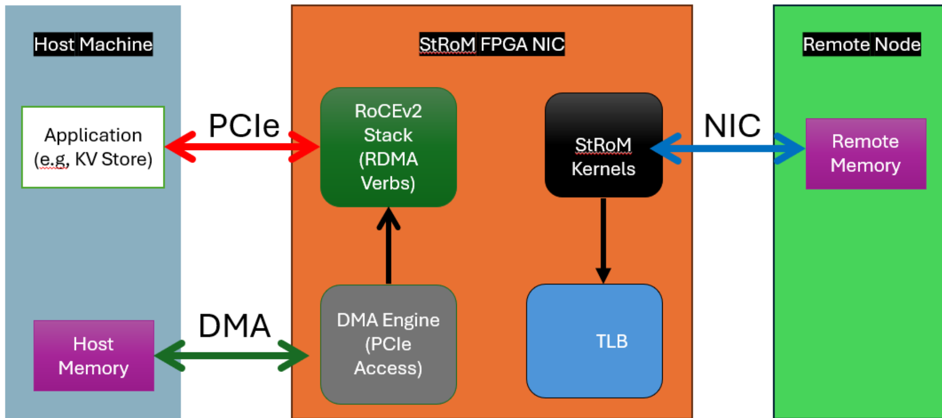

Figure 3.14: StRoM's SmartNIC Architecture.

3.2.6. SODA

The paper [27] introduces **SODA** (SmartNIC-Offloaded DisAggregated memory), a noval runtime library in the aim of improving the performance disaggregated memory systems. SODA leverages an off-path SmartNIC (NVIDIA BlueField-2 DPU) to offload memory management tasks such as data staging, caching and prefetching. The BlueField-2 DPU involves 8 ARM Cortex-A72 cores, 16GB of DDR4 memory and leverages 100Gb/s RoCE for RDMA (DMA bandwidth 10.3GB/s for writes, 9.4GB/s for reads), achieving up to 14.3GB/s host-DPU bandwidth. The key features offered by SODA include NUMA-aware optimizations and task aggregation and pipelined data movement for reducing network traffic and latency. Moreover two caching strategies are offered, static caching for frequently accessed data (e.g, graph vertex data) and dynamic caching with prefetching. The mentioned optimizations reduce network traffic by up to 42%, while delivering a 7.9x speedup over node-local SSDs in benchmarks on Friendster graph. Moreover, the hit rates are increased by 93% for dynamic caching in predictable workloads. When scaling is considered, the shared DPU agent reduces traffic by 25% for co-running processes. Without SODA, considering MemServer for instance, high network traffic and CPU overhead are expected with direct host-memory node communication. SODA does have limitations when it comes to coherence restrictions (single writer only), workload-dependent caching efficiency (dynamic caching can increase traffic by 117% if hit rates are low and finally DPU resources constraints (small DRAM and low-power cores). Despite these trade-offs,

SODA demonstrates that SmartNIC offloading can significantly enhance disaggregated memory performance when optimized for specific access patterns.

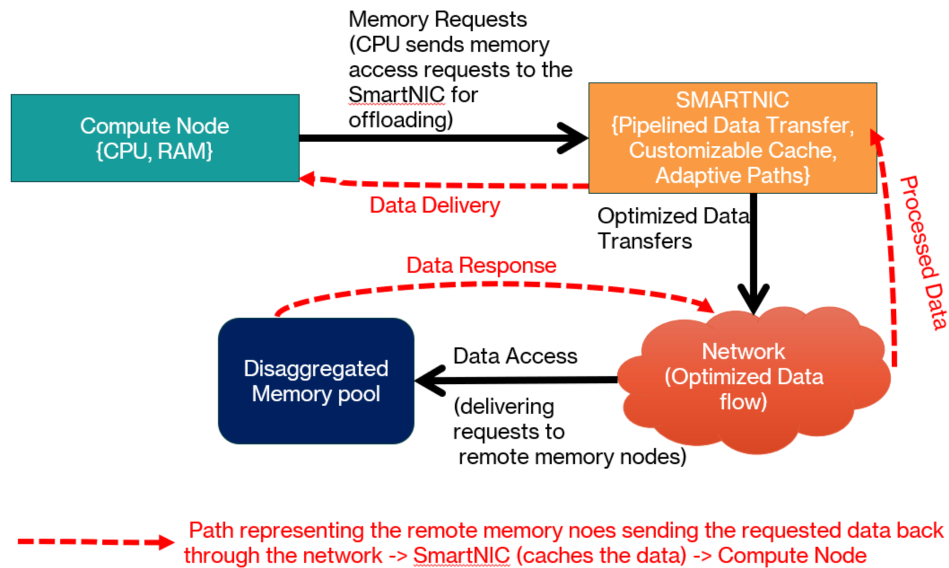

Figure 3.15: SODA components flow of data and control path.

4 | Architecture

In this chapter, we will discuss the architecture of our final solution and of the intermediate step we took to get there. We will also go into detail for each of the main components.

4.1. OpenNIC with HBM

As our first step, we added a new HBM subsystem to OpenNIC, since our design will rely on the Alveo board's high-bandwidth memory. In order to connect this, we modified the QDMA that vanilla OpenNIC comes with, adding an extra AXI memory mapped interface. We then connected this interface to the HBM. Additionally, we want to be able to access the HBM from the network. We thus created a memory controller with the goal of initiating memory mapped transactions from packets coming from the network in a stream interface.

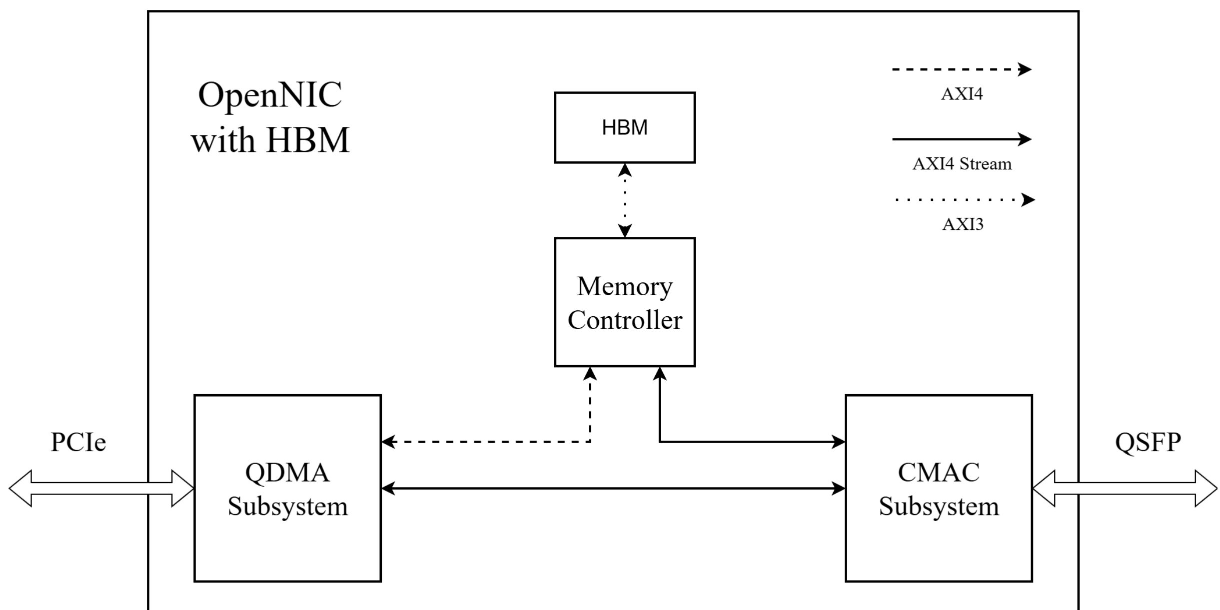

Figure 4.1: Architectural diagram of the OpenNIC Shell connected to High-Bandwidth Memory (HBM).

4.1.1. HBM

In order to make use of the on-board HBM, we added the HBM controller ip. This provides 32 AXI3 memory mapped interfaces. The HBM interfaces cannot be directly connected to the QDMA or other more recent ips, which all feature the AXI4 protocol rather than its previous iteration. To solve this issue, we used the SmartConnect ip provided by Xilinx to connect the two versions.

The HBM has several parameters that allow to change its behaviour and performance. For our design case, the choice of whether to allow each interface to reorder AXI transactions is very important. This presents a trade-off between throughput and latency, as enabling this option allows transactions to be swapped to a more convenient order, increasing throughput at the cost of increasing latency too as transactions are reordered. Since we are looking to minimise latency, we disabled this option.

Another key aspect of the HBM's behaviour is the global switch. When this option is enabled, each of the 32 interfaces can access all available memory locations. In contrast, when it is disabled, each interface can access only its own section of memory, which is 512 MB in our case. Of course, this global switch comes at the cost of latency, but we rely on it to be able to be more flexible with our design, so we kept this option enabled.

4.1.2. Memory Controller

Inside the memory controller, we use the Datamover ip to handle the memory transactions. This ip allows us to both store an AXI stream packet inside the HBM with the S2MM function, and to extract data to a stream with the MM2S function. All operations of the Datamover ip are started with the use of commands sent to the ip. These commands and their status response are handled within the memory controller.

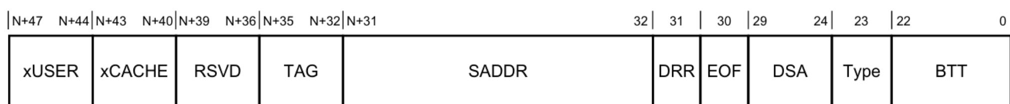

Figure 4.2: Diagram showing the Datamover ip basic command word layout.

The Datamover ip provides the option to enable a store-and-forward feature in each of the two directions of data. This allows it to initiate the transaction only after a portion of the data has arrived. We keep this option disabled as we try to minimise latency.

4.2. OpenNIC with Replication

The goal of our design is to enable per-key leader-based replication of a key-value store directly on the smart NIC. Therefore, we evolved the previous step to include a replication engine, whose goal is to decide the memory transactions to run and the packets to send out based on the replication packets that are received. On the receiving side, we filter packets that are meant for replication and divert them to the engine, while the rest are redirected to the QDMA as with the previous design. Meanwhile, on the transmitting side we arbiter between packets coming from the engine and the QDMA. This design allows normal networking to occur in parallel with replication on the NIC. For now, we consider key payloads to be 1∼KB, which provides a reasonable balance between replication throughput and hardware buffering. This value, however, is not fixed: depending on the application, users may adjust the payload size to trade off between latency, bandwidth efficiency, and resource utilization on the FPGA.

For the key-value store, we need to perform both reads and writes in the HBM from the network. So, our design must allow clients to send a read request to any node in the network and receive a response, and to send a write request to leaders and receive an acknowledgment of when the operation has been replicated across all nodes. This implies that each leader is responsible of replicating write requests to the other nodes and waiting to receive their acknowledgements.

