THESIS TO OBTAIN THE DEGREE OF DOCTOR OF THE UNIVERSITY OF MONTPELLIER

In Automatic and Microelectronic Systems (SyAM)

Doctoral School: Information, Structures and Systems (I2S)

Research Unit: LIRMM

Modeling and Analysis of Cache-Coherent Chip-to-Chip Interconnects

Presented by Luis BERTRAN ALVAREZ

March 27, 2025

Under the supervision of Pascal BENOIT & David NOVO

Thesis Committee:

Frédéric PÉTROT, Professor, TIMA, INP Grenoble, CNRS	Reviewer
Enrique VALLERO GUTIERREZ, Associate Professor, University of Cantabria	Reviewer
Abdoulaye GAMATIÉ, Professor, LIRMM, University of Montpellier, CNRS	Examiner, Moderator
Stephen BUSCH, ASIC architecture manager, Eviden	Examiner
Alejandro NOCUA, Senior application engineer, Synopsys	Examiner
Pascal BENOIT, Professor, LIRMM, University of Montpellier, CNRS	Thesis Director
David NOVO, Research Scientist, LIRMM, University of Montpellier, CNRS	Thesis Co-Director

UNIVERSITÉ DE MONTPELLIER

ABSTRACT

The slowdown of Moore's Law has led to a paradigm shift in High-Performance Computing (HPC). As transistor density increases in silicon chips have decelerated, computer architects are forced to explore alternative approaches. This shift has given rise to chiplet-based System-on-Chip (SoC) architectures, where logic is distributed across multiple dies within a single package, instead of traditional monolithic single-chip multicore architectures. While this transition offers improved cost-efficiency and yields, it also introduces new challenges in design, evaluation, and optimization.

Chiplet-based designs present both challenges and opportunities. They introduce varying intercore latencies but also enable post-tape-out heterogeneous integration, high-bandwidth on-package interconnects, and cache coherence across multiple interconnected chips. New protocols such as CCIX, UCIe, and OpenCAPI have emerged to facilitate communication between interconnected chips and maintain coherence. This approach allows for flexibility, ranging from single-chiplet CPUs for edge applications to multi-chiplet CPUs for high-performance cloud computing, all operating under a single OS environment.

Maintaining the coherence in modern multicore systems is already a challenging task. New protocols must frequently be designed and even if a new protocol is not a radical departure from previous protocols, designing and validating it are tedious, bug-prone processes. The extension of cache coherence from traditional monolithic multicore systems to multiple chiplets using independent cache-coherence protocols significantly increases design complexity. To address these challenges without resorting to time-intensive RTL designs, simulations, or costly tape-outs, high-level simulation is essential. However, existing architecture simulators, designed for monolithic chips, often lack the flexibility, detailed modeling capabilities, and OS control at the node level required to accurately simulate modern coherent interconnect protocols. Moreover, simulators with such capabilities are frequently proprietary, slowing down academic research and obscuring simulation details.

This thesis aims to explore the modeling of coherent interconnect protocols for chiplet-based systems, focusing on multicore cache coherence and chip-to-chip interconnect protocols. We address several key limitations in existing solutions. Current

multi-chiplet simulations often use multiple parallel simulator instances interconnected by external network simulators, each running its own OS, which poorly represents the targeted architectures. Additionally, monolithic network-on-chip (NoC) models that have been extended to multi-chip architectures often fail to extend coherence across all chips. Furthermore, open-source simulation platforms are typically complex and not natively compatible with multiple single network-on-chip handling.

To tackle these issues, this thesis makes several main contributions. First, we present a methodology for modeling and designing models in modern, open-source system-level simulators. We present gem5 and explain all the key aspects of the simulator, as well as debugging strategies and considerations.

Secondly, we discuss the challenges of accurately modeling coherent chip-to-chip interconnects, including chip-to-chip link management, maintaining coherence across multiple chips, and inter-chip routing. These insights provide a foundation for understanding the complexities involved in simulating modern multi-chip architectures, independently of the targeted simulator.

Lastly, we present an implementation of a chip-to-chip interconnect model in gem5. This solution extends the Arm CHI protocol to multiple chips while maintaining coherence, allowing full-system simulation of real-world applications while maintaining a single-chip programmer's perspective. We provide solutions to various modeling challenges and a methodology for calibrating the simulated model with a real hardware platform.

This thesis offers a comprehensive package that includes both the methodology for designing new simulation models and addressing development challenges, as well as a complete implementation of a coherent chip-to-chip interconnect model. Our contribution represents an important step towards enabling accurate simulation of cache-coherent multi-chip architectures and aims to inspire future work in optimizing such architectures. By providing open-source tools and methodologies, we hope to accelerate research in this critical area of computer architecture, enabling more efficient and powerful computing systems in the future.

RÉSUMÉ

Le ralentissement de la loi de Moore a conduit à un changement de paradigme dans le calcul haute performance (HPC), forçant les architectes informatiques à explorer des alternatives. Ce changement a donné naissance aux architectures System-on-Chip (SoC) basées sur des chiplets, où la logique est distribuée sur plusieurs dies au sein d'un même package, remplaçant les architectures multicœurs monolithiques traditionnelles. Bien que cette transition améliore la rentabilité et les rendements, elle introduit de nouveaux défis en matière de conception, d'évaluation et d'optimisation.

