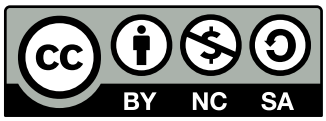
Aalto University
School of Science

Master's programme in Mathematics and Operations Research

Modelling Compute Express Link Performance in Multiprocessor Architectures

Juha Ponkkonen

Master's Thesis
2024



Aalto University
School of Science

Author Juha Ponkkonen

Title Modelling Compute Express Link Performance in Multiprocessor Architectures

Degree programme Mathematics and Operations Research

Major Systems and Operations Research

Supervisor D.Sc. (Tech.) Harri Hakula

Advisors Jari Karppinen, Bela Berde

Collaborative partner Nokia

Date 30 July 2024 **Number of pages** 59 **Language** English

Abstract

This thesis examines the performance of Compute Express Link (CXL) within multiprocessor systems, taking an in-depth look into understanding its operational mechanics and its influence on system efficiency. CXL technology, a recent innovation in high-speed interconnects, is pivotal in addressing the bandwidth and latency challenges prevalent in multiprocessor environments. The research focuses on modelling the integration of CXL across different processor architecture setups to assess its effectiveness in enhancing memory related performance and minimizing latency.

Through simulations using MATLAB and SIMULINK, this thesis models the interaction between CXL-equipped systems and standard processing units, illustrating the potential enhancements in data transfers and system throughput. Key findings indicate that CXL significantly improves the efficiency of data-intensive operations by facilitating faster communication between CPUs and peripheral devices, thereby optimizing memory coherence and access times.

The thesis also explores the scalability of CXL implementations and their potential to streamline resource allocation, which is critical for achieving higher operational efficiencies in cloud computing and enterprise data centers. By simulating various configurations and workload scenarios, this research substantiates the role of CXL in advancing the capabilities of modern computing infrastructure, proposing that it can serve as a fundamental component in the next generation of high-performance computing systems.

Keywords CXL, Compute Express Link, NUMA, memory, modelling, Simulink

Aalto-yliopisto
Perustieteiden
korkeakoulu

Tekijä Juha Ponkkonen

Työn nimi Compute Express Linkin suorituskyvyn mallintaminen moniprosessorisissa arkkitehtuureissa

Koulutusohjelma Mathematics and Operations Research

Pääaine Systems and Operations Research

Työn valvoja TkT Harri Hakula

Työn ohjaajat Jari Karppinen, Bela Berde

Yhteistyötaho Nokia

Päivämäärä 30 heinäkuu 2024 **Sivumäärä** 59 **Kieli** englanti

Tiivistelmä

Tämä diplomityö tutkii Compute Express Linkin (CXL) suorituskykyä moniprosessorijärjestelmissä, keskittyen perusteellisesti sen toimintamekanismeihin ja vaikutukseen järjestelmän tehokkuudessa. CXL-teknologia, joka on ajankohtainen innovaatio nopeiden liitäntöjen kentällä, on keskeisessä asemassa ratkaisemassa moniprosessorijärjestelmissä yleisiä kaista- ja viivehaasteita. Tutkimus keskittyy mallintamaan CXL:n integrointia erilaisiin prosessoriarkkitehtuureihin sen tehokkuuden arvioimiseksi muistiin liittyvän suorituskyvyn parantamisessa ja viiveiden minimoimisessa.

MATLAB- ja SIMULINK-simulaatioiden avulla työssä mallinnetaan CXL-varustettujen järjestelmien ja tavallisten prosessointiyksiköiden vuorovaikutusta, havainnollistaen potentiaalisia parannuksia tietojen siirrossa ja järjestelmän suoritustehossa. Keskeiset havainnot osoittavat, että CXL parantaa merkittävästi tietointensiivisten operaatioiden tehokkuutta nopeuttamalla viestintää suoritin- ja oheislaitteiden välillä, mikä optimoi muistikoherenssia ja -käsittelyaikoja.

Diplomityössä tarkastellaan myös CXL-toteutusten skaalautuvuutta ja niiden potentiaalia resurssien jakamisen tehostamisessa, mikä on kriittistä korkeamman operatiivisen tehokkuuden saavuttamiseksi pilvipalveluissa ja datakeskuksissa. Simuloimalla erilaisia konfiguraatioita ja kuormitusskenaarioita tutkimus tukee näkemystä, että CXL voi toimia keskeisenä osana seuraavan sukupolven suorituskykyisiä tietojenkäsittelyjärjestelmiä.

Avainsanat CXL, Compute Express Link, NUMA, memory, modelling, Simulink

Contents

Abbreviations

CXL	Compute Express Link
GPU	Graphics Processing Unit
NUMA	Non-uniform memory access
CMP	chip multiprocessor
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
VM	Virtual Machine
MESI	Modified, Exclusive, Shared, Invalid coherence protocol
PCI Express	Peripheral Component Interconnect Express
LLC	Last layer cache
FPGA	Field-programmable gate array
NIC	Network Interface Controller
DIMM	Dual In-line Memory Module
PBR	Port Based Routing
SNC	sub-NUMA clustering
FIFO	First In, First Out

1 Introduction

In the rapidly evolving landscape of computing technology, the demands for higher data throughput and reduced latency in data centers and high-performance computing environments are continually escalating. As applications grow increasingly complex and data-intensive, traditional multiprocessor systems frequently encounter significant performance bottlenecks. Among these, main challenges are memory bandwidth limitations, interconnect inefficiencies and cache coherency, significantly impacting system performance and efficiency. Addressing these challenges is crucial for the advancement of computing technologies.

Compute Express Link (CXL) emerges as a pivotal innovation in this context. Introduced as a high-speed interconnect aimed at improving the performance of servers by allowing high-bandwidth, low-latency communication between CPUs and devices such as memory expanders and GPUs, CXL represents a significant shift in data center architecture. This thesis focuses on the role of CXL in mitigating traditional bottlenecks in multiprocessor systems, particularly examining its effectiveness in enhancing memory accessibility, reducing latency, and increasing overall system throughput. As the adoption of Compute Express Link gains momentum across various sectors of technology, particularly in data centers and high-performance computing, it becomes imperative to thoroughly understand and predict its impacts on system performance. CXL introduces a novel approach to handling data coherency and memory sharing across high-speed interconnects, which changes the dynamics of processor-memory interactions and communication. Given the complex nature of these interactions and the novel integration of hardware components required by CXL, traditional analytical methods or simple experimental procedures may not suffice to capture the full spectrum of its performance implications.

Simulation and modelling play a pivotal role in translating theoretical specifications of CXL into practical, actionable insights. By creating detailed models of CXL-enabled systems, researchers and engineers can estimate the performance impact and inefficiencies in the architecture before moving into real-world deployments. Modelling allows testing various scenarios and what-if analyses to understand how different setups and workloads will perform with CXL. Results of modelling can provide design and deployment strategies, helping businesses and technology providers maximize their investments in CXL technology. Moreover, modelling provides a foundation for academic and industry researchers to collaboratively explore the potential of CXL, fostering innovation and leading to advancements in computing technology that might not be possible through experimental methodologies alone due to cost, scalability, or feasibility constraints.

There are multiple challenges in estimating the benefits of CXL. CXL's operation involves intricate interdependencies between the CPU, memory modules, and I/O devices. These relationships are not only complex but are also significantly different from those in traditional architectures. Modelling allows for detailed exploration of

these interdependencies in a controlled environment, enabling a better understanding of how changes in one component affect others. The benefits of CXL can vary widely depending on the specific system configurations, such as the type of CPUs used, the number of devices connected via CXL, and the nature of the workload. Through modelling, various configurations can be simulated and analyzed to understand the best use cases and configurations for CXL. In addition the rapid pace of development in CXL technologies and standards means that new features and capabilities are continually being added. Modelling these changes before they are fully implemented in hardware can provide valuable insights into their potential impacts, guiding future developments and helping manufacturers make informed decisions. Also as systems scale, the impact of architectural changes can become more pronounced. Modelling helps predict how CXL might perform as systems grows in complexity, ensuring that the technology is scalable and future-proof.

The primary aim of this thesis is to model the performance of CXL in a variety of multiprocessor architectures, providing a comprehensive analysis of its potential to benefits for the system. Through detailed simulation and performance evaluation, this thesis seeks to quantify the improvements CXL offers and explore the optimal configurations and scenarios where CXL can be most beneficial. This exploration is critical, not only for validating the theoretical benefits of CXL but also for identifying practical challenges and considerations in its implementation. By delving into the capabilities and impact of Compute Express Link, this thesis contributes to a deeper understanding of modern solutions to age-old problems in computer architecture, paving the way for more efficient, powerful, and scalable multiprocessor systems.

In this thesis two different type of discrete event based models are created with Simulink modelling software. The different models are dual processor system and the extended model that is a four processor system. Both models include baseline version and CXL version that are compared to each other. Baseline models are tuned with actual parameters from AMD processor system study. Models are created with subsystem components connected to each other. Task system releases different types of tasks into the system, and performance is examined through various metrics related to these tasks. Multiple different scenarios based on different cache hit rates, CXL device utilization and other parameters are studied with the models.

1.1 Main results and implementation contributions

The CXL models presented performance improvement over baseline versions for both dual and extended models. Improvement was present with all different task types examined. CXL device improved systems performance in poorly optimized systems, where cache hit ratios are low. Performance was also boosted in highly optimized systems with high cache hit rates.

The created models are parameterized and have modular structure, which allows to examine performance in complex processor systems. The models can be further

developed to for example examine even more larger processor systems, or study performance with specific memory placement algorithms.

1.2 Structure of the thesis

This thesis is structured as follows: Literature Review, which consists of a presentation of a Chip Multiprocessors and NUMA architecture followed by a review of the existing research on multiprocessor system bottlenecks. Following this, CXL technology is explained through its protocols and technical information. Continuing to Modelling and Results, which includes description of the modelling techniques and simulation tools used to evaluate the performance of CXL in multiprocessor systems. Presentation and discussion of the results from the performance simulations, providing insights into how CXL impacts the system. Analysis of the implications of the findings, considering both the enhancements and the limitations observed in the CXL implementation. Final section consists of conclusion and future work. Summarization of the thesis' contributions to the field of computing technology, along with recommendations for future research directions based on the findings.

2 Literature review

In this chapter Chip Multiprocessors and NUMA architecture are introduced as context for modelling. At the end of the chapter bottlenecks of multiprocessor systems are discussed. These bottlenecks are the primary problems that CXL technology aims to fix.

2.1 Chip Multiprocessors

Chip Multiprocessors (CMPs) represent a notable evolution in processor technology and architecture. CMP is a single chip consisting of group of uniprocessors. Design motivation is enhanced computational efficiency and performance, which is gained with multiple cores. From architectural perspective CMPs consists of multiple processor cores that share a single integrated circuit. These cores usually share resources such as caches and memory controller, while independent execution units are maintained for each core. This design makes it possible to execute several instruction streams concurrently with the architecture. This is a notable performance gain compared to the single-core processors. In Chip multiprocessors cores have methods of communication between each other. This is established through shared caches and interconnects. Important aspect for the architecture is for all of the cores to have the correct view of the shared memory, which is ensured by mechanisms for cache coherency and memory consistency [1].

Basis of CMPs working mechanism comes from parallel processing. Each core operates independently and can run separate threads or parts of a program simultaneously. Parallelism excels best with applications designed to take advantage of multi-threading and multi-tasking environments, providing improvements in processing speed and efficiency [1]. Ability to execute multiple concurrent threads significantly boosts system performance. However this efficiency introduces complexities in memory management, specially when the number of cores increase. Critical component for improving efficiency of CMPs is interconnect technology. Interconnect in CMP systems can be concluded to be a major component in memory hierarchy and overall performance. Main bottleneck in terms of interconnect performance comes from latency [2]. Considering the scaling of core counts in CMPs, the traditional assumption of equal access times becomes less effective, which leads to a importance of advanced memory access architectures.

2.2 Non-Uniform Memory Access Architecture

Non-Uniform Memory Access (NUMA) is a suggested solution to the scaling challenges related to multiprocessor computer systems. In the thesis, models are based on NUMA architecture. NUMA architectures provide a modular approach to memory access. Each processor or group of processors is attached to its own memory, forming a 'node'. Key aspect in this system is fast access to local memory that is within the same node. Accessing other node's memory, the remote memory, is slower. This design inherently reduces the contention for memory access and aims to scale up processor count without significant performance degradation. NUMA system typically consists of several nodes. Each of the nodes comprises of one or more processors (CPUs) with their own local memory and a memory controller. The memory within the node is local to the CPU's inside that node, and are accessible much faster compared to other nodes. Connections inside a single node is operated through an interconnect network. The network's design is general point considering performance of the overall system. Topology choice affects many factors such as bandwidth, latency and fault tolerance. Coherency in NUMA systems is handled with cache protocols such as MESI (Modified, Exclusive, Shared, Invalid). Coherency protocols in NUMA systems are important to maintain most recent data for processors, even if it has been modified by another processor on a different node [3].

Cache system plays a crucial role specially in modern NUMA multi-core processors, which typically feature a three-tier cache structure including a shared Last Level Caches (LLC). Specifically, L1 and L2 caches are dedicated to individual cores, while the L3 cache (LLC) is shared across all cores within a single NUMA node. As a result, L1 and L2 caches are faster to data-access than the L3, that is in turn faster for retrieving data from the main memory. The capacity of each layer varies and is a important factor in system performance. A cache with limited capacity might frequently evict working sets, increasing reliance on slower memory access. At the top of the hierarchy is the NUMA node. The programs access NUMA node via integrated memory controller that is owned by the node. Memory latency is defined by which core accesses which memory. NUMA nodes are interlinked by connection pathways and to access memory outside the local node (remote memory), interconnects must be navigated, potentially causing higher latency and causing congestion at the memory controllers. All cores within the same NUMA node usually share resources such as the memory controller, LLC and interconnect links. NUMA architecture necessitates careful consideration of cache-line placement strategies to optimize overall system performance and reduce coherence traffic [3].

Since these systems have difference in access times for local and remote memory, operating system plays a critical role in memory allocation, process scheduling and balancing memory access across the system to minimise the performance impact on memory access. Advanced operating systems include NUMA-aware schedulers. These schedulers keep track of memory usage patters and try to optimise memory accesses by allocating processes to nodes where the required data is already present or

nearby.