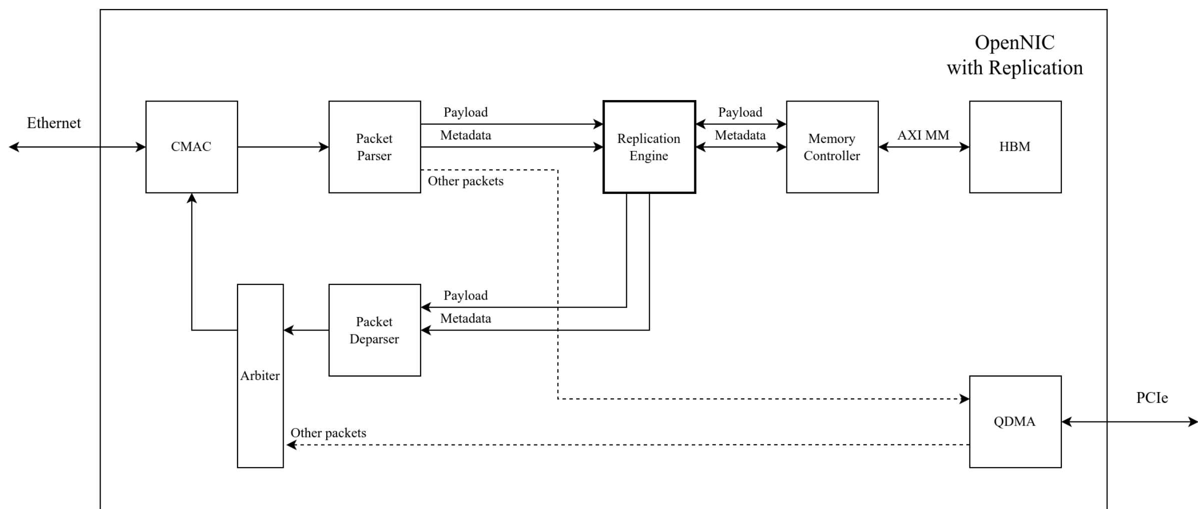

Figure 4.3: Diagram showing OpenNIC with the replication mechanism.

4.2.1. Key-Value Store

The memory controller is almost the same as in the previous iteration, with the exception that it is also responsible for hashing the key into an address for the HBM. It is controlled by the replication engine, which decides the operations needed and waits for their completion.

The memory controller is left separate from the replication engine in order to be more flexible. For example, given the nature of the HBM and its 32 interfaces, if more throughput is required, it would be simple to extend the memory controller to make use of more interfaces. This change would be transparent to the replication engine module.

In our key-value store, we opted for keys of 8 bytes and buckets of 1024 bytes. These sizes are taken with consideration of most applications of in-memory key-value stores, but can be very easily modified with parameters. As the size of our HBM is 16 GB, we hash the 8 bytes key into a 24 bit address, in order to be able to address every bucket inside the memory.

4.2.2. Replication Engine

The replication engine is the core of this design. It receives information from the packet parser regarding the incoming request, such as the opcode. Based on this, it is responsible for requesting memory transaction to the memory controller, replicating the received write request to all replicas in case the node is the leader for the involved key, and keeping track of all replicas' acknowledgment to know when the update is completed in every node. Also, when a memory transaction finishes, it is responsible for sending the correct response to the node which sent the request.

The engine uses two simple finite state machines: one to control packets incoming from the network and one to control data incoming from the memory controller. The network FSM has a state to broadcast a write request to replicas; during this, new requests can still arrive, so they are buffered and processed once the broadcast is finished. Finally, the engine also uses BRAM to keep track of outstanding write operations, including the number of acks received.

4.2.3. Parser and Deparser

We used the Vitis Networking P4 ip to process the incoming packets and build the outgoing ones. This ip greatly simplifies packet processing, as it converts a P4 design into an FPGA design. Thanks to this we can also remain very flexible regarding our packet structure, as

modifying the P4 design is much less demanding than rewriting a SystemVerilog module.

The job of the packet parser is to process incoming packets, arriving directly from the CMAC, and recognizing the replication ones. For the latter, it will extract relevant information from the headers, and forward it to the replication engine. In general, the most important fields extracted are the opcode, key, and networking information such as sender's MAC and IP addresses. For non replication packets, the parser simply signals the downstream filter to redirect them to the QDMA, allowing normal network communication.

The deparser does the opposite. It receives the necessary information from the replication engine, and with this it creates the headers and appends them to the incoming payload. The resulting packet is then forwarded to the CMAC, after passing through an arbiter. The arbiter's goal is to allow packets to come from both the replication engine and the QDMA.

4.2.4. Packet Structure

For the sake of simplicity, we designed a minimal replication header that contains all the necessary information to carry out the operations. We make use of the Ethernet, IP, and UDP network stack. We chose UDP as it is a connectionless protocol, giving us an advantage in latency. The packets then have the replication header containing the opcode, id, and key. The opcode signals what the contents of the packet represent. For example we can have read, write, read result, or write acknowledgement. Then, the id is used by the client to distinguish between outstanding requests, if it needs to do so. The response will always carry the same id value as the request. The packet's payload depends on the opcode. In the case of a write request or read result this represents the value for the given key. In other cases where necessary, we add some padding in order to reach the minimum Ethernet frame size of 64 bytes.

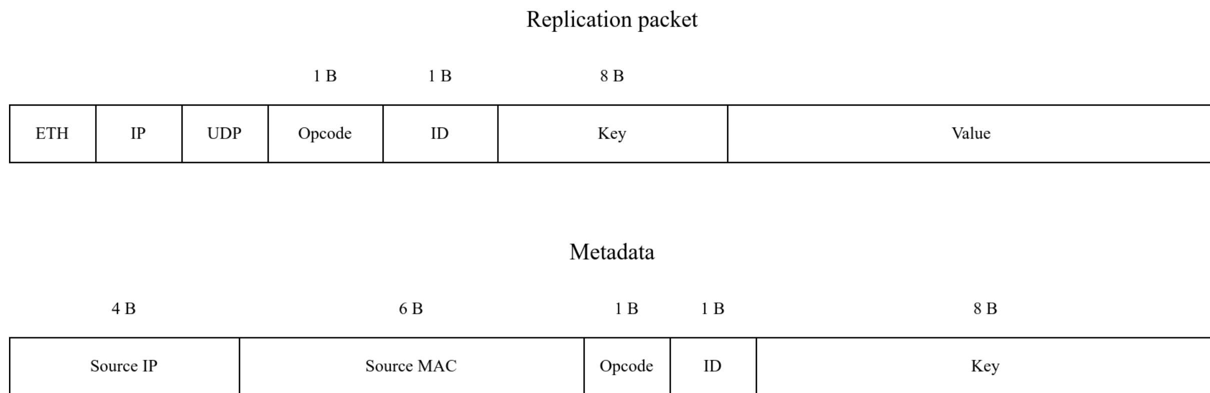

Figure 4.4: Diagrams showing the structure of replication packets and the metadata

We also have a custom structure for the metadata being exchanged between our modules in the replication subsystem. This metadata contains the minimum information that is needed to be taken from a request to be able to form a response. The value (or payload) is an exception, as it is exchanged with AXI stream interfaces.

5 | Experimental Results

In this chapter, we present experimental results of our architecture. We detail the outcomes of our final implementation and discuss the intermediate approaches verified in simulation that shaped its design. To begin, we will introduce the key evaluation metrics, including latency and resource utilization, that we use to assess performance of our design.

5.1. Evaluation Metrics

To assess the effectiveness of the proposed architecture, and the different design path prototyping that shaped it, we rely on a set of quality metrics that capture both performance and resource efficiency. These metrics provide a common basis for comparing different framework and simulation outcome results.

5.1.1. Latency

The term Latency refers to the time delay experienced by a data packet or transaction when passing through the system. In other words, it is the duration between the moment data is injected at the input and the moment it is observed at the output. Latency is commonly measured in nanoseconds and it directly reflects the responsiveness of the design. In this chapter, we record the average latency for all latency measurements done in our experiments.

5.1.2. Resource Utilization

The term Resource utilization refers to the fraction of available FPGA hardware resources consumed by a given design. In other words, it represents how much of the device's logic and memory fabric is occupied in order to implement the desired functionality. This can include lookup tables (LUTs), flip-flops (FFs), block RAMs (BRAMs), and digital signal processing (DSP) slices. Resource utilization is commonly reported as a percentage of total available resources, providing insight into area efficiency, scalability, and potential design limitations.

5.2. Experimental Setup

This section details the experimental setup used to generate the results presented in this chapter, encompassing the simulation, hardware, and benchmarking environments.

5.2.1. Simulation Setup

In order to shape and mold our final architecture, we had to develop a simulation framework in order to validate our initial ideas and take critical design decisions. Our framework consisted of the Xilinx OpenNIC [29]; which was extended with an HBM controller, enabling us to access the on-board memory. Communication between the OpenNIC's CMAC and QDMA interfaces and the HBM controller, was established through a custom controller, which includes the Datamover and Smart Connect IPs. The Smart Connect was the unit responsible for translating AXI4 to AXI3, while the datamover was responsible for translating AXI stream into AXI4. The QDMA communicated directly with the SmartConnect via its AXI4 buses, but the CMAC first had to interact with a data mover because of its streaming nature.

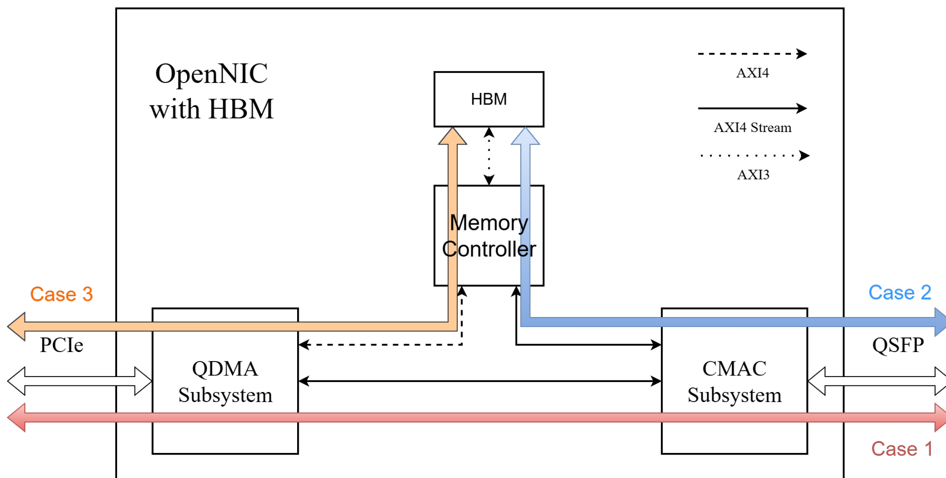

Figure 5.1: Modified OpenNIC with different simulation datapath.

