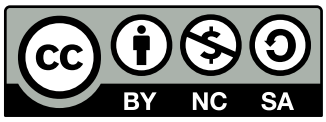
Aalto University
School of Science

Master's programme in Mathematics and Operations Research

Techno-Economic Analysis of Compute Express Link in Radio Networks

Juho Saranpää

Master's Thesis
2025



Aalto University
School of Science

Author Juho Saranpää

Title Techno-Economic Analysis of Compute Express Link in Radio Networks

Degree programme Mathematics and Operations Research

Major Systems and Operations Research

Supervisor Harri Hakula

Advisor Jari Karppinen

Collaborative partner Nokia Solutions and Networks Oy

Date 14 March 2025 **Number of pages** 63 **Language** English

Abstract

This thesis examines the transformative role of Compute Express Link (CXL) in addressing critical challenges in modern computing, such as resource utilization, scalability, and performance optimization. With a focus on Nokia's Cloud-RAN product, the research highlights how CXL's memory pooling, hardware cache coherency, and high-speed interconnect capabilities enable efficient resource allocation, lower CPU overhead, and reduced latency. By decoupling memory from compute nodes and supporting coherent, multi-host shared memory, CXL provides significant advantages over traditional paradigms like PCIe and kernel-mediated Linux networking.

Through the development of comprehensive cost and performance models, the thesis lays a foundation for evaluating the economic and technical impact of CXL adoption. These models assess key metrics, including memory bandwidth utilization, CPU efficiency, and system throughput, while integrating dimensions such as power consumption and hardware provisioning. While specific simulations highlight the potential of CXL to enhance throughput and optimize resource utilization, these models are designed as a framework for future studies. When real-world data becomes available, the models can be applied to quantify performance and cost implications at a high level, offering a versatile tool for analyzing CXL and other emerging technologies.

The research further explores CXL's compatibility with multi-tenancy principles, enabling flexible and scalable resource sharing in cloud environments. By reducing stranded resources and overprovisioning, CXL supports cost-effective, logically isolated multi-tenant architectures.

This work contributes to the field by not only demonstrating CXL's potential benefits but also providing adaptable models for evaluating its impact. These models ensure that future studies can effectively assess CXL's performance and cost efficiency, guiding the design and optimization of next-generation computing architectures as real-world deployments and data mature.

Keywords Compute Express Link, Memory Pooling, Linux Networking, Cost Modeling , High-speed Interconnect, Multi-tenancy, Cloud Computing

Aalto-yliopisto
Perustieteiden
korkeakoulu

Tekijä Juho Saranpää

Työn nimi Compute Express Link-teknologian teknillistaloudellinen analyysi radioverkoissa

Koulutusohjelma Mathematics and Operations Research

Pääaine Systems and Operations Research

Työn valvoja Harri Hakula

Työn ohjaaja Jari Karppinen

Yhteistyötaho Nokia Solutions and Networks Oy

Päivämäärä 14.3.2025 **Sivumäärä** 63 **Kieli** englanti

Tiivistelmä

Tämä diplomityö tutkii Compute Express Linkin (CXL) roolia modernin laskennan keskeisten haasteiden, kuten resurssien hyödyntämisen, skaalautuvuuden ja suorituskyvyn optimoinnin, ratkaisemisessa. Tutkimuksessa keskitytään erityisesti Nokian Cloud-RAN-tuotteeseen ja tarkastellaan, kuinka CXL:n muistin jakaminen, välimuistin koherenssi ja nopea liitäntä mahdollistavat tehokkaan resurssien kohdentamisen, pienemmän suoritinylikuorman ja alhaisemman viiveen. Erottamalla muistin laskentasolmuista ja tukemalla koherenttia, usean isännän jakamaa muistia CXL tarjoaa merkittäviä etuja perinteisiin lähestymistapoihin verrattuna.

Työssä kehitetään kattavia kustannus- ja tehokkuusmalleja, jotka luovat pohjan CXL:n käyttöönoton taloudellisten ja teknisten vaikutusten arvioinnille. Näissä malleissa tarkastellaan keskeisiä mittareita, kuten muistikaistan käyttöä, suorittimen tehokkuutta ja järjestelmän suorituskykyä, ja niihin sisällytetään myös energiankulutuksen näkökulma. Vaikka simulaatiot osoittavat CXL:n potentiaalin parantaa suorityskykyä ja optimoida resurssien käyttöä, mallit on suunniteltu alustaksi tuleville tutkimuksille. Kun oikeata dataa on saatavilla, malleja voidaan hyödyntää suorituskyvyn ja kustannusvaikutusten arviointiin laajalla tasolla, tarjoten monipuolisen työkalun CXL:n ja muiden kehittyvien teknologioiden analysointiin.

Tutkimus tarkastelee myös CXL:n yhteensopivuutta monikäyttäjäperiaatteiden kanssa, mikä mahdollistaa joustavan ja skaalautuvan resurssien jaon pilviympäristöissä. Vähentämällä käyttämättömiä resursseja ja ylimitoitusta CXL tukee kustannustehokkaita, loogisesti eristettyjä monikäyttäjäarkkitehtuureja.

Tämä työ tukee tutkimusta paitsi osoittamalla CXL:n mahdolliset hyödyt myös tarjoamalla mukautettavia malleja sen vaikutusten arviointiin. Nämä mallit varmistavat, että tulevat tutkimukset voivat tehokkaasti arvioida CXL:n suorituskykyä ja kustannustehokkuutta, ohjaten seuraavan sukupolven laskenta-arkkitehtuurien suunnittelua ja optimointia.

Avainsanat Compute Express Link, Kustannusarviointi, Pilvilaskenta, Radioverkot, Verkkoarkkitehtuuri, Tehokkuusmallit

Preface

First, I express my deepest gratitude to my family and friends for their great support throughout this thesis research. Your belief in me has been a source of motivation and strength.

I alsortfelt thanks to my supervisor, Dr. Harri Hakula, for his guidance, insightful feedback, and patience during the development of this thesis. His expertise and mentorship have been important in shaping this work. I am equally grateful to my Nokia advisor, Jari Karppinen, whose practical insights and professional advice have been crucial in connecting theoretical concepts to real-world applications.

The process of creating this thesis has been a great learning experience, broadening my understanding of fields beyond my own background in mathematics. I have explored aspects of computer science, electrical engineering, and telecommunications engineering, gaining an understanding of the interdisciplinary nature of modern technological advancements.

Completing this thesis has been a rewarding challenge, and I am grateful to everyone who has supported me along the way. Thank you for being part of this milestone in my academic journey.

Helsinki, 27 February 2025

Juho Saranpää

Contents

Abstract 3

Abstract (in Finnish) 4

Preface 5

Contents 6

1 Introduction 9
- 1.1 Motivation to this thesis . . . 9
- 1.2 Main results and findings . . . 9
- 1.3 Structure of the thesis . . . 10

2 Compute Express Link (CXL) 11
- 2.1 Motivation for CXL . . . 11
- 2.2 Generations . . . 13
  - 2.2.1 CXL 1.0/1.1 . . . 14
  - 2.2.2 CXL 2.0 . . . 14
  - 2.2.3 CXL 3.0 . . . 15
  - 2.2.4 CXL 3.1 . . . 16
- 2.3 Protocol types . . . 16
  - 2.3.1 CXL.io . . . 16
  - 2.3.2 CXL.cache . . . 17
  - 2.3.3 CXL.memory . . . 18
- 2.4 Device types . . . 18
  - 2.4.1 Type 1 . . . 18
  - 2.4.2 Type 2 . . . 19
  - 2.4.3 Type 3 . . . 20

3 Literature Review 22
- 3.1 Performance Studies - Microsoft Pond . . . 22
- 3.2 Performance Studies - Intel . . . 24
- 3.3 Cost models . . . 26
  - 3.3.1 Power Management in CXL . . . 28
  - 3.3.2 Power Consumption in Cloud RAN . . . 28
  - 3.3.3 Cost Study - Google . . . 29
  - 3.3.4 Cost studies - Yale & ByteDance . . . 31

4 Message-based communication in Nokia Cloud-RAN 34
- 4.1 Introduction . . . 34
- 4.2 Problem description: From messaging to data sharing . . . 34
- 4.3 Messaging in communication networking . . . 38
  - 4.3.1 Microservices and Messaging . . . 38
  - 4.3.2 Container Network Interface . . . 39

4.3.3 Kubernetes CNI 40
4.3.4 Linux Network messaging 40
4.3.5 Kernel by-pass and Dataplane Development Kit 43

5 Techno-Economic Analysis **45**
5.1 Total Cost Model 45
5.2 Performance models 47
5.2.1 Round-Trip Time (RTT) modeling 47
5.2.2 Throughput Model 52
5.3 Use Cases 54
5.3.1 Memory Pooling 54
5.3.2 Multi-Tenancy 57

6 Conclusions **59**

7 Contributions and Future Directions **60**

Abbreviations

ASIC	Application-Specific Integrated Circuit
CAPEX	Capital Expenditures
CNI	Container Network Interface
CPU	Central Processing Unit
CXL	Compute Express Link
DDR	Double Data Rate (Memory)
DIMM	Dual Inline Memory Module
DLLP	Data Link Layer Packets
DPDK	Data Plane Development Kit
GP-GPUs	General-Purpose Graphics Processing Units
GPUs	Graphics Processing Units
HBM	High-Bandwidth Memory
HDM	Host-Managed Device Memory
HTTP	Hypertext Transfer Protocol
IPC	Inter-Process Communication
MA	Microservices Architecture
MESI	Modified, Exclusive, Shared, Invalid (Cache Coherence Protocol)
NIC	Network Interface Card
NUMA	Non-Uniform Memory Access
OPEX	Operational Expenditures
PBR	Port-Based Routing
PCIe	Peripheral Component Interconnect Express
PDM	Private Device Memory
QoS	Quality of Service
RDMA	Remote Direct Memory Access
RPC	Remote Procedure Call
TCO	Total Cost of Ownership
VM	Virtual Machine
zNUMA	Zero-Core Virtual NUMA

1 Introduction

The rapid growth of cloud computing and high-performance computing has created an urgent need for more efficient resource utilization and scalable infrastructure. Compute Express Link (CXL) is a new interconnect technology that responses to this need of the new progress in data centers. CXL enables high-speed, low-latency communication between CPUs, memory, and accelerators. Also, CXL offers solutions such as memory pooling, dynamic resource allocation, and enhanced performance scalability.

This thesis explores the impact of CXL on modern computing environments, focusing on its ability to improve resource sharing and system efficiency. The study develops a framework for evaluating both the direct and indirect effects of CXL, including performance modeling and a total cost model. This research explores key use cases like memory pooling and multi-tenancy to show how CXL improves resource use, lowers hardware costs, and boosts scalability. It highlights how CXL solves major cloud and distributed computing challenges, making infrastructure more efficient and flexible.

The findings presented in this thesis aim to deepen the understanding of CXL's capabilities and establish its potential for next-generation computing systems. Through analytical models and use case evaluations, this work provides valuable insights into how CXL can improve resource management, enabling more cost-effective and scalable solutions for future workloads.

1.1 Motivation to this thesis

The study focuses on assessing the potential impact of CXL, especially concerning its implications for Microservices Architecture (MA) in telecommunications equipment systems. The aim is to evaluate the benefits in terms of cost, taking into account various factors, including performance, system complexity, implementation and maintenance costs (CAPEX and OPEX), application migration, and energy consumption. The ultimate goal is to provide a comprehensive understanding of the techno-economic aspects of CXL, ideally backed by relevant cost data.

The study faces challenges due to the early version of available CXL hardware, ongoing development of Linux drivers, and the CXL software bundle. In cases where empirical data is unavailable, simulations are conducted using in-house tools designed for CXL. The work is forward-looking, intended to examine the potential operational advantages of CXL in the coming years.

1.2 Main results and findings

This thesis shows that Compute Express Link is a powerful technology that improves resource utilization, scalability, and performance in cloud computing. By enabling memory pooling, reducing CPU load, and providing faster connections, CXL has a possibility to make systems more efficient and cost-effective. A key contribution of this work is the development of models that help evaluate CXL's impact on system performance and costs. The research also highlights CXL's benefits for cloud

computing by reducing wasted resources and improving multi-tenant environments. With its scalability and ongoing improvements, especially in CXL 3.0-specification, this technology is set to play a major role in future computing.

1.3 Structure of the thesis

This thesis is structured as follows: It begins with Compute Express Link section which provides an overview of CXL technology, detailing its protocols (CXL.io, CXL.cache, CXL.memory), device types, and generational advancements, highlighting its role in addressing memory coherence and scalability challenges. The Literature review evaluates the cost implications of CXL adoption through models and case studies, showing its potential to reduce memory provisioning costs and optimize total cost of ownership.

The thesis continues with Message-Based Communication in Nokia Cloud-RAN, showcasing CXL's role in enhancing Kubernetes-based networking by bypassing kernel bottlenecks. In Techno-Economic analysis, simulation-based evaluations of Round-Trip Time (RTT) and throughput highlight CXL's benefits in reducing CPU overhead, improving memory bandwidth, and enabling scalability. Use Cases and Future Directions explore practical applications such as memory pooling and multi-tenancy while identifying areas for further research, including integration with emerging interconnect technologies. Finally, the conclusions summarize the findings, highlighting the transformative potential of CXL to shape efficient and scalable computing architectures.

2 Compute Express Link (CXL)

This section explores the development and capabilities of CXL across its generations, focusing on how it has been developed to address the growing demands of modern computing systems. Starting with its introduction, the section provides an overview of key features in CXL 1.0/1.1, 2.0, and 3.0, with a comparative summary in Table 1. Each generation is examined in detail, highlighting contributions to memory pooling, resource sharing, and system scalability. The section also discusses core protocols—CXL.io, CXL.cache, and CXL.memory. Finally, the classification of CXL devices into Types 1, 2, and 3 is presented, highlighting their distinct roles in enhancing the memory and computing infrastructure. Together, this section provides a comprehensive view of how CXL is shaping the future of high-performance and distributed computing.

2.1 Motivation for CXL

CXL CXL has a broad definition that includes graphics processing units (GPUs), general-purpose graphics processing units (GP-GPUs), field-programmable gate arrays (FPGAs), and different purpose-built accelerators and storage devices. Traditionally, these devices have used the Peripheral Component Interconnect-Express® (PCIe®) serial interface. PCIe architecture features a host device, typically a CPU or chipset, and a multitude of peripheral devices, such as graphics cards, network cards, storage devices, and expansion cards. Employing a point-to-point topology, PCIe establishes direct connections between each peripheral device and the host through dedicated serial links referred to as "lanes." These lanes, composed of pairs of differential signaling lines, facilitate the bidirectional transmission of data. The PCIe architecture is a hierarchical structure comprising distinct layers, including the physical layer, the data link layer, and the transaction layer. [1].

Although the PCIe interface has been effective for various devices, it also exhibits inherent limitations. These limitations led to the development of CXL, which solves four key challenges and offers a more efficient way to connect devices. [2].

The first challenge addressed by CXL pertains to achieving coherent access to both system and device memory. Traditional systems establish coherence with system memory through DDR, integrating it with the CPU cache hierarchy. In contrast, PCIe devices access system memory non-coherently, passing through the host's root complex (RC) to maintain consistency with CPU caching semantics. This non-coherent access limits the PCIe device's ability to cache system memory for exploiting temporal or spatial locality, hindering the execution of atomic sequences of operations. Moreover, memory attached to a PCIe device is accessed non-coherently from the host, preventing its mapping to the cacheable system address space. While non-coherent accesses are suitable for streaming I/O operations, they pose challenges for accelerators, particularly in emerging applications such as Artificial Intelligence (AI), Machine Learning (ML), and smart network interface cards (NICs). In these scenarios, devices aim to simultaneously access specific portions of data structures with the CPU, utilizing device-local caches without the need for complete data structure transfers. [2].

The second challenge addressed by CXL is the issue of memory scalability, driven by the escalating demand for both memory capacity and bandwidth in tandem with the exponential growth of computational requirements. The conventional DDR memory, however, fails to keep pace with this growing demand, leading to constraints on memory bandwidth per CPU. This limitation is primarily attributed to the pin-inefficiency of the parallel DDR interface, which complicates scaling by adding DDR channels, thereby escalating platform costs and introducing signal integrity challenges. PCIe pins show promise because they offer higher memory bandwidth per pin. For example, a x16 Gen5 PCIe port operating at 32 GT/s offers 256 GB/s with only 64 signal pins, surpassing the capabilities of DDR5-6400, which provides 50 GB/s with approximately 200 signal pins. Moreover, PCIe supports longer reach with retimers, enabling the relocation of memory farther away from CPUs and the utilization of more than 15W of power per DIMM, ultimately enhancing performance. Despite these advantages, PCIe faces a significant drawback as it does not support coherency, and memory attached to devices cannot be mapped to the coherent memory space. Consequently, PCIe has not been able to replace DDR in addressing the growing demand for memory scalability. [2].

The third challenge addressed by CXL is the inefficiency due to resource stranding, particularly evident in modern data centers. Stranded resources occur when one resource remains idle while another resource, such as computing power, is fully utilized. This happens because compute, memory, and I/O devices are tightly linked within a single server. As a result, servers must be overprovisioned with memory and accelerators to handle peak workloads, leading to wasted resources. For example, when a server runs an application needing more memory or accelerators than available, it cannot borrow these resources from an underutilized server in the same rack. Instead, it must contend with the performance consequences of page misses. Conversely, servers with all cores engaged by workloads often have unused memory. This stranding of resources has significant repercussions, including adverse effects on power consumption, cost, and sustainability. Low resource usage due to stranding have been observed at big technology companies such as Alibaba, AWS, Google, Meta, and Microsoft. Solving this issue is crucial for maximizing resource use and reducing power, costs, and environmental impact in modern data centers. [2].