Usually NUMA systems performance is evaluated by specific benchmarks focusing on memory latency, bandwidth and processor interconnects. Remote-to-local access ratio, cache hit/miss rates and note level memory utilisation are commonly analysed metrics [4, 5]. Optimizing NUMA can be done either on software- or hardware-level. Software level techniques involve setting process/thread affinity and explicit memory allocation policies. Hardware-level approaches focus on improving inter-node communication channels to increase bandwidth and reduce latency.

Considering NUMA performance one of the important aspects is data locality. Computations should ideally occur close to where the data resides so access speeds are maximized and latency is minimized. This phenomenon affects directly to performance as local memory accesses are faster than remote ones. Other influential factor is memory access patterns. Memory access patterns are able to enhance performance of NUMA system, if local access is favoured. Local accesses reduce the reliance on slower interconnects used for accessing remote memory. Efficient shared resources management affects NUMA system performance. When multiple processors or cores try to access simultaneously shared resources, such as caches or memory controllers, performance degradation can occur. Other important factors related to performance are scheduling algorithms for NUMA systems. Algorithm for orchestrating memory that takes account for data locality and cache contention is important for achieving high performance. Other than NUMA optimizations, interconnect efficiency is crucial for NUMA systems, since data transfers between processors and memory banks must be handled [6].

2.3 Bottlenecks of Multiprocessor systems

In this chapter the bottlenecks of multiprocessor systems that are relevant to the CXL technology are discussed. Performance-memory related performance gap, interconnect bottlenecks and scalability are the chosen bottlenecks for further examination. All of these bottlenecks can be seen as a motivation for the need of CXL technology.

2.3.1 Performance-memory performance gap

One of the main bottlenecks in the multiprocessor systems lies in the bandwidth. Reasoning for this comes from well known Von-Neumann bottleneck problem. Von-Neumann presented computer architecture that is based on Central Processing Unit (CPU), which is formed by Control Unit and the Arithmetic Logic Unit (ALU). CPU operates with an input/output (I/O) subsystem and memory executing computer programs which are a stream of instructions. Computer programs performs I/O operations and process the data stored in memory. Key idea of the architecture is that memory content is defined entirely by how it is interpreted, meaning data and instructions are stored in the memory system the same way. Von Neumann architecture led to knowledge of Memory Access bottleneck which is still relevant to date when examining modern multiprocessor systems. Separation of CPU and memory has led to performance limitations in foundational level. CPU development has increased potential performance at much faster rate than performance in memory has evolved [7].

The overall system performance is limited by system's slowest component, meaning that this developmental imbalance between CPUs and memory presents a bottleneck. Primary reasons for this bottleneck is the requirement to supply the CPU with data from the memory, which leads to lower performance and higher energy. Today, the disparity between CPU speeds and memory access capabilities has become increasingly pronounced. Over recent decades, CPU speeds have approximately doubled every two years, in line with Moore's Law, whereas the improvements in memory access times and bandwidth have lagged significantly behind. This growing gap highlights a critical and intensifying bottleneck in computing system performance. This is referred sometimes as the memory wall [8]. On top of this most of the energy goes to the memory access and data transfer in modern computers compared to the processing operations in CPU [9]. DRAM technology plays a pivotal role in the architecture of main memory systems in computers. This is primary due to its cost-effectiveness and high storage density. Over the previous decades, the capacity of DRAM has consistently doubled approximately every two to three years. Resulting from this, there was a time when off-chip main memory was capable of providing the processor with data at sufficient rate. However, in contemporary settings, as processor performance enhances by roughly ten times more annually than memory latency, it now requires dozens of cycles for data to move between the processor and main memory [10].

2.3.2 Interconnect Bottlenecks

Interconnects are critical component in multiprocessor systems for providing performance-efficient data transfer between many system components. Interconnects are applied to systems between components such as CPUs, memory and input/output devices. Current state of art interconnect are for example PCI Express and Compute Express Link that is in the process [11, 12]. Interconnect performance affects system performance by bandwidth, latency and the ability to handle concurrent data transfers. High bandwidth and low latency are essential factors for achieving optimal performance, specially with the need to be able to handle large volumes of data. Recent advancements in technology and increased demands on system bandwidth have exposed significant bottlenecks in these interconnects. This bottleneck arises since the adoption of high-bandwidth access links, which increase data transfer rates notably, but matching similar advances have not emerged for interconnect technologies. This leads to interconnect congestion in host interconnects [13].

2.3.3 Scalability

Increasing demand for memory capacity and bandwidth for example in High-Performance Computing leads to need for more scaled out systems. However, increased amount of processing units and memory devices in the system is not scalable solution since it is expensive and inefficient in terms of resources. Additionally, problem of a resource imbalance rises since memory devices are dedicated to a processor in standard systems [14, 15]. Scaling the system with more memory devices presents a performance problem, since system needs to maintain coherency. Coherency protocols in scaled multiprocessor systems impact overall system performance by latency [16]. Need for coherence protocols in the scaled multiprocessor systems lead to being a bottleneck when the protocols are not optimized for maximal efficiency. Performance is affected by increased latency and also increased energy consumption [17].

3 CXL

Compute Express Link (CXL) is industry open standard dynamic multi-protocol technology. It is designed to provide support for memory devices and accelerators. Benefit of CXL is low-latency and high-bandwidth connectivity between host and processor, accelerators and memory expansion devices. This standard is running on the PCI express (PCIe) 5.0 physical layer infrastructure [18]. CXL has three protocols: CXL.io, CXL.cache and CXL.mem introduced in CXL 1.0. Three types of CXL devices are defined. Type 1 devices are devices with cache. These devices are accelerators, usually referred as SmartNIC, that use CXL.io and CXL.cache protocols. Type 1 devices have coherent cache that enables accelerator to implement unlimited number of atomic operations and to implement any chosen ordering model. Type 2 devices use all three protocols. These devices are accelerators with memory attached to the device. Type 2 performance benefits come from huge bandwidth between memory and accelerator. CXL device used in this work is Type 2. Type 3 devices use CXL.io and CXL.mem protocols and are referred as memory expanders. Benefit with Type 3 device is usage of device-attached memory as host attached memory [19].

3.1 CXL.io

CXL.io protocols base functionality handles basic device related operations such as configuration, discovery and initialization. Protocol is built on top of the PCIe architecture and utilizes non-coherent load/store semantics for general I/O operations. CXL.io uses a split-transaction approach that requests and their completions are handled asynchronously and independently. Transactions are packaged into a Transaction Layer Packet (TLP). This packet is not dependent on other transactions, allowing asynchronous transaction processing. Transactions use credit-based system, where each transaction has buffer requirement based "credits". Consuming credits in transactions helps manage the flow of data and prevents system from becoming over flooded from too many requests. Quality of service is addressed in CXL.io by incorporating two virtual channels to manage different types of traffic in the system. As example latency sensitive traffic can be separated from big load data transfers. Additionally traffic from different memory access types can be separated to these different channels [20].

3.2 CXL.cache

CXL.cache is the caching protocol in CXL technology designed to enhance capabilities of devices connected to a host system. It takes advantage of MESI (Modified, Exclusive, Shared, Invalid) coherence protocol. CXL.cache allows the host to manage all coherence activities. The CXL device does not need to directly interact with other caches, which simplifies device operations and design. The host ensures that all connected devices are in synchronization and keeps track of coherence status of data. Protocol operates with three different channels in two directions. Channels are for Requests, Responses and Data, operating. Directions are Host-to-Device and Device-to-Host. This structure optimizes performance by allowing information to flow

smoothly and independently between the host and the devices. To maintain coherency in the system CXL.cache uses Snoop Messages. A Snoop Message ensures that any changes in data are recognized by all devices that might access this data. Additionally, protocol uses Global Observation Messages. These messages indicate the coherence state of data at particular cache line. They ensure that device in the system knows the current state of the data it accesses. This is important for achieving system stability and optimal performance. CXL.cache supports devices operating with the usage of virtual addresses by using Address Translation Services (ATS) of PCIe. ATS manages virtual to physical address translations. This method makes sure that multiple virtual machines or containers can operate independently on the same hardware without data interference. Overall, the protocol enhances the efficiency and reliability of the system by managing caching and synchronization in multiple devices. CXL.cache supports quality of service by allowing separation of different traffic types [20].

3.3 CXL.mem

The CXL.mem protocol is designed to allow direct memory access by connected devices. It supports the management of Host-managed Device Memory (HDM). HDM allows the host to access and control device memory as if it were directly connected to system memory. Host-Managed Device Memory, being the key functionality of protocol, supports multiple types of memory media by translating host physical addresses into the device's specific memory addresses. CXL.mem has two communication channels, Master-to-Subordinate (M2S) and Subordinate-to-Master (S2M). M2S direction includes a Request channel and Request-with-Data (RwD) channel establishing direct memory operation requests from the host to the device. S2M direction features a Data-Response channel and Non-Data-Response channel. These channels allow devices to send data and responses back to the host. HDM-H and HDM-D (Host-only Coherent and Device-managed Coherent) are coherency protocols inside CXL.mem. HDM-H expands the host's memory without additional coherence management and HDM-D includes cache management features that let the device to manage and track the host's caching. It supports cache state management and snooping to support data integrity. CXL.mem has a feature called Bias Flip flow, where devices can change the cache state at the host. This feature ensures that device and the host are synchronized in terms of cache management and data coherency [20].

3.4 CXL devices

CXL devices are classified generally in 3 seperate groups: Type 1, Type 2 and Type 3. This classification is based on the protocols used in the devices. Type 1 Device uses CXL.io and CXL.cache protocols. Type 1 devices work as a accelerators without memory. Advantage that Type 1 device brings with CXL.cache is cache coherency. It is possible for this accelerator to implement any ordering model chosen and implement an unlimited number of atomic operations. Size of cache used in the device depends on the host's snoop filtering capacity. Main use cases for these type of devices are PGAS, NIC or NIC atomics [20].

Type 2 devices are Accelerators with memory. These are accelerators equipped with their own local memory, for example FPGAs and GPUs. These devices can directly map part of this memory to the systems cacheable memory. Due to this, type 2 devices can perform complex processing tasks efficiently. Type 2 devices apply all of the protocols. The full utilization of protocols allow the device to integrate deeply with the systems memory architecture. Example applications for the Type 2 device include machine learning, video processing and high-speed data-analysis. Type 3 devices are memory expanders. This type of device focuses on expanding the capacity and memory bandwidth of the system. They support a wide range of different memory types and memory tiers. Protocols applied to Type 3 devices include CXL.io and CXL.mem. First protocol is used for general device management and second for managing attached memory as cacheable, which enhances the overall memory resources available to the host system. Example application for Type 3 device are providing extended memory resources for data-intensive applications [20].

3.5 CXL technology

CXL defines interconnect protocols between CPUs and devices. Protocols consists of mixture of hardware and software techniques. General key component of CXL devices is PCIe serial interface. Idea behind creating CXL system is to overcome common challenges that exist in modern computing systems. One major challenge arises from coherent access to device and system memory. This happens when mixing traditional DDR-connected system memory and PCIe device memory. Historically in the architecture of these systems works in a way that CPU cache hierarchy can cache the system memory accessed through DDR, while the PCIe devices have to resort to non-coherent reads/writes when accessing the system memory. Generally this means that PCIe devices lack the capability to exploit the temporal or spatial locality benefits that are offered by the caching. Interactions of these sort of devices with the system memory are routed through the host's root complex, ensuring that PCIe consistency is maintained with the prevalent CPU caching semantics. More in depth, the following scenario is similary constrained: when the host accesses memory connected to a PCIe device, it does so non-coherently, with the responsibility of each access squarely on the shoulders of the PCIe device. This prevents mapping device memory to the cachable system address space, creating an inherent asymmetry in

access mechanisms. Non-coherent memory accesses of these types are handled with streaming I/O functions like storage or network operations, which are necessary for a liner data flow. Common model regarding accelerators is to reallocate entire data constructs for the system memory to the accelerator for dedicated operations and then back to the main memory. Software-level solutions are employed to prevent any concurrent access between the CPU and the accelerators and to ensure data integrity. This challenge presents limitation for emerging applications such as AI, ML and NICs. In these scenarios the ideal would be to allow devices to concurently access fragmented data structures alongside the CPU. This would leverage device-local caches following elimination the need for the extensive data transfers. The challenge is also related to the burgeoning domain of processing-in-memory which emphasizes computation proximity to data storage. Non-existance of standardised mechanism for PIM devices to coherently accessing data potentially stored in the CPU cache hierarchy forces developers to use intricate programming models. Such complexities increase developmental overheads [20, 19].

The second challenge comes from memory scalability conundrum. The increase in computational demands mandates a proportional rise in bandwidth and memory capacity. This exponential correlation describes well the needs of modern computing paradigms. While being widely recognised issue, there exists a mismatch between this demand and the supply capabilities of DDR memory, causing a bottleneck in memory bandwidth available per CPU. Predominant factor for this issue arises from the inherent pin-inefficiency associated with the parallel DDR interface. Solving this issue by augmenting the number of DDR channels concurrently introduces complex singnal integrity predicaments and increases system overall costs. PCIe, as seemingly ideal alternative, deliver enhanced memory bandwidth per pin efficiency. As comparison example DDR5-6400 yields 50 GB/s bandwidth at the cost of aprrox. 200 signal pins. PCie port x16 Gen5 can potentially offer bandwidth of 256 GB/s, utilising only 64 signal pins. PCIe's also offer extended reach, enabling memory components to be set greater distances from CPUs. In theory this flexibility could allow using more power than 15W of power per DIMM, resulting in improved performance. However, PCIe comes with key limitation: lack of coherency support. Inability to map device-attached memory to a coherent memory domain exists [20, 19].

One key challenge is resource stranding. Resource stranding is a common cause of inefficiencies that present-day data centers are struggling with. Generally resource is referred to be stranded when there exists an under-utilisation of a particular resource, such as memory, even as another corresponding resource, like compute, is saturated to its capacity. This results to each server needing to be over-provisioned with accelerators and memory to handle peak capacity demanding workloads. For example, if a server hosting an application that occasionally experiences spikes in memory or accelerator requirements beyond its provisioned capacity is considered, it is not possible to leverage idle memory or accelerators from another less-strained server within the same infrastructure. The result is a performance degradation marked by page misses. Oppositely, in scenarios where all computational cores of a server

are actively engaged, there often exists a surplus of memory not utilised. Resource stranding inflicts negative effects wide perspective. Addition to performance penalties, resource stranding causes negative effects to power consumption, sustainability metrics and financial overheads. These inefficiencies are common in the industry [20, 19].