Based on the design presented above, three possible paths were identified for packet traversal inside the NIC. These scenarios are analyzed in detail, with their corresponding simulation results provided later in this section. For now, we begin by describing the simulation infrastructure used to model and simulate these scenarios.

There are two points of packet injection in the OpenNIC-based simulation environment, as illustrated in Figure 5.1. These correspond to the CMAC and the QDMA interfaces. On the QDMA side, we used Xilinx's QDMA simulation infrastructure to emulate PCIe

transfers between host and QDMA. This framework provides SystemVerilog tasks to configure the QDMA, feed it with descriptors, and simulate a host memory, enabling both memory-mapped and streaming queue simulations. For the CMAC interface, since the module itself is not instantiated in the simulation environment, we use a custom AXI Stream simulation framework in OpenNIC. We developed Python modules to generate text files containing the desired type of packets traffic intended to be fed to the CMAC interface. The Scapy library was used to construct these packets, which were then converted into AXI Stream beats. These files were subsequently read by our testbench, which drove the interfaces according to our simulation requirements.

After verifying the designs in simulation, we also deployed them on the Alveo U55C accelerator card. Each datapath was tested by instantiating its design on the board and validating functionality with the appropriate driver. Specifically, the OpenNIC driver was used for **Case 1** and **Case 2** , while the Xilinx QDMA driver was employed for **Case 3**. In all cases, the on-board CMAC was operated in hardware loopback mode (enabled through the `pcimem` interface), allowing us to confirm that packets were correctly transmitted and received before moving on to combining the pieces together for the building the architecture and the final prototyping.

5.2.2. Hardware Setup

The hardware testbed where we tested our replication protocol architecture consists of two AMD Alveo U55C accelerator cards, each installed in a separate server via the PCIe slot. On each board, PCIe provides the interface between the FPGA and the host system. For inter-node communication, the U55C boards expose dual QSFP28 ports. In our setup, one port from each board is directly connected using 100∼Gb/s 0.5 meters Mellanox cables, providing a high-throughput and low-latency data path dedicated to replication traffic.

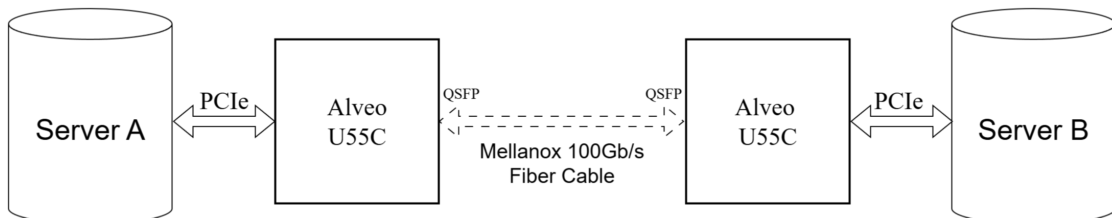

Figure 5.2: Diagram showing our two-node replication hardware setup.

Each FPGA is configured with the replication baseline architecture. In this two-node

setup, **both boards act as leaders for a subset of keys**, handling ordering and update propagation for those keys, while simultaneously acting as followers for keys led by the other board. This distributed leadership model allows the system to validate correctness and performance with minimal deployment complexity, avoiding the overhead of larger multi-node clusters.

To test the system, we crafted packets according to the structure defined in the previous chapter (read, write, write-to-leader, and normal UDP operations) using the Scapy library. These packets were sent to the board through the PCIe interface, which is exposed as an Ethernet interface once the OpenNIC driver is loaded. From there, packets traverse the standard QDMA datapath of the sender board and are forwarded through its CMAC into the QSFP link, ultimately arriving at the CMAC of the receiver board for further processing.

Latency measurements were carried out by instantiating an Integrated Logic Analyzer (ILA) on both boards, a component whose described in the following subsection.

5.2.3. ILA

The Integrated Logic Analyzer (ILA) is an IP core provided by Xilinx that can be instantiated within the FPGA fabric. It enables designers to probe internal signals at runtime, offering real-time visibility into the behavior of hardware modules.

By integrating the ILA into our design, we were able to observe the functionality of individual datapath components, validate the correctness of the implemented architecture and conduct latency measurements during hardware execution.

5.2.4. Benchmark Environment

We designed a benchmark environment to evaluate the replication baseline against different reference points. The first setup consists of a software replication benchmark, where packets are processed through a U55C board and then handled by a software-based replication layer written in C. This setup is also instrumented with ILAs for latency measurements. Packets are fed to the target NIC, through another NIC with QDMA drivers loaded. The second benchmark runs on vanilla OpenNIC armed with an ILA, using two U55C boards connected directly to each other in order to measure the latency of the unmodified datapath components, (no replication). Together, these two benchmarks provide the comparison needed to assess the overhead and extra resources introduced by our replication design.

5.3. Simulation Case 1: CMAC - Host Memory Communication

The first simulation case, focuses on the data transfer between the **CMAC** and the **Host Memory**. This path is already implemented in the standard OpenNIC (Vanilla OpenNIC). The scenario is split into the two possible directions: the host writing a packet to be sent out to the network, and packets entering the CMAC and written to the host memory. The measurements obtained from this simulation case, can be seen as a benchmark for evaluating the performance of the next simulation case, which involves accessing the HBM from the CMAC.

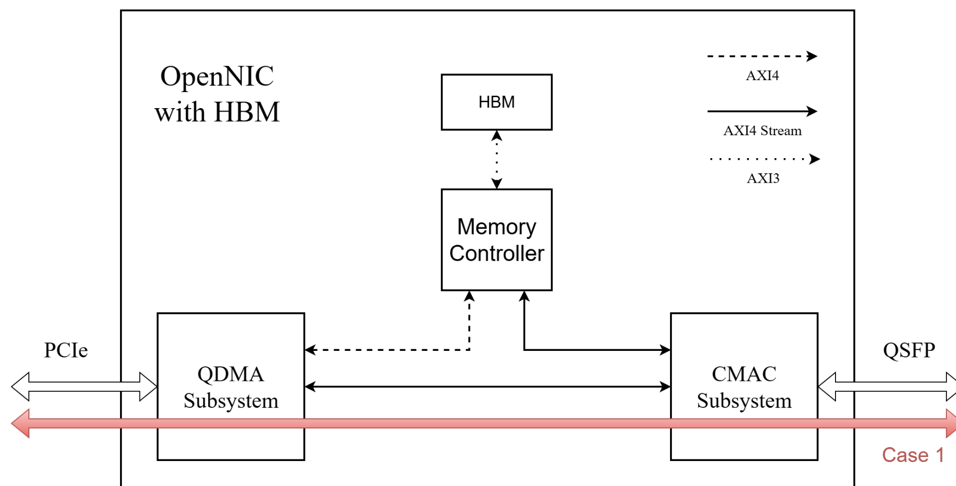

Figure 5.3: The red arrow indicates the path the packets take in this simulation case.

5.3.1. Operation Flow

In order to understand the results of our experiments, we need to focus on the behaviour of the QDMA. Transactions are always initiated by the host by updating the producer index register, which notifies the QDMA that a new descriptor is available. The QDMA will eventually fetch the descriptor from the host memory, and begin the operation.

For an outgoing packet (Host to Card direction), the QDMA will fetch the descriptor as soon as the producer index is updated. For an incoming packet (Card to Host direction), the QDMA will fetch the descriptor when a new packet is received after the producer index is updated.

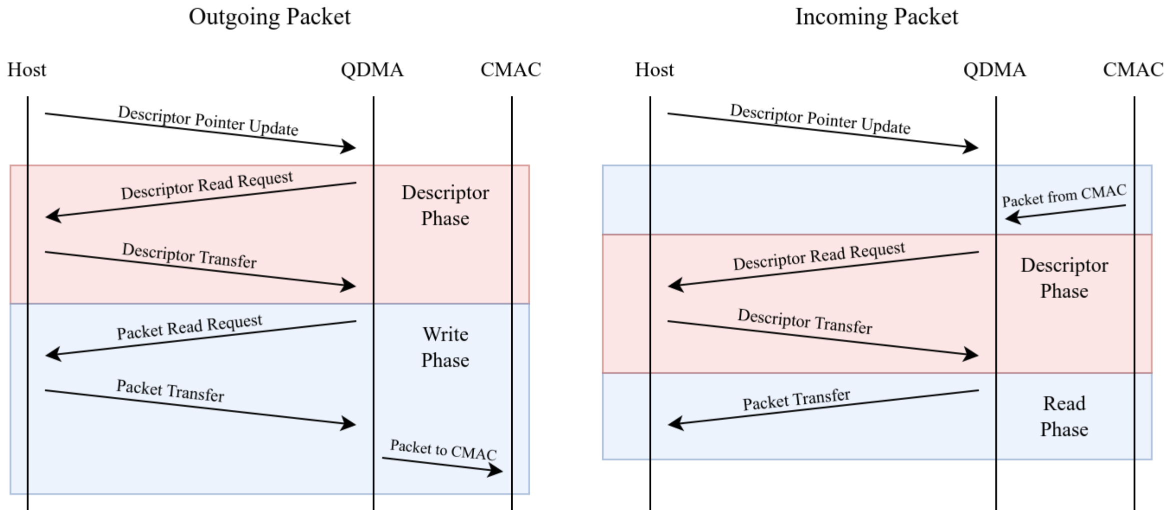

Figure 5.4: Flow diagram of a packet transfer for case 1.

In the Figure 5.4 above, we show a diagram of this process. Also, to better understand the source of latency in our simulation cases, we divided each operation into two sections. In the red section we isolate the QDMA's descriptor fetch procedure, while in the blue section we isolate the rest of the read or write. From this diagram we can expect that sending a packet would take more than receiving one, because the former has one less PCIe trip with respect to the latter. Meanwhile, we can expect the descriptor phases to be similar between the two directions, even considering the fact that the descriptor for the incoming direction is only 8B against the 16B of the opposite direction.

5.3.2. Simulation Results

We start measuring the descriptor phase when the QDMA starts sending the first read request; then we stop measuring when it finishes receiving the descriptor. For an outgoing packet, we measure the write phase from when the QDMA receives the descriptor to when the packet finishes exiting the CMAC. For an incoming packet, the read phase is split as shown in Figure 5.4. We start the timer when we start sending the packet to the CMAC, and stop when the QDMA starts sending the descriptor read request. Then, we start again when the QDMA receives the descriptor, and stop when the host finishes receiving the packet. In Table 5.1 shown below we can see the results of our simulations for different packet sizes.