The fourth challenge addressed by CXL involves the issue of fine-grained data sharing in distributed systems. Distributed systems often rely on fine-grained synchronization, where updates are small and latency-sensitive, causing work blocks during updates. Examples of such scenarios include partition/aggregate design patterns in web-scale applications like web search, social network content composition, and advertisement selection. In these systems, query updates are typically under 2kB, such as a search result. Distributed databases, which depend on kB-scale pages, and distributed consensus with even smaller updates, also fall within this category. Sharing data at such a fine granularity amplifies the impact of communication delays in typical data center networks, dominating the wait time for updates and consequently impeding crucial use cases. For instance, transmitting 4kB at 50GB/s (400Gbit/s) takes under 2 microseconds, but communication delays on current networks often exceed 10 microseconds. To address this challenge, a coherent shared-memory implementation

is proposed as a solution, aiming to reduce communication delays to sub-microsecond levels, as further elaborated in later sections. [2].

2.2 Generations

Since its launch in 2019, CXL has gained momentum in the industry, combining various companies and standards. Initially, there were competing interconnect standards like OpenCAPI, GenZ, and CCIX. However, CXL's membership has grown to around 250 companies. Intel released CXL 1.0 in March 2019, and the consortium released CXL 1.1 in September 2019, introducing compliance testing mechanisms. Subsequently, CXL 2.0 and CXL 3.0 were published in November 2020 and August 2022, respectively, expanding usage models while maintaining backward compatibility. Over time, the industry has widely adopted CXL, making it a central standard. [2]. Key differences between CXL-generations are seen in Table 1. Next, each generation is presented in more detail. More detailed information about the generations and features can be found in the latest specification [3].

Table 1: Features of different CXL generations, highlighting the increasing capabilities with each new iteration. The growth in features is especially notable in CXL 2.0 and CXL 3.0, showcasing advancements in memory pooling, switching, and scalability [4].

Feature	**CXL 1.0/1.1**	**CXL 2.0**	**CXL 3.0**
Release Date	2019	2020	2022 H1
Max Link Rate	32 GT/s	32 GT/s	64 GT/s
Flit 68 Byte (up to 32 GT/s)	✓	✓	✓
Flit 256 Byte (up to 64 GT/s)			✓
Type 1, Type 2, and Type 3 Devices	✓	✓	✓
Memory Pooling w/ MLDs		✓	✓
Global Persistent Flush		✓	✓
CXL IDE		✓	✓
Switching (Single-level)		✓	✓
Switching (Multi-level)			✓
Direct Memory Access for Peer-to-Peer			✓
Enhanced Coherency (256 Byte Flit)			✓
Memory Sharing (256 Byte Flit)			✓
Multiple Type 1 & Type 2 Devices per Root Port			✓
Fabric Capabilities (256 Byte Flit)			✓

2.2.1 CXL 1.0/1.1

CXL 1.0, launched in March 2019, offers dynamic multiplexing for various protocols, including I/O (CXL.io), caching (CXL.cache), and memory semantics (CXL.memory). It establishes a unified memory space between the host CPU and connected CXL devices, enabling resource sharing and improving performance, while also simplifying the software stack and reducing data movement. [2].

CXL is similar to PCIe in its asymmetric protocol design. In a CXL setup, the host processor includes a 'Root Complex' (RC) for each CXL link, connecting to an 'End Point' device. The host processor manages cache coherency, and system configuration is performed by software through instructions executed in the host processor, generating the necessary configuration transactions to access each device.

CXL supports native link widths of x16, x8, and x4, with x2 and x1 widths available in degraded mode. Degraded mode is activated when there are higher-than-expected error rates on a PCIe link, causing it to automatically switch to narrower widths and lower frequencies. Native data rates of 32.0 GT/s (CXL 1.0/2.0) and 64.0 GT/s (CXL 3.0) are supported, while degraded mode allows for 16.0 GT/s and 8.0 GT/s data rates.

2.2.2 CXL 2.0

CXL's second generation introduces resource pooling, allowing the allocation of the same resources to different hosts at different times. This dynamic resource reassignment addresses resource stranding by breaking the tight coupling of resources to individual hosts. If one host runs a compute-intensive workload but doesn't use its assigned device memory, operators can reassign it to another host running a memory-intensive workload. This flexibility allows operators to provision memory based on the average case rather than the worst-case memory capacity for various workloads, saving significant memory. This resource pooling concept can also be applied to other resources like accelerators.

CXL 2.0 introduces several key features, including Hot-Plug, Single Level Switching, Quality-of-Service (QoS) for Memory, Memory Pooling, Device Pooling, and Global Persistent Flush (GPF). Hot-Plug, previously not possible in CXL 1.1, enables adding CXL resources after the platform boots, allowing for traditional physical hot-plug and dynamic resource pooling.

To support a single level of switching, CXL standardizes address decoding for CXL.memory regions in hierarchical decoders (HDM). This approach follows the PCIe memory decode model, allowing for decode at each switch, eliminating the need for the host or switch to fully decode all agents in the hierarchy. Multi-host connections and device pooling are facilitated by representing the CXL topology as a virtual hierarchy (VH) for each host, which includes switches and virtual bridges. CXL 2.0 is limited to directed tree topologies with at most one path between each host and device, and scalability is limited to a single switch level.

Device Pooling builds upon multi-host switch support, allowing devices to be dynamically assigned to one host at a time. CXL defines Single-Logical-Device (SLD) for standard devices and Multi-Logical-Device (MLD) for devices whose resources

can be divided into logical devices (up to 16) that can be assigned to different hosts simultaneously.

In scenarios where resources may become oversubscribed, CXL 2.0 addresses Quality-of-Service issues by introducing a DevLoad field in CXL.memory Response messages. This field informs the host about the observed load in the device it is accessing, enabling the host to adjust its request rate based on this load information. The reference model defines load-based adjustments to the injection rate to ensure efficient resource utilization.

2.2.3 CXL 3.0

CXL 3.0 aims to address data sharing challenges in large distributed systems by expanding the resource pooling introduced in CXL 2.0. This expansion reaches a much larger scale with multi-level switching and supports protocols for up to 4096 end devices. These end devices can include host CPUs, memory, accelerators, I/O devices, and more. The goal is to dynamically compose systems based on specific workloads, ultimately delivering power-efficient performance and reducing total cost of ownership (TCO) in large distributed systems.

CXL 3.0, built on PCIe 6.0 technology, doubles the transfer rate to 64GT/s without adding latency compared to previous generations. This results in a maximum aggregate raw bandwidth of 256GB/s for x16 width links. CXL 3.0 leverages PCIe 6.0's lightweight FEC and strong CRC for error-free transmission using 256B flits on PAM-4 signaling to achieve 64GT/s. It introduces a latency-optimized flit variant that further reduces latency by breaking up the CRC in 128B sub-flit granular transfers to mitigate store-and-forward overheads in the physical layer. The new 256B flit format remains backward compatible with previous CXL generations.

CXL Type-2 devices, known as accelerators with memory, gain the ability to back invalidate the host's caches, a feature called enhanced coherency. This enhances memory management and coherency for host-managed device-attached memory (HDM), allowing for more efficient memory mapping and management. It also enables direct peer access to HDM memory without host involvement and the ability for Type 2/Type 3 devices to back invalidate a cache line through the host processor.

Building upon the concept of memory pooling introduced in CXL 2.0, CXL 3.0 introduces memory sharing. Memory sharing allows CXL-attached memory to be coherently shared across hosts using hardware coherency, enabling simultaneous access by multiple hosts without software coordination. This facilitates the creation of clusters of machines for solving large problems through shared memory constructs.

CXL 3.0 introduces fabric capabilities, departing from traditional tree-based architectural structures. It supports up to 4096 nodes with a scalable addressing mechanism called Port Based Routing (PBR). These nodes can include CPU hosts, CXL accelerators with or without memory, PCIe devices, and Global Fabric Attached Memory (GFAM) devices, which can be accessed by multiple nodes using port-based routing. CXL fabric opens up possibilities for building powerful systems comprising compute and memory elements tailored for specific workloads.

2.2.4 CXL 3.1

The CXL 3.1 specification introduces various features, including enhancements to CXL Fabric capabilities and extended requirements for Fabric Decode and Routing. It defines the Fabric Manager API for Port Based Routing (PBR) Switch and enables host-to-host communication with the Global Integrated Memory (GIM) concept.

Additionally, the specification facilitates direct Peer-to-Peer (P2P) CXL.mem support through PBR Switches and introduces the Trusted Execution Environment Security Protocol (TSP). Memory Expander functionality sees improvements, and Extended Meta Data supports up to 32-bits per cache line of host-specific state.

The CXL 3.1 specification also provides improved visibility into errors related to CXL memory devices and offers expanded visibility and control over the Reliability, Availability, and Serviceability (RAS) aspects of CXL memory devices. Notably, it maintains full backward compatibility with CXL 2.0, CXL 1.1, and CXL 1.0. [3].

2.3 Protocol types

This subsection explores the core protocols of the CXL framework: CXL.io, CXL.cache, and CXL.memory. The CXL.io protocol handles key tasks such as device discovery, initialization, and direct memory access (DMA), leveraging a PCIe-based, asynchronous, packetized protocol that ensures efficient communication through flow control mechanisms and virtual channels for quality of service. CXL.cache enables devices to coherently cache host memory using the MESI protocol, with the host managing coherence and address translation, ensuring streamlined cache operations and consistency across systems. Finally, CXL.memory provides devices with the capability to expose memory as Host-Managed Device Memory (HDM), supporting both memory expansion and accelerator scenarios. It incorporates flexible coherence models and ensures efficient, ordered memory access for advanced computing workloads. Together, these protocols form a comprehensive framework for high-performance memory and I/O management. [3].

2.3.1 CXL.io

The CXL.io protocol serves various functions like device discovery, configuration, initialization, I/O virtualization, and direct memory access (DMA) with non-coherent load-store semantics [3]. It it based on Peripheral Component Interconnect Express (PCIe). Unlike early PCI-based architectures, CXL.io uses a split-transaction, credit-based, packetized protocol. This means that completion of CXL.io requests arrives independently and asynchronously, allowing for concurrent transactions. CXL.io employs three flow-control classes (FCs): posted (P) for memory writes and messages, non-posted (NP) for transactions requiring completion like memory reads and configuration/I/O read/writes, and completions (C). CXL.io adheres to PCIe ordering rules across these FCs to ensure forward progress and maintain a producer-consumer ordering model.

Additionally, CXL.io enforces two virtual channels (VCs) to guarantee quality of service (QoS). Traffic with varying requirements and latency characteristics is segregated into different VCs to minimize top-of-queue blocking in the system. For instance, bandwidth-intensive bulk traffic and latency-sensitive isochronous traffic can be placed in separate VCs with distinct arbitration policies. Similarly, DRAM accesses and persistent memory accesses are assigned to different VCs due to their disparate latency and bandwidth characteristics, preventing interference between the two.

CXL.io relies on standard PCIe DLLPs for tasks like sharing credits, ensuring reliable TLP delivery, and power management. It also leverages the PCIe configuration space and enhances it for CXL-specific purposes. This approach enables compatibility with the existing device discovery mechanism. It is anticipated that PCIe device drivers will be updated to utilize new features like CXL.cache and CXL.mem, while system software will be responsible for programming the associated registers related to these new capabilities. [3].

2.3.2 CXL.cache

The CXL.cache protocol enables a device to cache host memory using the MESI coherence protocol with a 64-byte cache line size [2]. The protocol simplifies device management by having the host handle coherence tracking for peer caches, and the device never directly interacts with peer caches. It employs three channels in each direction: Host-to-Device (H2D) and Device-to-Host (D2H), including Request, Response, and Data channels. These channels operate independently, except that a Snoop message from the host in the H2D Request channel must push a prior Global Observation (GO) message in the H2D Response for the same cache line address, indicating coherence state (MESI) and the coherency commitment point.

CXL.cache uses host physical addresses, while caching devices, like CPUs, operate on virtual addresses. Devices implement a Device Translation Look-aside Buffer (DTLB) to cache page table entries, and they use the Address Translation Services (ATS) of PCIe (and CXL.io) to fetch virtual-to-physical translations and access control. CXL extends ATS to convey whether access to an address is allowed to use CXL.cache or limited to CXL.io. The DTLB is non-coherent, so the host processor is responsible for tracking entries pending in DTLBs and initiating invalidation when necessary.

The D2H Request channel includes commands for Read, Read0, Read0-Write, and Write, allowing the device to request coherence state and data, coherence state only, direct data write, and cache eviction. The H2D Response channel provides coherence state information, and the H2D Data channel delivers data.

The H2D Request channel allows the host to change coherence state in the device, referred to as "Snooping" (Snp), with the device updating its cache and potentially returning data. The protocol ensures proper ordering to handle conflicting accesses, specifically Req-to-Snoop and Eviction-to-Snoop cases, where the order of GO messages and Snoop messages is crucial for correct processing and maintaining cache coherence. [2].

2.3.3 CXL.memory

The Memory Protocol in CXL allows a device to expose Host-managed Device Memory (HDM), which the host can manage and access as if it were native DDR memory. This protocol is media-independent, employing straightforward reads and writes with Host Physical Addresses that the device translates into its media address space. For devices that wish to cache this memory, advanced semantics enable direct caching and tracking of host caching by the device. The protocol utilizes two channels in each direction: Master-to-Subordinate (M2S) and Subordinate-to-Master (S2M), offering Request, Request-with-Data (RwD), Non-Data-Response (NDR), and Data-Response (DRS) channels.

The two primary use cases for CXL.mem are "host memory expander" and "accelerator memory exposed to the host," with corresponding protocol requirements designated as HDM-H and HDM-D. HDM-H doesn't assume any coherence protocol, while HDM-D includes cache state and cache snooping attributes. The coherence model allows devices to change host cache state using the CXL.cache request, referred to as the "Bias Flip" flow, which influences how the host handles the data. This flow ensures coherent memory access within the system and utilizes ordered channels for HDM-D addresses to prevent race conditions. [3].

2.4 Device types

There are three different profiles for CXL devices described in the CXL 1.0/1.1 specification. The following sections describe the devices that can be attached via CXL.

2.4.1 Type 1

CXL Type 1 devices have unique requirements, specially benefiting from a fully coherent cache in the device. Thus, standard producer-consumer ordering models are in many cases inadequate for these devices. For instance, a need to perform complex atomics beyond PCIe's standard operations might be challenging. Basic cache coherency allows these devices to implement custom ordering models and an unlimited range of atomic operations, requiring only a small cache that can be tracked by the host's snoop filter capacity. CXL accommodates such devices with its optional CXL.cache link, enabling cache coherency transactions through the CXL.cache protocol. [3]. A basic example of a Type 1 device is seen in Figure 1.

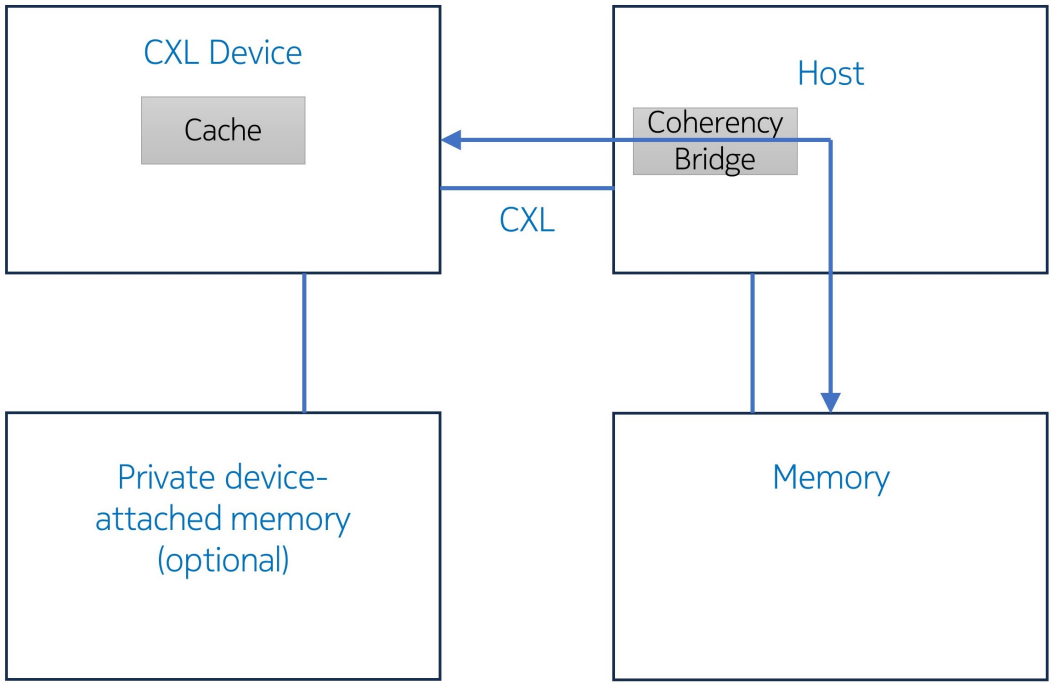


Figure 1: This figure provides a basic example of a CXL Type 1 device. Unlike standard producer-consumer ordering models, these devices require advanced cache coherency to support complex atomic operations. CXL addresses this need through its optional CXL.cache link, allowing cache coherency transactions via the CXL.cache protocol. This enables more efficient data ordering and atomic operations while minimizing cache size requirements.

2.4.2 Type 2

CXL Type 2 devices feature both a fully coherent cache and attached memory, such as DDR or High-Bandwidth Memory (HBM) [3]. These devices use a high-speed data link between the accelerator and device-attached memory to achieve their performance. CXL's primary goal is to allow the Host to efficiently transfer data to and from device-attached memory without introducing significant software and hardware burden that would negate the accelerator's benefits.

It's crucial to differentiate HDM from traditional I/O and PCIe Private Device Memory (PDM). For example, a GPGPU with attached GDDR treats device-attached memory as private, making it inaccessible to the Host and lacking coherence with the rest of the system. This private memory is managed only by the device's hardware and driver and is mainly used for intermediate storage of large datasets. The drawback of this approach is the need for high-bandwidth data transfers between Host memory and device-attached memory for operand inputs and result outputs. A basic example of a type 2 device is seen in Figure 2.