The final challenge is navigating fine-grained data sharing in distributed ecosystems. Essential attribute of contemporary distributed systems is their reliance on fine-grained synchronisation. Such synchronisation is characterised by frequent yet short and latency-sensitive updates, a scenario where tasks are invariably contingent on prompt updates. Many of web-scale applications incorporate this model. In these ecosystems amount of updates related to queries often doesn't exceed 2kB, what is typical with individual search results. Many examples can be seen in distributed databases, where system leans heavily to kB-sized pages and distributed consensus mechanisms that hinge on even more granular updates. Data sharing paradigm brings forth an inherent challenge. Considering the size of updates, communication latency to standard data center networks becomes major factor determining the wait time for updates. Integration of a coherent shared-memory architecture presents a promising avenue [20, 19].

The emergence of the Compute Express Link displays a conscious effort to confront challenges presented. Since the launch of CXL's specification, three distinct generations exists of the technology. Each generation retains backward compability and introduces new aspects on the CXL protocols. CXL 1.0 is based on layering coherency and memory semantics on top of existing PCIe framework. Motivation of CXL 1.0 is to confront coherency challenge and memory scaling. It further establishes a coherent interface, paving the way for mainstream adoption of PIM systems and their associated programming paradigms. Notably, CPUs gain the ability to cache device memory, addressing the intricacies of fine-grained data sharing in the realm of heterogeneous computing. The memory connected to a CXL device can now be seamlessly mapped onto the system's cacheable memory arena, enforcing heterogeneous computation and bettering the constraints of memory bandwidth and capacity. More in-depth CXL 1.0 persists in endorsing the non-coherent producer-consumer semantics inherent to PCIe [20, 19].

With CXL 2.0 Focus shifts to overcoming resource stranding challenge by facilitating resource pooling to multiple hosts. This pooling mechanism acts as an fix to resource stranding and fragmentation, allowing dynamic reallocation of resources such as memory across distinct hosts over temporal intervals without need of system reboot. Reallocation is made possible by the introduction of CXL switches, creating a compact network interlinking hosts and memory devices. CXL 3.0 focus is on addressing resouce stranding at expansive scale through the incorporation of multi-tiered CXL switching, enabling creation of dynamically configurable systems at rack level. Simutainiously challenge arising from data sharing is addressed by sanctioning fine-grained memory sharing across the confines of host peripheries [20, 19].

3.6 CXL 2.0 Protocol Enhancements

Compute Express Link 2.0 introduces multiple enhancements over the base 1.0 version of CXL protocol. CXL 2.0 incorporates PCIe hot-plug mechanisms, that allows addition or removal of CXL resources dynamically. This is possible even when the platform is booted. Hot-plug mechanisms benefit by enabling dynamic resource management and reduction in system downtime. Single-Level switching is introduced to simplify the address decoding process for CXL.mem address regions. This enables routing and switching without requiring full decode by host or any switch. Benefits from the Single-Level switching come from possibility to create multi-host connections, which enable simple device pooling and system designs. With CXL 2.0 memory and device pooling is possible. Dynamic assignment of memory and device resources to different hosts in CXL network enhances resource utilization and flexibility. As one of the key features of 2.0 protocol Multi-Logical Device (MLD) support is presented. It enables a single physical device's memory to be segmented into multiple logical devices, which can be separately assigned to different hosts. By allowing each host to view and managing only the devices and memory segments allocated to it, management efficiency and security is improved [20].

3.7 CXL 3.0 Protocol Enhancements

CXL 3.0 protocol is the most significant evolution in CXL standard, addressing the challenges of data sharing and resource management. The version expands the 2.0 version by enhancing scalability, improving bandwidth and reducing latency. Protocol supports multi-level switching and extends protocol support for up to 4096 end devices. Devices include for example memory units, hosts and accelerators. This benefits the system significantly enhancing power efficiency and reducing total cost of ownership by allowing the creation of dynamically composable systems. CXL 3.0 introduces fabric topology and enhanced bandwidth. Fabric topology support permits multiple paths between any source and destination pair, which reduces latency and congestion while boosting the bandwidth efficiency by direct peer-to-peer access between devices [21, 20].

Improvements in the protocol are made also to shared coherent memory and message passing. Shared coherent memory is made possible across multiple hosts which allows multiple systems to share data structures and perform synchronization with low latency. Unordered I/O (UIO) and Back-Invalidate (BI) fabric support features are introduced to enable more efficient memory access patterns in distributed system. UIO allows unordered read/write operations which enhances throughput and BI supports direct memory access by devices, leading to better data coherency. One main improvement of CXL 3.0 is Port Based Routing (PBR). PBR simplifies the message routing process by using a 12-bit identifier for each port. The simplification allows more scalable routing mechanism that relies less on the physical and virtual hierarchy of the network. Simplification leads to reduced latency and overheads in hierarchical routing methods. PBR establishes a flat network topology which is

managed locally at each switch. This leads to possibility to scale the network for theoretically thousands of endpoints, since routing load is distributed across multiple switches that can independently manage their connections and traffic. PBR provides efficient multi-path routing. Messages can be dynamically routed through multiple paths in the device network, depending on networks status and traffics attributes. This capability optimizes bandwidth utilization and increases overall system reliability. Example of CXL 3.0 device pool is presented in Figure 1. Figure resembles system architecture that is possible to be created with CXL pooling [21, 20].

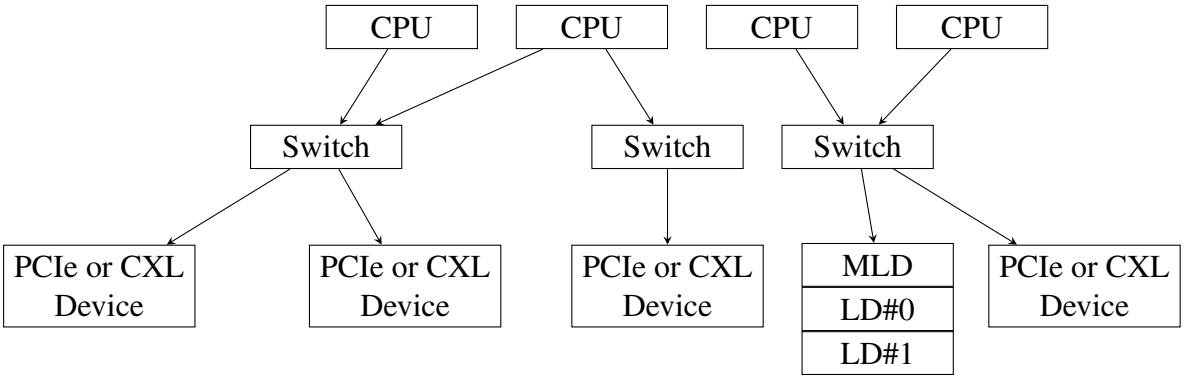


Figure 1: Pooling example by using CXL Switches [21]. Multiple CPUs can be connected to same CXL devices through switches, forming a pooled system.

3.8 CXL Microarchitecture Latencies and Bandwidth

CXL.mem and CXL.cache latencies have been estimated in multiple different system setups. For CPU to Type 3 Single Logical Device connecting to DDR memory latency is estimated to be 170ns. CPU to pooled or shared memory on a direct attach multi-headed Type 3 device for DDR memory latency consists from <100ns CPU side load to use, 50ns stack round trip, 10ns flight time and 10ns Retimer round trip, if applied. Same latency is for device access to direct connect memory on host processor across a CXL link with CXL.cache. For CPU to Type-3 memory through a CXL switch latency is estimated to be 250ns. Device caching latency to a Type-3 memory connected through CPU latency consists of <100ns, 70ns CXL switch with link flight times and latency of 80ns for CXL Type 3 device. Message to a peer CPU or Device on local Shared Memory Controller or through a CXL switch is estimated to have a latency of 220ns. For a message to a peer Device or CPU on remote Shared Memory Controller latency is estimated to be 270ns. Same is estimated for similar message through 2 CXL switches [20].

3.9 Understanding Flits in CXL Architecture

Flits, or flow control digits, are fundamental units of data used in the transport layer of the Compute Express Link (CXL) architecture. In CXL, flits are used to encapsulate data along with necessary control information which facilitates the efficient movement

of data across the CXL fabric. Flits enable the high-speed transfer of data between the CPU and devices such as memory expanders, accelerators, and other I/O devices [20].

3.9.1 CXL Flit Structures and Bandwidth Calculations

For CXL 1.0, 1.1, and 2.0, the standard flit size is configured as 68 bytes, which is composed of a 64-byte payload, a 2-byte protocol-ID, and a 2-byte CRC (cyclic redundancy check). The protocol-ID is crucial as it is processed by the logical physical layer to identify the flit type. The payload typically contains four 16-byte slots, accommodating either headers or data, referred to as G-slots. In contrast, CXL 3.0 proposes two flit sizes optimized for different needs: a 256-byte flit for capacity optimization and a 128-byte flit tailored for reduced latency [20].

3.9.2 Bandwidth and Latency Specifications

Table 1: Effective Bandwidth and Latency Specifications for Various CXL Configurations [20].

Configuration	**Effective Link Efficiency (*Link_eff*)**
CPU to Type-3 device with DDR memory (68-byte flit)	0.924 (Sync HDR on), 0.939 (Sync HDR off)
CPU to pooled/shared memory on MLD Type-3 device via SMC (256-byte flit)	0.938 (15 of 16 slots used for data)
CXL.io overhead (DLLP estimated at 2%)	0.906 (Sync HDR on), 0.92 (Sync HDR off)

In Table 1 the link efficiency (*Link_eff*) calculations factor in various overheads such as synchronization headers (Sync HDR), skip ordered sets (SKP), and the basic flit overhead. Notably, these efficiencies highlight how CXL accommodates the protocol overhead to ensure robust data integrity and synchronization across complex multi-device environments [20].

3.9.3 Traffic Mix and Efficiency Calculations

For varied traffic mixes in CXL.io, such as READ, WRITE, and a 50-50 mix, the bandwidth efficiency (*B_eff*) is influenced by the proportion of data payload per transaction layer packet (D) and the associated overhead:

- READ transactions use the formula:

$$B_{eff} = Link_{Eff} \times \frac{D}{D + 3 + FT\_CRC} \text{ (READ)}$$

- WRITE transactions apply:

$$B_{eff} = Link_{Eff} \times \frac{D}{D + 4 + FT\_CRC} \text{ (WRITE)}$$

- For a balanced READ and WRITE mix:

$$B_{eff} = Link_{Eff} \times \frac{D}{D + 8 + 2 \times FT\_CRC} \text{ (50-50)}$$

Each calculation takes into account the specifics of CXL protocol operations, reflecting the critical role of flit structure and overhead management in achieving high efficiency and low latency in CXL networks [20].

4 CXL Studies

In this chapter three applicable studies are presented. In all of the studies different implementation of CXL prototype is created and examined. These studies provide valuable insight on actual CXL performance. The results of the three presented studies are compared later to the results of modelling in this thesis.

4.1 CXL Study 1

The study by Sun et al. revealed multiple important aspects of the CXL enabled systems. One major aspect found is following: CXL controller design and/or memory technologies lead to true CXL memory device give a wide range of memory access latency and bandwidth values. CXL memory device can generally outperform emulated CXL devices by up to 26% in terms of latency and 3-66% in bandwidth efficiency. This enhanced performance is largely due to the architectural design of true CXL memory, which does not incorporate caches or CPU cores that modify caches, although it is recognized as a NUMA node. To operate this, the CPU uses an on-chip hardware structure designed to quickly check cache coherence for accesses to CXL memory. These distinctions are crucial as they can significantly alter the conclusions drawn from previous studies regarding the performance attributes of CXL memory and, by extension, the effectiveness of system-level proposals [22].

Additionally, the sub-NUMA clustering (SNC) mode isolates the Last Level Cache (LLC) among SNC nodes by ensuring that CPU cores within an SNC node exclusively evict their L2 cache lines to LLC slices within the same node. However, this LLC isolation is disrupted when CPU cores access CXL memory, as L2 cache lines from CXL memory may be evicted to LLC slices in any SNC node. As a result, accessing CXL memory potentially allows applications to benefit from an effectively 2–4 times larger LLC capacity compared to accessing local DDR memory. This compensates significantly for the longer latency typically associated with accessing CXL memory, particularly in applications that frequently utilize the cache. Conversely, for complex applications such as social network microservices exhibiting millisecond-scale latency, the latency increments when using CXL memory were marginal. This suggests that the inherent delay associated with CXL memory does not significantly affect the overall end-to-end latency for such applications, pointing to the application-specific sensitivity to CXL's memory latency [22].

The research highlighted importance of CXL system optimization. Just allocating half of the memory pages to CXL memory, as per default operating system policies, actually lowered throughput. This was despite the theoretical increase in total bandwidth achieved by integrating DDR and CXL memory, indicating that mere access to increased bandwidth does not automatically translate to improved performance.

4.2 CXL Study 2

In Gouk et al. study CXL system implementation is presented where system performance is enhanced by facilitating direct memory access to pooled resources via CXL.mem protocol, effectively overcoming the limitations observed in traditional RDMA-based memory pooling systems. In the study developed prototype DirectCXL is examined and important findings are found. It is revealed that DirectCXL offers a performance output approximately seven times superior to that of conventional RDMA-based systems across multiple real-world workloads. This is achieved primary by the reduction of redundant memory copies and the direct memory access provided by CXL, which substantially reduces access latency [23] .

The prototype system consists of CXL host processors, CXL switch and CXL devices. Four compute hosts are connected to four CXL devices through a CXL switch. Each of the CXL devices are prototypes built on customized CXL memory blade which uses 16-nm FPGA and eight DDR4 DRAM modules. Prototype demonstrates disaggregated memory with set of network infrastructure components to make disaggregated memory connected to the host in a scalable manner. Benefits for the prototype come from the aspect that it does not require any data copies between the remote and host memory. This leads to exposition of the true performance of disaggregated systems [23].

The findings from the DIRECTCXL study highlight CXL's potential to alleviate traditional bottlenecks in multiprocessor systems. The study affirms that CXL can significantly mitigate latency and enhance bandwidth efficiency. This capability is vital for applications that demand rapid memory access, such as large-scale data processing and real-time analytics. The efficiency of the DIRECTCXL approach in memory utilization and pooling mirrors the improved performance observed in this thesis' CXL simulations, where tasks that required extensive memory bandwidth demonstrated enhanced throughput and reduced latency.