	Outgoing packet [ns]		Incoming packet [ns]	
Packet Size	**Descriptor**	**Write**	**Descriptor**	**Read**
128B	540	941	532	886
256B	540	971	536	915
512B	540	1005	536	977
1KB	540	1120	536	1038
2KB	540	1236	528	1166

Table 5.1: Case 1 results. All times shown are in nanoseconds.

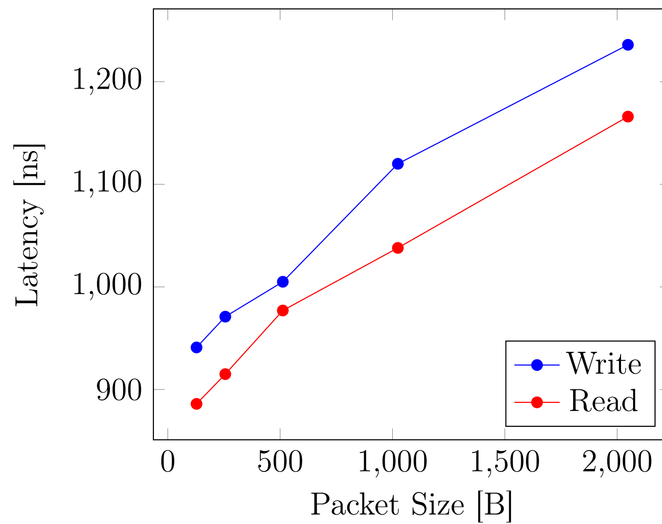

Figure 5.5: This figure shows the latency as a function of packet size for both write and read operations in **Case 1**. The results are obtained from simulation.

According to the quality metric 5.1.1, lower latency is better; therefore, write operations perform better than reads, as reads consistently exhibit higher latency, especially for larger packet sizes. We further observe that the initial gap between writes and reads is small, but it widens progressively as the packet size increases.

5.3.3. More Fine-Grained Results

As shown in Figure 5.6 below, we can further divide the path of a packet inside OpenNIC into four sections: host to QDMA (green), inside the QDMA subsystem (blue), from the QDMA to the CMAC (yellow), and inside the CMAC subsystem (purple).

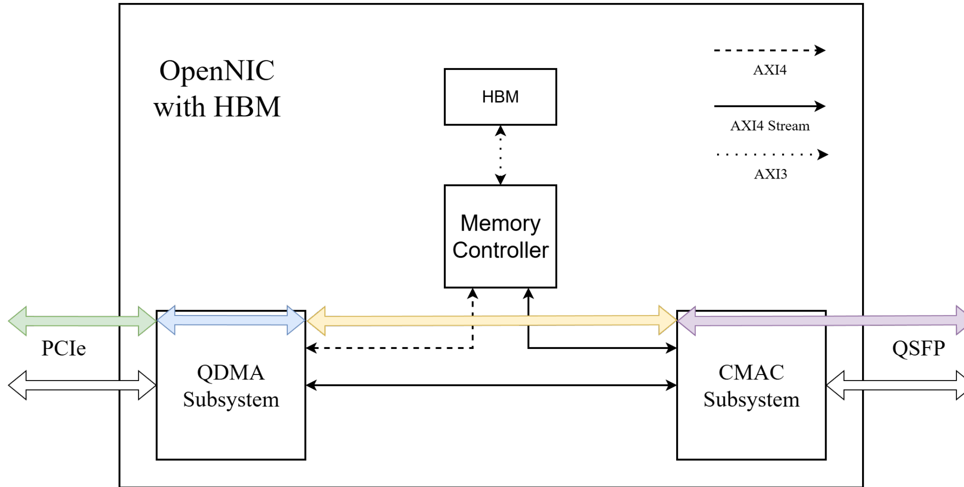

Figure 5.6: The communication path of simulation case 1 divided into four sections.

	Outgoing packet [ns]				**Incoming packet [ns]**			
Packet Size	**Host-QDMA**	**QDMA**	**QDMA-CMAC**	**CMAC**	**CMAC**	**CMAC-QDMA**	**QDMA**	**QDMA-Host**
128B	560	320	16	45	62	16	520	288
256B	576	328	16	51	75	16	520	304
512B	600	328	16	61	105	16	520	336
1KB	672	344	16	88	162	16	488	372
2KB	740	344	16	136	276	16	428	446

Table 5.2: Case 1 results for the fine-grained communication path. All times shown are in nanoseconds.

We can see two key differences from this table. The time spent communicating over PCIe is greater for outgoing packets because the QDMA has to send additional memory read requests, adding an additional trip. Meanwhile, incoming packets spend more time inside the QDMA, because they have to wait for the QDMA to start sending the descriptor read request.

5.4. Simulation Case 2: CMAC - HBM Communication

This simulation case explores a direct communication path between the **CMAC** and the **HBM**. This scenario is used when incoming packets entering the CMAC need to be cached inside the HBM. A key challenge in this design is bridging the AXI Stream interface of the CMAC with the AXI memory mapped interface of the HBM. Such bridging is done by incorporating a Controller built on top of Xilinx's Datamover ip.

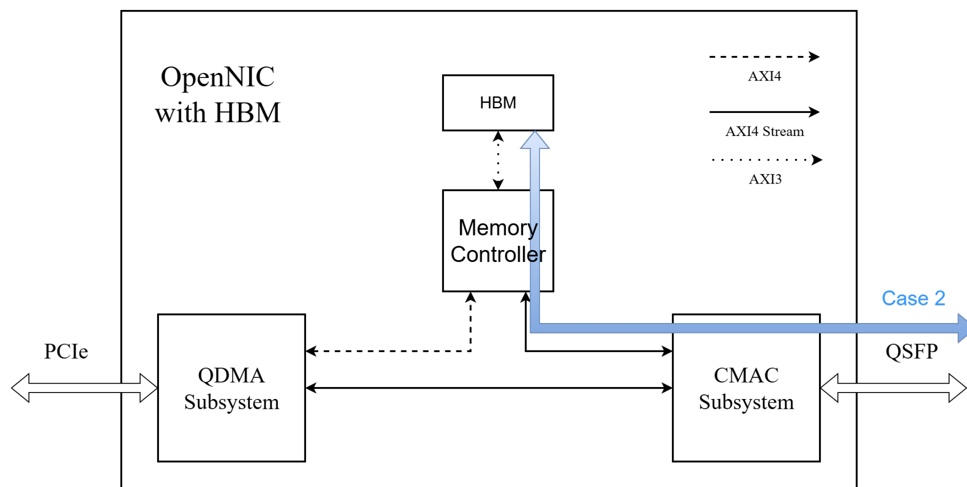

Figure 5.7: The red arrow indicates the path the packets take in this simulation case.

5.4.1. Operation Flow

In order to understand the results of this case, we must elaborate on the functionality of the **AXI DataMover**. Transactions are initiated by the DataMover by providing it with either **S2MM** or **MM2S** commands. These commands are processed by the DataMover and translated into either read or write transactions. In simulation, the commands are sent to the DataMover externally via a command generation unit. However, this will not be the case in the hardware implementation, where commands will be assembled within the bridging hardware itself.

In a *write-to-HBM* scenario, an S2MM command, along with an incoming packet from the CMAC, are sent to the DataMover. These are then fetched by the DataMover resulting in a "packet forward" to the HBM. On the other hand, in a *read-from-HBM* scenario, an MM2S command is sent to the DataMover, which initiates a read request to the HBM to obtain the packet and then forwards it to the CMAC.

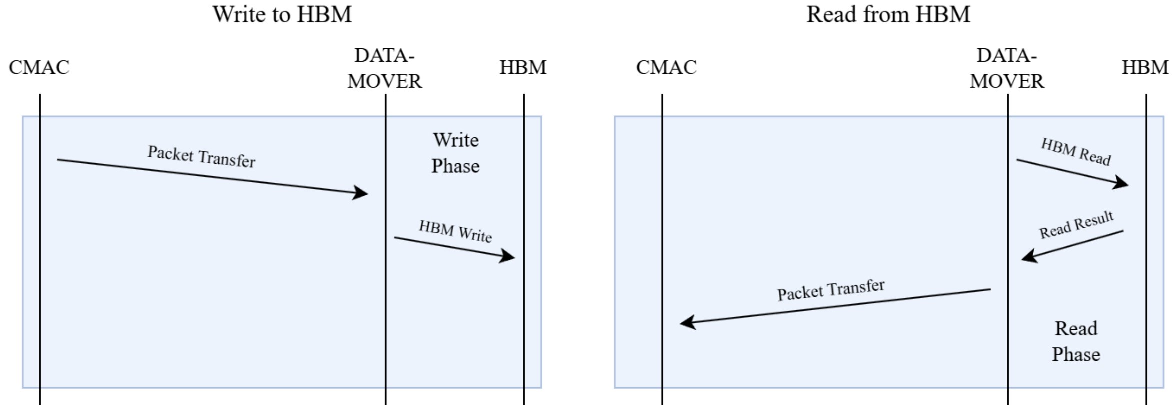

Figure 5.8: Flow diagram of a packet transfer for case 2.

In the Figure 5.8 above, we show a diagram of this process. Also, to better understand.

The source of latency in our simulation case, can be noticed by scrutinizing the diagram. We note that, by observing the diagram, overall we expect that reading a packet would take more than writing one. This is due to the extra datapath traversal involved in reading a packet. Meanwhile, we expect that the DataMover fetching phase to be the same in both directions.

5.4.2. Simulation Results

For writing to the HBM, we measure the time from when the CMAC receives a packet to when we receive the HBM's ACK in the form of the AXI b-channel in the DataMover. For reading from the HBM, we start the timer when the DataMover hardware receives the read command, and stop it when the CMAC finishes receiving the packet. In table 5.3 below we show the results for different packet sizes.

Packet Size	Write to HBM [ns]	Read from HBM [ns]
128B	261	635
256B	281	660
512B	325	722
1KB	417	834
2KB	593	1064
4KB	949	1536

Table 5.3: Case 2 results, All times shown are in nanoseconds.

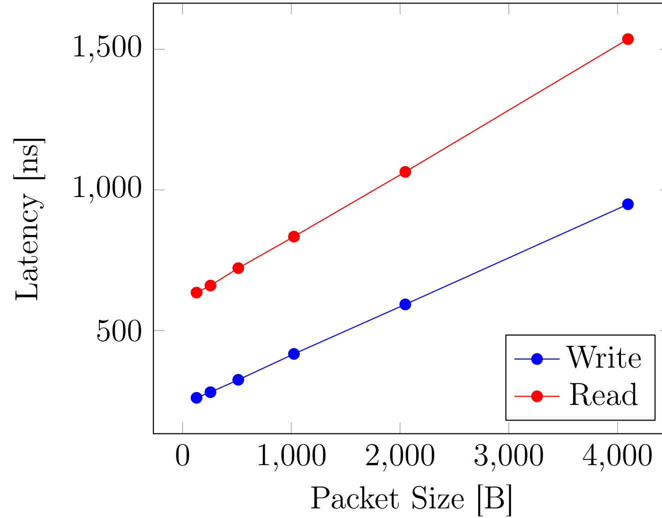

Figure 5.9: This figure shows the latency as a function of packet size for both write and read operations in **Case 2**. The results are obtained from simulation.