Les conceptions basées sur les chiplets présentent à la fois des défis et des opportunités, introduisant des latences variables entre les cœurs tout en permettant une intégration hétérogène post-tape-out et des interconnexions à haute bande passante. De nouveaux protocoles comme CCIX, UCIe et OpenCAPI ont émergé pour faciliter la communication entre les puces interconnectées. Cette approche permet une flexibilité allant des CPUs à chiplet unique pour les applications edge aux CPUs multi-chiplets pour le cloud computing haute performance.

Le maintien de la cohérence dans les systèmes multicœurs modernes est un défi, nécessitant une conception et une validation fréquentes des protocoles. L'extension de la cohérence de cache à plusieurs chiplets augmente significativement la complexité de conception. Bien que la simulation puisse répondre à ces défis sans recourir à des conceptions RTL chronophages ou des tape-outs coûteux, les simulateurs d'architecture existants manquent souvent de flexibilité et de capacités nécessaires pour les protocoles d'interconnexion cohérents modernes. De plus, les simulateurs capables sont souvent propriétaires, entravant la recherche académique.

Cette thèse explore la modélisation des protocoles d'interconnexion cohérents pour les systèmes basés sur les chiplets. Nous abordons les principales limitations des solutions existantes : les simulations multi-chiplets actuelles utilisent souvent plusieurs instances parallèles de simulateur avec des environnements OS séparés, représentant mal les architectures ciblées. De plus, les modèles network-on-chip monolithiques étendus échouent souvent à maintenir la cohérence entre les puces, et les plateformes open-source manquent généralement de support natif pour la gestion de multiple network-on-chip.

Nos contributions incluent : premièrement, une méthodologie de modélisation dans les simulateurs système open-source modernes, présentant gem5 avec ses aspects clés et stratégies de débogage. Deuxièmement, une analyse des défis de la modélisation des interconnexions cohérentes puce à puce, fournissant des directives pour la simulation d'architectures multi-puces. Enfin, nous implémentons un modèle d'interconnexion puce à puce dans gem5, étendant le protocole Arm CHI à travers plusieurs puces tout en maintenant la cohérence et permettant une simulation full-system.

Cette thèse fournit à la fois une méthodologie pour concevoir des modèles de simulation et une implémentation complète d'un modèle d'interconnexion cohérent puce à puce. Notre contribution fait progresser la simulation précise des architectures multi-puces cohérentes en cache et vise à inspirer de futurs travaux d'optimisation. Grâce aux outils et méthodologies open-source, nous cherchons à accélérer la recherche en architecture informatique, permettant des systèmes informatiques plus efficaces.

Acknowledgments

Before diving into the core content of this thesis, I would like to thank many people without whom this PhD manuscript would not exist. I believe many of them do not realize the extent of their contribution.

First, I would like to thank my thesis director Pascal Benoit. His incredible support through the years helped me approach problems with a different point of view. His knowledge of academia was also of the utmost importance.

David Novo, my PhD thesis co-director, taught me everything I know about research. I learned how to approach complicated tasks, organize my work and dissect my bad habits. I look up to his consistency and clear-headedness and will work hard to match them someday. I can confidently say that without his support, this project would not have been completed. I feel lucky to have had such an amazing support during my PhD.

During this PhD, I also had the support of the team at Eviden (formerly Atos). Their supervision has greatly helped build this project. They also enabled a bit part of the project by providing the hardware, which allowed our main publication. I would like to thank Alejandro, Stephen and Ghassan.

During this three-year journey between Atos in Paris and the LIRMM in Montpellier, I spent a lot of time with many other PhD students. From the former colleagues: Théo, Loïc; The newest team members: Enzo and Felipe; and all the great friends spending time with me at and outside the lab: Paul, Soraya, Rafael (akaṘaf), Xhesila (akaẊhes), Gianmarco, Kamilia, Lila, Mohamed, Thomas, Amaury, Paul, and many others. All the moments we shared, and their support, has helped me get through the hardest times of the PhD.

Along with my lab friends, I also need to thank my precious friends, some far and some close, but who made my PhD experience so much more enjoyable: Marc, Gaël, Anouk, Hugo, Mathias, Kevin, Fabrice and Vincent. I would like to express my gratitude to my beloved girlfriend, Xhezaira, for her incredible love and support, being a ray of sunshine during many trying times. All these relationships have been a major drive to go through my PhD with a positive mindset.

Finally, I am grateful for the love and support from my family over the years. Many thanks to my parents and my dear brother Manuel (aka. Manu) and sister Anouk for their unconditional help. They have always been there, and coming back home to a warm family has been an incredible help.

This thesis only exists thanks to all of you.