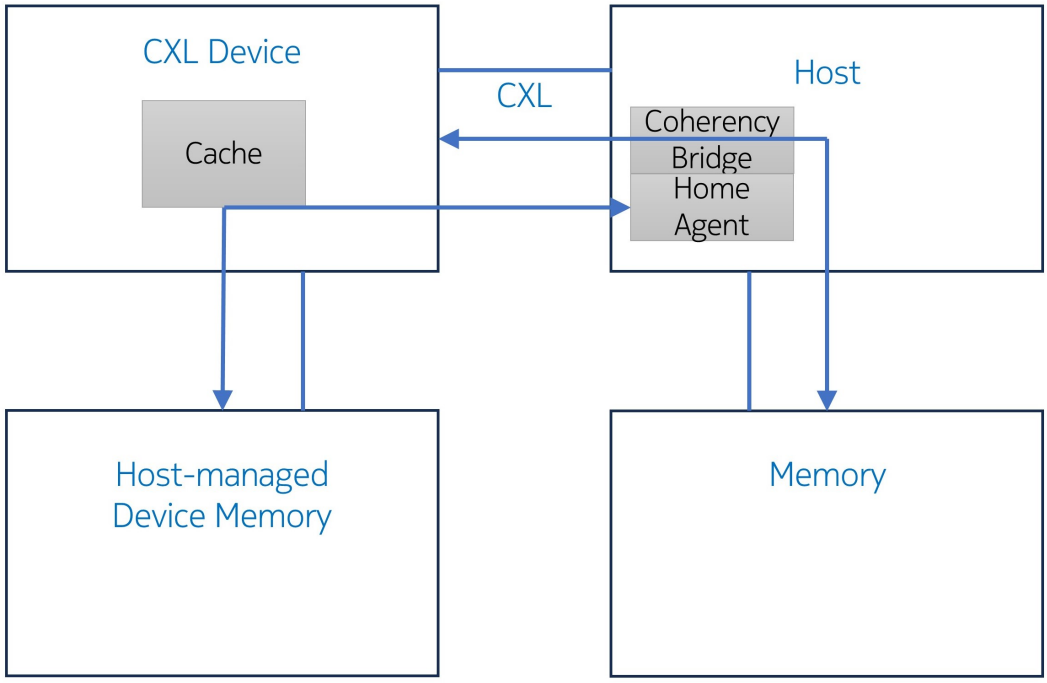


Figure 2: This figure provides a basic example of a CXL Type 2 device, which includes both a fully coherent cache and attached memory, such as DDR or High-Bandwidth Memory. These devices use a high-speed data link between the accelerator and device-attached memory to enhance performance. The key advantage of CXL is that it allows the host to efficiently transfer data to and from device-attached memory without adding unnecessary software or hardware complexity.

There are two ways for ensuring device coherence of Host-managed Device Memory (HDM). The first method utilizes CXL.Cache to maintain coherence, known as "Device coherent," and it's identified by the suffix "D" (HDM-D). The second method relies on the dedicated channel in CXL.mem called Back-Invalidation Snoop, and it's denoted by the suffix "DB" (HDM-DB). [3].

2.4.3 Type 3

CXL Type 3 devices support both CXL.io and CXL.mem protocols. Unlike traditional accelerators that operate on host memory, these devices don't utilize CXL.cache for requests. Instead, they focus on CXL.mem to fulfill Host-initiated requests. The CXL.io protocol serves functions like device discovery, enumeration, error reporting, and management, and it can also be used for other I/O-specific applications by the device.A basic example of a Type 3 device, a memory expander, is seen in Figure 3. [3].

This architecture is memory technology-agnostic and allows for various memory organization possibilities based on the Host's support. When Type 3 device memory is exposed as HDM-DB, it enables the device to manage coherence with the host directly, facilitating in-memory computing and direct access via UIO on CXL.io. [3].

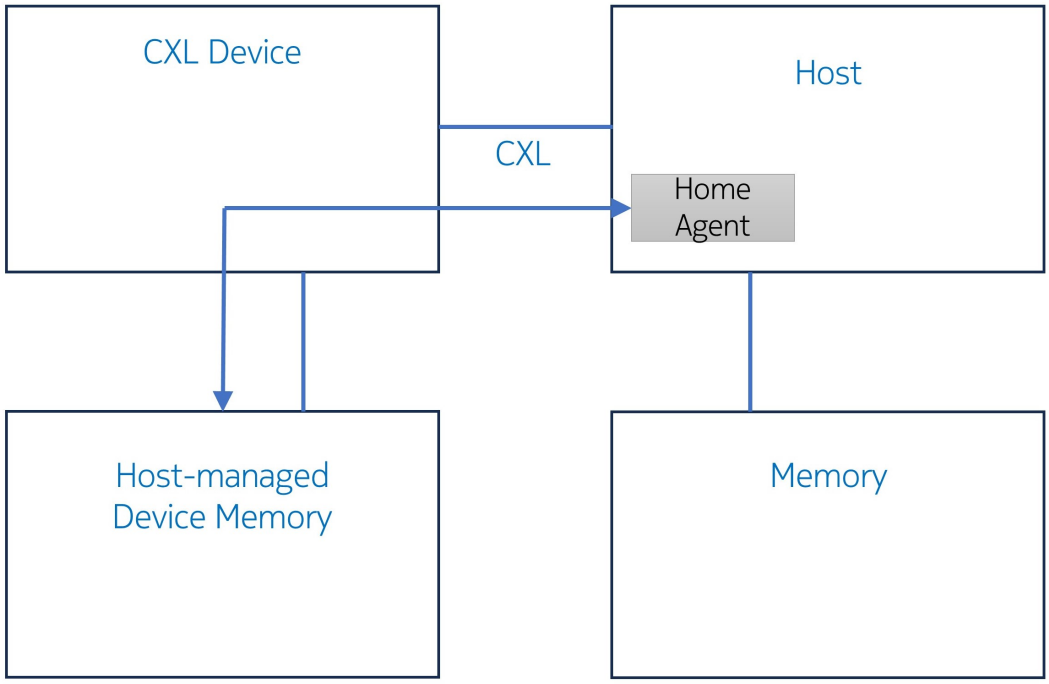


Figure 3: This figure provides a basic example of a CXL Type 3 device, which supports both the CXL.io and CXL.mem protocols. Unlike traditional accelerators that rely on host memory and use CXL.cache, Type 3 devices do not handle cache requests. Instead, they primarily use CXL.mem to respond to host-initiated memory requests.

3 Literature Review

Memory constitutes a significant cost in data centers and cloud servers, accounting for 40% (Meta) and 50% (Azure) of a server's overall cost. This financial pressure has spurred research into methods such as far memory, memory compression, and Intel Optane memory to reduce RAM needs and costs. The consequences of insufficient or excess memory include stranded cores or stranded memory. CXL offers lower latency and hardware support for cache coherence, enabling efficient access to remote memory. This has led to the exploration of CXL memory pools in data centers and cloud networks, potentially reducing memory requirements and overall costs according to recent research. [5].

Currently, most available CXL hardware is based on version 1.1, with some vendors incorporating selected features from version 2.0. Versions 3.0 and 3.1 depend on improvements in PCIe, set to be achieved in the ratified Gen 6 of that protocol. In addition, all major x86 server-class CPU vendors have adopted CXL. [6]. Intel supports CXL from Sapphire Rapids (SPR) CPUs and Agilex7 FPGAs, covering all three protocols. AMD plans CXL support in Genoa CPUs and SmartNIC devices. ARM has announced CXL 2.0 support in V2, N2, and E2 series CPUs. [2].

This section begins by examining two performance studies that evaluate the impact of CXL on modern computing systems. The first study by Microsoft investigates the Pond system, which employs CXL-based memory pooling to optimize resource utilization and reduce costs in public cloud environments, focusing on its effect on NUMA systems and latency-sensitive workloads [7]. The second study by Intel examines the performance differences between true and emulated CXL memory, exploring factors such as latency, bandwidth efficiency, and cache interactions [8]. Following these performance assessments, the section transitions to cost models, presenting theoretical frameworks and real-world examples to evaluate the Total Cost of Ownership (TCO) for CXL-enabled systems. These models incorporate both capital and operational expenditures, providing insights into the economic implications of using CXL in data centers and cloud environments.

3.1 Performance Studies - Microsoft Pond

Li *et al.* [7] introduces Pond, a pioneering system that achieves both competitive cost and equivalent memory performance to that of a single Non-Uniform Memory Access (NUMA) node in public cloud environments.

In a NUMA system, memory is organized into NUMA nodes. Each node encompasses memory with consistent access characteristics for a specific processor. Nodes exhibit affinity to both processors and devices, with locally attached devices having optimal performance when accessing memory within a NUMA node. Memory allocated from the most suitable NUMA node for a processor is termed "local node." For instance, in a system with one node per socket, each having four cores, the NUMA architecture ensures specific memory access characteristics for processors within each node. [9].

Pond employs a combination of hardware and system techniques, relying on the

Compute Express Link (CXL) standard for efficient cacheable load/store accesses to pooled memory across different processor architectures. Despite higher latencies in CXL accesses compared to same-NUMA-node accesses, Pond mitigates this impact through innovative systems support for CXL-based pooling.

The feasibility of Pond is grounded in four key insights. Firstly, analysis of Azure production clusters indicates that pool sizes between 8-16 sockets result in substantial Dynamic Random Access Memory (DRAM) savings. Second, by emulating various memory access overheads, Pond identifies workloads suitable for allocation in pool memory, emphasizing the challenge of achieving same-NUMA-node performance for specific workloads. Machine learning models are trained to identify insensitive workloads for pre-allocation on the Pond memory pool.

Third, observations from Azure measurements reveal that approximately 50% of virtual machines (VMs) utilize less than 50% of their allocated memory. Allocating untouched memory from the pool has minimal performance impact, particularly for latency-sensitive VMs, when exposed to the VM's guest OS as a zero-core virtual NUMA (zNUMA) node. This concept holds true, contrary to prior assumptions, and effectively biases memory allocations away from the zNUMA node.

Fourth, Pond optimally allocates CXL memory with same-NUMA-node performance by predicting VM latency sensitivity and the amount of untouched memory. In cases of incorrect predictions, Pond introduces a novel monitoring system that detects poor memory performance and triggers a mitigation, migrating the VM to use only same-NUMA-node memory. Importantly, all inputs for training and running Pond's machine learning models are obtained from existing hardware telemetry with minimal overhead.

First, the study focuses on assessing memory stranding and untouched memory at Azure through the analysis of production data. The dataset comprises measurements from 100 cloud clusters over a 75-day period, encompassing mainstream first-party and third-party VM workloads. These clusters are chosen to be representative of the majority of the server fleet, with similar deployment years and spanning various major regions globally. Each cluster's trace includes millions of per-VM arrival/departure events, detailing time, duration, resource demands, and server-id.

In summary, the analysis reveals varying degrees of memory stranding, ranging from 3-27% at the 95th percentile, with occasional outliers reaching 36%. Almost all virtual machines (VMs) can fit within a single NUMA node. The implementation of memory pooling across 16-32 sockets demonstrates the potential for a 10% reduction in cluster memory demand, indicating substantial cost reductions. However, the success of memory pooling hinges on allocating a significant portion of Dynamic Random Access Memory (DRAM) to memory pools. Providers embarking on DRAM pool implementation with cross-NUMA latencies must exercise careful management to mitigate potential performance impacts.

Secondly, the study characterizes the performance impact of CXL latency for typical workloads in Azure's datacenters. The evaluation involves 158 workloads under two scenarios of emulated CXL access latencies, representing a 1.82-times and 2.22-times increase in memory latency. The comparison is made with workload performance under NUMA-local memory placement.

Under a 182% increase in memory latency, the study observes that 26% of the 158 workloads experience less than a 1% slowdown with CXL, while an additional 17% see less than a 5% slowdown. Conversely, 21% of workloads face more than a 25% slowdown. Variations are noted among different workload classes, with graph processing workloads generally exhibiting higher slowdowns, albeit with significant variability within each class. Azure's proprietary workloads are less impacted, with 6 out of 13 production workloads showing minimal effects (<1%), attributed to their NUMA-awareness and data placement optimizations.

Under a 222% increase in memory latency, the effects intensify. Approximately 23% of the 158 workloads experience less than a 1% slowdown, while 14% see less than a 5% slowdown. More than 37% of workloads face greater than a 25% slowdown. The study observes a magnification of effects seen under lower latency, indicating that workloads performing well under a 182% increase also tend to perform well under a 222% increase, while those severely affected by the former are even more impacted by the latter.

Finally, the study evaluates the performance of pond with a prototype setup. This prototype setup uses production servers at Azure and similarly-configured lab servers. The production servers include two configurations: Intel Skylake 8157M sockets with 384GB of DDR4 each or two AMD EPYC 7452 sockets with 512GB of DDR4 each. The Intel configuration exhibits 78ns NUMA-local latency, 80GB/s bandwidth locally, and 142ns remote latency with 30GB/s bandwidth (equivalent to 3/4 of a CXL ×8 link). On the AMD configuration, the measurements are 115ns NUMA-local latency and 255ns remote latency. The BIOS settings disable hyper-threading, turbo-boost, and C-states. Performance baselines are established using VMs entirely backed by NUMA-local DRAM.

In summary, with a pool size of 16 sockets, Pond achieves a 9% reduction in overall DRAM requirements under a 182% latency increase and a 7% reduction under a 222% latency increase. In comparison, the Static approach reduces DRAM by 3%. The 7-9% reduction leads into an overall reduction of 3.5% in cloud server cost.

3.2 Performance Studies - Intel

The Intel study [8] involves running a set of benchmarks to not only compare the performance of true CXL memory with emulated CXL memory but also to analyze the intricate interplay between the CPU and CXL memory in detail.

The evaluation employs a server with two Intel Sapphire Rapids (SPR) CPU sockets, each socket equipped differently. One socket contains eight 4800 MT/s DDR5 DRAM DIMMs (128 GB) across eight memory channels, while the other socket has only one 4800 MT/s DDR5 DRAM DIMM to emulate the bandwidth and capacity of CXL memory. The Intel SPR CPU integrates four CPU chiplets, each with up to 15 cores and two DDR5 DRAM channels. Users can choose to use the chiplets either as a unified CPU or as separate NUMA nodes in the SNC mode, offering flexibility for resource isolation. Hyper-threading is turned off, and the CPU core clock frequency is set to 2.1 GHz for more predictable performance.

Three CXL memory devices are utilized, featuring different CXL IPs (ASIC-based

hard IP and FPGA-based soft IP) and DRAM technologies (DDR5-4800, DDR4-2400, and DDR4-3200). The CXL protocol's flexibility allows seamless accommodation of various memory technologies, including DRAM, persistent memory, flash, and emerging memory technologies, potentially resulting in different latency and bandwidth characteristics among CXL memory devices.

The study observes a reduction in memory access latency with the implementation of full-duplex CXL and Ultra Path Interconnect(UPI) interfaces. The use of these interfaces leads to a significant decrease in memory access latency, particularly demonstrated by the "memo" microbenchmark. This benchmark reveals a 76% lower load latency for emulated CXL memory when compared to Intel MLC (Memory Latency Checker), highlighting the advantages derived from the full-duplex capabilities of these interfaces.

The latency associated with accessing true CXL memory devices is found to be highly dependent on the specific design of the CXL controller. The study introduces different CXL controllers, including CXL-A, CXL-B, and CXL-C, each exhibiting varying levels of latency. Notably, CXL-A shows only a 35% longer load latency than DDR5-R, while CXL-C demonstrates almost a threefold increase in load latency compared to DDR4-2400. These variations underscore the impact of CXL controller design on memory access performance.

A noteworthy finding is the comparison between emulated CXL memory and true CXL memory in terms of latency. Emulated CXL memory is observed to have longer memory access latency than its true CXL counterpart. This difference arises from the cache coherence checks involved with emulated CXL memory, which requires coordination with the remote CPU. In contrast, true CXL memory, despite being exposed as a remote NUMA node, benefits from on-chip structures, resulting in lower latency.

The study explores the impact of cache coherence overheads on store latency, particularly due to cache write-allocate policies. The latency of st to emulated CXL memory is found to increase more significantly than st to true CXL memory. This discrepancy is attributed to the higher cache coherence overheads associated with emulated CXL memory, especially during cache coherence checks with the remote CPU.

Bandwidth efficiency is identified as strongly dependent on the efficiency of CXL controllers. Different CXL controllers, such as CXL-A, CXL-B, and CXL-C, exhibit varying levels of bandwidth efficiency, influenced by factors like memory read/write ratios. The study emphasizes the importance of considering these controller-specific characteristics when assessing the overall efficiency of CXL memory systems.

The investigation underscores the competitive bandwidth efficiency offered by true CXL memory, particularly concerning store operations, when compared to emulated CXL memory. Emulated CXL memory experiences more pronounced bandwidth efficiency degradation, primarily due to the additional overhead associated with cache coherence checks.

An essential aspect explored in the study is the interaction between CXL memory and the CPU's cache hierarchy. This interaction differs significantly from local DDR memory, impacting Last-Level Cache (LLC) hit/miss and interference characteristics.

Understanding these interactions is deemed crucial for analyzing the performance of applications utilizing CXL memory, especially in the context of Sub-NUMA Clustering (SNC) modes.

In summary, load instructions on CXL-attached memory can be 35% slower than NUMA, while stores show slightly lower overheads. The efficiency of memory transfer is measured in terms of nominal bandwidth capacity, with CXL interconnects demonstrating 46% efficiency for loads, compared to 70% for NUMA. Surprisingly, stores to a CXL device can be 12% more bandwidth efficient than to a neighbor NUMA socket, attributed to the bypassing of coherency checks on the CXL device.

3.3 Cost models

Understanding and accurately estimating the costs associated with a data center is crucial for effective planning and decision-making. Cost models provide a structured approach to evaluate both capital expenditures (CapEx) and operating expenditures (OpEx), offering insights into initial investments and ongoing expenses. These models not only help in forecasting long-term financial commitments but also enable comparisons between different infrastructure configurations, scalability options, and efficiency measures. By exploring cost models, we can analyze the trade-offs between upfront investments and operational savings, as well as identify areas where optimizations can lead to significant cost reductions over time. [10], [11].