According to the quality metric 5.1.1, lower latency is better; therefore, write operations perform better than reads, as reads consistently exhibit higher latency, especially for larger packet sizes. From the graph, we also observe a widening gap in latency between reads and writes to the HBM . Moreover, we can deduce that the latencies grow linearly with the packet size, exhibiting an $\mathcal{O}(n)$ latency trend for both reading and writing.

5.4.3. More Fine-Grained Results

We can further divide the read and write operations into three sections. In a write scenario, we first measure the time spent in the CMAC (Purple) from when it receives the first beat of a packet until the first beat exits. In the second (Yellow) Path associated with the DataMover, we measure the time from when the first beat of the packet enters the DataMover until the first beat of packet is written in the memory-mapped write request. Finally, in the third section (Red) representing the communication path between the DM and the HBM, we start measuring from when the first beat of the packet is posted on the write request bus until the AXI b-channel in the DataMover is asserted.

Conversely in a read scenario, the measurement procedure changes a bit. In the first (Purple) section, we measure from when the packet has fully entered the CMAC until it has fully emigrated the CMAC. Regarding the second section in (Yellow), we measure two time intervals and combine them together. The first time interval is considered from when a read command is sent to the DataMover until the memory mapped read request is posted on the bus. Yet, for the second time interval; we consider the time from when the

DataMover receives the packet from the HBM until it fully exits the DataMover. Finally, for the third section (Red), we consider the time interval from when the read request is posted until the data is sent from the HBM to the DataMover.

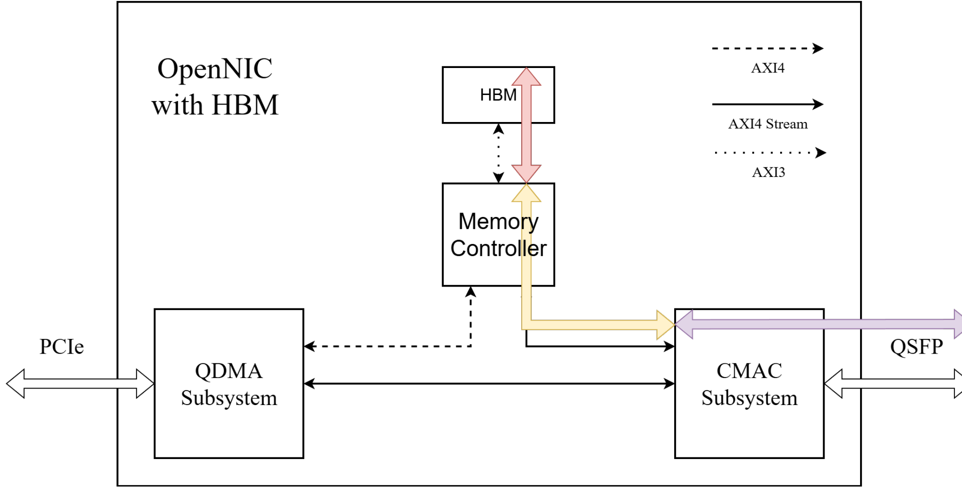

Figure 5.10: The communication path of simulation case 2 divided into three sections.

	Write to HBM [ns]			**Read from HBM [ns]**		
Packet Size	**CMAC**	**DM**	**DM-HBM**	**DM-HBM**	**DM**	**CMAC**
128B	73	20	168	524	48	63
256B	78	20	184	540	44	76
512B	89	20	216	576	44	106
1KB	117	20	296	636	44	154
2KB	165	20	408	764	44	256
4KB	265	20	664	1036	44	456

Table 5.4: Case 2 results for the fine-grained communication path. All times shown are in nanoseconds.

We can notice, that in both read and write operations, most of the time spent is on the DataMover-HBM communication part. Moreover, we notice that the time spent in the DM is much higher when it comes to reading from the HBM versus writing to HBM.

5.5. Simulation Case 3: HBM - Host Memory Communication

This simulation case focuses on measuring the latency of data transfers between two memories, the **HBM** and the **Host Memory**. The idea here is to synchronize the packets we cached in HBM with the host memory. One challenge in this design is modifying the OpenNIC's QDMA to accommodate for both the AXI Stream and AXI memory mapped interfaces, as the latter is needed to communicate with the HBM.

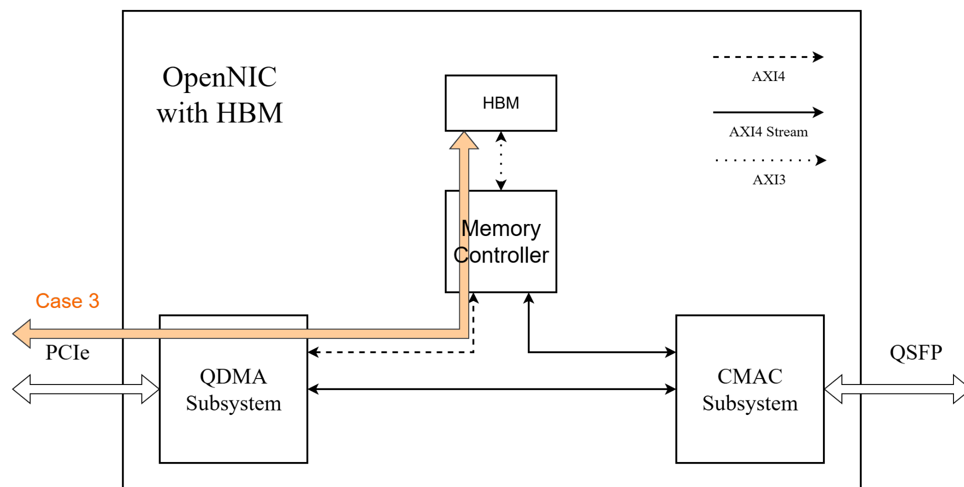

Figure 5.11: The red arrow indicates the path the packets take in this simulation case.

5.5.1. Operation Flow

Like for the first simulation case, we divided the operations into two sections, one for the descriptor fetch and one for the actual write or read operation. In both directions, the QDMA will fetch the descriptors as soon as it receives the updated producer index.

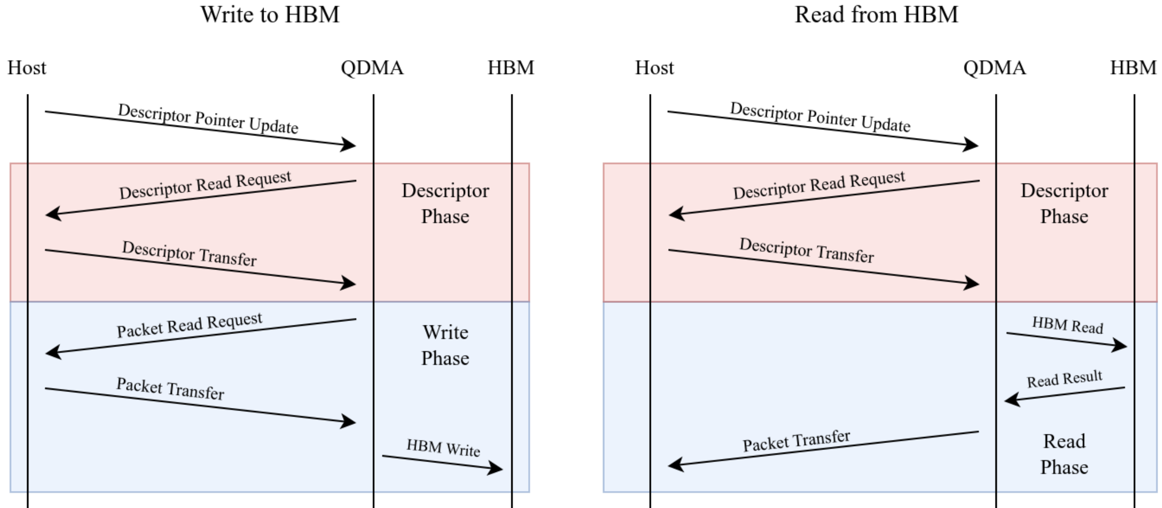

Figure 5.12: Flow diagram of a packet transfer for case 3.

From the diagram shown in the Figure 5.12, we can expect the two descriptor phases to take a similar amount of time; in this case, the descriptors are all of the same size, 32B. We can also expect that reading should take less time, because it avoids an extra trip over PCIe that the write needs for sending the packet read requests.

5.5.2. Simulation Results

We measure the descriptor phase exactly as in the first simulation case presented in section 5.3. For writing to the HBM, we measure the time from when the QDMA receives the descriptor to when we receive the HBM's ACK in the form of the AXI b-channel. For reading from the HBM, we start the timer when the QDMA receives the descriptor, and stop it when the host finishes receiving the packet. In Table 5.5 below we show the results for different packet sizes.

	Write to HBM [ns]		**Read from HBM [ns]**	
Packet Size	**Descriptor**	**Write**	**Descriptor**	**Read**
128B	544	1160	540	956
256B	540	1200	544	992
512B	544	1276	544	1072
1KB	544	1463	540	1136
2KB	544	1576	540	1264
4KB	544	1828	536	3617

Table 5.5: results. All times shown are in nanoseconds.

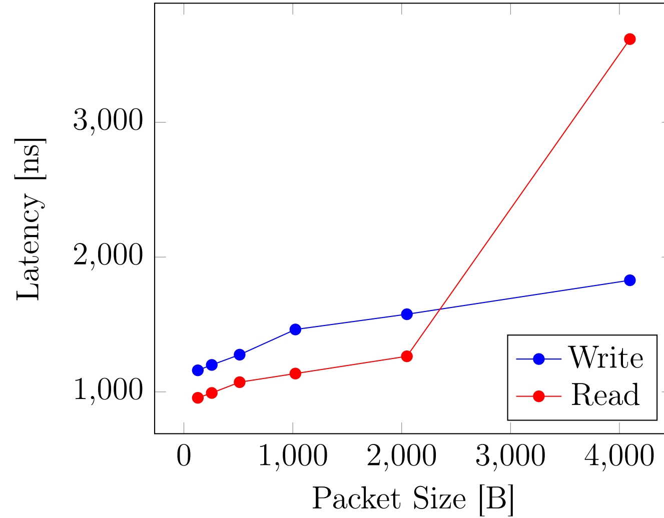

Figure 5.13: This figure shows the latency as a function of packet size for both write and read operations in . The results are obtained from simulation.