4.3 CXL study 3

In Li et. al study memory pooling system Pond is proposed. Pond applies CXL in to the cloud systems. Pond is created based on extended memory controller (EMC) that offers CXL ports to connected devices. EMC implements the pool by multiple DDR5 channels accessed through a collection of CXL ports with PCIe 5 speeds. Memory pool is NUMA-friendly design and pool memory is visible by zNUMA node for VMs that use both local and pool memory. Pond uses machine learning based scheduler for VMs resource scheduler. Predictions for workloads are based on latency sensitivity and untouchability of memory. Prediction models are formed as an optimization problem which balances the prediction model by taking account of target rate of false positive latency intensive workloads, untouched memory, target rate of overpredictions and performance degradation margin [24].

Pond introduces a practical CXL-based memory pooling system designed to optimize DRAM utilization in cloud platforms. By analyzing production traces from Azure, the study highlights that memory stranding—where unused memory remains while CPU resources are fully allocated—can be mitigated through effective memory pooling strategies. This system not only reduces the hardware costs associated with DRAM but also enhances the overall efficiency of resource usage. One of the most significant findings of this study is that Pond can maintain performance within 1-5% of traditional same-NUMA-node memory allocations for virtual machines (VMs). This is achieved through strategic pooling across 8-16 sockets, minimizing the latency impacts commonly associated with larger pool sizes. Pond leverages the CXL interconnect standard to enable cacheable load/store accesses to pooled memory, which dramatically reduces the latency typically associated with memory accesses in disaggregated systems. This feature is pivotal in maintaining the high performance of cloud services despite the physical separation of memory resources [24].

The insights from the Pond study are particularly relevant to the focus of this thesis on CXL performance modelling in multiprocessor architectures. The ability of Pond to closely approximate same-NUMA-node performance using CXL-based memory pooling provides a compelling case study of CXL's potential to overcome traditional memory architecture limitations. It also validates the thesis hypothesis that CXL can effectively address interconnect bottlenecks and enhance memory access efficiency in complex systems. A critical aspect of Pond's success lies in its use of machine learning models to predict optimal memory allocation for VMs, ensuring efficient use of pooled resources. This approach aligns with the methodologies discussed in this thesis, where predictive modelling plays a crucial role in anticipating system performance and optimizing CXL configurations [24].

5 Modelling

In this section the modelling of the CXL performance is presented. Starting from explaining the motivation of the modelling, following an in-depth presentation of the modelling methods, components and functionalities. After the models are introduced, results are displayed and discussed.

5.1 Motivation

Goal of the model is the ability to examine performance of multiprocessor systems including CXL device based on hardware features such as component latencies, bandwidth and architecture structure. Key goal of model presented in this work is modularity and wide range of parametrization of the system. At current time actual CXL 3.0/3.1 devices do not yet exist and processor offer limited compability for these devices. CXL specification is public and estimated latencies exists, which makes it possible to study and predict CXL's system-wide effects [20]. Importance of modelling rises from the findings of the study by Sun et al. [22]. Emulated CXL performance differed in wide range compared to the actual created device. At the moment CXL is being researched and developed at a fast pace, so the need for parametrized models is evident. There is a need to be able to investigate performance impacts of new technology in efficient way.

5.2 Structure of the model

Model is created with MATLAB Simulink, Simevents package. Model is discrete-event based model that represents multiprocessor systems with different setups. Models' structure is based on implementation created in study by Brandberg and Di Natale [25]. In the study a model was created representing multicore architecture with Simevents using three different components: Task system, Cores and Memories. This structure was used as a basis for NUMA and CXL models in this work. Components are changed to represent NUMA system. Previous work focused on verification of memory access delays. Model presented in this work focuses on modelling performance of the architectures based on different system setups. Performance modelling is based on task throughput of the system. Task entities are unit of flow in the system, that include various variables for time measuring purposes. Components are built by using entity servers that process task entities based on service time parameters.

5.3 Components

Model components are created by Simevents blocks presented in Figure 2. The Entity generator block generates entities, that are discrete items of interest that are defined in a discrete-event simulation. These entities carry scalar data. In the model entities represent memory requests, defined by different task attributes. Release gate has simple functionality as gating entities. Release gate can be opened through a message

to the block. Entity queue stores entities based on order of arrival. Each element at the head of the queue departs when the downstream block is ready to accept it. Queues in the model are first-in-first out FIFO queues. The Entity Server block functions by holding onto entities for a specified duration known as the service time. While an entity is in the server, it is considered to be in the process of being served. The server has the capability to handle several entities at once and sends each one out through its output port once service is complete. However, the release of entities can be delayed if the output port is currently unavailable or blocked. Server blocks have ability to implement service time as MATLAB function. Other basic components used in the model include entity input and output switches and go-to tags to create the system. Components as nodes, processors and memory devices are created inside a subsystem block for implementation reasons. MATLAB function blocks are used for a need to apply MATLAB script based functionalities.

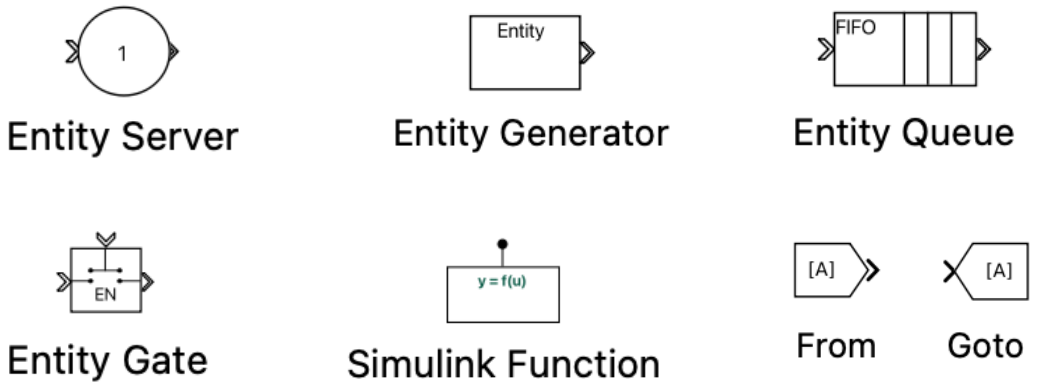


Figure 2: Functional SimEvents components used in MATLAB model.

Tasks represent memory requests in computer systems. Task system consists of three subsystems which represent different types of tasks examined in the model. Different task types are Memory intensive, Processor intensive and mix of previous two, called 50/50. Difference in these task types are created with memory latency and processing latency parameters. Along with these parameters Task entity has memory page dependency. Modelled as memory requests, task entities want to access certain randomly generated memory page. Task generation subsystem consists of entity generator followed by queue, release gate and server. Entity generator generates task entity units. Queue is used to determine initial generation amount of tasks and also the limit of active tasks in the system. Server generates memory page dependency for task entity. Release gate allows to generate new task entity when previous is completed. Task entities include different helper parameters for functionality, for example task entities include data on which node task has flown through.

NUMA nodes are created based on Brandberg's [26] core component. Node subsystem is presented in Figure 3. Nodes generally process the task entities. Functionality of node includes forwarding task to the memory devices, executing and

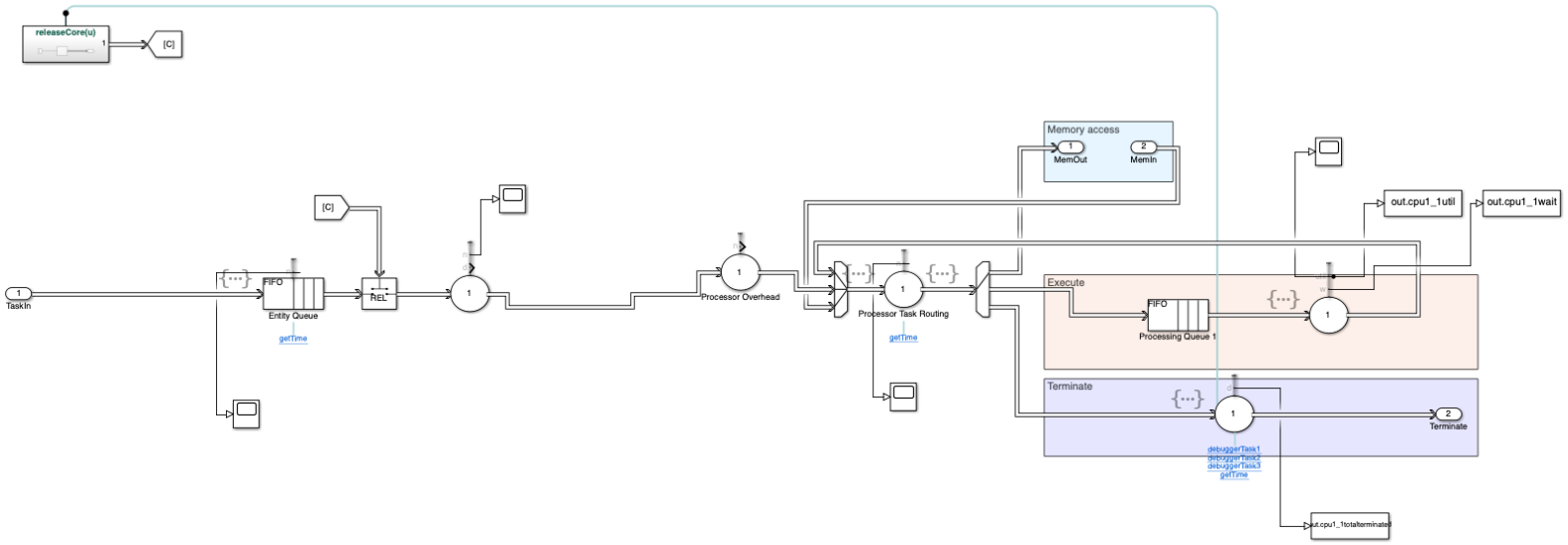


Figure 3: View of the node subsystem. Task entities flow from left to right. Right side of the node subsystem has input and exit port for task entities to move into memory subsystems, and to receive tasks back from the memory subsystem.

terminating a task. Structure of the node component includes queue, release gate, and servers. Queues in node component represent processing queue and execution queue. Servers are separated to add parametrized latency for node memory routing and node processing overhead. Release gate frees node for new tasks when task is executed. Basic functionality of node is following: Task entity enters node, where it is forwarded to the memory devices. After flowing through memory devices, task returns to node for execution. After execution task is terminated and Task system is informed about termination of task.

Memory components in the model include layer two and three caches and deep memory modelled as local memory devices. Deep memory has local memory device for each node. For example Node 1 has local memory device 0. Other nodes sees memory device 0 as shared memory, so the NUMA architecture is represented. These deep memory components consists of servers and two separate queues for local and shared memory. Memory devices omit a certain memory range for each device. Tasks are given a random memory page dependency in task generation. Based on the memory page, tasks have a designated memory device based on memory devices ranges. Deep memory devices will process tasks based on task's memory time parameter. After deep memory device, task is routed back to the node it arrived from.

Between nodes and deep memory exists layer two and three caches. Layer two caches, referred as L2, are local caches for each node. Functionality of L2 caches include hit rate, based on percentage of nodes local memory devices memory range. L2 caches consists of servers. First server handles hit-rate functionality, following different paths for cache hit and miss. If cache hit occurs in L2 cache, task entity returns to the node it arrived from. In case of cache miss, task entity goes to layer 3 cache, referred as L3 cache. L3 cache is shared cache for nodes in the processor system. L3 cache inputs task traffic from all the nodes in the processor and also socket traffic. L3 cache has hit and miss functionality. Task entities will be forwarded to the deep memory if cache miss occurs, otherwise they return back to the initial node. L3 cache consists of server and different path for hit and miss. L3 hit operations server is separated outside the L3 cache for implementation practicalities.

If task entity's memory page is not found on the processor system that entity is processed in, it is forwarded to other processor system through socket system. After L3 miss, Deep Memory operations server passes task entity to other processor, if task entity's destination exists there. When task entity arrives other processor system, it enters socket operations server. Socket operations server has functionality of adding bandwidth timing effect. After this task entities flow to the L3 cache in the current processor system.

5.4 Baseline model

Baseline model represents two AMD EPYC Rome processors as presented in Velten et al's study [27]. Study provides specific parameters for the baseline model and

examines memory hierarchies of two leading server processor architectures. The study is driven by the need to understand underlying hardware differences that do beyond superficial specifications, taking a closer look on complexity of processor architectures that have profound implications on performance. Figure 4 shows the top layer of the baseline model. Task subsystem, two processor subsystems and helper functions can be seen in the figure.

Focus on the research is to compare and analyze the memory performance characteristics of AMD's EPYC Rome and Intel's Cascade Lake SP processors. These architectures are scrutinized for memory access latencies, bandwidth capabilities and the effects of cache coherence protocols. The examination of the systems includes testing and benchmarking to measure and compare memory access patterns, latencies and bandwidth across different cache levels and main memory, providing a detailed view of how architecture performs [27].

Main interest regarding the model is the AMD EPYC Rome architecture, that the baseline model represents. In the study general concept of architecture is presented: AMD EPYC Rome processors utilize a combination of Core Complex Dies (CCD) and an I/O die, interconnected through AMD's Infinity Fabric. This design supports up to eight CCDs, with each CCD hosting up to eight Zen 2 cores, enabling up to 64 cores per processor. The I/O die facilitates communication between CCDs and external components, such as PCIe lanes and memory. Each CCD contains two Core Complexes (CCX), each with up to four Zen 2 cores. L1 and L2 caches are dedicated to each core, while a shared 16 MiB L3 cache is available for all cores within a CCX [27].

Rome processors feature up to four NUMA nodes, configurable through BIOS settings, which can impact data access latencies and bandwidth. Each core within a CCX holds a slice of the L3 cache, managing transfers and cache coherency between L2 caches within the CCX. Rome cores use three Address Generation Units (AGUs) to support up to two 256-bit loads and one 256-bit store per cycle. Hardware prefetchers are utilized for L1, L2, and L3 caches to minimize cache misses [27].

Figure 5 presents the processor subsystem. In the top left in the figure we have task distributor function, connected to four NUMA node subsystems. NUMA nodes are connected to L2 caches, which are then connected to L3 shared cache. L3 cache leads to Local Memory subsystems that can be seen in the right on the Figure 5. In the bottom left corner is the socket connection for the processor.