The Total Cost of Ownership (TCO) is the comprehensive sum of all costs over an n-year period, categorized into CapEx and OpEx. Capital expenditures encompass depreciated costs for acquiring servers, software licensing, networking, storage, power and cooling equipment, and facilities (Example in Table 2). Operational expenditures consist of real estate, power, cooling, support, and maintenance costs, as well as administrator and personnel expenses (Example in Table 3). OpEx is calculated based on the daily operational costs to maintain and operate the devices. [12].

Table 2: Example CapEx for a Data Center. This describes key cost categories and important considerations. The categories include infrastructure, IT equipment, power and cooling systems, security, software, and network infrastructure. Each row highlights factors that influence costs, such as performance requirements, energy efficiency, scalability, and regulatory compliance.

CapEx Category	Considerations	CapEx (€)
Infrastructure	Location, size, redundancies, construction costs	x
IT Equipment	Performance requirements, scalability, technology lifecycle	x
Power Systems	Power capacity, redundancy, energy efficiency	x
Cooling Systems	Cooling capacity, energy efficiency (PUE)	x
Security Systems	Regulatory compliance, risk mitigation	x
Software	Compatibility, scalability, vendor lock-in	x
Network Infrastructure	Bandwidth requirements, future expansion	x

Table 3: Example OpEx for a Data Center. This describes key cost categories and important factors. The categories include power consumption, maintenance, real estate rent, and overall operational expenditures. Each row highlights factors affecting costs, such as energy efficiency, hardware upkeep, rental space, and total ongoing expenses.

OpEx Category	Considerations	Annual Cost
Operating Power	Power consumption per server, energy efficiency, and electricity rates	x €
Maintenance Costs	Regular hardware and software maintenance, replacement cycles	x €
Real Estate Rent	Data center area in square meters, rent per square meter	x €
Operational expenditures	Sum of all operating expenses	x €

The surge in AI-driven demand has significantly impacted the capital expenditures of leading cloud providers, driving substantial investments in infrastructure and technology. Amazon Web Services (AWS) reported triple-digit growth in AI-related revenue, far outpacing its overall growth, and plans to allocate $75 billion in 2024 primarily for AI services. CEO Andy Jassy noted that generative AI demand is expanding three times faster than AWS did at a comparable stage. Microsoft increased its CaPex by 34% year-over-year for Q3 2024, with an estimated 13.3% of spending directed toward AI infrastructure, the highest proportion among its peers. Similarly, Google (Alphabet) saw a 35% year-over-year rise in CaPex, with approximately 6.8%

invested in AI-related infrastructure. These figures highlight AI's pivotal role in shaping cloud providers' financial strategies. [13].

3.3.1 Power Management in CXL

CXL implementations are mandated to incorporate Physical Layer Power Management, encompassing both protocol-specific Link Power Management and CXL Physical Layer power management. This responsibility falls on the ARB/MUX Layer, which oversees the coordination of power management states among various protocols on both sides of the links. [3].

To enhance power management for CXL-connected devices across the entire system, a hierarchical architecture is established. This framework treats discrete devices as autonomous entities, allowing for local execution of thermal and power management. Coordination with the processor is achieved through Vendor-defined Messages (VDMs) over CXL. The communication for coordination involves PM2IP and IP2PM messages. Additionally, the system can support simplified protocols. [3].

In terms of CXL Physical Layer, it supports L1 and L2 states as defined in the PCIe Base Specification. While entry and exit conditions adhere to PCIe specifications, the CXL ARB/MUX takes charge of directing the entry and exit from Physical Layer Power Management states. [3].

CXL Link Power Management facilitates Active Link State Power Management (ASPM), supporting L1 and L2 power states. In 256B Flit mode, there is support for L0p negotiation as well. The PM Entry/Exit process is segmented into three phases. In 68B Flit mode, if the LTSSM undergoes Recovery before the ARB/MUX vLSM transitions to the PM state, the PM Entry process must recommence from Phase 1. However, in 256B Flit mode, PM entry handshakes remain unaffected by Link Recovery transitions. [3].

3.3.2 Power Consumption in Cloud RAN

The power consumption of cloud networks is a critical concern due to its significant environmental and economic impact. As digital services continue to expand, data centers now consume a staggering 30 billion watts of electricity globally, equivalent to the output of 30 nuclear power plants. [14] . This massive energy consumption not only contributes to rising operational costs for businesses but also leads to substantial greenhouse gas emissions. Cloud computing, while offering numerous benefits, can potentially exacerbate this issue if not managed efficiently. However, when optimized, cloud services can be up to 93% more energy-efficient than on-premises data centers. [15].

Nokia conducted a study to analyze power consumption in their cloud network product, using two setups utilizing different vendors. The research measured the total power consumption of each setup, specifically breaking down the contributions of CPU and memory. To maintain confidentiality, the actual consumption values were replaced with percentage figures in the study. These values (Table 4) were calculated using equations (1) & (2). This approach underscores the significant role that CPU and

memory play in the overall energy efficiency of cloud network operations, highlighting the critical need for optimizing these components to reduce power usage and enhance sustainability in cloud environments.

$$\text{CPU Power Percentage} = \frac{\text{Total CPU Power}}{\text{Total Power Consumption}} \times 100 \tag{1}$$

$$\text{Memory Power Percentage} = \frac{\text{Total Memory Power}}{\text{Total Power Consumption}} \times 100 \tag{2}$$

Table 4: Power consumption data for Vendor 1 and Vendor 2, presenting the percentage of total power consumption attributed to CPU and memory usage under two different test setups.

Vendor 1			**Vendor 2**		
T (°C)	**CPU (%)**	**MEM (%)**	**T (°C)**	**CPU (%)**	**MEM (%)**
20	41.5	8.3	20	54.5	9.8
25	41.4	8.3	25	53.3	10.0
30	40.7	8.2	30	51.5	10.3
35	40.5	8.2	35	50.7	10.1
40	39.1	8.0	40	46.6	9.5
45	36.1	7.4	45	38.6	8.1

The data (Figure 4) highlights the impact of increasing chamber temperature on the power consumption distribution between the CPU and memory in two vendors' systems. Vendor 1 demonstrates relatively stable CPU power consumption, decreasing slightly from 41.5% to 36.1%, and memory power consumption also shows a small reduction from 8.3% to 7.4%. In contrast, Vendor 2 exhibits a more pronounced decline in CPU power consumption, dropping from 54.5% at 20°C to 38.6% at 45°C, with memory power consumption decreasing from 9.8% to 8.1% over the same temperature range. These trends indicate that Vendor 1's components are less sensitive to increasing chamber temperatures compared to Vendor 2, suggesting better thermal efficiency. Vendor 2's steeper decline in CPU power indicates potential thermal management challenges that may require optimization to maintain consistent performance in higher temperature environments.

3.3.3 Cost Study - Google

Google scholars argued against the profitability of CXL memory pooling in their study [5]. The primary challenge with implementing CXL memory pools lies in the cost factor. While the main advantage of a CXL memory pool is to reduce the overall RAM requirements of data centers and cloud systems, the associated infrastructure costs are a significant consideration. Currently, servers are provisioned to accommodate the maximum footprint of all virtual machines (VMs) or containers simultaneously,

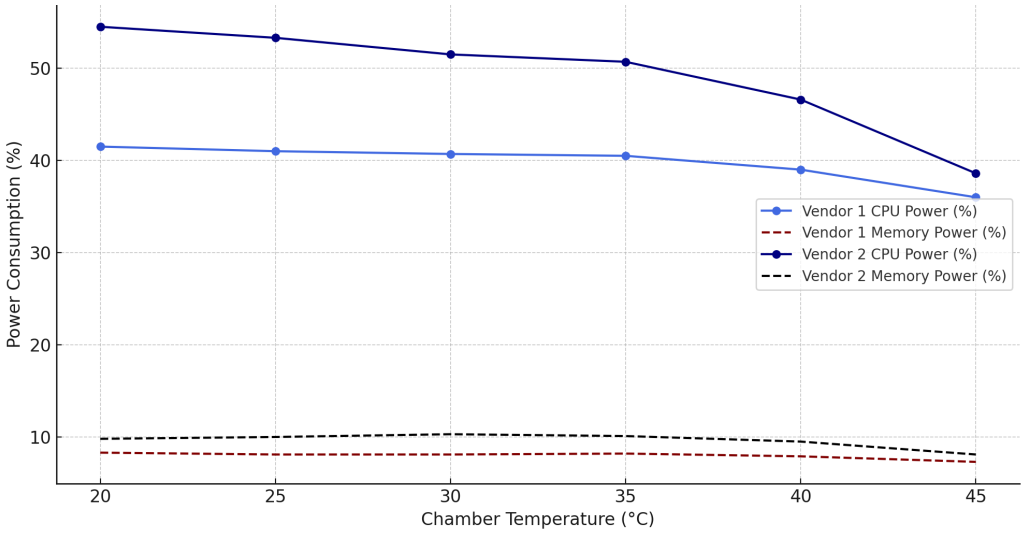


Figure 4: Plot of power consumption (%) for Vendor 1 and Vendor 2. This illustrates the effect of increasing chamber temperature on the distribution of power consumption between the CPU and memory in the test setups.

resulting in high memory usage. CXL memory pools enable servers to provision for expected use and store cold data in the pool when VMs reach their maximum footprint. However, the implementation of CXL requires a parallel network infrastructure, different from Ethernet, involving a CXL appliance at the top of the rack (or server) with dedicated cabling to all servers. This infrastructure adds to the overall cost of implementing CXL technology.

While the idea of a shared pool to meet peak memory needs and lower per-server memory costs seems beneficial, there are practical issues. The assumption that memory is interchangeable and can be reduced by a small fraction (e.g., 7-9%) may not be applicable in reality. In addition, the analysis overlooks the expenses associated with additional cabling and networking infrastructure required for CXL implementation. Cloud and datacenter servers face limitations in DRAM capacity due to discrete steps, and small reductions in memory may not result in cost savings. Modern server CPUs, such as those by Intel and AMD, have 8 or 12 DDR channels. To maximize memory bandwidth, servers populate every channel, but DIMMs are available in specific sizes (e.g., 32GB, 48GB, 64GB), and each channel must have the same-sized DIMM. This constraint leads to servers having fixed configurations with predetermined amounts of memory, restricting flexibility.

According to Pond's findings, allocating 25% of VM memory on average to a shared pool results in minimal performance slowdowns, with only 1% of VMs experiencing more than a 5% slowdown. This can be accomplished, for instance, by replacing 128GB DIMMs with 96GB DIMMs. However, it's crucial to note that while this allocation is feasible, it doesn't actually reduce the total amount of memory required. Servers still demand the same overall memory capacity; the only difference is that a portion of it is now housed in a CXL-connected memory appliance.

Attempting to implement CXL memory pools faces challenges, as reducing server RAM by 7% or 9% is not feasible. Instead, servers must cut their RAM by 25%, and the CXL memory pool extracts 7-9% from this reduced amount. This necessitates targeting a specific memory amount in the CXL pool device, which proves challenging due to the need to populate every socket for maximum throughput and the large jumps in DIMM size. While there are specific configurations where this approach can be successful, it imposes constraints on the overall system, limiting choices in memory, core count, and the degree of CXL pooling.

They also see a challenge with CXL memory pools in the significant increase in software complexity. Experimental results on real hardware reveal that, for random accesses, CXL devices exhibit slower performance than the optimal values indicated in standard documents. While CXL has high latency, its substantial throughput allows competitive transfer of larger memory blocks (a few kilobytes) compared to DRAM. However, achieving this involves explicitly copying remote memory into local memory, transforming the CXL pool from directly accessed memory to a distant memory cache. The absence of commercially available CXL devices and non-disclosure agreements limiting result publication without prior approval adds further complications, relying on experimental findings from specific papers for conclusions.

This paper's performance evaluation is based on the two existing CXL benchmarking papers [7],[8]. In these researches, they use CXL 1.0/1.1 versions, which have only the first CXL features. At this moment, there are no papers available yet on benchmarking newest CXL versions (3.0 & 3.1). Therefore, the missing features of the newest CXL version effect the performance comparison.

3.3.4 Cost studies - Yale & ByteDance

The study [16] conducted by Yale and ByteDance researchers investigates the performance of ASIC (Application-Specific Integrated Circuit) CXL memory within different data-center scenarios. They aim to understand how CXL memory, when integrated through specially designed policies and strategies, can impact critical performance metrics such as throughput, latency, and cost reduction. They are focusing on commercial CXL 1.1 Type-3 devices, using CXL.io and CXL.mem protocols for memory expansion in single-server environments.

They are using an experimental test platform consists of three servers with specific configurations. Two of these servers are designated for the CXL experiments. Each of these servers is equipped with dual Intel Xeon 4th Generation CPUs, known as Sapphire Rapids, 1 TB of 4800 MHz DDR5 memory, and two 1.92 TB SSDs. Additionally, each server includes two A1000 CXL Gen5 x16 ASIC memory expander modules from AsteraLabs. Each module provides 256 GB of 4800 MHz memory, resulting in a total of 512 GB of additional memory per server. Both A1000 memory modules are attached to socket 0 of the respective servers. The third server serves as the baseline and is configured identically to the CXL experiment servers, with dual Intel Xeon 4th Generation CPUs, 1 TB of 4800 MHz DDR5 memory, and two 1.92 TB SSDs, but without the CXL memory expanders. This baseline server is used for initiating client requests and running workloads that strictly utilize the main memory during the

application assessments. All three servers are interconnected via 100 Gbps Ethernet links to ensure high-speed communication and data transfer between them.

The cost analysis reveals that adopting CXL memory expansion offers significant benefits for data center applications, including comparable performance and operational cost savings. However, determining the Return on Investment (ROI) for such innovative technology poses a significant challenge. Although detailed technical specifications and benchmark performance results are available, accurately forecasting Total Cost of Ownership (TCO) savings remains difficult due to the complexity of simulating benchmarks at production scale and the limited availability of CXL hardware. Traditional cost models, which could provide such forecasts, require extensive internal and sensitive information that is often inaccessible.

To address this, the researchers propose an Abstract Cost Model designed to estimate TCO savings without relying on internal or sensitive data. This model uses a set of metrics obtainable through microbenchmarks and a few empirical values that are easier to approximate or access, providing a viable means to evaluate the economic viability of implementing CXL technology.

Using a capacity-bound application like Spark SQL as an example, the model demonstrates that additional capacity enabled by CXL memory reduces data spillage to SSDs, resulting in higher throughput and fewer servers needed to meet performance targets. By dividing the execution time into segments processed by main memory, CXL memory, and SSD storage, the researchers establish a method to approximate execution times and thus, cost savings.

The model measures baseline performance (throughput when most data is spilled to SSD), relative performance when the entire working set is in main memory (MMEM), and relative performance when the entire working set is in CXL memory. These measurements are then used to formulate a cost model that estimates the TCO savings.

For example, with certain performance ratios and cost assumptions, the model suggests that using CXL memory could reduce the number of servers by approximately 32.71%, leading to an estimated TCO saving of 25.98%.

The Abstract Cost Model (Table 5) provides an accessible way to estimate the benefits of using CXL memory, guiding the design of next-generation infrastructure. The model is adaptable, allowing for the inclusion of additional infrastructure expenses such as the cost of CXL memory controllers, switches, PCBs, and cables as fixed constants. However, it currently focuses on one application type at a time, which poses a challenge for data centers evaluating cost savings for multiple distinct applications with shared resources, an area identified for future investigation.

$$TCO_{saving} = 1 - \frac{TCO_{CXL}}{TCO_{baseline}} = 1 - \frac{N_{CXL}}{N_{baseline}} R_t$$

$$\frac{N_{CXL}}{N_{baseline}} = \frac{CR_c(R_d - 1)}{R_c R_d (C + 1) - CR_c - R_d}$$

Table 5: This table defines parameters used in the abstract cost model. The parameters include throughput measurements for different memory configurations (SSD, main memory, and CXL memory), memory capacity ratios, and the number of servers required in baseline and CXL-equipped clusters. Additionally, it includes a parameter for comparing the total cost of ownership between standard and CXL-enabled servers.

Parameter	Description	Example Value
P_s	Throughput when (almost) the entire working set is spilled to SSD on a server. Normalized to 1 in the cost model.	1
R_d	Relative throughput when the entire working set is in main memory on a server, normalized to P_s.	10
R_c	Relative throughput when the entire working set is in CXL memory on a server, normalized to P_s.	8
D	The MMEM capacity allocated to each server. For completeness only, not used in the cost model.	-
C	The ratio of main memory to CXL capacity on a CXL server. E.g., 2 means the server has 2× MMEM capacity compared to CXL memory.	2
N_{baseline}	Number of servers in the baseline cluster.	-
N_{cxl}	Number of servers in the cluster with CXL memory to deliver the same performance as the baseline.	-
R_t	Relative TCO comparing a server equipped with CXL memory vs. a baseline server. E.g., if a server with CXL memory costs 10% more than the baseline server, this parameter is 1.1.	1.1

4 Message-based communication in Nokia Cloud-RAN

This section explores the transformative role of CXL in enabling efficient, low-latency communication within Nokia Cloud-RAN. CXL reduces CPU overhead and accelerates data sharing by introducing cache-coherent, multi-host shared memory, offering significant advantages over traditional networking paradigms like IPC. It addresses critical challenges in modern data center environments, such as resource utilization, scalability, and unsynchronized memory access, while facilitating faster intra-rack communication. Through performance models and real-world use cases, we demonstrate how CXL optimizes microservices architectures, enhances Kubernetes-based networking, and supports scalable multi-tenancy, making it a cornerstone technology for future cloud infrastructures.

4.1 Introduction

Compute Express Link 3.0 enables multi-host shared memory with hardware cache coherency across compute nodes, reducing CPU usage by allowing applications to share pointers to data instead of serializing and transmitting it. However, this introduces safety risks, as shared memory removes the isolation between sender and receiver that networking typically provides. This can lead to unsynchronized memory access if the sender modifies data while the receiver is processing it, which could cause significant issues, especially in scenarios involving mutually distrustful applications. [3].