Table of Contents

Abstract i

Résumé iii

Acknowledgments v

Table of Contents vii

List of Figures ix

Listings . . . xi

Listings xiii

1 Introduction 1

1.1 Context . . . 1

1.2 Contributions . . . 5

1.3 Thesis outline . . . 7

I Background 9

2 Multi-chip cache coherence 11

2.1 Cache coherence fundamentals . . . 11

2.2 Coherent chip-to-chip interconnects . . . 17

2.3 CHI and CCIX protocols . . . 20

2.4 Summary . . . 25

3 The gem5 simulation framework 27

3.1 Computer architecture simulation platforms . . . 27

3.2 Overview of gem5 . . . 30

3.3 Simulation workflow in gem5 . . . 31

3.4 The Ruby memory subsystem . . . 33

3.5 SLICC: cache coherence protocol specification . . . 36

3.6 Debugging techniques in gem5 . . . 41

3.7 Summary 46

II Contributions **47**

4 Extending cache coherence across multi-chip architectures **49**
4.1 Related work 49
4.2 Intra-chip message routing strategies 53
4.3 Inter-chip message routing strategies 55
4.4 Maintaining coherence across chips 59
4.5 Inter-chip link management for coherence 60
4.6 Summary 62

5 Full-system simulation of cache-coherent interconnects **63**
5.1 Motivations 63
5.2 Implementing the C2CI in SLICC 64
5.3 Addressing port instantiation challenges 72
5.4 Integrating KVM with dual networks 77
5.5 Iterative development and validation 78
5.6 Inter-chip handshake mechanism 83
5.7 Extending CHI for multi-chip systems 86
5.8 Model complexity and development effort 90
5.9 Interface calibration 91
5.10 Results 96
5.11 Summary 98

6 Conclusion and future directions **101**
6.1 Summary of contributions 101
6.2 Future work 103
6.3 Concluding remarks 104

Bibliography **107**

List of Figures

1.1 Processor performances compared to the VAX11-780 with era names. Source: [44] . . . 2
1.2 Chiplet design principle with monolithic chip on the left and system-in-package on the right. . . . 2

2.1 Cache hierarchy schematic in a simple multicore system. . . . 12
2.2 Step-by-step diagram of a simple multicore system without cache-coherence. . . . 13
2.3 Cache-coherence solving data consistency issues in a simple 2-core architecture. . . . 16
2.4 Relationship between the OSI reference model and SoC/NoC layers. . . . 18
2.5 Network-on-chip in a mesh topology. Right side details the router. . . . 18
2.6 High-level architecture with two dual-core interconnected chips with private L1s, private L2s, shared L3. . . . 20
2.7 Network-on-chip in a mesh topology, with CPU cores, LLC, memory controllers and a representation of what CHI implementations define. . . . 21
2.8 Example architecture with simple CHI nodes and read flow diagram. . . . 23

3.1 gem5 precision depending on models and modes. Source: [17] . . . 31
3.2 gem5 simulation input, outputs and runtime interface. . . . 31
3.3 gem5 full-system simulation timeline with optional CPU switching and checkpointing. . . . 32
3.4 Life of a memory request in Ruby, from the core to the network. . . . 35
3.5 Simple architecture with a network-on-chip and caches, as well as a simplified view of the FSM of the L1 cache described in SLICC. . . . 37
3.6 SLICC and gem5 compilation flow. . . . 37
3.7 SLICC state machine main components with SLICC keywords (in red). . . . 38
3.8 Checkpoint time savings in the simulation timeline. . . . 45
3.9 Different simulation modes of gem5. Checkmarks show supported simulation modes and red cross shows unsupported modes. . . . 46

4.1 Diagram of a mesh network-on-chip with a request from the L2 cache, address ranges and downstream clients. . . . 54

4.2 Flow diagram of a read request with response data and acknowledgment. 54
4.3 High-level C2C (a) and reference (b) architecture with CHI components and C2CIs. 55
4.4 Flow diagram of an off-chip read example with useful routing and forwarding information. 57
4.5 High level architecture of two chips with CHI NoCs, interconnected through CCIX with an arrow representing access to DRAM-1. CXG refers to CCIX Gateway, interfacing the chips. 58
4.6 Different interpretation of a snoop from a C2CI depending on its origin. 60
4.7 Flow diagram with two requests arriving at the same time to the two C2C link interfaces and causing a deadlock. 61

5.1 Port structure of the C2CI. 65
5.2 Port connection problem when using Ruby ports between two controllers of different networks. 72
5.3 SNF controller with I/O network-side ports and internal Ruby message buffers, themselves connected to the memory_out_port gem5 port, provided by the AbstractController parent class. 74
5.4 C2CI-0 to C2CI-1 communication link with C2C-side ports. Detailed view of gem5 classic ports (i.e., `c2c_out_port` and `c2c_in_port`) structure with method names. 75
5.5 Single chip to multi-chip using Chip-0's SNF as request entry point. . . . 79
5.6 High-level architecture of the memory expansion architecture milestone with address ranges. 80
5.7 Read request transaction with times (cycles). 81
5.8 Interface-interface high-level architeture with address ranges. 81
5.9 Fully coherent chip-to-chip architecture. 82
5.10 Flow diagram of two requests to the same address arriving at the same time to the C2CI and creating a deadlock. 84
5.11 Handshake mechanism flow diagram with two requests arriving simultaneously. 85
5.12 Flow diagram showcasing a problematic access pattern requiring off-chip sharer tracking. 87
5.13 Flow diagram showcasing a problematic access pattern requiring a tracking of the HNF and CHI message extension. 89
5.14 Hardware platform picture and architecture diagram. 92
5.15 Access time distribution on the hardware testbed (top) and the tuned simulation (bottom). 94