Velten et al's research [27] provides detailed parameters to tune created model. Socket-socket RAM-latencies as presented in Appendix, Table 6. Models parameters are identical in both socket systems, but are set separately. In the model these parameters are defined as `mem_deep_X_Y` where X represents row and Y represents column of the RAM-latencies matrix form [27]. These variables set service time to the shared memory path server in memory devices. Parameter for local memory access times for deep memory is `mem_X_local_t` where X represents number of the memory device.

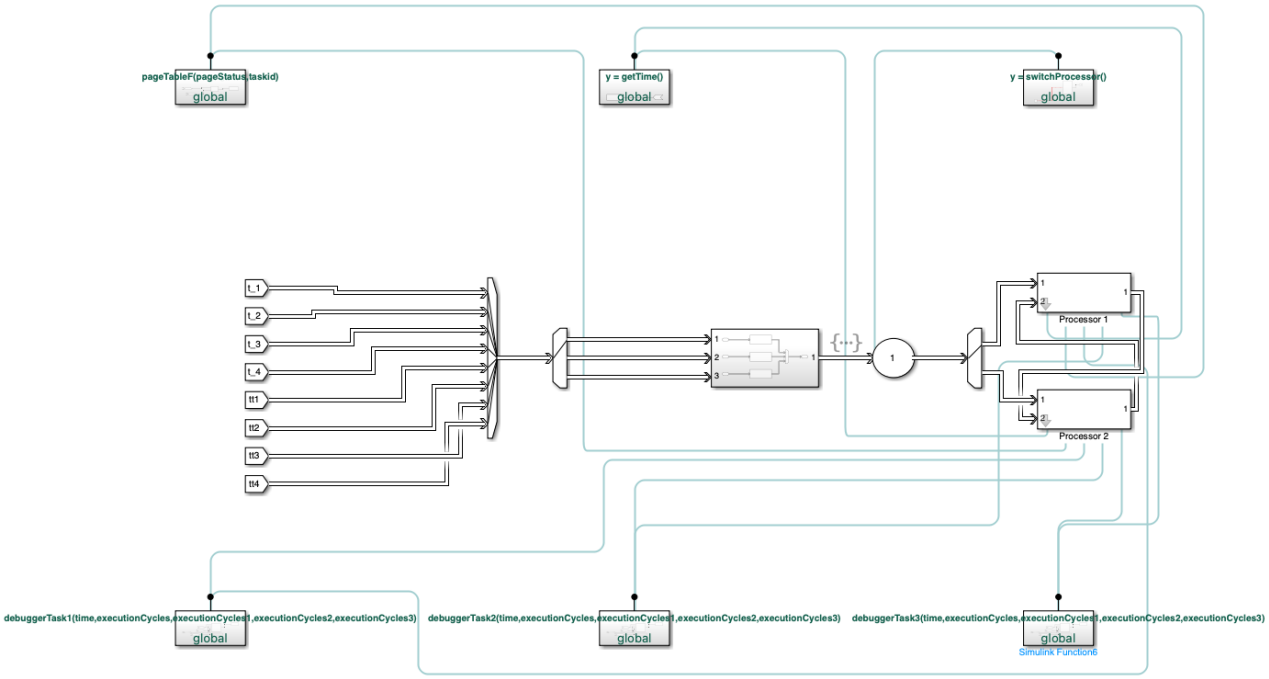


Figure 4: View of the baseline model structure. Tasks are created in the task generator subsystem, which is in the middle of the view, flowing to the processor subsystems in the right side of the model. After tasks are terminated from the simulation, termination signals are received from the left side of the model, back to the task generator subsystem. At the top and bottom of the baseline model view are helper functions of the simulation.

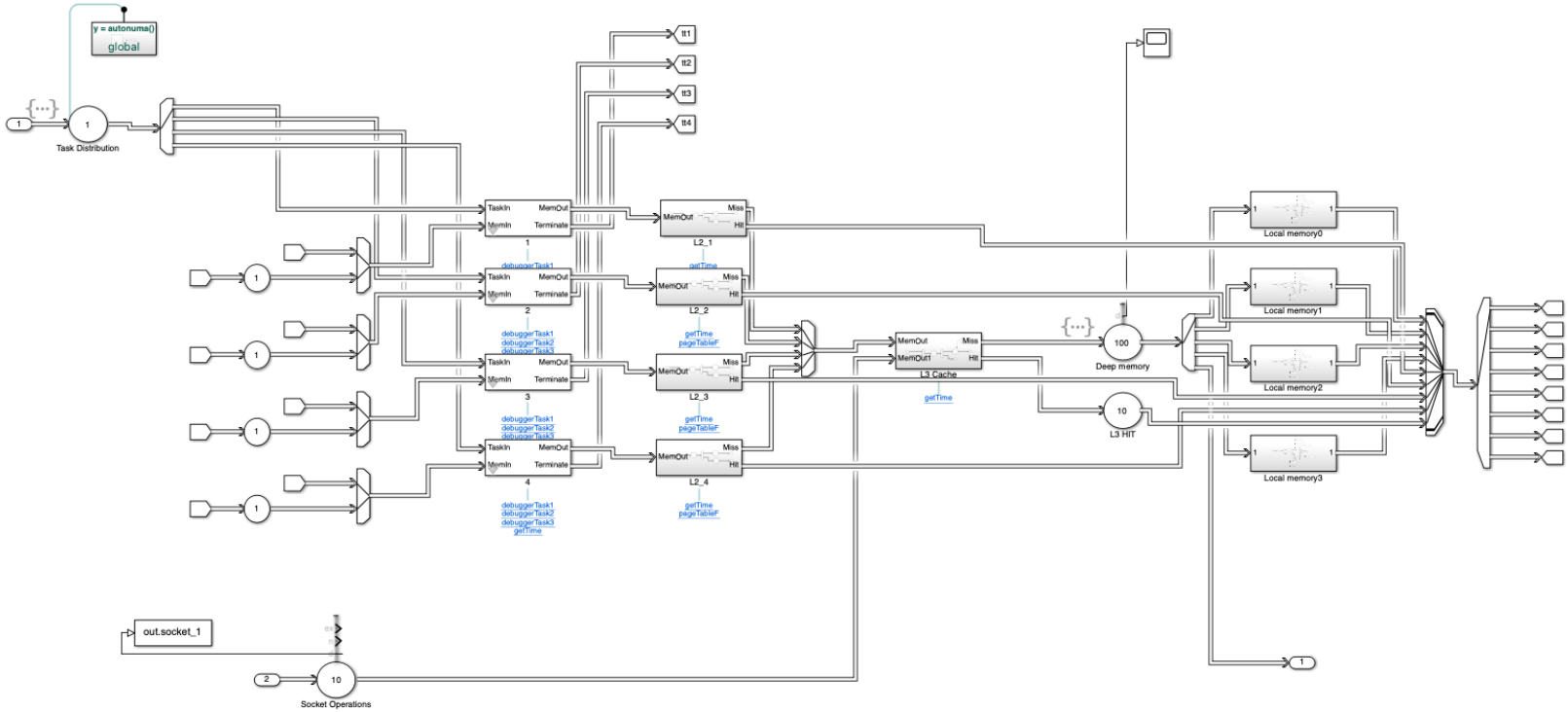


Figure 5: View of the processor subsystem in baseline model. Tasks are received from the top left corner of the subsystem view, where they are forwarded to the node subsystems. After node systems tasks flow into L2 caches and shared L3 cache, located in the middle of the view. Right side of the processor has the deep memory subsystems. Socket connection to other processors is located at bottom of the subsystem, input connection in bottom left, and output connection in bottom right.

These parameters define service time in the local memory path server in memory devices.

Caching parameters for L2 and L3 caches are the hit rate `L2_hitrate` and `L3_hitrate`. L2 hitrate takes percentile of local memory devices memory range to represent memory in cache. L3 hit rate decides hits and misses based on probability. Cache hit memory latency is defined for L2 devices per device and for L3 cache. These variables are `L2_X_hit_t`, where X represents number of L2 cache device and `L3_hit_X` where X represents cache hit memory latency, depending on the node that task entity is coming from. L2 hit time parameters set the service time inside L2 cache subsystem hit route's server. L3 hit time parameter sets service time in L3 hit server inside processor subsystem.

Node parameters include node processing overhead defined as `pro_X_overhead`, where X represents node number. Node overhead sets service time inside node subsystem. Other Processor related parameter is `pro_X_queue_exe_n` which defines nodes processing queue's capacity. This is set inside node subsystem's execution path queue block. Parameter to set "process scheduling" is defined as `task_dist_t`, which sets task distributor server's service time. Socket to Socket traffics bandwidth is defined as parameter `soc_bw`. This parameter is percentage value which calculates entity's memory time times bandwidth for implementation reason. Tuning and setting parameters feature is accomplished by creating masks for sub subsystems and linking these into processor subsystem. This way all of the components parameters can be set in one place.

As in the study, baseline model consists of two modelled representations of AMD EPYC Rome processors connected with AMD's Infinity Fabric. Each Processor has 4 NUMA nodes, 4 local L2 caches, shared L3 cache and 4 deep memory devices.
In the model process scheduling is out of the scope of the work and is represented as simplified version. Task entities alternate in order to each socket, and populate NUMA nodes in order. Since same process scheduling is used in all of the models, results can focus on hardware features. Caching mechanism is based on random generation and probability and advanced caching systems are out of the scope of this work.

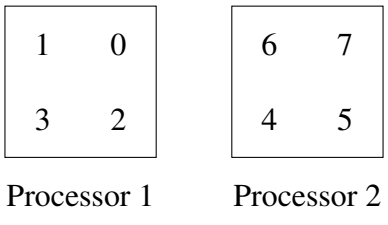
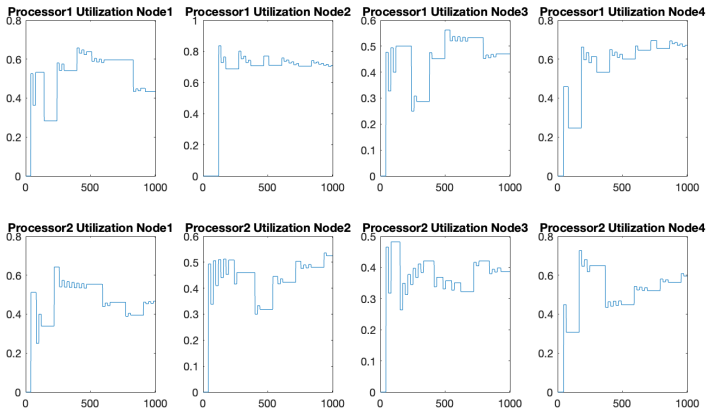


Figure 6: Illustration of node placement in the model.

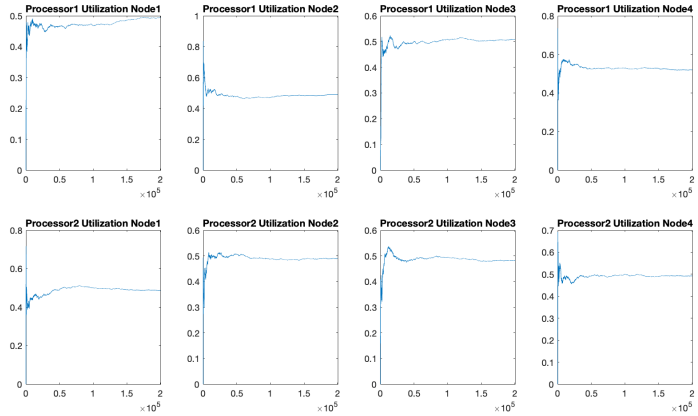
Figure 6 represents NUMA-node placements in the processors. Placement is recreation of researched system in Velten et al. work [27]. The placement affects node to node latencies based on distance. Node latency is smaller when node distance is small and vice versa. Node latencies for nodes inside same processor are set to fixed

value, that is higher than latency between processor 1 and 2, but smaller than the node latency when accessing same memory.

Model has a feature to track utilization of the nodes, and example utilizations can be seen in Figure 7. Task system is configured to have 16 active tasks in both dual and extended model. That way it is possible to compare also the baseline models to each other. Utilization levels with 16 active tasks reach 50% in average on the dual processor model.



(a) Subfigure 7a: The first 1000 simulation seconds illustrate 16 active tasks populating the nodes in the beginning of the simulation run. Utilization varies between 80% to 40% in the nodes.



(b) Subfigure 7b: Utilization view for the full simulation run. Node utilization becomes stabilized in the system at estimated 50% in all nodes. The system is well capable of handling 16 active tasks and in the full run the nodes process the tasks faster than new nodes enter. This results in node utilization to stabilize at full simulation time.

Figure 7: Result view of node utilization in dual processor baseline model for cache hit rates at (L2 = 0.5 , L3 = 0.75). Ran with 16 active tasks in the system.

5.5 Baseline results

Baseline simulations were ran on 200 000 simulation seconds. To find out effects of different tuneable variables, simulation was ran on different cache hit rates. Results presented in Table 2 are based on different cache hit rate settings. In top of the table cache hit rates are represented as (X,Y), where X is L2 cache hit rate as percentile and Y is L3 hit rate. Processor related parameters are socket-to socket traffic related to each processor, total socket traffic and amount of tasks processed. Number of different task types processed in the simulation is also shown in Table 2. Reason for multiple simulation runs with different hit rates is to show wide-range effects of CXL device when proceeding with comparison model. Key parameters to observe are total tasks and amount of different task types processed.

Table 2: Result output parameters measured with different cache hit rates by (L2 hit rate, L3 hit rate).