Moreover, CXL 3.0 introduces a new transport layer by enabling multiple hosts to communicate through fast, byte-addressable, cache-coherent shared memory. With CXL, hosts can map the same shared memory region, allowing updates via load/store instructions to be instantly visible to all connected hosts without explicit communication. Compared to RDMA and HTTP, CXL-based RPC frameworks could offer significant performance improvements in communication latency. [3].

In scenarios where up to 32 servers are connected to a shared memory pool via CXL, this architecture would limit memory sharing to a single rack (32–64 nodes). For longer-range communication, CXL is expected to coexist with conventional networking like TCP and RDMA. This setup suits datacenter environments, where microservices distributed across multiple servers typically communicate using RPCs, enabling faster intra-rack communication through shared memory while relying on RDMA for larger distances. [17].

4.2 Problem description: From messaging to data sharing

Nokia Cloud RAN, or Virtualized Radio Access Network (vRAN), is an advanced Radio Access Network solution designed to meet the demands of 5G networks and beyond. It disaggregates traditional RAN hardware and software, leveraging a cloud-native architecture to provide agility, scalability, flexibility, and openness. Nokia Cloud RAN enables Communication Service Providers (CSPs) and enterprises to enhance

cost and resource efficiency while introducing new services for the enterprise space. By utilizing shared edge infrastructure for edge cloud deployments, it supports the deployment of 5G radio capacity and paves the way for 6G evolution. A key feature of Nokia Cloud RAN is its integration of specialized hardware acceleration through the Nokia Cloud RAN SmartNIC, which ensures performance parity with purpose-built RAN environments and delivers an energy-efficient solution for demanding Layer 1 processing. It complements existing network assets, facilitates synergies with enterprise on-premises clouds, and provides seamless interworking with classical RANs. [18].

Nokia's Cloud RAN product uses Kubernetes networking to manage its distributed architecture (Figure 5), which comprises nodes and clusters interconnected via a Container Network Interface (CNI). In the current implementation, inter-node communication relies on the traditional Linux networking stack, which, while functional, introduces latency and inefficiencies that can impact overall system performance.

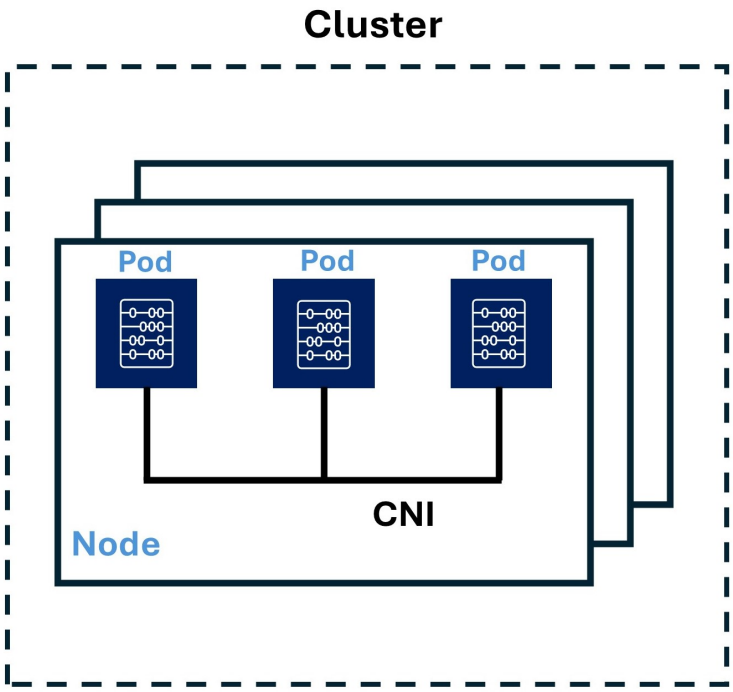


Figure 5: This figure illustrates the current system architecture, consisting of nodes and clusters interconnected via CNI. In this implementation, pod-to-pod communication relies on the traditional Linux networking stack, which introduces latency and inefficiencies that can affect overall system performance.

As the demand for low-latency, high-efficiency communication in cloud-native environments grows, Nokia is exploring the adoption of CXL technology to enhance inter-node messaging (Figure 6). CXL offers a promising alternative by enabling the creation of high-speed CXL socket connections that effectively bypass the kernel, reducing latency and improving resource utilization. Integrating CXL into the Cloud

RAN product could significantly optimize data transfer and scalability, aligning with the evolving requirements of next-generation cloud-based networks.

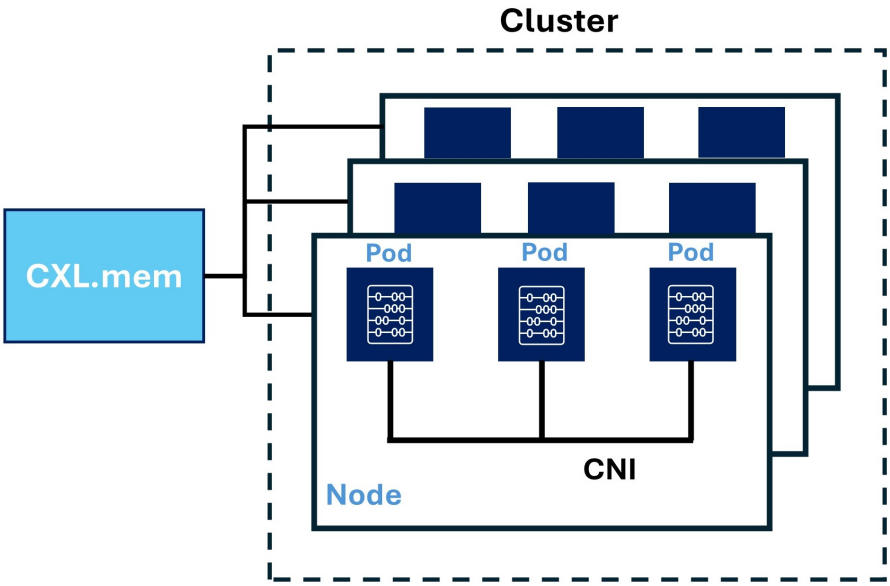


Figure 6: This figure illustrates the CXL system architecture, where nodes are connected to a CXL device using the CXL.mem protocol. This approach creates a CXL socket, enabling pod-to-pod communication entirely within user space. By bypassing the transition into kernel space, this method reduces latency, lowers CPU overhead, and improves overall network performance.

The current Linux networking stack used by Cloud RAN relies on transitioning between user space and kernel space when communication between pods is required (Figure 7). This involves copying data from user space into kernel space for processing, which introduces additional latency and consumes CPU resources, especially in scenarios involving high-frequency or large-volume pod-to-pod communication. As Cloud RAN systems scale and the demand for low-latency, high-efficiency networking increases, this traditional approach becomes a bottleneck.

A promising future alternative involves leveraging CXL technology to establish a CXL socket, enabling pod-to-pod communication to occur entirely within user space (Figure 8). This paradigm would eliminate the need for data to transition into kernel space, thereby reducing latency, minimizing CPU overhead, and optimizing the overall performance of the networking stack. By bypassing the kernel entirely, a CXL-based approach not only streamlines the data path but also aligns with the growing need for high-performance, scalable networking solutions in modern cloud-native architectures.

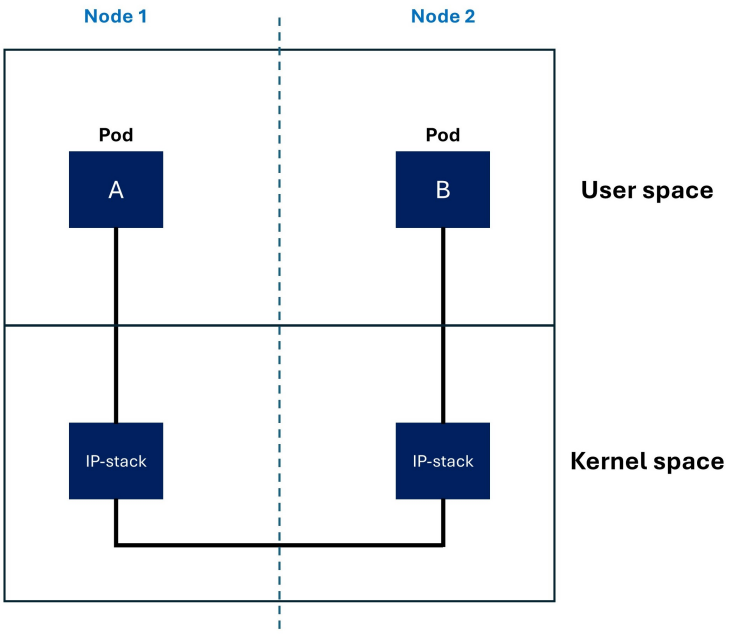


Figure 7: This figure illustrates Pod-to-Pod communication, showing how packets are transmitted through the IP stack in kernel space. In this traditional approach, pods rely on the kernel networking stack to send and receive packets between them.

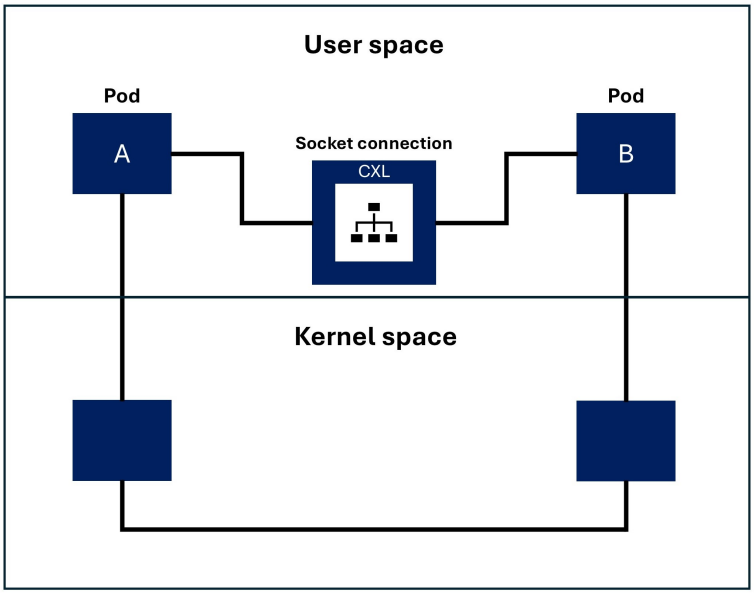


Figure 8: This figure illustrates Pod-to-Pod communication with CXL, demonstrating how CXL enables the creation of a CXL socket connection within user space. By using CXL, packets can bypass kernel space, improving communication efficiency between pods.

4.3 Messaging in communication networking

This section explores key technological advancements in modern computing, focusing on microservices, container network interfaces (CNI), Kubernetes networking, and innovations in Linux networking stacks. It examines the principles of microservices architecture, emphasizing modularity, scalability, and decentralized data management, alongside messaging paradigms crucial for their operation. The discussion transitions into the role of CNIs in containerized environments, contrasting approaches like Docker's CNM and Kubernetes-supported CNI, and highlighting their impact on efficient container orchestration. Kubernetes is presented as a robust platform that leverages CNIs to optimize resource utilization and ensure seamless inter-Pod communication. The limitations and challenges of traditional Linux networking stacks are analyzed, particularly in the context of high-speed and low-latency demands, showcasing the evolution of network processing techniques like kernel bypassing and frameworks such as DPDK. Together, these advancements highlight the synergy between emerging technologies in cloud-native and networking domains, offering insights into scalability, performance optimization, and operational efficiency.

4.3.1 Microservices and Messaging

While the core concepts behind microservices, such as dividing services into functions that communicate through programming interfaces, are not new, recent advancements in cloud computing have allowed these principles to be extended in novel ways. The cloud has facilitated rapid, flexible, and scalable implementations of microservices, evolving beyond the traditional, rigid approaches of Service-Oriented Architectures (SOA). This evolution is driven by the demand for interchangeable and adaptable components that leverage the inherent advantages of cloud infrastructure. [19].

Microservices architecture emphasizes modularity, scalability, and decentralization, allowing for a more flexible and maintainable approach to software development. As discussed in [20], one of its core principles is the decomposition of a system into small, independent components, each referred to as a microservice. These microservices encapsulate all the resources, such as business logic and data, necessary for their functionality, enabling them to operate autonomously. This modular design allows for easier deployment, replacement, and modification of individual services without affecting the rest of the system. By structuring services in this way, microservices facilitate a more agile development process, where updates and improvements can be made incrementally, enhancing the system's overall adaptability and scalability.

Another key characteristic of microservices is that they are typically organized around business capabilities rather than technical layers. Each microservice is responsible for a specific business function, such as managing a shopping cart or handling payment processing, and encapsulates all related functionalities and data within its scope. This organizational model aligns with the structure of business operations, making it easier to scale specific services based on demand or changes in business requirements. This approach also promotes the autonomy of development teams, as each team can focus on building and maintaining a specific service without

the need for constant coordination with other teams. This reduces dependencies and increases the efficiency of development and deployment processes.

Data management in microservices is also decentralized, with each microservice handling its own data storage and management. Unlike traditional centralized databases, where data is shared across multiple components, each microservice maintains its own data model and storage solution. This allows each service to select the most appropriate storage technology, whether it be a relational database, file system, or other data storage mechanisms. While this decentralized data management introduces complexities, such as managing distributed transactions, it provides greater flexibility and enables services to operate independently, further enhancing the scalability and resilience of the overall system.

In microservices architecture, messaging plays a critical role in enabling communication between services. Microservices are considered "smart endpoints" because they encapsulate all necessary resources for their functionality. These endpoints communicate via messaging, which is typically asynchronous and lightweight. Microservices use lightweight message buses like ZeroMQ for simple, reliable communication. This decoupling of the service logic from the messaging mechanism enhances the flexibility and longevity of the services.

The "dumbness" in microservices refers to the minimal role of the message bus. Microservices use RESTful protocols, allowing for decentralized and choreographed service interactions, focusing on message-passing without requiring extensive middleware, which improves scalability and reduces complexity. [20].

4.3.2 Container Network Interface

As the number of cloud-deployed applications grows, the overhead associated with virtualization remains a significant concern [21]. Traditional virtual machines (VMs) rely on virtualized hardware and require a complete operating system (OS) instance, which results in a large footprint. This limitation reduces the number of VMs that can be consolidated on a single physical machine, and the prolonged OS startup time renders VMs inefficient for hosting short-lived applications. In contrast, container-based virtualization mitigates these challenges by sharing OS libraries and the kernel among applications, each operating within an isolated namespace, or container.

As container adoption expands across various sectors and the scale of containerized applications continues to increase, there is a growing demand for a standardized container network specification. Currently, two main standards for container network interfaces exist. The Container Network Model (CNM), proposed by Docker, consists of modules such as sandbox, network, and endpoints, and has been adopted by organizations like VMWare and Weave. In contrast, the Container Network Interface (CNI), a community-driven standard developed by Google and CoreOS, offers a more streamlined and flexible design. CNI is supported by platforms such as Apache Mesos and Kubernetes. Additionally, several open-source projects, such as Calico, are compatible with both standards. [21].

4.3.3 Kubernetes CNI

Nokia uses Kubernetes which is the leading container orchestration platform used by cloud service providers (CSPs) to optimize cloud resource utilization [22]. It offers the flexibility to run various containerized cloud applications on both physical and virtual cloud infrastructures. In Kubernetes, the "Pod" represents the fundamental unit for deployment, scaling, and management, containing one or more containers that share resources such as networking. Pods can be dynamically scaled to meet workload demands and ensure resilience. The increasing adoption of microservices and function-as-a-service architectures requires the support of numerous containers and efficient communication between them. To meet these demands, Kubernetes automates, scales, and secures its orchestration and networking processes for large-scale deployments. The platform utilizes the Container Network Interface (CNI) as the foundation for its networking, assigning each Pod a unique IP address for cluster communication. CNI plugins manage inter-Pod communication and various network operations.

Kubernetes networking relies on both Linux and Kubernetes namespaces, which serve distinct purposes. Linux network namespaces provide network isolation for each Pod, allowing independent operation from the host and other Pods. In contrast, Kubernetes namespaces facilitate the division of a physical cluster into multiple virtual clusters, promoting flexible resource management and network policy application. While these namespaces are not fully isolated, they help allocate resources and implement network policies across different user workloads. [22].

The Kubernetes networking model addresses four types of communication: intra-Pod (within a Pod), inter-Pod (between Pods), Service-to-Pod, and external-to-service communication. Kubernetes specifies the network model but relies on CNI plugins for implementation, with key requirements being IP addressability of Pods and communication without network address translation (NAT). Popular CNI plugins include Flannel, Weave, Calico, Cilium, and Kuberouter. Kubernetes Network Policy allows for traffic control and isolation, enforcing rules for ingress and egress traffic and determining which Pods can communicate with each other. Once a policy is applied, unapproved traffic is blocked, providing fine-grained control over network interactions. [23].

4.3.4 Linux Network messaging

The slowdown of Moore's Law, the end of Dennard scaling, and the rapid adaptation of high-bandwidth links have placed traditional host network stacks under considerable strain [24]. Over recent years, while datacenter access link bandwidth and the associated computational demands for packet processing have increased by 4 to 10 times, technological advancements in critical host resources—such as CPU clock speeds, core counts, cache sizes, and NIC buffer capacities—have remained relatively stagnant. Consequently, the challenge of designing CPU-efficient host network stacks has gained prominence. To address this issue, the research community has proposed a variety of solutions, including optimizations to the Linux network stack, hardware

offloading techniques, Remote Direct Memory Access (RDMA), clean-slate userspace network stacks, and specialized host networking hardware. However, a comprehensive understanding of the CPU inefficiencies in traditional Linux network stacks is essential to inform the design space for these solutions. This task is inherently complex due to the size and intricacy of the Linux network stack, which consists of numerous tightly integrated components forming an end-to-end packet processing pipeline.