5.16 Flow diagram with two different access patterns for local and remote access. 95
5.17 High-level C2C (left) and reference (right) architecture with CHI components and C2CIs. 96
5.18 Execution times 97
5.19 Memory vs C2C link bandwidth 98
5.20 Normalized C2C traffic types 99

6.1 High-level architecture diagram with two RNF per chip and 3 chips. . . 104

LISTINGS

3.1 Sequencer load request in a debug trace 34
3.2 Declaration of the MI cache FSM using SLICC. 38
3.3 Declaration of the states of the MI cache controller in SLICC. 39
3.4 Declaration of the events of the MI cache controller in SLICC. 39
3.5 Declaration of the TBE using a structure. 39
3.6 Declaration of the "response from network" port. 40
3.7 Declaration of transitions using SLICC. 40
3.8 Declaration of an action using SLICC. 41
3.9 Declaration of traffic generators in Python. 42
3.10 Declaration of two debug flags inside the SLICC code. 42
3.11 Command line argument to execute gem5 with a debug flag. 43
3.12 Sample of a trace generated with the RubyGenerated debug flag. 43
5.1 SLICC C2CI port declaration. 65
5.2 Execution trace sample showing a transition. 66
5.3 C2CI getState method in the generated C++ code. 67
5.4 Action using the triggerQueue. 68
5.5 Sample of the event declaration of the C2CI controller. 69
5.6 SLICC compiler modifications to program the wakeup function differently. 76
5.7 Hack to select the correct memory controller. 78
5.8 Use example of the hasDownstreamClient in the SLICC code of the C2CI controller. 85

CHAPTER 1

INTRODUCTION

1.1 Context

Computing power has been pivotal in advancing research and development across many fields of human knowledge. This includes disciplines like physics [1], biology, mathematics [25], chemistry [43] among many others. Progress in these fields has been significantly tied to the ever-growing power of computing devices. Historically, this growth has been steadily following two key semiconductor scaling principles: Dennard Scaling [31] (or MOSFET Scaling) and Moore's law [63]. The Dennard scaling states that as transistors become smaller, their power density stays constant. Moore's law is the observation and prediction that the number of transistors in integrated circuit (IC) doubles approximately every two years. These two principles have driven the industry, which in turn has facilitated advancements in the aforementioned scientific fields. However, as of 2025, it is widely acknowledged that these laws are no longer viable [33, 52]. For instance, Intel's 10nm node manufacturing was delivered in 2019, five years after the previous 14nm generation [6]. Figure 1.1 shows the performances of processors compared to the *VAX11-780* from 1980 to 2015. The VAX11-780 is a well-known *minicomputer* from 1978. The performances metrics are found using the SPECintCPU benchmark. As we can see, the growth in processor performance has plateaued.

This change from the expected exponential transistor count scaling has gradually shifted the computing devices optimization from semiconductor design to computer architecture [44].

1.1.1 Multi-chip architectures

A Network-on-Chip (NoC) is a communication subsystem that applies networking principles to on-chip communications, replacing older bus-based designs. Similar to other networks, NoCs route messages between different components through a network of routers and links, enabling a more scalable communication across the chip.

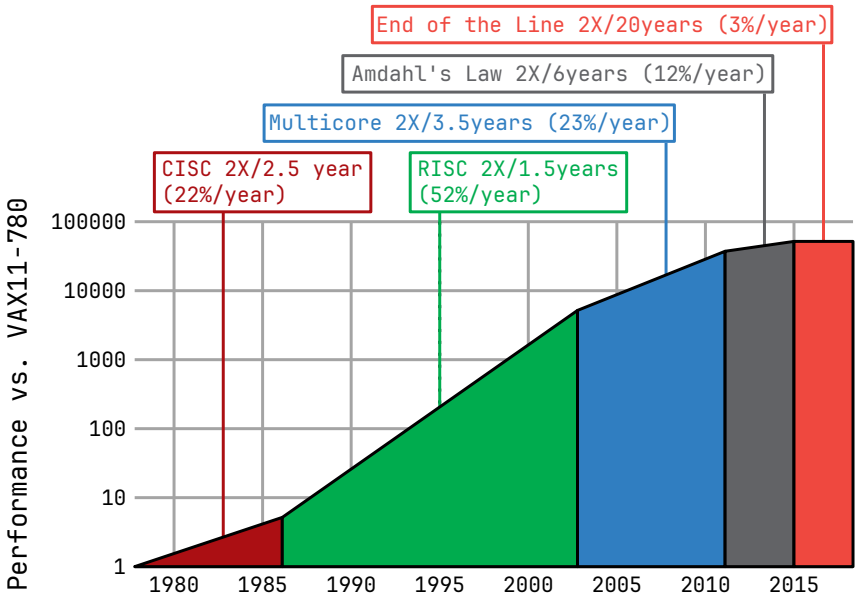


Figure 1.1: Processor performances compared to the VAX11-780 with era names. Source: [44]