Parameter	**(0,0)**	**(0.25,0.25)**	**(0.5,0.5)**	**(0.25,0.75)**	**(0.5,0.75)**	**(1,1)**
Pro_1_traffic	2593	2354	1959	1225	1194	0
Pro_2_traffic	2587	2332	1965	1221	1183	0
Total_traffic	5180	4686	3924	2446	2377	0
Pro_1_tasks	5097	6263	7700	9767	9800	12841
Pro_2_tasks	5091	6269	7700	9765	9802	12833
Total_tasks	10188	12532	15400	19532	19602	25674
Task_1_n	3331	4282	5460	7438	7484	10601
Task_2_n	3108	3638	4251	4970	5003	5897
Task_3_n	3749	4612	5689	7124	7115	9176

In the results functionality of model is apparent, as when hit rates reach 100 percent, socket-to socket traffic goes to zero. Since all of the memory request's memory locations can be found in cache, requests return to node after cache hit. Second observation from the Table 2 can be made about total amount of tasks processed. Lowest amount of tasks processed occur when there is no cache hits and socket traffic is highest. Reason for previous lies in deep memory latencies and socket latencies. Having to access deep memory is much more latency heavy compared to memory roundtrip from caching devices. Socket latency is also higher compared to task having only access local memory. Amount of total tasks processed in the system increases when cache hit rates increase. Difference comparing zero cache hits to 100% hit rate is about 15000 tasks. Looking in depth on amount of different task types, it can be noticed that major increase of processed tasks happen with task type 1 and 3. Task 2 increases only slightly when total amount of tasks processed increases. This happens because type 2 task is processing time dependent, and other tasks depend on memory time. Amount of tasks processed is same for processor system 1 and 2, because work balancing distribution is set as 50% for two different processors. Simulations were run

in 200 000 simulation seconds, to have enough variance between different settings. Reasoning for this is to clearly see system wide effects of different components and in following to see how CXL device makes a difference to the system in various aspects.

5.6 CXL model

CXL model introduces type 2 CXL device to the model. Addition of CXL device to the system can be seen in the Figure 8 at the right. Device works as shared memory with caching capabilities. Device includes cache and 2 modelled memories. Cache functionality includes hit-rate based on probability. CXL devices workload is defined by a set memory range, which defines percentage of memory requests that will use CXL memory. Variable `cxl_range` defines amount of memory requests processed by the CXL device. Memory processing time is defined as parameter `ddr_t`. Main CXL parameters are `cxl_load_to_use` and `cxl_protocol_t` which first defines latency occurring from devices load to use and second defines protocol latency. Type 2 CXL device functions as fast, shared memory between processors. In current state of real life CXL implementations, device is technically seen as CPU-less NUMA node. Similarly to the model, device is initialized with some memory range it serves. In NUMA system CXL device works as a complimentary element with Infinity Fabric. Figure 9 presents a view inside CXL device subsystem.

5.7 CXL model results

The introduction of CXL shows a substantial improvement in the number of tasks completed compared to the non-CXL setup under similar cache hit scenarios, as seen in Table 3. Notably, as cache performance improves, the difference in task completion between the CXL and non-CXL setups also increases, underscoring CXL's effectiveness in optimizing task handling in a cache-efficient environment. The percentage improvement ranges from about 4.3% to 13.7% as cache efficiency varies. Best improvement occurs at zero cache hit rates, and the least improvement at (0.5, 0.75). The tasks assigned to the CXL device (cxl_n) show that CXL is actively handling a significant portion of computational or data management tasks, contributing directly to the increased task completion rates observed in the CXL-enabled system. This reflects that CXL effectively offloads certain operations from the processors, enabling them to handle other tasks more efficiently and to manage increased loads.

As for traffic, data indicates the volume of data exchanged between processors and memory. Generally, both setups show decreased traffic as cache efficiency improves, which is expected as better cache hits reduce the need for external memory accesses. The CXL model consistently shows lower traffic than the non-CXL model for similar cache hit scenarios, suggesting that CXL's enhanced memory access capabilities allow for more efficient data handling and reduced reliance on inter-processor communications.

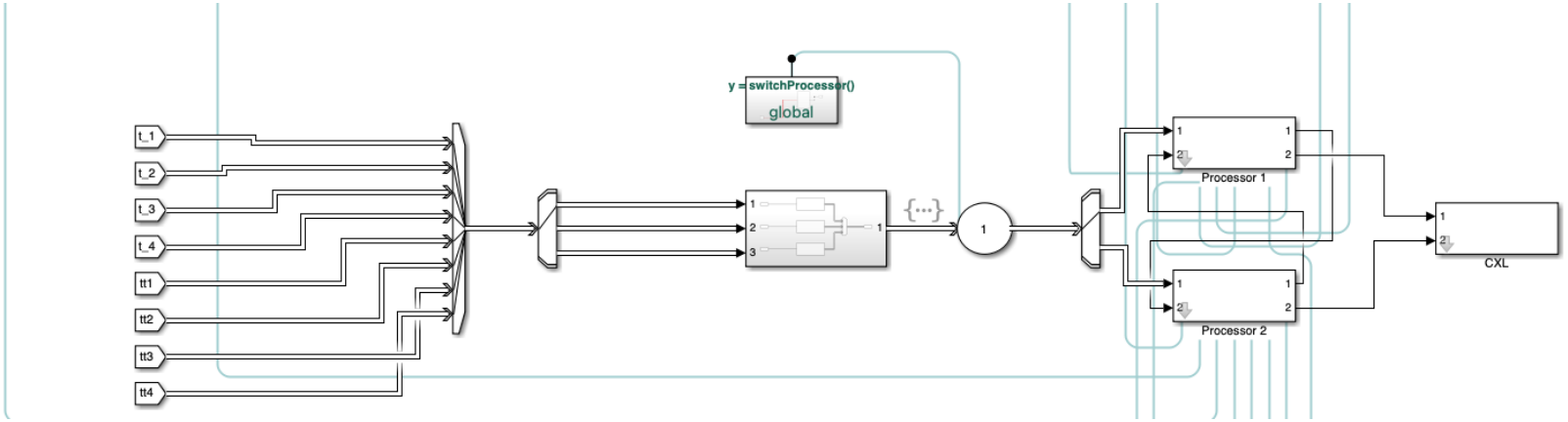


Figure 8: View of the CXL model. CXL component is located in the left side of the model, and is connected to the processor subsystems.

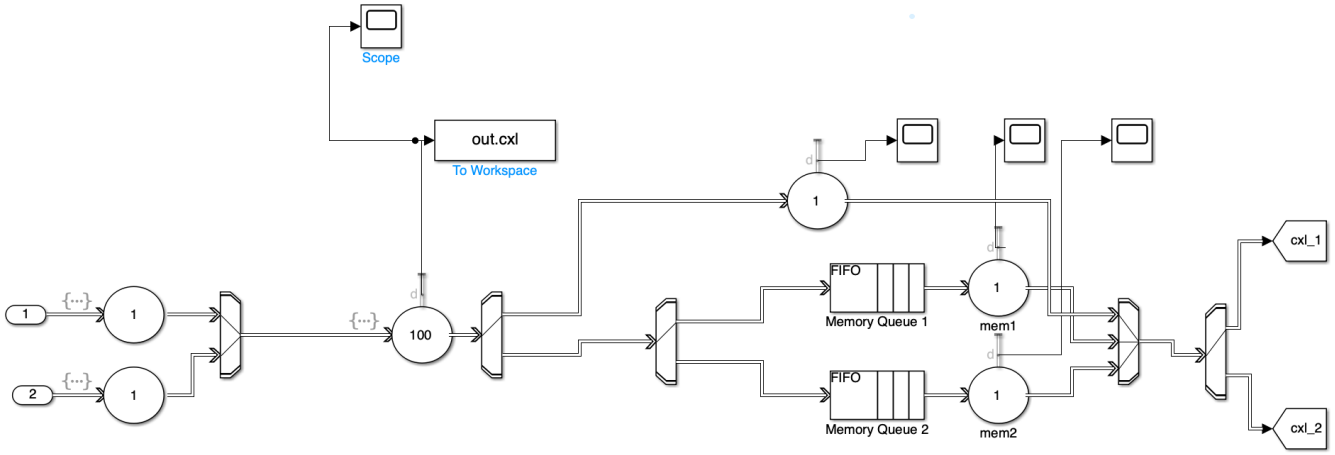


Figure 9: Representation of CXL device subsystem. Tasks flow from left to the CXL cache component. After cache, there is two deep memory components represented with memory queues and server blocks. Tasks exit the CXL device from the right side of the view.

Table 3: Result output parameters for CXL model measured with different cache hit rates by (L2 hit rate, L3 hit rate)

Parameter	**(0,0)**	**(0.25,0.25)**	**(0.5,0.5)**	**(0.25,0.75)**	**(0.5,0.75)**	**(1,1)**
Pro_1_traffic	2169	1926	1545	957	944	x
Pro_2_traffic	2147	1941	3969	963	933	x
Total_traffic	4316	3867	3091	1920	1877	x
Pro_1_tasks	5793	6900	8344	10287	10215	x
Pro_2_tasks	5794	6903	8353	10291	10223	x
Total_tasks	11587	13803	16697	20578	20438	x
Task_1_n	3914	4835	6037	7972	7945	x
Task_2_n	3457	3868	4531	5136	5081	x
Task_3_n	4216	5100	6129	7470	7412	x
cxl_n	2909	2606	2113	1286	1318	x

5.8 Extended Baseline model

Extended model consists of four processors connected to each other, as seen in Figure 10, forming a 16 NUMA-node system. Task routing system increases to 16 paths, which are seen in the left on the figure. Model has same task system as baseline model with same parameters. Figure 11 demonstrates node placement on multiprocessor system. Node latencies are based on the study by Velten et al. [27]. Latencies are mirrored from original values from Processor 1 and 2 to represent related latencies for rest of the nodes. Extended model uses 256 different latency variables. Node interaction latencies within the same processor are set to fixed point latency. These node-to node interaction latencies are inputted in the model in form of matrix.

Additions in the processor subsystem, as visible in Figure 12, come from task routing system, since there needs to be a connection to all of the system's nodes. Also socket system is extended to three way connection, so that all of the processors can communicate with each other.

5.9 Extended Baseline model results

Table 4: Result output parameters for extended baseline model measured with different cache hit rates by (L2 hit rate, L3 hit rate).

Parameter	**(0,0)**	**(0.25,0.25)**	**(0.5,0.5)**	**(0.25,0.75)**	**(0.75,0.75)**	**(1,1)**
Pro_1_traffic	3247	2863	2314	1445	1553	0
Pro_2_traffic	3162	2876	2372	1518	1543	0
Pro_3_traffic	3107	2804	2369	1552	1440	0
Pro_4_traffic	3456	3130	2552	1667	1601	0
Total_traffic	12972	11673	9607	6182	6137	0
Pro_1_tasks	4646	5543	6844	8719	8717	11898
Pro_2_tasks	4640	5545	6844	8717	8715	11896
Pro_3_tasks	4643	5545	6843	8717	8714	11896
Pro_4_tasks	4643	5545	6847	8720	8714	1896
Total_tasks	18572	22178	27378	34873	34860	47584
Task_1_n	5065	6687	9071	13102	13265	21069
Task_2_n	5383	5992	6707	7449	7432	8322
Task_3_n	8124	9499	11600	14322	14163	18193

Performance metrics of the Extended baseline model can be seen at Table 4. The analysis between both the standard and extended baseline models reveal a distinct pattern of efficiency as cache hit rates improve. Initially the extended model shows higher traffic, caused by increased inter-processor communication. However, as cache

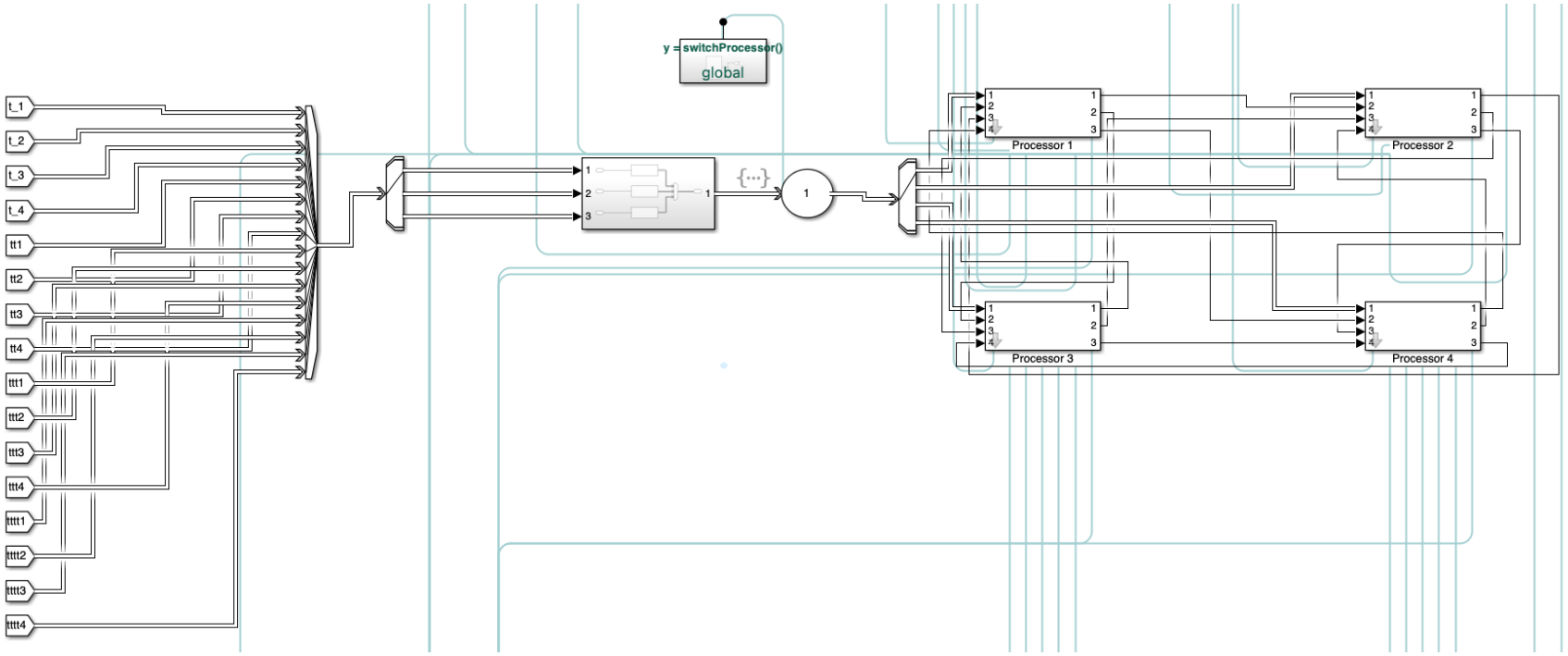


Figure 10: Representation of Extended baseline model. The view is of the outermost layer. Four processor subsystems are visible at the right side of the view. The left side shows 16 different task termination signal paths flowing into the task generator subsystem.