The Linux network stack (Figure 9) has distinct data paths for the sender-side (from application to NIC) and the receiver-side (from NIC to application). It tightly integrates multiple components into a unified, end-to-end packet processing pipeline. In the context of the kernel space having three layers—Transport (L4), Networking (L3), and Data Link (L2)—the sender and receiver processes can be described as follows:

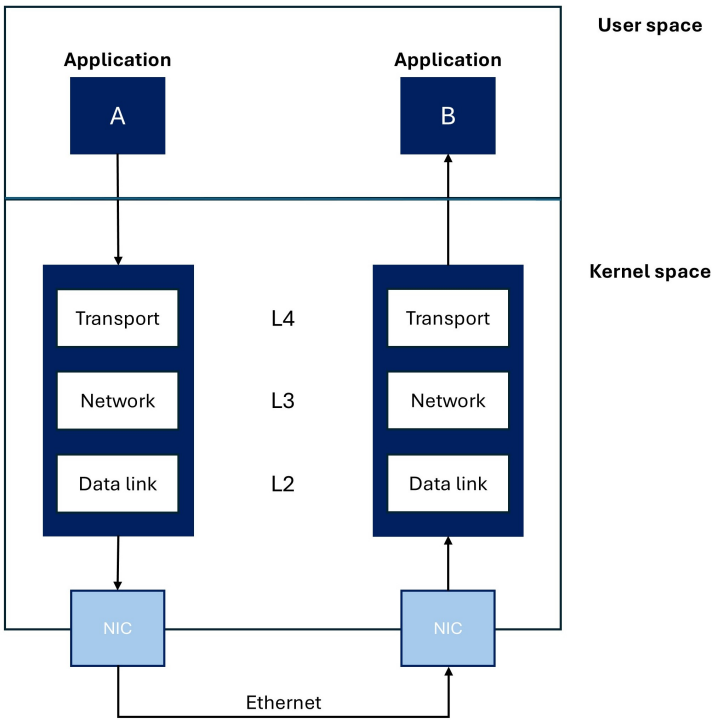


Figure 9: This figure illustrates the architecture of the Linux networking stack, describing how application data packets travel through user space, kernel space, and the network interface card. The figure highlights the flow of packets as they pass through Layer 2, Layer 3, and Layer 4 of the Linux networking stack, demonstrating the processing steps involved in packet transmission and reception.

When an application issues a write system call, the Transport Layer (L4) initiates the creation of socket buffers (SKBS), which act as containers for packet data. The kernel copies data from the userspace buffer into these skbs in the Network Layer (L3). The TCP/IP protocols in this layer handle the segmentation of skbs into smaller chunks, typically sized according to the Maximum Transmission Unit (MTU). This process uses a feature called Generic Segmentation Offload (GSO), optimizing the

handling of large data blocks. Once segmented, the skbs are passed to the Data Link Layer (L2), where the data is prepared for physical transmission. The skbs are placed in the transmit (Tx) queue of the Network Interface Card (NIC) driver, awaiting transmission. Using Direct Memory Access (DMA), the NIC fetches the packet data directly from the kernel buffers without additional CPU intervention, ensuring efficient communication between the kernel and the hardware.

On the receiving end, when a packet arrives, the Data Link Layer (L2) is first engaged. The NIC, equipped with receive (Rx) queues and memory buffers, transfers incoming frames from the network to kernel memory using DMA. The NIC then signals the driver through interrupts, allowing the CPU to process the new data. NAPI polling is triggered to efficiently process incoming frames in bursts, reducing the overhead of frequent interrupts. The driver allocates SKBS to hold the incoming packet data. In the Network Layer (L3), protocols like IP handle the packet's routing and other layer 3 operations. To optimize performance, SKBS may be merged through Generic Receive Offload (GRO) or its hardware counterpart, Large Receive Offload (LRO), which reduces the number of packets that need processing. Finally, the Transport Layer (L4) takes over for any remaining processing, such as ensuring proper TCP sequencing. The SKBS are then delivered to the application's receive queue. Data copying between the kernel and userspace occurs only once, with the kernel operations being optimized by manipulating metadata and pointers to the skbs, avoiding unnecessary data duplication. [24].

The Linux network stack is optimized for general-purpose networking, supporting a range of protocols like IPv4, IPv6, TCP, and UDP, and functions well for applications requiring up to 1 Gbit/s [25]. However, it struggles to maintain performance at higher speeds, such as 10 Gbit/s, where packet loss becomes a problem due to system limitations. The primary bottleneck is the CPU, which has limited cycles per second. The more CPU cycles consumed per packet, the fewer packets can be processed, restricting throughput.

Additionally, memory management introduces further inefficiencies. Per-packet memory allocation and deallocation, the complexity of the sk_buff data structure, and multiple memory copies all contribute to significant CPU overhead. These design choices make the Linux network stack suitable for general use but not for high-speed packet processing. To address these challenges, specialized frameworks—discussed in later sections—introduce optimizations to reduce CPU cycles per packet and improve memory handling, enabling higher-speed network applications. [25].

In the Linux networking stack, latency refers to the time delay experienced by a network packet as it travels through the stack from the source to the destination [26]. It states the time taken for a packet to be processed by the operating system's network layers (both in the kernel and user space) and the physical network interface, along with the time spent traversing the network to its destination. Latency operates independently from throughput, meaning that increasing throughput—by processing or sending more messages—does not inherently reduce latency. The significance of latency varies by application. Latency-sensitive applications face performance degradation when resources stall while awaiting data from the network. If no other threads can be scheduled during this wait, the application is considered latency-

dependent. In contrast, applications that can mask latency by scheduling multiple threads shift toward a throughput-oriented approach. However, spawning many threads to hide latency may reduce system efficiency due to increased overhead from context switching, thread migration, and contention for shared resources.

A significant portion of TCP/IP latency arises from the software interface, where the asynchronous nature of communication requires the receiving node to interrupt a processor and identify the appropriate application through protocol stacks. This leads to context switching and data copying before the message is processed, introducing delays. Additionally, techniques like interrupt moderation, used by NICs to reduce processing overhead, can further add to latency by batching packets, even if some packets are more latency-sensitive than others.

TCP/IP has much higher latency than hardware-based protocols like Infiniband and Myrinet because TCP/IP relies heavily on software for packet handling, including packetization, segmentation, reassembly, and data movement. In contrast, Infiniband and Myrinet perform these tasks directly in hardware, allowing data to be delivered straight to user space without requiring transitions between kernel space and user space or costly memory copies. This hardware-based approach significantly reduces latency compared to TCP/IP's software-intensive process. [26].

4.3.5 Kernel by-pass and Dataplane Development Kit

Between 2010 and 2012, researchers introduced methods to use static, pre-allocated buffers with minimal metadata for common packet operations, such as IPv4 packet forwarding. A key innovation in these systems was the move to perform all packet processing in user space, bypassing the traditional operating system stack by transferring packets directly between the NIC and user space. These approaches also utilized batching techniques for system calls and device access, with a write barrier preceding them. This research led to the development of general frameworks for fast user-space packet I/O, such as netmap and The Data Plane Development Kit (DPDK). [27].

DPDK (Figure 10) is designed to improve packet processing performance by bypassing the traditional Linux network stack [25]. It allows applications to directly interact with network hardware, avoiding the overhead of kernel-based packet handling. DPDK operates in user space, using techniques such as polling (instead of interrupts) and zero-copy mechanisms to minimize delays and maximize throughput.

Unlike the Linux network stack, which relies on context switches, system calls, and memory copies between kernel and user space, DPDK maps network hardware resources directly into user space, enabling faster packet processing. This approach significantly reduces latency and CPU overhead, making DPDK ideal for high-performance networking environments like telecommunications, data centers, and real-time processing systems. [25].

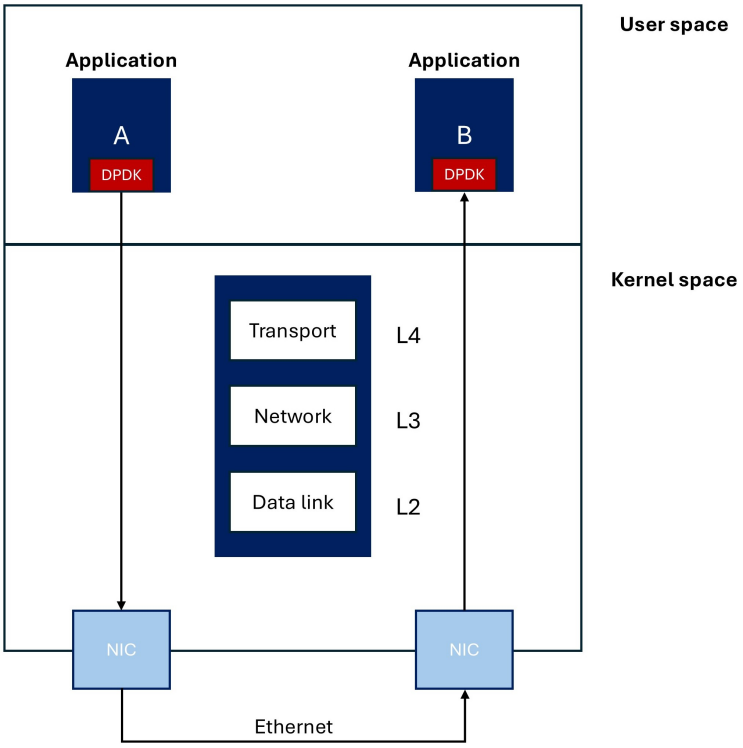


Figure 10: This figure illustrates the architecture of DPDK Kernel Bypass, showing how DPDK improves packet processing performance by bypassing the traditional Linux network stack. By operating in user space, DPDK enables applications to directly interact with network hardware, eliminating the overhead associated with system calls, context switches, and kernel-based packet handling, resulting in lower latency and higher throughput.

5 Techno-Economic Analysis

This section examines a techno-economic analysis of CXL-enabled networking stacks. It introduces a Total Cost Model that breaks down costs into different parts, such as CPU processing, memory access, network overhead, power consumption, and hardware costs. Unlike traditional models, this approach considers real-world factors like caching inefficiencies and power consumption changes to provide a more accurate cost analysis. The section also explores performance models, focusing on how CXL affects Round-Trip Time (RTT) and data throughput. Example simulations compare systems with and without CXL, showing how it can improve efficiency. Finally, the section looks at practical use cases, such as memory pooling and multi-tenancy, demonstrating how CXL helps optimize resource utilization and scalability. By combining cost, performance, and real-world use cases, this section provides an understanding of the benefits of CXL.

5.1 Total Cost Model

The Total Cost Model provides a comprehensive framework for evaluating the economic impact of CXL-enabled networking stacks by breaking down costs into specific, measurable components. Unlike traditional CapEx and OpEx models, which aggregate costs into broad categories, this approach incorporates granular elements such as CPU processing, memory access, network overhead, power consumption, and hardware acquisition and maintenance costs. By integrating these components, along with non-linear dynamics such as caching inefficiencies, network congestion, and power utilization under varying workloads, the model offers a detailed and realistic representation of real-world system costs. Moreover, it directly links costs to performance metrics like round-trip time (RTT) and bandwidth efficiency, enabling a focused comparison between CXL-enabled systems and traditional Linux kernel stacks. This method not only highlights the cost savings and performance benefits introduced by CXL but also provides a flexible and adaptable framework for evaluating its advantages across diverse scenarios.

The total cost (C_{total}) of a CXL enabled networking stack, presented in Figure 8, can be modeled as the sum of multiple cost components: CPU processing costs, memory access costs, network overhead costs, power consumption costs, and hardware acquisition and maintenance costs. This model allows for a direct comparison with the costs of a traditional Linux kernel networking stack, incorporating both the advantages introduced by CXL and the inherent inefficiencies of kernel-mediated operations.

$$C_{\text{total}} = C_{\text{cpu}} + C_{\text{memory}} + C_{\text{network}} + C_{\text{power}} + C_{\text{hardware}}$$

Non-linearities arise in several cost components. The CPU cost (C_{cpu}) is determined by the processing time per packet (T_{cpu}), the CPU utilization (U_{cpu}), and the cost per CPU cycle (C_{cycle}). For larger packet sizes, the cost grows quadratically due to increased computational complexity such as fragmentation, modeled as:

$$C_{\text{cpu}} = T_{\text{cpu}} \cdot U_{\text{cpu}} \cdot C_{\text{cycle}} \cdot \left(1 + \alpha_{\text{cpu}} \cdot \left(\frac{S}{S_{\max}}\right)^2\right),$$

where α_{cpu} is a scaling factor for non-linear effects, S is the packet size, and $S_{\max}$ is a normalization constant.

Memory access costs (C_{memory}) are influenced by the number of packets processed (N_{packets}), the probability of cache hits (P_{cache}), and the costs of accessing cache and DRAM (C_{cache} and C_{dram}, respectively). Cache hit probability decreases exponentially with increasing packet size due to reduced caching efficiency:

$$P_{\text{cache}}(S) = P_{\text{base}} \cdot e^{-S/S_{\max}},$$

where P_{base} is the base cache hit probability, and $S_{\max}$ is a scaling parameter. Memory cost is then calculated as:

$$C_{\text{memory}} = N_{\text{packets}} \cdot \left(P_{\text{cache}} \cdot C_{\text{cache}} + (1 - P_{\text{cache}}) \cdot C_{\text{dram}}\right).$$

Network overhead costs (C_{network}) grow non-linearly due to retransmissions and congestion. This behavior is captured using a power-law relationship between packet size and transmission cost:

$$C_{\text{network}} = \frac{N_{\text{packets}} \cdot \text{RTT}}{B_{\text{link}}} \cdot C_{\text{transmission}} \cdot \left(\frac{S}{S_{\text{ref}}}\right)^{\beta},$$

where RTT is the round-trip time, B_{link} the link bandwidth, $C_{\text{transmission}}$ the cost per transmitted packet, S_{ref} a reference size, and β an exponent capturing the severity of non-linearity.

Power consumption costs (C_{power}) exhibit non-linear growth due to inefficiencies at higher system utilization, modeled as:

$$C_{\text{power}} = \left(P_{\text{cpu}} \cdot T_{\text{cpu}} + P_{\text{memory}} \cdot T_{\text{memory}} + P_{\text{nic}} \cdot T_{\text{network}}\right) \cdot \left(1 + \alpha_{\text{power}} \cdot U^2\right),$$

where P_{cpu}, P_{memory}, and P_{nic} denote the power consumption rates of the CPU, memory, and NIC, respectively; α_{power} is a scaling factor for inefficiency; and U is system utilization. Finally, hardware costs (C_{hardware}) include acquisition and maintenance expenses, with savings from CXL factored in:

$$C_{\text{hardware}} = C_{\text{base}} + C_{\text{maintenance}} - \text{Savings due to CXL}.$$

By summing these components, the total costs for CXL-enabled and traditional networking stacks are expressed as:

$$C_{\text{total,CXL}} = C_{\text{cpu,CXL}} + C_{\text{memory,CXL}} + C_{\text{network,CXL}} + C_{\text{power,CXL}} + C_{\text{hardware,CXL}},$$

$$C_{\text{total,base}} = C_{\text{cpu,base}} + C_{\text{memory,base}} + C_{\text{network,base}} + C_{\text{power,base}} + C_{\text{hardware,base}}.$$

The relative cost improvement can then be evaluated as:

$$\Delta C = \frac{C_{\text{total,base}} - C_{\text{total,CXL}}}{C_{\text{total,base}}} \times 100\%.$$

By integrating non-linear behaviors into the cost model, this framework accurately reflects real-world inefficiencies and advantages. CXL's ability to reduce CPU overhead, improve cache efficiency, and lower active power consumption results in significant cost savings and performance improvements over traditional networking stacks.

5.2 Performance models

Next, we examine performance models for CXL-enabled systems, focusing on its impact on Round-Trip Time (RTT) and throughput. RTT is modeled as the sum of CPU processing, memory access, and network transmission latencies, with CXL reducing CPU and memory latencies via kernel bypass and increased memory bandwidth. The throughput model evaluates data handling capacity, showing how CXL's enhancements to memory bandwidth and CPU efficiency sustain higher rates even for larger packet sizes. Simulations compare baseline and CXL-enabled systems under varying packet sizes, demonstrating significant performance improvements in both RTT and throughput. Compact visualizations illustrate CXL's ability to optimize latency and maximize throughput in high-performance networking environments.

5.2.1 Round-Trip Time (RTT) modeling

A mathematical simulation model for CXL in Latency and Round-Trip Time (RTT) modeling focuses on quantifying how CXL's memory access optimizations (like kernel bypass and high-speed direct memory access) impact the overall latency and round-trip communication time between endpoints, such as pods or containers in a distributed environment. We use available bandwidth instead of theoretical bandwidth because it reflects the actual portion of memory bandwidth that can be utilized by an application or process under real-world conditions. While theoretical bandwidth represents the maximum achievable rate under ideal scenarios, it does not account for factors like hardware contention, system overhead, inefficient memory access patterns, and workload-specific characteristics, which reduce the bandwidth effectively accessible to the application. By focusing on available bandwidth, especially in memory-intensive tasks, we can accurately model and optimize performance, identify bottlenecks, and assess improvements such as those enabled by technologies like CXL, which reduce kernel overhead and increase the fraction of bandwidth accessible to applications.

The generic RTT is composed of three components:

$$\text{RTT}_{\text{total}} = L_{\text{cpu}} + L_{\text{memory}} + L_{\text{network}} \tag{3}$$

The generic equation for RTT describes the fundamental components of processing, memory access, and network latency (Table 6). With the introduction of CXL, the

Table 6: Latency Parameters in the RTT Model.