Traditionally, researchers and mainstream chipmakers have improved chip architectures within monolithic multicore chips. This means that cores and other components are interconnected with a network-on-chip on a single die. As die sizes increase to accommodate more components, manufacturing yields decrease significantly due to the higher probability of defects in larger silicon areas [92]. Accordingly, and following the decline of Moore's law, computer architects have been pressed to come up with new solutions. A particularly interesting paradigm that emerged is called *chiplet-designs* [64]. With this new approach, the architecture is implemented with a multitude of smaller chiplets. Chiplets are manufactured separately and interconnected together in a System-in-Package. Figure 1.2 shows how the chiplet-design approach creates similar logic as a monolithic chip design by interconnecting chiplets in a system-in-package. Chiplet-based architectures are solving this problem by decreasing die sizes and allowing the use of different process nodes as well as the use of chiplets tailored for special applications.

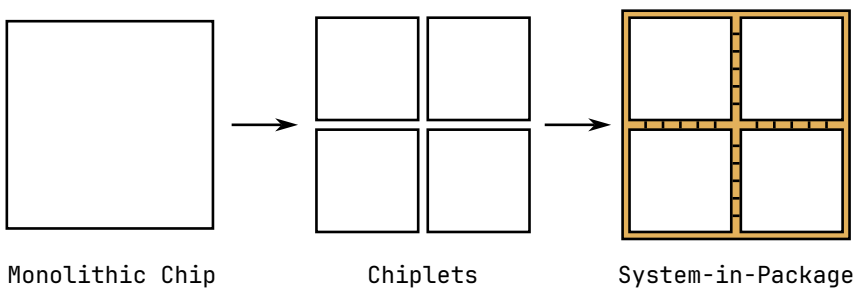


Figure 1.2: Chiplet design principle with monolithic chip on the left and system-in-package on the right.

Designing architectures with chiplets has three main advantages. First, smaller dies allow for a better silicon yield. Smaller portions are lost when having silicon defects. Moreover, smaller dies also allow for a better utilization of the wafer's circular surface. Second, it allows the use of different process nodes within a single chip [58]. This allows to cut prices by using more mature process nodes for chiplets that don't need critical density. Third, the chiplet design allows to build chips over the maximum chip size imposed by the lithographic reticle size [3].

A particular chiplet-based design approach includes multiple cache-coherent multicore chips that, when connected, maintain coherency across all interconnected chips. This design approach is particularly interesting since it allows the design of complex cache-coherent multicore architectures by combining multiple chiplets. As such, the same chiplet can be used for different market segments, from edge to high performance. In coherent multi-chip architectures, chiplets are interconnected using high-density in-package interconnect physical links as well as coherency protocols. CCIX [27], UCIe [82] and OpenCAPI [85] are examples of open standards proposed to enable such systems. With these protocols, cache coherency is maintained between multiple chiplets. This means that, first, all the cores are working on a unified view of memory. Second, the complete system-in-package chip works under a single OS environment, effectively obfuscating the complexity from the programmer's perspective. The emergence of chiplet-based architectures represents a shift in computer architecture design, driven by the need for application-specific optimization and manufacturing flexibility. The increasing complexity of heterogeneous multi-chip systems, interconnects, necessitates robust simulation frameworks to evaluate design trade-offs and validate architectural decisions before committing to costly implementations.

1.1.2 System-level simulation

Open-source, reliable, precise and flexible high-level simulation of multicore architectures has become increasingly crucial. As the complexity of new architectures grows, so does the design space that needs to be explored. In this context, simulation offers three primary advantages that make it an indispensable tool for both industry and academia. First, simulation provides a fast and cost-effective method for performance predictions. It allows architects to evaluate design choices and estimate the performances without resorting to computationally intensive RTL simulations or expensive tape-outs. This significantly reduces development costs and accelerates the iterative design process. Second, simulation enables hardware-software co-design, a critical aspect of modern system development. By providing emulated platforms, simulation allows software engineers to start their work long before the first physical chips are produced. This strategy decreases time-to-market. Third, simulation enables architectural research. It

provides researchers with a flexible environment to model and test innovative designs for key architectural components. This ability is essential for pushing the computer architecture research.