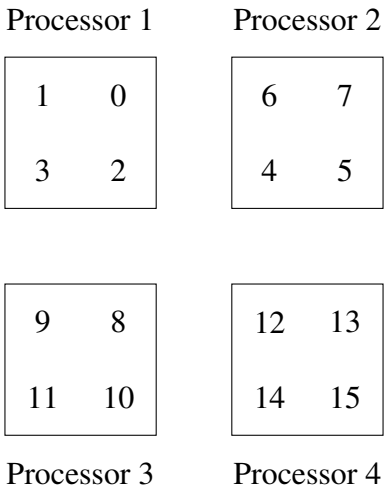


Figure 11: Node Placement Illustration for Extended Baseline model

efficiency increases both models are able to reduce their inter-processor traffic significantly, with the extended model maintaining a consistent performance enhancement over the standard model. This demonstrates efficiency for handling higher loads for four processor system.

The extended baseline model shows superior task handling capabilities in all of the scenarios, substantially outperforming the standard model in the total number of tasks completed. Total tasks completed increases over 50% in all of the examined cases with the Extended baseline model compared to the Dual processor baseline model. This performance gap widens as the cache efficiency improves, emphasising the advantages of additional processors in managing larger volumes of tasks simultaneously. The increase in task completion not only underscores the benefits of scaling processor numbers in multiprocessor systems but also illustrates the effective parallel processing that becomes possible with more computational resources. Examining the different task types the extended model's performance advantage is apparent. It handles all types of tasks with greater efficiency. The ability to maintain high performance across various task types is key representation of a well-rounded multiprocessor system capable of adapting to different computational demands.

Both models are created to represent multiprocessor architectures that have performance properties of modern high performance systems. Both of the models show good scaling properties as they reduce traffic and increase task completions with higher cache efficiencies. The extended model with its additional processor outperforms the standard baseline model in terms of examined performance metrics, demonstrating near-linear scalability in performance. This indicated that the modelling purpose of extended baseline model is fulfilled. The architecture of the model demonstrates increased processing power maintaining high efficiency under expanded operational demands.

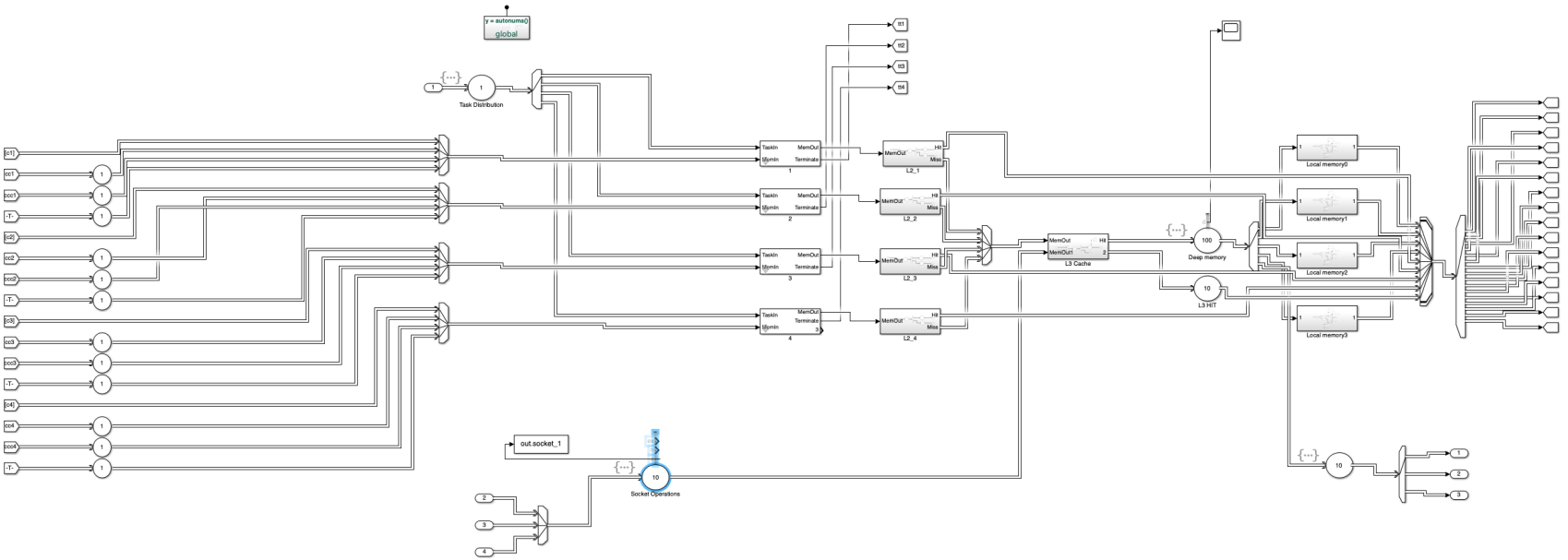


Figure 12: Extended baseline model Processor subsystem. Difference to the dual model is 16 input and output signals for the tasks. Socket connection is formed to all three other processor subsystems, as seen in the bottom of the model.

5.10 Extended CXL model

Extended CXL model consists of the Extended baseline model with CXL device included. CXL device is applied to the multiprocessor system by being connected to all processors. Device co-exists with Infinity Fabric processor to processor connection. System performance is studied with same parameters as the extended baseline model. Model is presented in Figure 13. Added CXL device subsystem is in the left of the model.

5.11 Extended CXL model results

Table 5 show performance results for extended CXL model. Both models show increased task completions as cache efficiencies improve, yet the CXL model consistently reports higher numbers of tasks completed in all scenarios. CXL model performs better even in highly optimized system configuration, completing 38247 tasks, when cache hit rates are high (0.75, 0.75). Task completion performance is increased by 9.28% and 9.72% in highly optimized systems, compared to the baseline model. At low cache efficiencies (0,0) and (0.25,0.25) CXL model shows notable advantage in competing tasks and managing traffic. Extended CXL model completes 22820 and 26415 tasks, over performing the Baseline models 18572 and 22178 completed tasks in low optimized setups. Task completion performance increase is 22.87% and 19.1% for low optimized setup. Results indicate enhanced capability to handle operations even under suboptimal cache conditions. Results for low hit rates imply that CXL reduces dependency on cache hits by streamlining data transfer directly through its high-speed interconnect, thereby minimizing the typical performance penalties of cache misses. Total traffic decreases 60% in average for both low optimized systems and highly optimized system. Performance at high cache hit rates is improved regarding task completion and inter-processor traffic.

Results suggest that substantial enhancements at high cache efficiencies demonstrate CXL's capability to leverage optimal cache conditions to maximize system throughput and efficiency. CXL model not only sustains performance but also amplifies it, taking full advantage of reduced latency and increased bandwidth. In both models memory-heavy tasks (Task 1) show the largest increase in completions when cache efficiency improves. However CXL model handles these tasks more effectively. Processor-heavy tasks (Task 2) and 50/50 task (Task 3) also exhibit better completions in the CXL model.

Next comparisons of CXL range in low optimized systems and highly optimized systems is examined. Low optimized multiprocessor system is defined as 25% hit rate for both L2 and L3 cache. For Highly optimized system hit rate values of 75% is used. These different system settings are examined to study effects of CXL device in relation to applied CXL handling range. CXL range represents percentage value on how much of the systems workload is allocated to the CXL device. Ranges are examined up to 75%, but main focus is from 30-50%, since it is more realistic amount for CXL system level utilization. If CXL range is set at 50% it would mean that CXL

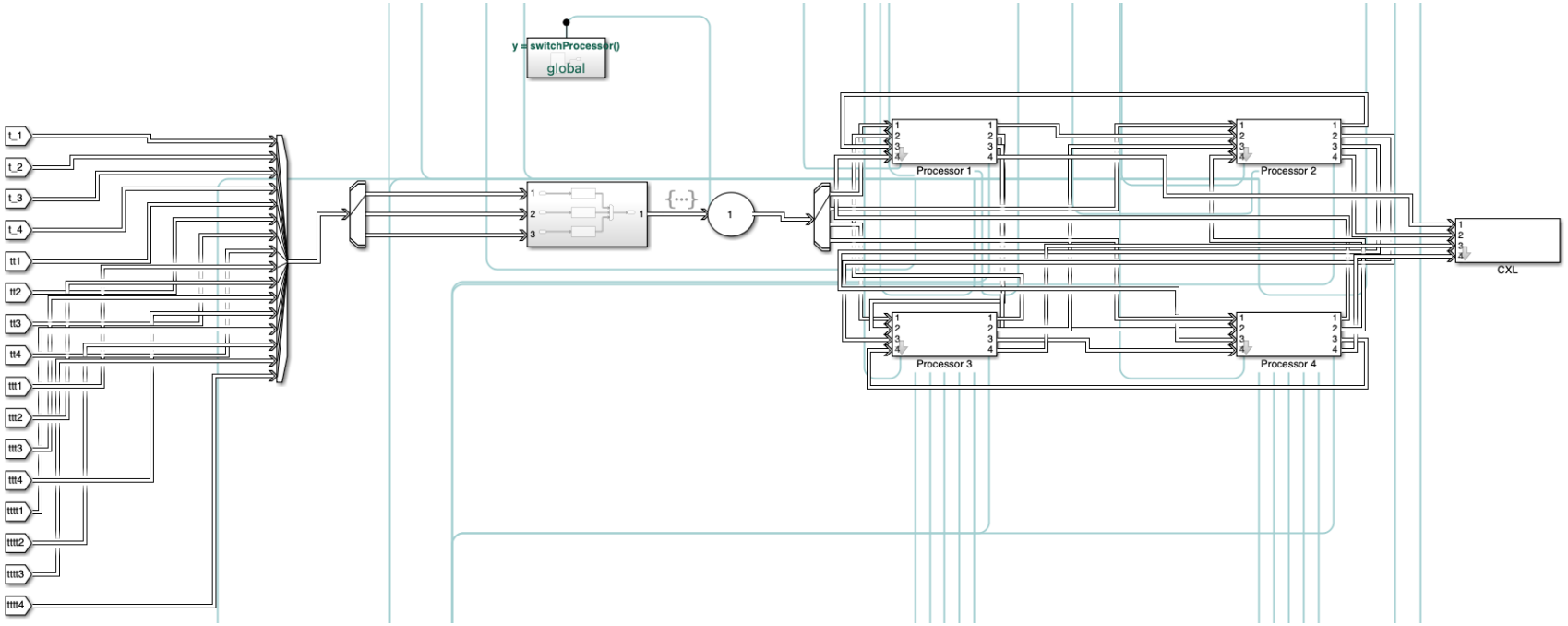


Figure 13: View of the outermost layer of Extended CXL model. CXL component is located at the right side, connected to the four processor subsystems.

Table 5: Result output parameters for extended CXL model measured with different cache hit rates by (L2 hit rate, L3 hit rate).

Parameter	**(0,0)**	**(0.25,0.25)**	**(0.5,0.5)**	**(0.25,0.75)**	**(0.75,0.75)**	**(1,1)**
Pro_1_traffic	1957	1682	1383	819	847	x
Pro_2_traffic	1910	1709	1357	855	807	x
Pro_3_traffic	1945	1728	1347	841	871	x
Pro_4_traffic	2218	1829	1488	915	873	x
Total_traffic	8030	6948	5575	3430	3398	x
Pro_1_tasks	5705	6604	770	9525	9562	x
Pro_2_tasks	5704	6601	7768	9528	9563	x
Pro_3_tasks	5704	6605	7768	9527	9560	x
Pro_4_tasks	5707	6605	7767	9529	9562	x
Total_tasks	22820	26415	31073	38109	38247	x
Task_1_n	7032	8800	11137	15095	15109	x
Task_2_n	6025	6493	7085	7678	7710	x
Task_3_n	9763	11122	12851	15336	15428	x
cxl_n	10648	9361	7342	4423	4396	x

device handles half of the memory requests in the system.

Figure 14 showcases the number of tasks completed in both system configurations across the extended CXL ranges. In both the highly and low optimized systems increase of the CXL range leads to higher amount of tasks completed. The highly optimized system consistently outperforms the low optimized system in terms of tasks completed across all CXL ranges. This suggests that the optimizations involving enhanced cache hit rates, and better utilization of CXL capabilities contribute significantly to the performance gains. Low optimized system gains more improvement in Task completion than highly optimized system when CXL range is increased. Reasoning for this lies in higher actual CXL device utilization when applied in systems with low cache hit rates. Device gains more memory request traffic in Low optimized system. In both cases systems performance increases when CXL is allocated with more workload. The increase in tasks completed correlates with the theoretical benefits of CXL in reducing memory latency and improving bandwidth. This supports the assertion that CXL can significantly enhance the performance of multiprocessor systems by providing more efficient memory access and usage. Also the trend highlights the importance of system optimization in conjunction with CXL technology to maximize performance gains, especially in systems with high memory demands.

Figure 15 shows processor-to-processor and CXL memory request traffic in relation to CXL range. In both systems, socket traffic decreases as the CXL range increases,

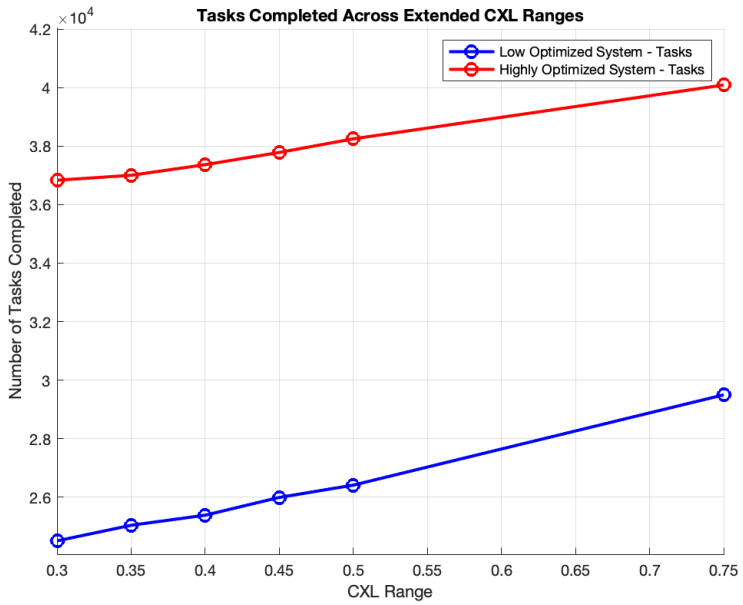


Figure 14: Number of tasks completed across extended CXL ranges. Evidently, the increase in CXL range leads to higher amount of tasks completed. Low optimized system can be seen benefiting more from the CXL range increase than the highly optimized system regarding task completion.

reflecting a reduction in the need for inter-processor communication via traditional system interconnects. This is a positive outcome, as reduced socket traffic typically correlates with fewer bottlenecks in data transfers between processors and memory. CXL traffic increases with the CXL range, especially noticeable in the low optimized system. This indicates that more data transfer is occurring through the CXL interface, which is designed to handle high-speed, efficient communication between the host processor and devices like memory expanders. The reduction in socket traffic while CXL traffic increases suggests that CXL is effectively offloading traffic from traditional interconnects. This shift is a crucial aspect of CXL's role in enhancing system performance by managing data flows more efficiently. Results for traffic imply the disparity in CXL traffic between the low and highly optimized systems could indicate different levels of reliance on CXL for performance improvements. The low optimized system's higher increase in CXL traffic could reflect a more significant offloading of tasks to CXL to compensate for less effective internal optimizations.