Parameter	Definition	Formula
L_{cpu}	CPU processing latency per packet	-
L_{memory}	Memory access latency per packet, dependent on available memory bandwidth	-
L_{network}	Network transmission latency	-
L_{memory}	Memory latency calculated from packet size and available memory bandwidth	$\frac{\text{packet size}}{B_{\text{mem}}^{\text{available}}}$

latency characteristics change due to the improved efficiency of memory access and CPU processing. To reflect these CXL-specific changes, we introduce two reduction factors, α and β. They represent the decrease in CPU processing time and memory access time, respectively. The RTT equation for CXL (5) uses these factors, showing how CXL accelerates packet processing by reducing both CPU and memory-related delays. Specifically, the CPU processing time is scaled by (1 - α), and the memory access time is scaled by (1 - β). Also, the CXL system has its own variable for available memory bandwidth (Table 7).

$$\text{RTT}_{\text{CXL}} = (1-\alpha) \cdot L_{\text{cpu}} + (1-\beta) \cdot L_{\text{memory}} + L_{\text{network}} \tag{4}$$

$$\text{RTT}_{\text{CXL}} = (1-\alpha) \cdot L_{\text{cpu}} + (1-\beta) \cdot \frac{\text{packet size}}{B_{\text{mem}}^{\text{available,CXL}}} + L_{\text{network}} \tag{5}$$

Table 7: CXL-Specific Parameters in the RTT Model.

Parameter	Definition	Formula
α	Reduction factor for CPU processing time	-
β	Reduction factor for memory access time	-
$B_{\text{mem, CXL}}^{\text{available}}$	Available memory bandwidth with CXL	-
$(1-\alpha) \cdot L_{\text{cpu}}$	Reduced CPU processing time	-
$(1-\beta) \cdot L_{\text{memory}}$	Reduced memory access time with CXL	$(1-\beta) \cdot \frac{\text{packet size}}{B_{\text{mem}}^{\text{available,CXL}}}$

Next, we create a simulation model for CXL in the context of RTT. This simulation is a performance modeling exercise designed to assess and quantify the impact of CXL on the latency and round-trip time of network communication. This simulation aims to illustrate how CXL can optimize communication between nodes, such as pods or containers in distributed environments, by reducing the latency involved in CPU processing and memory access.

The simulation can be categorized as performance modeling and comparative analysis. It uses mathematical equations to represent RTT in a network communication

setup, allowing us to see how CXL affects key components of latency. The RTT is modeled as the sum of CPU processing time, memory access time, and network transmission time. By introducing reduction factors specific to CXL-enabled environments, the simulation clearly compares systems that use traditional kernel-mediated memory access and those that leverage CXL's enhanced memory capabilities.

The main objective of this simulation is to break down and analyze RTT into its core components and to demonstrate how CXL influences each. The components include CPU processing time, memory access time, and network transmission latency. In a typical network setup, memory access and CPU processing can contribute significantly to overall latency, especially in memory-bound or CPU-intensive applications. By reducing the processing load on the CPU and allowing direct memory access, CXL can reduce these components, leading to lower RTTs. This comparative analysis helps visualize the potential performance benefits CXL brings to network communication.

The simulation calculates RTT for a range of packet sizes in two scenarios: one without CXL and one with CXL. It uses baseline values for CPU time, memory bandwidth, and packet size to calculate RTT_{noCXL} and RTT_{CXL} (all parameters are listed in Table 8). By adjusting the reduction factors α and β , as well as comparing different bandwidths, the simulation shows how CXL's kernel bypass and high-speed memory access capabilities reduce latency in memory-intensive applications. This comparative analysis helps illustrate that CXL-enabled systems can handle larger data transfers more efficiently and maintain lower latency even as packet size increases.

Table 8: Parameters used in RTT-simulation.

Parameter	Definition	Value in Simulation
L_{cpu}	Baseline CPU processing time	0.5 ms
$B_{\text{mem}}^{\text{available}}$	Baseline memory bandwidth without CXL	20 GB/s
$B_{\text{mem}}^{\text{available,CXL}}$	Memory bandwidth with CXL	40 GB/s
L_{network}	Baseline network latency	1 ms
Packet size	Range of packet sizes	500 bytes to 5mb
α	CPU processing reduction factor with CXL	0.3
β	Memory access reduction factor with CXL	0.4

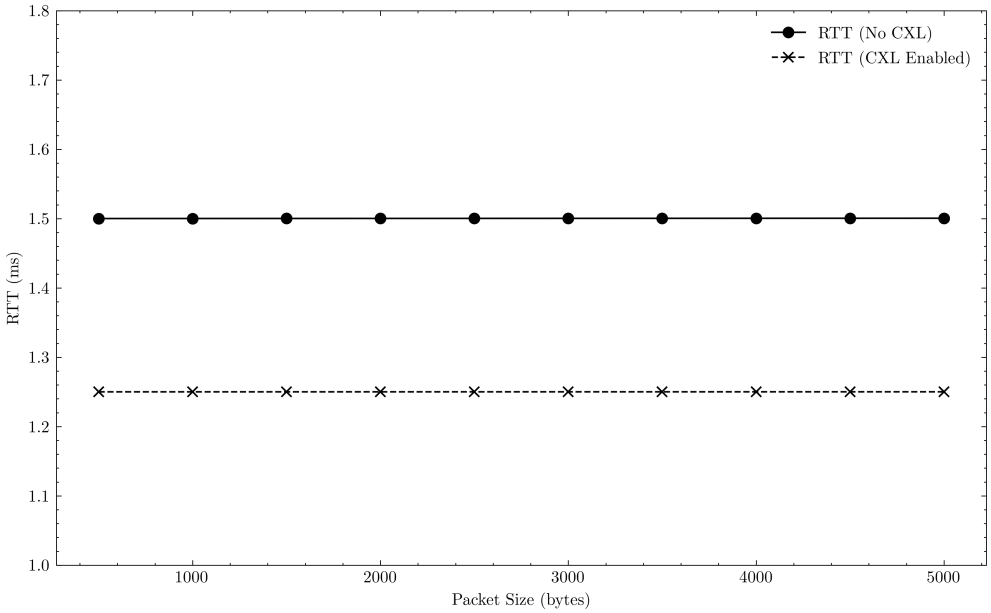


Figure 11: Round-Trip Time (RTT) Comparison with and without CXL (up to 5000 bytes)

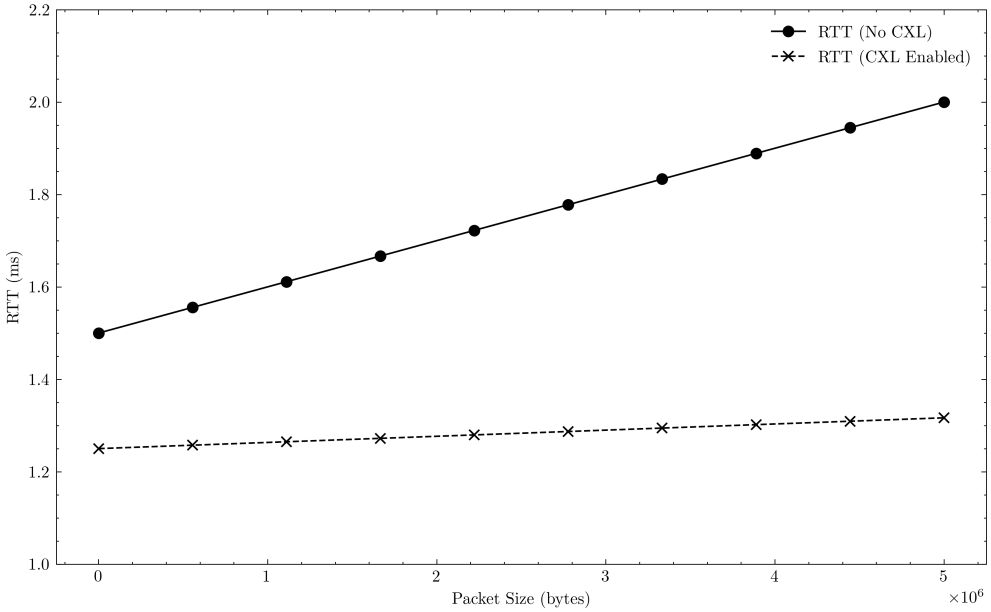


Figure 12: Round-Trip Time (RTT) Comparison with and without CXL (up to 5MB)

The output of the simulation includes plots showing how RTT scales with packet size. Without CXL, RTT increases more steeply as packet size grows due to limitations in memory bandwidth and increased CPU processing load. However, with CXL, the RTT curve grows more slowly, indicating reduced latency. This is because the enhanced memory bandwidth provided by CXL reduces the L_{memory} component, and kernel bypass reduces the L_{cpu} component.

While CXL primarily impacts L_{cpu} and L_{memory} , its indirect influence on $L_{network}$ comes from reducing queuing delays (L_{queue}) and minimizing retransmissions ($L_{retransmission}$). Faster packet processing and memory access reduce queueing backlogs and improve overall network efficiency, indirectly reducing network-related delays.

The output of the simulation includes plots showing how RTT scales with packet size. Without CXL, RTT increases more steeply as packet size grows due to limitations in memory access delay (M) and increased CPU processing delay (C). However, with CXL, the RTT curve grows more slowly, indicating reduced latency. This is because the enhanced memory bandwidth provided by CXL reduces M, and kernel bypass reduces C.

While CXL primarily impacts CPU processing delay and memory access delay, it also indirectly influences network delay by reducing queuing delay (Q) and minimizing retransmission delay (R). Faster packet processing and memory access lower queuing backlogs and improve overall network efficiency, which indirectly reduces network-related delays.

The earlier model of RTT (5) can be extended by breaking down network delay into its components: propagation delay (P), transmission delay (T), queuing delay (Q), network processing delay (X), and retransmission delay (R). This refined model accounts for how data packets experience latency at various stages of the network, from the physical transmission of signals to delays caused by congestion, device processing, and packet retransmissions. By substituting network delay with its expanded formula, the model becomes:

$$\text{RTT}_{\text{total}} = C + M + (P + T + Q + X + R), \tag{6}$$

providing a more detailed understanding of RTT and enabling a granular analysis of how CXL's optimizations, such as reduced queueing delays and lower retransmission rates, indirectly enhance network latency. All variables used are listed in Table 9. This extended model serves as a framework for evaluating both direct and indirect effects of CXL on system performance.

Table 9: Definitions of variables in the extended version of RTT.

Variable	Definition
C	CPU processing delay
M	Memory access delay
P	Propagation delay
T	Transmission delay
Q	Queuing delay
X	Processing delay
R	Retransmission delay

5.2.2 Throughput Model

To simulate CXL's effect on data throughput, we need to model how CXL impacts the system's ability to process and transmit data efficiently. Throughput, typically measured in packets per second or bits per second, depends on factors such as memory bandwidth, CPU processing capabilities, and network transmission speed. By enabling kernel bypass and high-speed direct memory access, CXL can significantly enhance throughput by reducing CPU overhead and increasing memory bandwidth.

Throughput ($T_{\text{throughput}}$) depends on memory bandwidth, CPU processing capabilities, and network transmission speed. We define throughput as:

$$T_{\text{throughput}} = \min\left(\frac{B_{\text{mem}}}{\text{packet size}}, T_{\text{CPU}}, T_{\text{network}}\right). \tag{7}$$

The parameters in the throughput model are defined in Table 10.

Table 10: Parameters used in the throughput model.

Parameter	Definition
B_{mem}	Available memory bandwidth (in bytes per second)
T_{CPU}	Maximum packet processing rate determined by CPU capabilities
T_{network}	Maximum network transmission rate

CXL improves B_{mem} and T_{CPU} by enabling direct memory access and reducing CPU processing load. This results in higher throughput due to:

- Increased memory bandwidth: Faster memory access.
- Reduced CPU load: Kernel bypass reduces CPU cycles needed for packet handling.

The simulation we are conducting next is a performance modeling and comparative analysis aimed at evaluating how CXL impacts data throughput in a networked system. This type of simulation helps quantify the improvements in data handling capabilities introduced by CXL, focusing on two main enhancements: increased memory bandwidth and reduced CPU processing overhead. By simulating both baseline (without CXL) and CXL-enabled scenarios, we can observe the potential performance gains in terms of data throughput and understand CXL's practical benefits in high-performance environments.

This simulation is best described as a deterministic analysis. It uses fixed input parameters, such as memory bandwidth, CPU processing rate, and network rate, to produce predictable outcomes (Table 11). The objective is to determine the maximum achievable throughput under specific conditions and compare it between a traditional setup and a CXL-enhanced system. Additionally, it is a comparative analysis because it focuses on contrasting two states: a baseline without CXL and an improved state with CXL. This comparison highlights how CXL's features can boost system performance by providing better memory bandwidth and reducing CPU load.

The simulation is structured as a theoretical model that assumes ideal conditions for demonstrating potential throughput gains. It does not incorporate real-world network variability such as congestion, packet loss, or other system-specific overheads. Instead, it isolates key factors like memory access time and CPU processing capabilities to show how CXL technology can improve these metrics. While real-world applications may show slightly different results due to external factors, this model provides a clear and controlled environment to visualize the fundamental advantages of CXL.

CXL can significantly improve throughput by enabling direct memory access and reducing CPU workload. These improvements translate into higher available memory bandwidth and faster data handling. In practical terms, this means that data can be processed more efficiently, especially when handling larger packet sizes. The simulation models this by applying increased memory bandwidth and improved CPU processing rates to a CXL-enabled system. By comparing these with the baseline values, we can see how much more data the system can handle per unit of time with CXL.

Table 11: Parameters used in throughput simulation.

Parameter	Definition	Value in Simulation
$B_{\text{mem}}^{\text{noCXL}}$	Baseline memory bandwidth	20 GB/s
$B_{\text{mem}}^{\text{CXL}}$	CXL-enabled memory bandwidth	40 GB/s
Packet size	Range of packet sizes	500 to 50,000 bytes
T_{CPU}	CPU processing rate (without/with CXL)	5M / 8M packets/s
T_{network}	Network packet processing rate	10M packets/s

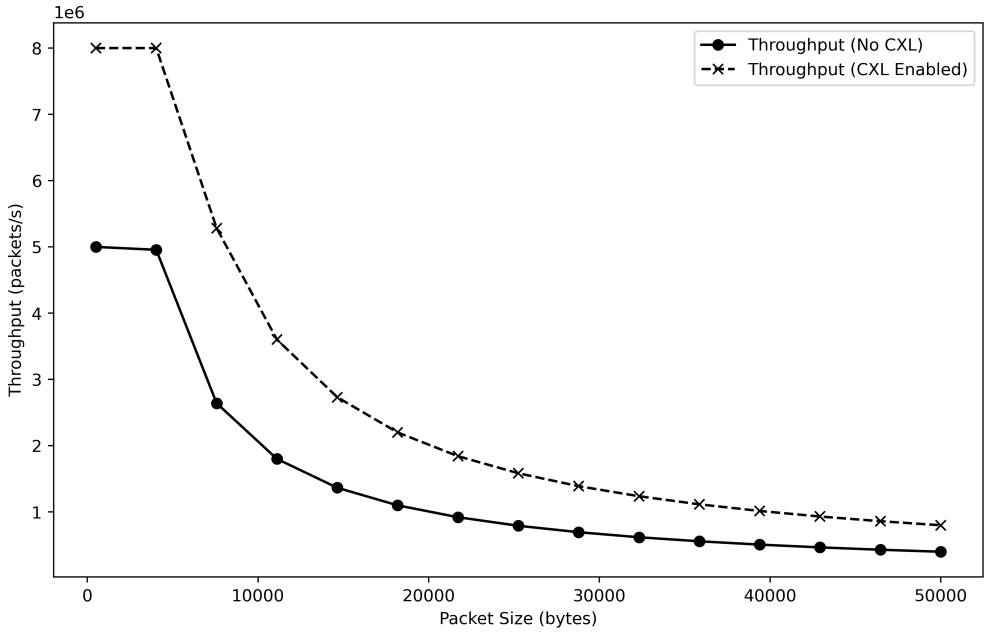


Figure 13: Simulation of data Throughput Comparison with and without CXL. This displays how the throughput decreases when packet size grows larger.

The results of the simulation (Figure 13) should show that without CXL, throughput is lower, especially for larger packet sizes where memory access time becomes the limiting factor. In contrast, the CXL-enabled scenario should demonstrate improved throughput due to increased memory bandwidth and reduced CPU load. This means that with CXL, the system can handle more data per second, allowing for more efficient processing of large packet sizes and enhancing overall performance.

5.3 Use Cases

This section explores the potential of CXL focusing on two key use cases: memory pooling and multi-tenancy. CXL's ability to decouple memory from compute resources enables shared memory pools that optimize resource utilization, reduce costs, and improve scalability. In multi-tenant environments, CXL facilitates dynamic allocation of memory resources across multiple workloads, enhancing performance isolation and enabling flexible resource provisioning.

5.3.1 Memory Pooling

Memory pooling, or memory disaggregation, enables sharing memory resources across multiple systems to improve utilization, elasticity, and resource management efficiency. By decoupling memory from individual compute nodes, it allows memory to be dynamically allocated based on real-time demands, reducing inefficiencies and overprovisioning. [28].

Memory pooling addresses the challenge of memory stranding, where memory remains unused because other resources, such as CPU cores, have been exhausted.

This imbalance often arises in cloud environments due to the difficulty of precisely matching cluster configurations to the diverse and fluctuating resource demands of applications. By disaggregating memory and pooling it through technologies like CXL, resources can be dynamically reassigned across multiple hosts. This flexibility eliminates the need to provision each server for worst-case scenarios, optimizing memory utilization and significantly reducing stranded resources. With CXL, memory pooling achieves low-latency access and high bandwidth, aligning closely with the performance of local DRAM, while enabling more efficient and scalable resource management. [7].

There are two primary methods for managing memory pooling: page-based and object-based. In the page-based approach, virtual memory techniques are used to enable seamless access to remote memory without requiring changes to application code. When a local memory page is unavailable, data is swapped between the host's DRAM and remote memory over a network connection, triggered by page faults. Although practical, this method relies heavily on the traditional memory hierarchy and suffers from latency due to frequent data movement and cache management. [28].

The object-based approach takes a different path, interacting with remote memory through custom databases like key-value stores instead of relying on virtual memory systems. This reduces issues like page faults and context switching but requires significant modifications to application code and interfaces. Both methods rely on high-performance network interfaces, such as RDMA, to transfer data between local and remote memory. However, these transfers introduce redundancies, such as multiple memory copies and software overhead, resulting in latency that is significantly higher than local DRAM access. [28].