Currently, there is a diverse ecosystem of open-source high-level simulators. These range from high-speed, scalable many-core systems simulators that prioritize simulation speed [28, 38] to slower more precise cycle-accurate simulators that provide detailed insight into the architecture [10]. This variety allows researchers to choose the most appropriate tool for their specific needs. However, these simulators face significant limitations when modeling emerging coherent multi-chip architectures. Most have been developed with monolithic chip design in mind and lack the flexibility required to accurately represent the complexity of multi-chip systems. Specifically, they often struggle to model the subtleties of cache coherence both within and between chips in multi-chip architectures. Moreover, many of these simulators lack detailed modeling of critical components such as Network-on-Chips (NoC), which are essential for accurately modeling the behavior of advanced multi-chip systems. Another limitation is the lack of OS control at the node level. This restricts the ability to model multiple chips operating under a single operating system (OS), a key characteristic of multi-chip designs.

Thus, in order to bridge the gap in coherent multi-chip architectures simulation, we provide a methodology to model cache-coherent chip-to-chip interconnects.

1.1.3 Related works

In this section, we provide an overview of the works related to the contribution of this thesis. Subsequent chapters will relate and extend this body of work.

Our work builds upon two main research directions in computer architecture. First, chiplet-based architectures have emerged as a response to the increasing limits of traditional transistor density and speed [52]. These architectures enable the integration of heterogeneous components from different vendors into a single System-in-Package while maintaining tight integration benefits [54]. The industry is actively developing coherent interconnect standards such as CCIX [27], CXL [89] and OpenCAPI [85] to address cache coherency challenges in these systems.

Second, computer architecture simulation platforms [5] provide essential tools for architectural exploration and performance prediction. While existing simulators excel at modeling traditional monolithic architectures, they typically lack comprehensive support for multi-chip coherent systems. A detailed discussion of the state-of-the-art of both areas is presented in Chapter 4 Section 4.1.

1.2 Contributions

In this thesis, to tackle the shortcomings in the state-of-the-art, we propose three main contributions.

1.2.1 Model design methodology in gem5

The first important step in developing a model is to identify the key elements required in a simulation platform. Simulation platforms can vary significantly in precision, which typically affects their simulation speed. This trade-off between accuracy and performance is one of the many critical considerations in the selection process. In our research, we carefully evaluated various options and explain our rationale for choosing the gem5 simulator to develop our model.

To justify our choice, we identify and analyze the connections between key features of the gem5 simulator and the specific requirements of our target simulation model. We focus on explaining crucial elements of gem5, such as the *Ruby* subsystem, which provides a flexible framework for modeling complex memory hierarchies, and the SLICC (Specification Language for Implementing Cache Coherence) Domain Specific Language (DSL). SLICC is a powerful tool to specify cache-coherence protocols, which is essential to work on chip-to-chip interconnects. We also explain special aspects of working with gem5 such as how simulation scripting works and how to effectively debug the simulation and behavior of the models.

Though this analysis, we demonstrate that gem5 possesses the necessary toolset to enable accurate modeling of coherent chip-to-chip interconnect protocols. Its modular architecture and robust modeling capabilities align with the demands of this project. This overview not only justifies our platform choice but also offers insights for other researchers in the field, interested in developing new simulation models or using gem5.

1.2.2 Extending cache coherence across chips

Second, we focus on identifying the key elements that are required for accurately modeling coherent interconnects in multi-chip systems. This part of the work addresses the challenges that must be overcome to maintain coherency across multiple chips, regardless of the chosen simulation platform. By exploring these challenges, we provide insights that can be broadly used in the field of computer architecture simulation. Moreover, we show the fundamental challenges and considerations that have been followed in the rest of the project.

We first give a detailed view of the research works related to our project, motivating chiplet-based architecture research and modeling. Then, we explain how messages are routed withing individual chips, focusing on the important concept used to route different types of messages (e.g., requests, responses, etc.). Then, we explore the complexity of extending this routing across multiple chips. We introduce the concept of chip-to-chip interface. We also examine the architectural requirements and additional metadata needed for proper message routing across chips. We then explain the challenges of maintaining coherence in multi-chip systems, focusing mainly on cache line sharer tracking and the handling of different coherence message types. Finally, we show the importance of managing the chip-to-chip link, addressing concurrency issues and the difficulties of handling multiple in-flight transactions.

1.2.3 Full-system simulation of cache-coherent interconnects

Third, we focus on the practical implementation of our methodology for simulating cache coherent interconnect. We develop *c2c-gem5*, a comprehensive model using the gem5 simulator, leveraging its advantages and addressing its limitations. We use the SLICC language to develop our model. We demonstrate how SLICC's capabilities allows us to develop a new component in gem5 called *chip-to-chip interface* that is the cornerstone of c2c-gem5. Although gem5 was originally designed for monolithic chip architectures, we detail the extensions that we have developed to overcome these limitations, enabling the simulation of multiple chips in a single gem5 instance. Mainly, we address the key challenges bridging independent NoCs within Ruby by extending the simulator kernel and the SLICC compiler, allowing the use of two interconnected network models in one Ruby environment. We implement a chip-to-chip link responsible for serializing and transmitting messages between chips, ensuring that the entire system behaves as if it were a single monolithic Network-on-Chip (NoC). This involves the development of a *handshake* mechanism that resolves concurrency issues in the C2C link. Our interconnect model is specifically designed to connect multiple chips implementing the Arm Coherent Hub Interface (CHI) coherence protocol, a state-of-the-art coherence solution for high-performance, scalable, multicore systems. We extend the CHI cache coherence protocol to operate seamlessly across multiple chips.