According to the quality metric 5.1.1, lower latency is better; therefore, write operations perform better than reads, as reads consistently exhibit higher latency, especially for larger packet sizes. From the graph, we also observe a massive spike in latency when reading 4KB or more from the HBM. This occurs because the operation inevitably crosses a 4KB address boundary, which is not permitted by the AXI4 specification, causing the read to be split into two transactions. Apart from this effect, the latencies for both reads and writes grow linearly with packet size, exhibiting an $\mathcal{O}(n)$ trend.

5.5.3. More Fine-Grained Results

We can further divide the read and write operations into three sections each. In the first we measure the time spent communicating over PCIe between the host to the QDMA (red), in the second the time spent inside the QDMA subsystem (blue), and in the third section the time spent communicating between the QDMA and the HBM (yellow).

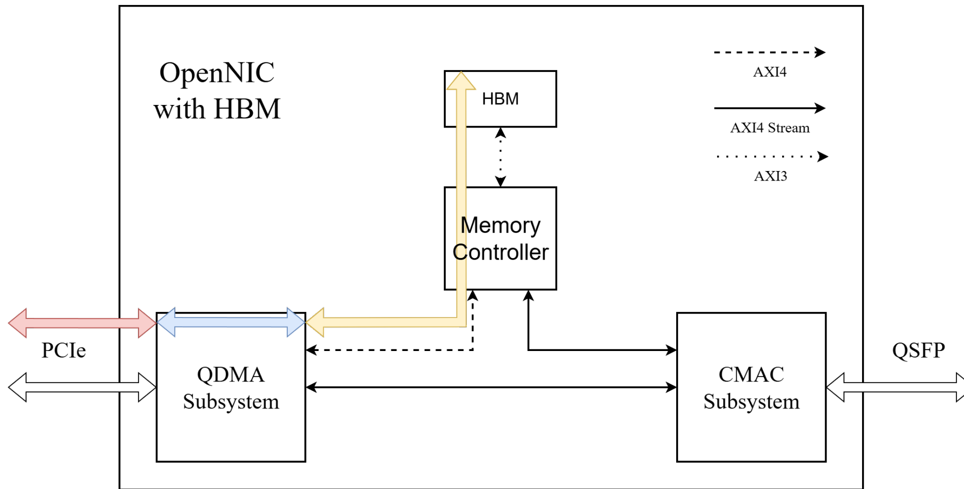

Figure 5.14: The communication path of simulation case 3 divided into three sections.

	Write to HBM [ns]			**Read from HBM [ns]**		
Packet Size	**Host-QDMA**	**QDMA**	**QDMA-HBM**	**HBM-QDMA**	**QDMA**	**QDMA-Host**
128B	560	408	192	284	384	288
256B	576	416	208	308	380	304
512B	604	432	240	340	396	336
1KB	679	480	304	404	332	400
2KB	740	404	432	528	212	532
4KB	916	228	684	1204	212	2480

Table 5.6: results for the fine-grained communication path. All times shown are in nanoseconds.

As for the first simulation case, write operations spend more time communicating over PCIe, due to having an additional trip caused by additional memory read requests.

5.6. Prototyping

In this section, we take a closer look at how our replication architecture performs in practice. We begin by outlining the architecture itself and measuring the latency of core replication operations such as reads, writes and writes to leader.

5.6.1. Functional Correctness

Before looking at performance, we first checked that the replication architecture worked as intended. In simulation, we tested different traffic patterns to make sure updates were applied in order, acknowledgments matched the right writes, and no data was lost or duplicated. The traffic injection mechanism for the architecture simulation, was the same as the one used in the previous simulation cases.

On hardware, the two-node setup confirmed that when a write is issued to a leader, it is correctly propagated to the follower and stored, as expected in a leader-follower replication protocol. With these checks in place, we could be confident in the design's correctness before moving on to performance results.

5.6.2. Prototype Results

Below we present the results obtained with our replication NIC design, focusing on latency as defined in Quality Metric 5.1.1. We will present different types of operations: reads, writes to replicas, and writes to leaders.

Reads and writes

We start with presenting results for reads and writes to replicas. While reads can be processed by any node, writes can only be processed by leaders. But, we would like to single out the latency also for writes being sent from leaders to replicas during replication, as this result can be better compared with the read operations.

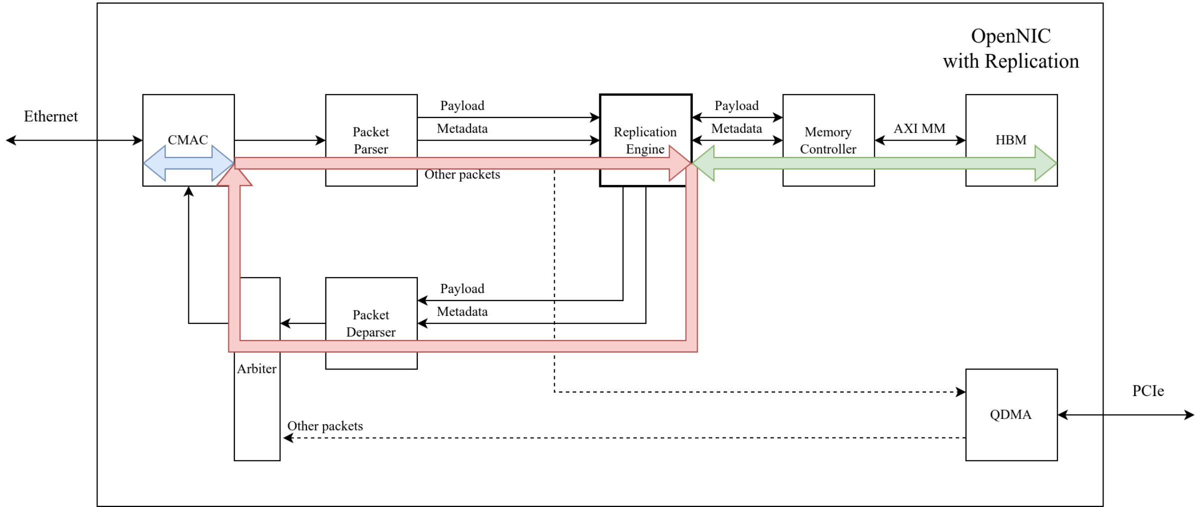

Figure 5.15: Diagram showing how the measurements are divided based on our replication NIC's architecture.

Average read [ns]			**Average write [ns]**		
873			856		
CMAC	**Replication**	**Memory**	**CMAC**	**Replication**	**Memory**
225	272	376	188	284	384

Table 5.7: Average latency for reads and writes to replicas

In Figure 5.15 we show what each column represents in Table 5.7. In blue we have the CMAC section, including its packet adapter. In red we have the full replication loop. Consisting of the packet parser, replication engine, packet deparser, and arbitrer. Finally, in green we single out the memory controller and the HBM latency as a whole.

	CMAC		**Replication**		**Memory**
	RX	TX	RX	TX	
READ	71	154	152	120	376
WRITE	117	71	164	120	384

Table 5.8: Fine-grained average latency results for read and write datapaths.

Furthermore, Table 5.8 presents fine-grained results for both RX and TX paths of each component. This provides a deeper view of where additional latency is introduced, allowing us to distinguish between RX and TX behavior in the CMAC and replication paths, while the memory is represented as a single access path.

In the replication component, the RX path introduces more latency than the TX path. Meanwhile, for the CMAC, the results show that during read operations the TX path is more expensive, which is expected since the entire packet must be sent back to the client; whereas during write operations the RX path dominates, as the whole packet is received and written.

Leader writes

We consider a leader write; the operation that occurs when a client sends a write request to a leader. This includes the time it takes for the leader to send write requests to replicas, wait for their acknowledgements, and send a final acknowledgement back to the client who initially sent the request. Of course our measurements in this case are going to depend on the network's latency, as we are taking into account the communication between multiple nodes. But, we are able to single out this added latency and present meaningful results.

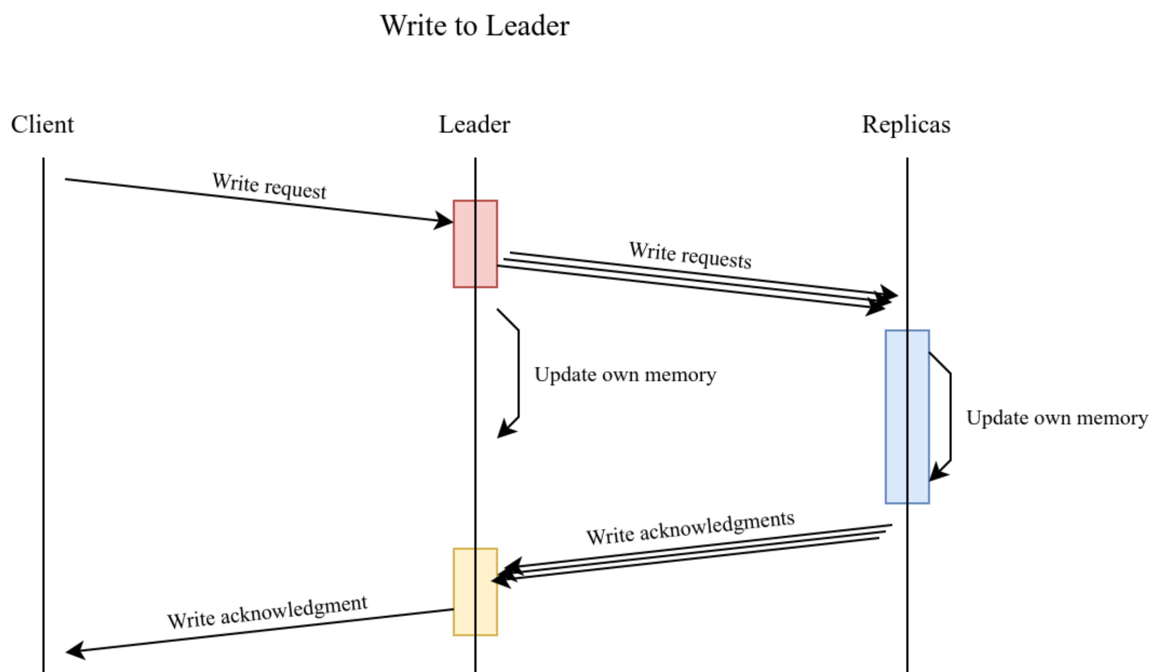

Figure 5.16: Diagram showing the flow of packet during a write to a leader.

Average Leader Write [ns]			
Without Network			With Network
1776			2352
Leader	Replica	Leader [Write ack]	Network
472	856	448	952

Table 5.9: Latency breakdown for a leader write operation, reported with and without network contribution.