CXL offers a transformative solution to these challenges. Its cache coherence capabilities enable it to address the inefficiencies of traditional memory pooling methods by reducing data movement overhead and improving resource sharing. By dynamically allocating memory resources with minimal latency, CXL not only optimizes utilization but also eliminates the need for overprovisioning, lowering hardware costs. [28].

In a traditional 3-node cluster (Figure 14), each node is equipped with dedicated memory sized for peak workload demands. This design often results in significant memory over-provisioning, as each node must independently account for workload spikes, leaving unused memory resources stranded during normal operations.

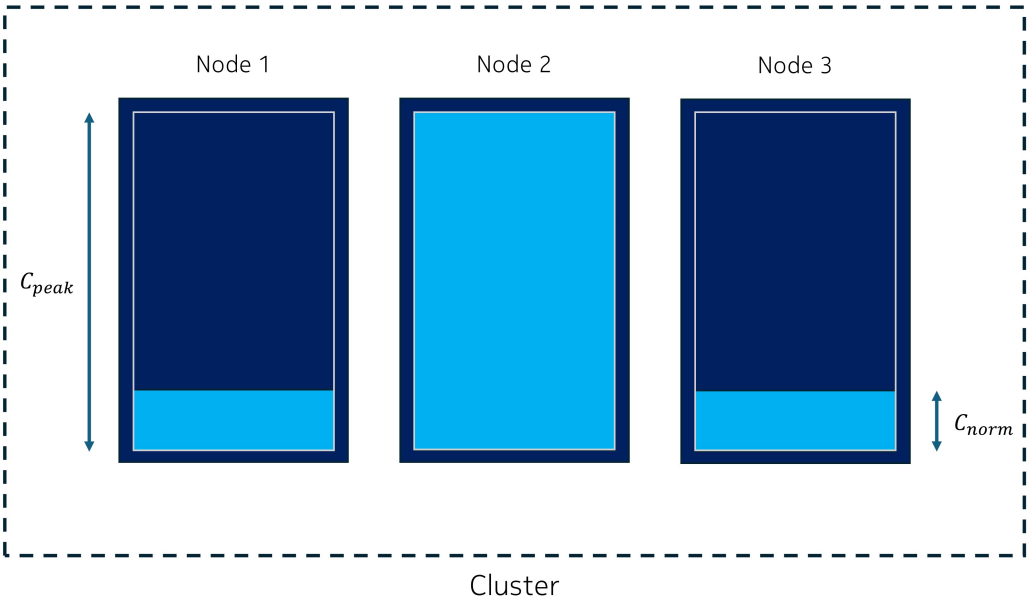


Figure 14: Capacity requirements for a 3-node cluster, displaying that in a traditional architecture, each node must be designed to handle the peak workload independently, ensuring system stability under maximum demand.

CXL-enabled memory pooling transforms this model by decoupling memory from individual nodes and enabling all three nodes to access a shared pool of memory. This approach ensures that memory resources are dynamically allocated based on actual workload demands, reducing the total capacity required to maintain performance across the cluster.

For example, consider a scenario where each node typically uses 32 GB of memory but is provisioned with 128 GB to handle peak loads. In a traditional setup, the total memory capacity across the cluster would be 384 GB (128 GB × 3 nodes). With CXL-enabled pooling, memory can be centrally managed and allocated dynamically, allowing the cluster to meet peak demands with a smaller total memory pool. If peak loads rarely occur simultaneously across all nodes, the cluster might require only 192 GB or less in the pooled memory—potentially a 50% reduction in capacity.

$$C_{peak} = 128 \text{ GB}$$

$$C_{norm} = 32 \text{ GB}$$

$$\sum C^{without}_{memory\ pooling} = C_{peak} + C_{peak} + C_{peak} = 384 \text{ GB}$$

$$\sum C^{with}_{memory\ pooling} = C_{peak} + C_{norm} + C_{norm} = 192 \text{ GB}$$

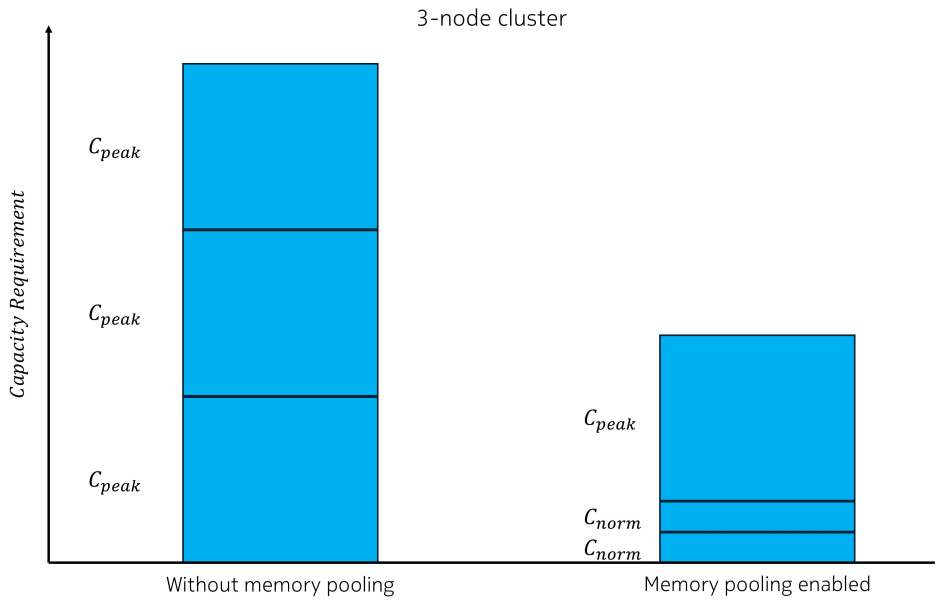


Figure 15: Capacity requirements for a 3-node cluster with and without memory pooling, illustrating the difference in total memory allocation. When memory pooling is enabled, the required memory is approximately half of that in a traditional 3-node system, demonstrating the efficiency gained through memory pooling.

This reduction in memory requirements translates to lower costs for hardware and reduced power consumption, as fewer DIMMs are needed. Additionally, pooled memory improves flexibility, as any unused capacity can be instantly reallocated to nodes experiencing temporary spikes in demand, avoiding bottlenecks and improving overall performance. By eliminating the inefficiencies of fixed memory provisioning, CXL-enabled memory pooling revolutionizes resource utilization in multi-node clusters.

5.3.2 Multi-Tenancy

Multi-tenancy is a cloud computing architecture where multiple tenants (users or organizations) share the same software or infrastructure resources, optimizing resource utilization and reducing costs. Unlike traditional single-tenant setups, where each tenant operates a separate, customized instance, multi-tenancy allows all tenants to use a shared application instance that is configurable to meet their unique needs. This setup leads to significant hardware resource sharing, as multiple tenants can share the same software and database instances, reducing the number of instances required and simplifying updates and maintenance. Multi-tenancy offers high configurability, enabling tenants to tailor their experience without needing separate instances. Data management within multi-tenancy can be achieved through methods like database isolation, schema separation, or shared tables, ensuring data security and separation. The approach can also involve virtualization or physical separation of resources, though these are costlier options. Ultimately, multi-tenancy enhances efficiency,

lowers costs, simplifies deployment, and creates opportunities for data aggregation and user behavior analysis, making it a cornerstone of modern cloud computing, especially in Software as a Service (SaaS) applications. [29].

CXL-enabled memory pooling not only optimizes resource utilization but also aligns seamlessly with multi-tenancy principles, enhancing the efficiency of shared infrastructure in multi-tenant environments. In a multi-tenant setup, multiple tenants (applications or organizations) share the same physical or virtual infrastructure, while maintaining logical isolation. CXL memory pooling introduces dynamic resource sharing, enabling the system to allocate memory flexibly based on the actual, aggregated demand of all tenants. [3]

In a traditional non-pooled setup, each tenant or node might require dedicated memory provisioned for its peak usage, much like in the described cluster scenario. This results in substantial over-provisioning and stranded memory resources, as each tenant's peak demand is unlikely to coincide with others. For example, if multiple tenants operate independently on a three-node cluster, each tenant's workload spike requires its node's memory to be fully provisioned, leading to the same inefficiencies seen in the traditional single-tenant model.

With CXL-enabled pooling, memory is decoupled from individual nodes and centrally managed. This allows tenants to share a common memory pool, with the system dynamically allocating resources based on real-time workload demands. Consider the earlier example where each node is provisioned for a 128 GB peak but typically requires only 32 GB. In a multi-tenant environment using pooled memory, tenants can collectively draw from a smaller shared pool, such as 192 GB, because not all tenants are likely to hit their peak demand simultaneously.

This model has profound implications for multi-tenancy, as it enables more efficient resource sharing and cost reduction by dynamically allocating memory from a centralized pool based on real-time tenant demands. Instead of provisioning separate, peak-capacity resources for each tenant, CXL-enabled pooling allows tenants to share a smaller total memory pool, reducing hardware costs and power consumption while maintaining performance. It also enhances scalability, as the shared pool can accommodate growing workloads or additional tenants without requiring dedicated resources. Logical isolation ensures data security, aligning with multi-tenancy principles, while centralized management simplifies maintenance and updates, benefiting all tenants simultaneously. By addressing inefficiencies and enabling flexible resource allocation, this model significantly enhances the cost-effectiveness and scalability of multi-tenant systems. [30],[3].

6 Conclusions

This thesis has comprehensively examined the potential of CXL in addressing modern computing challenges in resource utilization, scalability, and performance optimization. By leveraging CXL's capabilities, such as dynamic memory pooling, high-speed interconnects, and enhanced coherency, significant improvements in system efficiency, scalability, and cost-effectiveness have been demonstrated.

A key contribution of this work is the development of cost and performance models that provide a structured framework to evaluate the impact of CXL on system design and operation. These models quantify key metrics such as memory bandwidth utilization, CPU overhead reduction, and overall system throughput. Importantly, the models also incorporate cost dimensions, such as hardware provisioning, power consumption, and resource efficiency, making them versatile tools for analyzing the economic implications of CXL adoption.

The cost models are designed to be adaptable, allowing their application to new interconnect technologies alongside CXL. By focusing on essential metrics such as memory bandwidth, CPU processing capabilities, and network throughput, these models can evaluate both existing and upcoming interconnect technologies. This adaptability provides a robust framework for comparing interconnect solutions, ensuring their scalability and efficiency.

The research highlights several significant insights about the potential of CXL in modern computing systems. CXL-enabled memory pooling has proven to be highly effective in improving resource utilization and scalability by separating memory from individual compute nodes. This reduces stranded resources and over-provisioning, leading to significant cost savings; examples demonstrate that memory provisioning can be reduced by up to 50% in typical multi-node cluster setups. In performance optimization, the throughput models developed in this thesis reveal that CXL enhances system performance by increasing memory bandwidth and reducing CPU overhead, particularly in high-performance and large-scale workloads. The integration of cost efficiency into these analyses further validates that CXL's ability to dynamically allocate resources translates into lower hardware costs, reduced power consumption, and improved overall system economics.

Additionally, CXL has shown compatibility with multi-tenancy principles, enabling flexible resource allocation while maintaining logical isolation. This improves the efficiency of shared infrastructures and reduces operational costs, making it an important component for cloud computing architectures. Furthermore, the backward compatibility and continuous advancements of CXL, especially in CXL 3.0, highlight its scalability for large distributed systems. The introduction of features such as multi-level switching and fabric capabilities demonstrates its ability to support diverse applications, including artificial intelligence and data-intensive workloads, solidifying its role as a key enabler of future computing environments.

7 Contributions and Future Directions

This thesis contributes by not only showcasing the advantages of CXL but also providing a framework for evaluating any upcoming interconnect technology. The cost and performance models developed serve as a fundamental framework for assessing the efficiency and economic viability of emerging solutions in memory and compute architectures.

Future research can use this work by applying these models to evaluate new interconnect technologies, such as Gen-Z, NVLink. Furthermore, real-world validation through case studies in hybrid cloud, edge computing, or artificial intelligence workloads can further refine these models. Investigating integration with non-volatile memory, disaggregated storage, and advanced coherence protocols will also deepen our understanding of next-generation system architectures. CXL represents a new shift in resource management and interconnect technology. Its ability to optimize resource allocation, improve system performance, and reduce costs makes it a cornerstone of modern computing. The cost and performance models developed in this thesis provide a scalable and flexible framework to evaluate not only CXL but also other emerging interconnects, ensuring that the findings remain relevant as technology evolves. This adaptability ensures that the models will continue to guide the design and optimization of future high-performance and distributed computing systems.

References

[1] Santhosh Nagaraj Nag. Technical analysis of PCIe to PCIe 6: A next-generation interface evolution. *World Journal of Engineering and Technology*, 11(3):504–525, 2023.

[2] Debendra Das Sharma, Robert Blankenship, and Daniel S Berger. An introduction to the compute express link (CXL) interconnect. *arXiv preprint arXiv:2306.11227*, 2023.

[3] Compute Express Link Consortium. Compute Express Link (CXL) Specification, Revision 3.1 . https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-3.1-Specification.pdf, 2023. Accessed: 2025-01-04.

[4] DD Sharma. Compute Express Link 3.0 . https://www.design-reuse.com/articles/52865/compute-express-link-3-0.html, 2023. Accessed: 2025-01-04.

[5] Philip Levis, Kun Lin, and Amy Tai. A case against CXL memory pooling. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, pages 18–24, 2023.

[6] Alberto Lerner and Gustavo Alonso. CXL and the return of scale-up database engines. *arXiv preprint arXiv:2401.01150*, 2024.

[7] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. Pond: CXL-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 574–587, 2023.

[8] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, et al. Demystifying CXL memory with genuine CXL-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–121, 2023.

[9] Christoph Lameter. Numa (non-uniform memory access): An overview: Numa becomes more common because memory controllers get close to execution units on microprocessors. *Queue*, 11(7):40–51, 2013.

[10] Doaa Bliedy, Sherif Mazen, and Ehab Ezzat. Cost model for establishing a data center. *International Journal of Computer Science, Engineering and Applications*, 8:11–30, 2018.

[11] Jonathan Koomey, Pitt Turner, John Stanley, and Bruce Taylor. A simple model for determining true total cost of ownership for data centers. *Uptime Institute*, 2007.

[12] Katsantonis Konstantinos, Mitropoulou Persefoni, Filiopoulou Evangelia, Michalakelis Christos, and Nikolaidou Mara. Cloud computing and economic growth. In *Proceedings of the 19th Panhellenic Conference on Informatics*, pages 209–214, 2015.

[13] Alan Weissberger. Canalys gartner: Ai investments drive growth in cloud infrastructure spending. Technical report, IEEE ComSoc Technology Blog, 2024.

[14] Svystus Oleksii. How to Achieve Energy Efficiency and Sustainability in Cloud-Based Solutions. https://tech-stack.com/blog/how-to-achieve-energy-efficiency-and-sustainability-in-cloud-based-solutions/, 2023. Accessed: 2025-01-65.

[15] WSP. Cloud Computing Study for Microsoft. https://www.wsp.com/en-gl/insights/microsoft-cloud-computing-environmental-benefit-study, 2023. Accessed: 2025-01-65.

[16] Yupeng Tang, Ping Zhou, Wenhui Zhang, Henry Hu, Qirui Yang, Hao Xiang, Tongping Liu, Jiaxin Shan, Ruoyun Huang, Cheng Zhao, et al. Exploring performance and cost optimization with ASIC-based CXL memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 818–833, 2024.

[17] Suyash Mahar, Ehsan Hajyjasini, Seungjin Lee, Zifeng Zhang, Mingyao Shen, and Steven Swanson. Telepathic datacenters: Fast RPCs using shared CXL memory. *arXiv preprint arXiv:2408.11325*, 2024.

[18] Nokia. Cloud RAN . https://www.nokia.com/networks/mobile-networks/airscale-radio-access/cloud-ran/.html, 2025. Accessed: 2025-01-04.

[19] Alan Sill. The design and architecture of microservices. *IEEE Cloud Computing*, 3(5):76–80, 2016.

[20] Dharmendra Shadija, Mo Rezai, and Richard Hill. Towards an understanding of microservices. In *2017 23rd International Conference on Automation and Computing (ICAC)*, pages 1–6. IEEE, 2017.

[21] Kun Suo, Yong Zhao, Wei Chen, and Jia Rao. An analysis and empirical study of container networks. In *IEEE INFOCOM 2018-IEEE Conference On Computer Communications*, pages 189–197. IEEE, 2018.

[22] Shixiong Qi, Sameer G Kulkarni, and KK Ramakrishnan. Assessing container network interface plugins: Functionality, performance, and scalability. *IEEE Transactions on Network and Service Management*, 18(1):656–671, 2020.

[23] Kubernetes. Kubernetes: Services, Load Balancing, and Networking . https://kubernetes.io/docs/concepts/services-networking/#the-kubernetes-network-model, 2024. Accessed: 2024-10-08.

[24] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 65–77, 2021.

[25] Dominik Scholz. A look at Intel's dataplane development kit. *Network Architectures and Services*, 115, 2014.

[26] Steen Larsen, Parthasarathy Sarangam, Ram Huggahalli, and Siddharth Kulkarni. Architectural breakdown of end-to-end latency in a TCP/IP network. *International journal of parallel programming*, 37:556–571, 2009.

[27] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. Stackmap: Low-latency networking with the OS stack and dedicated NICs. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 43–56, 2016.

[28] Donghyun Gouk, Miryeong Kwon, Hanyeoreum Bae, Sangwon Lee, and Myoungsoo Jung. Memory pooling with CXL. *IEEE Micro*, 43(2):48–57, 2023.

[29] Isaac Odun-Ayo, Sanjay Misra, Olusola Abayomi-Alli, and Olasupo Ajayi. Cloud multi-tenancy: Issues and developments. In *Companion Proceedings of the10th International Conference on utility and cloud computing*, pages 209–214, 2017.

[30] Hussain AlJahdali, Abdulaziz Albatli, Peter Garraghan, Paul Townend, Lydia Lau, and Jie Xu. Multi-tenancy in cloud computing. In *2014 IEEE 8th international symposium on service oriented system engineering*, pages 344–351. IEEE, 2014.