To ensure the accuracy of our model, we use a calibration process. We extract key metrics from a real hardware platform and use these to calibrate our simulation model. This calibrated model is then validated using real-world multithreaded applications demonstrating its ability to accurately represent multi-chip systems.

Our results show the importance of model flexibility in achieving accurate simulation that closely match real-life performances. The calibrated model provides insights into the workloads and bandwidth utilization of the C2C link but also serves as a tool for predicting the behavior of multi-chip systems under different conditions.

1.3 Thesis outline

We structure this thesis around six chapters. In each chapter we give specific context and motivations for the section. In Chapter 2 we introduce the relevant background cache-coherence, coherent chip-to-chip interconnects as well as examples of two main cache-coherence and interconnect protocols Arm CHI and CCIX. In Chapter 3, we describe gem5 in more detail as well as our design methodology using this simulation platform. We explain why we chose gem5, what are the key parts of the simulator. We show the different parts of the simulator that are important when developing a model. We also explain different debugging strategies for the simulator. In Chapter 4, we explain how cache coherency is expended across chips. This means going from routing messages inside a chip to another chip as well as maintaining coherence and managing the C2C link. In Chapter 5, we detail our interconnect model contribution and showcase its functionality. We propose a modeling methodology called *c2c-gem5* that allows the modeling of multiple coherent interconnected chips inside a single unified OS environment. Finally, in Chapter 6, we conclude this thesis by summarizing the main contributions and discussing future research directions.

Part I

Background

CHAPTER 2

MULTI-CHIP CACHE COHERENCE

In this chapter, we introduce the fundamental concept and technologies that form the foundation of this thesis. We begin by exploring the principle of cache coherence, a critical mechanism for maintaining data consistency in multicore systems. Then, we explain coherent chip-to-chip interconnects, examining their crucial role in enabling seamless communication and data sharing between multiple chips in modern architectures. Lastly, we focus on two significant protocols for this thesis: the Arm Coherent Hub Interface (CHI) and the Cache-Coherent Interconnect for Accelerators (CCIX) and give a short context about the interconnect protocol landscape.

2.1 Cache coherence fundamentals

In order to understand cache-coherency, we will first describe the fundamental aspects of cache memory. We describe how cache memory function, then we go over how cache coherence is enforced.

Cache memory plays a crucial role in enhancing processor performance by bridging the gap in speed between the CPU cores and the main memory [84]. Cache exploits the principles of temporal (i.e., applications tend to reuse recently accessed data) and spatial (i.e., applications tend to reuse data with close addresses) locality, storing recently accessed data and instructions closer to the CPU for rapid retrieval. This reduces memory access times and limits the Von Neumann bottleneck [11] and overcome the memory wall problem [93]. Caches tend to be 10 to 100 times faster than off-chip DRAM [77].

This thesis will often refer to Cache Lines (CL). The cache line is the granularity at which data is managed withing the cache. It represents the smallest amount of data that can be transferred between cache and main memory in a single operation. In most cases, modern compute systems use a 64-byte cache line. Each cache line is associated with metadata such as a *tag* that identifies the memory block it corresponds to, and *status* bit to encode the state of the line (see Section 2.1.1).

When the CPU issues a request, the cache controller compares the tag bits of the requested cache line, and the ones stored in the cache. The request is a *hit* if the requested block state is valid, and the tag bits stored in the cache match the tag of the original request. Otherwise, it is a *miss*, and the controller needs to redirect the request to the next cache level or the off-chip DRAM. Effectively, this means that we call cache hit or miss when the cache controller has the cache line or not.

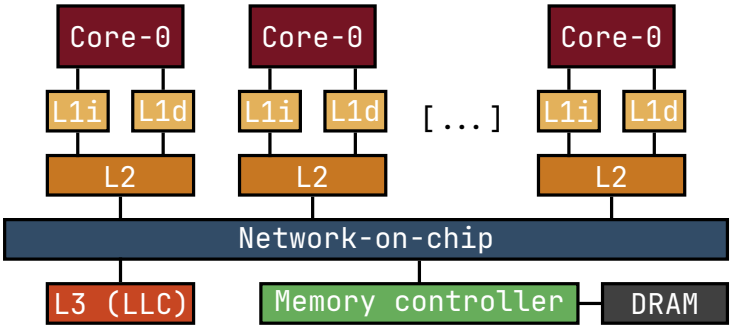


Figure 2.1: Cache hierarchy schematic in a simple multicore system.