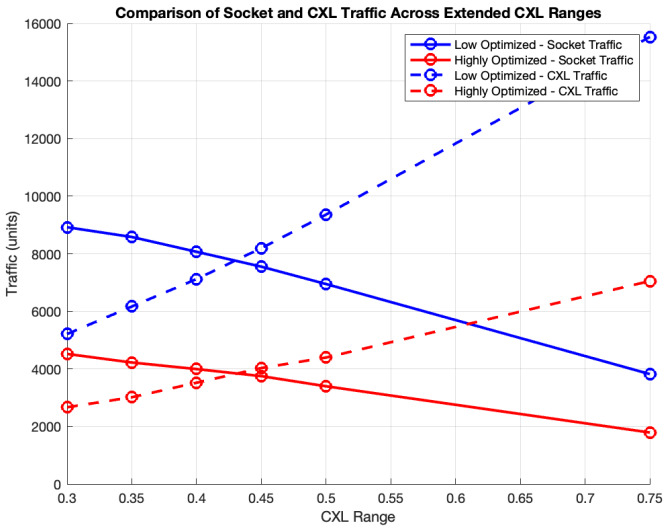


Figure 15: The comparison of socket and CXL traffic across extended CXL ranges. Increase in CXL range reduces socket traffic with both systems. Greater impact can be seen with the low optimized system, where socket traffic is notably lower in higher CXL ranges.

6 Conclusions and Implications of CXL Integration in Multiprocessor Systems

This thesis has extensively examined the performance dynamics of multiprocessor systems enhanced with Compute Express Link (CXL) technology. By using comparative analyses between standard and extended baseline models with and without CXL device, significant insights have been gathered regarding the operational efficiencies, task handling capabilities and overall system performance. Key findings about performance enhancement across model reveal that both the baseline CXL model and the extended CXL model demonstrated superior performance over their non-CXL counterparts, particularly in terms of increased task completions and reduced inter-processor traffic. The CXL models showed marked improvements in handling memory-intensive, processor-heavy and balanced tasks, with the extended CXL model exhibiting the most significant gains due to its additional processors. As for traffic management, a consistent reduction in traffic across all CXL models can be noticed as cache hit rates improved. The CXL technology notably minimized the system's reliance on traditional interconnects, facilitating more efficient and faster data transfers. In general the integration of CXL led to a substantial increase in the number of tasks completed, highlighting devices effectiveness in improving computational throughput. Task completion increase was evident also in already highly optimized system configurations. Considering the CXL workload range, benefits of the CXL increase the more device is utilized in the system. The more task workload is allocated to CXL device, the more benefits are gained in terms of performance, because CXL devices latency and bandwidth benefits.

The CXL model consistently shows improved performance across all cache hit rate scenarios. The percentage improvement ranges from about 4.3% to 13.7% in dual processor models and 9.3% to 22.9% in Extended model as cache efficiency varies, suggesting that CXL's benefits are more pronounced when cache efficiency is lower. The enhanced memory access capabilities provided by CXL (in real life systems, through memory pooling and efficient memory access mechanisms) likely contribute to these improvements. By reducing the need for frequent memory fetches from remote nodes and minimizing data traffic, CXL allows the processors to execute tasks more efficiently. Even in scenarios with higher cache hit rates, where the inherent system efficiency is already improved, CXL adds a noticeable boost in performance. This indicates that CXL not only helps in managing memory traffic but also optimizes how tasks are distributed and handled in the system.

The empirical evidence provided by the Gouk et al. study [23] substantiates the theoretical framework of this thesis, asserting that CXL technology is effective in resolving critical challenges faced by multiprocessor systems, especially in managing memory bandwidth and reducing access latency. The study not only underscores the practical benefits of CXL but also illustrates a viable implementation that could be considered for future architectural designs and optimizations in multiprocessor

environments. This real-world application of CXL as demonstrated in the study serves as a model for potential system enhancements and further technological advancements.

Regarding scalability and system design, the findings affirm that CXL integration is crucial for scaling up multiprocessor systems to handle more complex and larger workloads efficiently. System designers and architects are encouraged to consider CXL in early stages of the system development to achieve increased scalability and performance. In optimization strategies perspective, findings suggest that optimal CXL performance is achieved when device is introduced to highly optimized cache system. Future systems should integrate advanced cache optimization techniques with CXL technology to maximize performance benefits. This is supported by CXL study 1 which highlighted importance of CXL system optimization, and showed that best results are not achieved just by simply using default system settings with CXL device. The study 1 provides empirical evidence on the challenges and opportunities presented by CXL memory in multiprocessor architectures. The nuanced performance impacts highlighted by this study underscore the necessity for sophisticated memory management techniques in fully capitalizing on CXL technology. This supports the argument that while CXL offers significant advantages in terms of memory bandwidth and latency reduction, its successful integration into system architectures requires careful consideration of application-specific memory access patterns and advanced management policies. The demonstrated benefits of CXL in the model suggest that CXL technology is well-suited to scaling up in more complex or larger multiprocessor environments, potentially addressing some of the critical bottlenecks associated with traditional multiprocessor NUMA systems.

7 Further Development of the model

To further examine performance effects of the CXL devices in multiprocessor architectures, scheduler functions could be implemented to the model. Realistic task scheduler, for example based on existing AutoNUMA, could be integrated in to the task distribution server block. CXL-aware scheduler would be possible to implement in logical level. Dynamic task scheduler would create more in-depth results. Cache protocols are possible to develop to the model to achieve more realistic model. For example simplified version of MESI (Modified, Exclusive, Shared, Independent) protocol that CXL takes advantage of could be implemented into the model. Since the system is based on subsystem blocks connected to each other, scaling the system include more processors is possible. Large multiprocessor systems are possible to be created and studied with the model's blocks. The model has potential to be applied to test and develop CXL operational algorithms in terms of logic, since operational functions could be integrated in the Simulink's blocks. Models can be used on the performance estimations on these algorithms. Since subsystem structure of the model is scalable, model could be extended to represent large CXL based memory pool system. Performance and the bottlenecks of this sort of system would be possible to examine. Models are possible to be further developed to have actual task functionalities

in the system. For example tasks could complete mathematical calculations, and performance effects of these calculations could be studied.

References

[1] O. A. Olukotun, L. Hammond, and J. P. Laudon, *Chip multiprocessor architecture: techniques to improve throughput and latency*, vol. 3. Morgan & Claypool Publishers, 2007.

[2] D. Sanchez, G. Michelogiannakis, and C. Kozyrakis, "An analysis of on-chip interconnection networks for large-scale chip multiprocessors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 7, no. 1, pp. 1–28, 2010.

[3] Z. Yi, F. Chen, and Y. Yao, "A barrier optimization framework for NUMA multi-core system," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 5, p. e5527, 2020.

[4] L. Bergstrom, "Measuring NUMA effects with the STREAM benchmark," *arXiv preprint arXiv:1103.3225*, 2011.

[5] M. Liu and T. Li, "Optimizing virtual machine consolidation performance on NUMA server architecture for cloud workloads," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 325–336, 2014.

[6] Z. Majo and T. R. Gross, "Memory management in numa multicore systems: trapped between cache contention and interconnect overhead," in *Proceedings of the international symposium on Memory management*, pp. 11–20, 2011.

[7] I. Arikpo, F. Ogban, and I. Eteng, "Von Neumann architecture and modern computers," *Global Journal of Mathematical Sciences*, vol. 6, no. 2, pp. 97–103, 2007.

[8] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[9] R. B. Hur and S. Kvatinsky, "Memory processing unit for in-memory processing," in *2016 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, pp. 171–172, IEEE, 2016.

[10] D. Efnusheva, A. Cholakoska, and A. Tentov, "A survey of different approaches for overcoming the processor-memory bottleneck," *AIRCC's International Journal of Computer Science and Information Technology*, pp. 151–163, 2017.

[11] D. D. Sharma, "PCI Express 6.0 specification: A low-latency, high-bandwidth, high-reliability, and cost-effective interconnect with 64.0 gt/s PAM-4 signaling," *IEEE Micro*, vol. 41, no. 1, pp. 23–29, 2020.

[12] D. D. Sharma, "Compute express link," *CXL Consortium White Paper*, 2019.

[13] S. Agarwal, R. Agarwal, B. Montazeri, M. Moshref, K. Elmeleegy, L. Rizzo, M. A. de Kruijf, G. Kumar, S. Ratnasamy, D. Culler, *et al.*, "Understanding host interconnect congestion," in *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pp. 198–204, 2022.

[14] J. Sim, S. Ahn, T. Ahn, S. Lee, M. Rhee, J. Kim, K. Shin, D. Moon, E. Kim, and K. Park, "Computational CXL-memory solution for accelerating memory-intensive applications," *IEEE Computer Architecture Letters*, vol. 22, no. 1, pp. 5–8, 2022.

[15] S. Ryu, S. Kim, J. Jun, D. Moon, K. Lee, J. Choi, S. Kim, H. Kim, L. Kim, W. H. Choi, *et al.*, "System optimization of data analytics platforms using compute express link (CXL) memory," in *2023 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pp. 9–12, IEEE, 2023.

[16] P. Caheny, L. Alvarez, S. Derradji, M. Valero, M. Moretó, and M. Casas, "Reducing cache coherence traffic with a numa-aware runtime approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 5, pp. 1174–1187, 2017.

[17] C. Fensch, N. Barrow-Williams, R. D. Mullins, and S. Moore, "Designing a physical locality aware coherence protocol for chip-multiprocessors," *IEEE Transactions on Computers*, vol. 62, no. 5, pp. 914–928, 2012.

[18] D. D. Sharma, "Compute express link®: An open industry-standard interconnect enabling heterogeneous data-centric computing," in *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*, pp. 5–12, IEEE, 2022.

[19] C. E. L. Consortium, "Compute Express Link Specification - Revision 3.0, Version 1.0." https://www.computeexpresslink.org/download-the-specification, 2022 (accessed June, 2023).

[20] D. D. Sharma, R. Blankenship, and D. S. Berger, "An introduction to the compute express link (CXL) interconnect," *arXiv preprint arXiv:2306.11227*, 2023.

[21] C. E. L. Consortium, "Compute express link specification - revision 3.1, version 1.0." https://www.computeexpresslink.org/download-the-specification, 2023 (accessed March, 2024).

[22] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. 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*, pp. 105–121, 2023.

[23] M. Ha, J. Ryu, J. Choi, K. Ko, S. Kim, S. Hyun, D. Moon, B. Koh, H. Lee, M. Kim, *et al.*, "Dynamic capacity service for improving CXL pooled memory efficiency," *IEEE Micro*, 2023.

[24] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. 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*, pp. 574–587, 2023.

[25] C. Brandberg and M. Di Natale, "A Simevents model for the analysis of scheduling and memory access delays in multicores," in *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*, pp. 1–10, IEEE, 2018.

[26] Caroline Brandberg (2024). Analysis of Scheduling and Memory Access Delays in Multicores (https://www.mathworks.com/matlabcentral/fileexchange/66173-analysis-of-scheduling-and-memory-access-delays-in-multicores), MATLAB Central File Exchange. Retrieved May 13, 2024.

[27] M. Velten, R. Schöne, T. Ilsche, and D. Hackenberg, "Memory performance of AMD EPYC Rome and Intel Cascade Lake SP server processors," in *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, pp. 165–175, 2022.

A Appendix

Table 6: Variable values for baseline model

Variable	Value	Description
$mem0\_local\_t$	11.0	Node related local memory latency
$mem0\_local\_t$	11.0	
$mem1\_local\_t$	11.5	
$mem2\_local\_t$	14.4	
$mem3\_local\_t$	12.75	
$mem\_deep\_0\_4$	20.9	Node to Deep memory latency
$mem\_deep\_0\_5$	21.1	
$mem\_deep\_0\_6$	20.4	
$mem\_deep\_0\_7$	20.6	
$mem\_deep\_1\_4$	21.2	
$mem\_deep\_1\_5$	21.7	
$mem\_deep\_1\_6$	21.0	
$mem\_deep\_1\_7$	21.2	
$mem\_deep\_2\_4$	20.3	
$mem\_deep\_2\_5$	20.5	
$mem\_deep\_2\_6$	20.7	
$mem\_deep\_2\_7$	21.0	
$mem\_deep\_3\_4$	21.0	
$mem\_deep\_3\_5$	21.1	
$mem\_deep\_3\_6$	21.3	
$mem\_deep\_3\_7$	21.8	
$other\_node\_t$	15.0	
$L2\_1\_hit\_t$	12.65	L2 cache hit latency
$L2\_2\_hit\_t$	13.15	
$L2\_3\_hit\_t$	14.55	
$L2\_4\_hit\_t$	15.2	
$L3\_hit\_1$	12.05	L3 cache hit latency
$L3\_hit\_2$	12.55	
$L3\_hit\_3$	14.0	
$L3\_hit\_4$	14.6	
$pro\_1-4\_overhead$	5	Processor overhead
$pro\_1-4\_queue\_exe\_n$	25	Processor execution queue capacity
$socket\_bw$	1.0	Socket bandwidth multiplier
$L3\_hit\_other$	15	L3 hit latency for other than local nodes
$task\_dist\_t$	1.0	Task distribution latency

Table 7: Variable values for CXL component

Variable	Value	Description
$cxl\_hitrate$	0.3	CXL cache hitrate
cxl_hit_t	7.0	CXL hit latency
$cxl\_protocol\_t$	8.0	CXL protocol latency
ddr_t	8.0	DDR memory latency
$cxl\_load\_to\_use$	5.0	CXL load-to-use latency
cxl_bw	0.364	CXL bandwidth multiplier
cxl_range	varies	CXL memory range