In Figure 5.16, we show what each column represents in Table 5.9, similar to what we did previously. In red, we have the Leader section which represents the time it takes for the leader to receive a write request and replicate this to all other nodes. In other words, it accounts for the latency of the CMAC plus the Replication, but without the Memory. This is because storing in memory happens in the background; since the replication part sends the write request to the follower nodes without waiting for it's write to complete, unlike regular write acknowledgments. In blue we have the average time it takes for replicas to execute the write and send the acknowledgement back to the leader. We took this number from Table 5.7, allowing us to also single out the latency added by the network. Finally, in yellow we have the time it takes for the leader to send the final acknowledgment to the client, after it has received acknowledgments from all replicas. The later, accounts for also the latency of only the CMAC and replication, since no memory is involved in this operation. Note that the network latency at the end might vary with the cable length, in our case the cable was only 0.5 meters

Note that the measured network latency accounts for 2 round trips across a 0.5 meters optical cable, and this component grows linearly with cable length due to the propagation delay of light in fiber [4].

5.7. Benchmarking

In this section, we examine and compare how our design differs from a Vanilla OpenNIC, both in terms of latency and resource utilization, with a particular focus on the additional overhead introduced by our replication hardware. Moreover to put our results in perspective, we also benchmark the hardware implementation against a software-based replication model, showing the advantages of moving replication into the SmartNIC.

5.7.1. Vanilla OpenNIC Benchmark

To quantify the hardware overhead and latency overhead introduced by our replication hardware, we refer to the quality metrics 5.1.2 and 5.1.1.

Since the replication modules were implemented directly inside the OpenNIC shell, it is important to evaluate how much additional FPGA fabric they require on top of the vanilla design. On top of that, it is also essential to quantify the additional latency imposed by the replication modules on the OpenNIC shell datapath.

Resource Utilization

To start, we begin by analyzing the resource utilization. Table 5.10 reports resource utilization in terms of logic, memory, and specialized blocks, with a breakdown between the baseline OpenNIC shell, the added replication hardware, and the additional memory hardware.

Resource	Standard OpenNIC	Replication	Memory	% of U55C
LUT as Logic	92,361	7,007	13,151	8.6%
LUT as Memory	17,663	2,672	2,902	3.9%
CLB	25,818	1,985	4,817	20.0%
Block RAM	478	37	37.5	27.4%
DSP Slices	4	0	0	$<$0.1%

Table 5.10: FPGA resource utilization: baseline OpenNIC and additional overhead from replication and memory hardware, relative to U55C device capacity.

Overall, we can deduce from the table, that the OpenNIC shell consumes the majority of resources, as expected, while the additional replication hardware introduces moderate overhead. Specifically, replication logic adds about 7.5% more logic LUTs (7,007 on top of 92,361) and 2,672 additional LUTs configured as memory, while also increasing CLB usage by roughly 8%. Memory-related hardware contributes a further 13,151 logic LUTs and 2,902 memory LUTs, as well as 4,817 CLBs, which is consistent with the cost of buffering and interfacing with HBM. BRAM usage increases only slightly, with 74.5 blocks for both replication and memory components combined. Regarding DSPs, both memory and replication hardware do not make use of any.

The utilization figures also need to be understood in the context of the overall U55C device capacity, which demonstrate that the replication protocol can be integrated into the SmartNIC without exhausting resources, leaving room for further extensions or user logic.

Added Latency Analysis

The main difference between the Vanilla OpenNIC datapath and our replication baseline lies in the addition of two modules: the replication module and the memory module. While the CMAC component remains identical to the vanilla design, packets in our baseline pass through the mentioned extra stages, which introduce additional latency into the CMAC datapath. Each of these modules accounts for a measurable share of the added latency, as summarized in Table 5.11. In contrast, for reference in Table 5.12, we only consider the CMAC latency and not the QDMA, since its path remains unchanged in our design and is not involved in the replication mechanism. The QDMA path is only used for non-replication packets, and while it could be extended in the future to support replication-related operations, it is outside the scope of our evaluation.

Operation	Replication [ns]	Memory [ns]	Total Added [ns]
Read	272	376	648
Write	284	384	668
Write to Leader	544	384	928

Table 5.11: Added latency introduced by replication and memory modules compared to the Vanilla OpenNIC datapath.

Datapath	QDMA [ns]	CMAC [ns]	Total [ns]
RX Latency	360	88	448
TX Latency	504	162	666

Table 5.12: Baseline end-to-end latency of Vanilla OpenNIC for CMAC and QDMA datapaths.

As shown in Table 5.11, the replication and memory modules together add 648∼ns for reads, 668∼ns for writes, and up to 928∼ns for write-to-leader operations. In theory we only consider these values as the additional latency that impact the CMAC datapath.

5.7.2. Software replication Benchmark

To promote in-hardware replication, we have also conducted an experiment where packets go through the Vanilla OpenNIC, but this time the replication happens in software. Table 5.13 summarizes the results of measurements for basic read and write operations, as well as writes to a leader.

Operation	**Software [μs]**	**Hardware [ns]**
Read	45	873
Write	48	856
Write to Leader	164	1776

Table 5.13: Comparison of software- and hardware-based replication latency.

For comparison, our hardware baseline achieves significantly lower latencies across all operations. In particular, read and write operations complete in the sub-microsecond range, while the write-to-leader path, which involves forwarding updates to a follower, is also an order of magnitude faster than its software counterpart. This demonstrates the clear advantage of offloading replication into the SmartNIC datapath.

6 Conclusions

This thesis presented the design, implementation, and evaluation of a replication mechanism offloaded to a SmartNIC, with the goal of reducing both the persistence latency and communication overhead typically associated with software-based replication. Building on Xilinx's OpenNIC framework, we extended the architecture with dedicated replication and memory modules, and integrated them into the datapath alongside the CMAC and QDMA components.

Through simulation we demonstrated the functionality of the our approach, and with hardware prototyping on the Alveo U55C board, we have proven functional correctness of our architecture. Our results have shown a great achievement with regards to reducing the intended latencies, operations such as reads and writes achieve sub-microsecond latency, and even write-to-leader operations remain well below the equivalent software replication cost. Fine-grained measurements further revealed the contribution of individual components, to highlight the additional latencies introduced on the vanilla data-path.

From a resource perspective, the replication modules introduce only a moderate overhead relative to the Vanilla OpenNIC design, confirming that replication can be offloaded without prohibitive cost in FPGA fabric. The benchmarks against both Vanilla OpenNIC and software replication highlight the trade-off between added hardware complexity and the substantial performance benefits gained by moving replication into the NIC datapath.

Overall, this work demonstrates the feasibility and advantages of hardware-assisted replication, showing that SmartNICs can be leveraged to accelerate traditionally software-bound mechanisms. Future work could extend the QDMA path to support replication-related operations, optimize the memory subsystem to further reduce latency, introduce replication load balancer modules or an AI replication policy, or scale the design to multi-node clusters to validate its performance under real-world distributed workloads.

Bibliography

[1] M. K. Aguilera, N. Ben-David, R. Guerraoui, V. J. Marathe, A. Xygkis, and I. Zablotchi. Microsecond consensus for microsecond applications. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, Santa Clara, CA, USA, Nov. 2020. USENIX Association. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/aguilera.

[2] AMD. *AXI High Bandwidth Memory (HBM) Controller v1.0 Product Guide*, 2024. URL https://docs.amd.com/r/en-US/pg276-axi-hbm/Introduction. https://docs.amd.com/r/en-US/pg276-axi-hbm/Introduction [Accessed: 4/3/2025].

[3] I. AMD. Amd alveo™ u55c high performance compute card. https://www.amd.com/en/products/accelerators/alveo/u55c/a-u55c-p00g-pq-g.html [Accessed: 5/3/2025].

[4] F. Azendorf, A. Dochhan, and M. H. Eiselt. Accurate single-ended measurement of propagation delay in fiber using correlation optical time domain reflectometry. *ArXiv preprint*, 2022.

[5] A. Chiao. A practical guide to the storage hierarchy. https://arthurchiao.art/blog/practical-storage-hierarchy/, 2022. Accessed: September 15, 2025.

[6] Z. Duan, H. Lu, H. Liu, X. Liao, H. Jin, Y. Zhang, and S. Wu. Hardware-supported remote persistence for distributed persistent memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384421. doi: 10.1145/3458817.3476194. URL https://doi.org/10.1145/3458817.3476194.

[7] V. Gavrielatos, A. Katsarakis, and V. Nagarajan. Odyssey: The impact of modern hardware on strongly-consistent replication protocols. In *Sixteenth European Conference on Computer Systems (EuroSys '21)*, EuroSys '21, page 16, New York, NY, USA, 2021. ACM. doi: 10.1145/3447786.3456240.

[8] X. Inc. Alveo u55c data center accelerator card product brief, 2021. URL `https://www.xilinx.com/content/dam/xilinx/publications/product-briefs/alveo-u55c-product-brief.pdf`. Accessed: September 2025.

[9] S. Jha, J. Behrens, T. Gkountouvas, M. Milano, W. Song, E. Tremel, R. v. Renesse, S. Zink, and K. P. Birman. Derecho: Fast state machine replication for cloud services. *ACM Transactions on Computer Systems (TOCS)*, 36(2):49, Mar. 2019. doi: 10.1145/3302258.

[10] M. R. S. Katebzadeh, A. Katsarakis, and B. Grot. Reliable replication protocols on smartnics, 2025. URL `https://arxiv.org/abs/2503.18093`.

[11] A. Katsarakis, V. Gavrielatos, M. R. S. Katebzadeh, A. Joshi, A. Dragojevic, B. Grot, and V. Nagarajan. Hermes: a fast, fault-tolerant and linearizable replication protocol. In *Proceedings of the 2021 ACM Symposium on Cloud Computing*, Virtual Event, USA, 2021. ACM. University of Edinburgh, Intel, Microsoft Research.

[12] E. Kfoury, S. Choueiri, A. Mazloum, A. AlSabeh, J. Gomez, and J. Crichigno. A comprehensive survey on smartnics: Architectures, development models, applications, and research directions. *arXiv preprint arXiv:2405.09499*, 2024. URL `https://arxiv.org/abs/2405.09499`.

[13] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan. Hyperloop: group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 297–312, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355674. doi: 10.1145/3230543.3230572. URL `https://doi.org/10.1145/3230543.3230572`.

[14] C. H. Lam. Storage class memory. In *2010 10th IEEE International Conference on Solid-State and Integrated Circuit Technology*, pages 1080–1083, 2010. doi: 10.1109/ICSICT.2010.5667551.

[15] X. Ltd. Down to the tlp: How pci express devices talk (part i), 2012. `https://xillybus.com/tutorials/pci-express-tlp-pcie-primer-tutorial-guide-1` [Accessed: 4/3/2025].

[16] X. Ltd. Down to the tlp: How pci express devices talk (part ii), 2012. `https://xillybus.com/tutorials/pci-express-tlp-pcie-primer-tutorial-guide-2` [Accessed: 4/3/2025].

[17] Molex. Understanding qsfp transceivers for data centers and networking, 2020. URL https://www.molex.com/en-us/products/connectors/qsfp. Accessed: September 2025.

[18] A. Murat, I. Zablotchi, C. Burgelin, M. K. Aguilera, A. Xygkis, and R. Guerraoui. Swarm: Replicating shared disaggregated-memory data in no time. *arXiv preprint arXiv:2409.16258v1*, Sep 2024. URL https://arxiv.org/abs/2409.16258v1. École Polytechnique Fédérale de Lausanne (EPFL), Mysten Labs, VMware Research Group, Oracle Labs.

[19] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 327–341, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355674. doi: 10.1145/3230543.3230560. URL https://doi.org/10.1145/3230543.3230560.

[20] J. Postel. User Datagram Protocol. RFC 768, 1980. URL https://www.rfc-editor.org/rfc/rfc768.

[21] J. Postel. Transmission Control Protocol. RFC 793, 1981. URL https://www.rfc-editor.org/rfc/rfc793.

[22] A. Psistakis, F. Chaix, and J. Torrellas. Minos: Distributed consistency and persistency protocol implementation & offloading to smartnics. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, USA, 2024. University of Illinois Urbana-Champaign, USA and FORTH, Greece, IEEE. ISBN 979-8-3503-9313-2. doi: 10.1109/HPCA57654.2024.00076.

[23] K. Seemakhupt, S. Liu, Y. Senevirathne, M. Shahbaz, and S. Khan. Pmnet: In-network data persistence. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 804–817, 2021. doi: 10.1109/ISCA52012.2021.00068.

[24] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso. Strom: smart remote memory. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368827. doi: 10.1145/3342195.3387519. URL https://doi.org/10.1145/3342195.3387519.

[25] Y. Taleb, R. Stutsman, G. Antoniu, and T. Cortes. Tailwind: Fast and atomic RDMA-based replication. In *2018 USENIX Annual Technical Conference (USENIX*

ATC 18), pages 851–863, Boston, MA, July 2018. USENIX Association. ISBN 978-1-939133-01-4. URL `https://www.usenix.org/conference/atc18/presentation/taleb`.

[26] L. Thostrup, J. Skrzypczak, M. Jasny, T. Ziegler, and C. Binnig. Dfi: The data flow interface for high-speed networks. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1825–1837, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3452816. URL `https://doi.org/10.1145/3448016.3452816`.

[27] J. Wahlgren, R. Pearce, G. Schieffer, M. Gokhale, and I. Peng. Disaggregated memory with smartnic offloading: a case study on graph processing. *arXiv preprint arXiv:2410.02599*, 2024. arXiv:2410.02599v1 [cs.DC], 3 Oct 2024.

[28] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui. Apus: Fast and scalable paxos on rdma. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '17)*, SoCC '17, page 14, New York, NY, USA, 2017. doi: 10.1145/3127479.3128609.

[29] Xilinx. open-nic. URL `https://github.com/Xilinx/open-nic`. GitHub repository for open-nic project.

[30] E. Zamanian, X. Yu, M. Stonebraker, and T. Kraska. Rethinking database high availability with rdma networks. *Proc. VLDB Endow.*, 12(11):1637–1650, July 2019. ISSN 2150-8097. doi: 10.14778/3342263.3342639. URL `https://doi.org/10.14778/3342263.3342639`.

[31] T. Ziegler, J. Nelson-Slivon, V. Leis, and C. Binnig. Design guidelines for correct, efficient, and scalable synchronization using one-sided rdma. *Proc. ACM Manag. Data*, 1(2), June 2023. doi: 10.1145/3589276. URL `https://doi.org/10.1145/3589276`.

List of Figures

2.1 Peak bandwidth of storage media, networking, and distributed storage solutions. [5]. . . . 8
2.2 The Alveo U55C Board [3]. . . . 10
2.3 64GB HBM Two Stack Configuration [2]. . . . 11

3.1 How caching is used in Chaapar, as shown in [10]. . . . 23
3.2 The active-memory replication protocol, as shown in [30]. . . . 24
3.3 A replication operation with Tailwind. In red are the appended pieces of metadata. The receiver CPU stays idle. . . . 25
3.4 Example of HERMES Reconfiguration and Recovery. Adapted from A. Katsarakis et al.(2021), Hermes: a fast, fault-tolerant and linearizable replication protocol [11] 27
3.5 Architecture of SWARM-KV with a single key. Adapted from: A. Murat et al. (2024), Swarm: Replicating shared disaggregated-memory data in no time [18] 28
3.6 Diagram of MINOS-O SmartNIC offloading. Adapted from: Psistakis et al. (2024), Minos: Distributed consistency [22] 29
3.7 APUS RDMA powered paxos workflow pipeline 30
3.8 Diagram demonstrating Mu's architecture. Adapted from : M. K. Aguilera et al. (2020), Microsecond consensus for microsecond applications [1] . . . 31
3.9 Diagram demonstrating Derecho's RDMA-Powered Zero-Copyy SST Update Flow 32
3.10 Improvements of pessimistic latches, as shown in [31]. . . . 34
3.11 Round trip time of a request with PMNet, as shown in [23]. . . . 35
3.12 An RDMA write. Note that the sender's CPU has no way of knowing when data is in persistent memory. . . . 36
3.13 Distributed Radix Hash Join with DFI flows. Adapted from : L. Thostrup et al. (2021), DFI: The data flow interface for high-speed netwoks [26] . . 37
3.14 StRoM's SmartNIC Architecture. . . . 38
3.15 SODA components flow of data and control path. . . . 39

4.1 Architectural diagram of the OpenNIC Shell connected to High-Bandwidth Memory (HBM). 41
4.2 Diagram showing the Datamover ip basic command word layout. 42
4.3 Diagram showing OpenNIC with the replication mechanism. 43
4.4 Diagrams showing the structure of replication packets and the metadata . 46

5.1 Modified OpenNIC with different simulation datapath. 48
5.2 Diagram showing our two-node replication hardware setup. 49
5.3 The red arrow indicates the path the packets take in this simulation case. . 51
5.4 Flow diagram of a packet transfer for case 1. 52
5.5 This figure shows the latency as a function of packet size for both write and read operations in **Case 1**. The results are obtained from simulation. . 53
5.6 The communication path of simulation case 1 divided into four sections. . . 54
5.7 The red arrow indicates the path the packets take in this simulation case. . 55
5.8 Flow diagram of a packet transfer for case 2. 56
5.9 This figure shows the latency as a function of packet size for both write and read operations in **Case 2**. The results are obtained from simulation. . 57
5.10 The communication path of simulation case 2 divided into three sections. . 58
5.11 The red arrow indicates the path the packets take in this simulation case. . 59
5.12 Flow diagram of a packet transfer for case 3. 60
5.13 This figure shows the latency as a function of packet size for both write and read operations in . The results are obtained from simulation. 61
5.14 The communication path of simulation case 3 divided into three sections. . 62
5.15 Diagram showing how the measurements are divided based on our replication NIC's architecture. 64
5.16 Diagram showing the flow of packet during a write to a leader. 65

List of Tables

2.1 Compute Resources and Specifications of AMD U55C Board. 10

3.1 Taxonomy of replication protocols from the literatures 19
3.2 Analysis of the literature replication protocol architectures, the columns represent : Definition : A brief overview of the protocols design ; System Architecture : Hardware roles in replication ; Interface Type : Compatible storage format with it's replication interface ; Method : The implemented mechanism for replication ; Communication Protocol : Transport mechanism used for data replication ; Support for persistency: Durability of replication ; Open Source : Whether the implementation is publicly available 21
3.3 Analysis of Replication Protocols Literature, the columns represent : Description : A brief overview of the protocols design ; Latency (µs) : Average & tail latencies measured in microseconds ; Throughput (kops) : System throughput in thousands of operations per second ; CPU Util : CPU utilization during operation ; Memory overhead : Memory overhead measured in bytes per key/object 22
3.4 Group network primitives introduced by Hyperloop. 26
3.5 Analysis of papers regarding SmartNIC memory innovations, the Columns represent : Goal : A brief overview of the solution goal ; Contribution : The implemented mechanism capability ; Open Source : Whether the implementation is publicly available 33

5.1 Case 1 results. All times shown are in nanoseconds. 53
5.2 Case 1 results for the fine-grained communication path. All times shown are in nanoseconds. 54
5.3 Case 2 results, All times shown are in nanoseconds. 56
5.4 Case 2 results for the fine-grained communication path. All times shown are in nanoseconds. 58
5.5 results. All times shown are in nanoseconds. 60

5.6 results for the fine-grained communication path. All times shown are in nanoseconds. 62
5.7 Average latency for reads and writes to replicas 64
5.8 Fine-grained average latency results for read and write datapaths. 64
5.9 Latency breakdown for a leader write operation, reported with and without network contribution. 66
5.10 FPGA resource utilization: baseline OpenNIC and additional overhead from replication and memory hardware, relative to U55C device capacity. . 67
5.11 Added latency introduced by replication and memory modules compared to the Vanilla OpenNIC datapath. 68
5.12 Baseline end-to-end latency of Vanilla OpenNIC for CMAC and QDMA datapaths. 68
5.13 Comparison of software- and hardware-based replication latency. 69

Acknowledgements

Acknowledgments

I would like to thank everyone in the DEIB Department for their support and collaboration during my thesis work. In particular, I would like to express my sincere gratitude to my supervisor, Prof. Davide Zoni, for his guidance, encouragement, and constant availability throughout the course of this work. His advice and continuous support have been fundamental in shaping both the direction and the quality of this thesis.

I am equally grateful to my co-supervisor, Prof. Gianni Antichi, for his constructive feedback, helpful discussions, and high availability. His continuous feedback along the way greatly contributed to improving this research.

I would also like to thank the Ph.D. students Gabriele Montanaro and Andrea Motta, Post-doc Andrea Galimberti for their technical support and advice during the development of this work. A special thanks goes to Andrea Galimberti, who was always available to help, even outside of working hours, and whose assistance was invaluable. I am also thankful to Gabriele Montanaro, who held several sessions with us during the work to explain important concepts and provide guidance that proved very useful in practice.

A special thanks also goes to all the professors who taught me during the HPC program; without their guidance, I would not have developed the skills I have today.

My deepest gratitude goes to my family abroad — my father, brother, grandmother, cousins, uncles, and my aunt — for their unconditional love and encouragement from abroad. Their belief in me has been a constant source of strength. I am also thankful to my friends abroad, whose support and motivation accompanied me throughout this journey.

Finally, I want to thank all my classmates from the HPC program, for their friendship, collaboration, and for making this experience both rewarding and enjoyable. A special thanks goes to Giorgio Massimo Fontanive, Luca Muscarnera, and Filippo Mininno for their constant support and encouragement throughout the program.