Modern processors use a hierarchy of cache levels comprising L1, L2, L3 and sometimes L4. The highest cache level is also known as Last Level Cache (LLC). Each level has an increased size but longer access times. The L1 cache, separated into data and instruction caches, is the smallest and has the shortest access times. It is usually the closest to the CPU core. L2 caches are slower and bigger and are often dedicated to a single CPU core. The LLC is shared among multiple cores, providing a larger last level of on-chip memory. Figure 2.1 shows this cache hierarchy in a simple dual-core system with the Network-on-Chip (NoC) connecting the caches. The cache hierarchy allows to reduce production costs as faster SRAMs are more expensive, achieving the right tradeoff between speed and storage capacity. This hierarchy can be organized either as inclusive, where higher-level caches contain copies of data in lower levels, or exclusive, where each level stores unique data. The inclusive approach simplifies coherence protocols but may reduce effective cache capacity, while the exclusive cache is more storage efficient at the cost of complex management.

As previously stated, modern processors use a hierarchy of cache levels. Moreover, modern processors are also multicore systems. Multicore processors have become essential in modern computing due to fundamental limitations in single-core performance scaling and increasing computational demand. As Dennard scaling broke down, raising clock frequencies led to unsustainable power consumption and heat generation, capping single-core performance gains. Simultaneously, traditional methods of improving single-core efficiency through instruction-level parallelism reached diminishing returns. Multicore architectures address these challenges by exploiting thread-level parallelism, executing multiple threads concurrently to boost system

performance. The shift to multicore designs aligned with the evolving landscape of applications, particularly in fields like scientific computing and more recently artificial intelligence, which exhibit a high degree of parallelism. However, the introduction of multiple cores brings new challenges, especially in managing shared data across caches. When multiple cores operate on shared data, inconsistencies can arise as each core may have its own local cache with potentially outdated versions of the data. This scenario can lead to data races and compromises parallel programs.

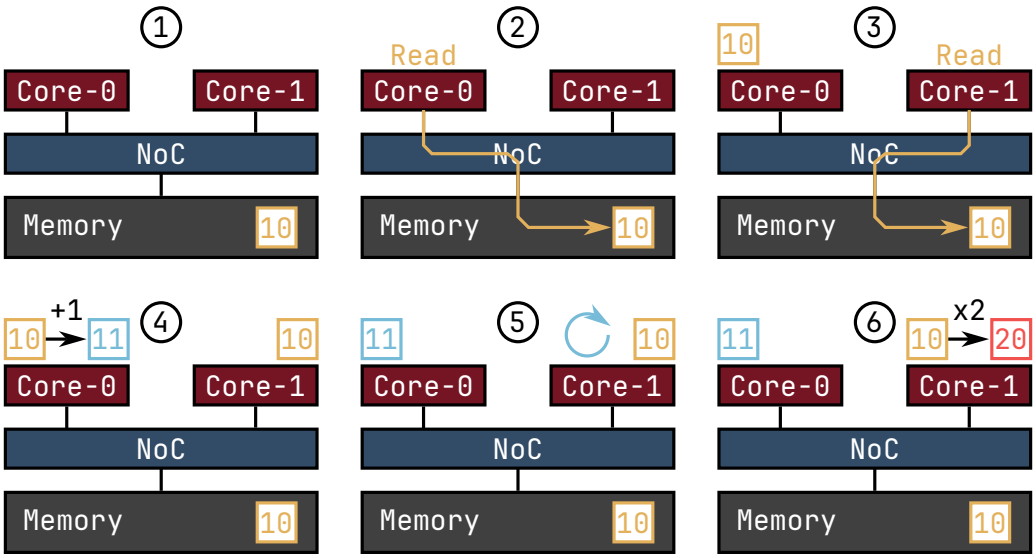


Figure 2.2: Step-by-step diagram of a simple multicore system without cache-coherence.

Figure 2.2 shows a simple two-core system accessing a memory block with the value 10. In step 2, core 0 reads the data, storing it in its cache. In step 3, core 1 accesses the same data. In step 4, core 0 adds 1 to its local copy of the data. In step 5, core 1 reads the data again. Since the data is present in its local cache, the data is read from the cache. In step 6, core 1 multiplies the data by two. Since the data wasn't up to date, the result is erroneous. To address this issue, cache coherence protocols are implemented. These protocols ensure that all caches maintain a consistent view of shared data, automatically propagating changes made by one core to all other cores that have cache the same data. By maintaining data consistency across multiple caches, cache coherence enables efficient and correct execution of parallel programs on multicore systems.

2.1.1 Cache-coherence protocols

Cache coherence is the process that ensures that the changes in values of shared data are propagated through the system in a timely manner. As seen in Figure 2.2, modern multicore systems cannot operate correctly on shared data without cache coherence. Cache coherence is also important because it abstracts away the distributed

memory problem from the programmer's perspective, allowing for easier software development. However, maintaining coherence in memory systems is a challenging task. Each cache can hold shared copies of a cache line. When increasing the number of cores, and by extension the number of cache memories, tracking all copies of shared cache lines and ensuring that each core is working with an up-to-date version of the data becomes increasingly challenging.

In a multicore system, consider that more than one cache holds a copy of the location X. The following conditions are necessary to achieve cache coherence [72]:

- In a read made by processor P to the location X that follows a write by the same processor P to X, with no writes to X by another processor occurring between the write and the read instructions made by P, X must always return the value written by P.

- In a read made by a processor P1 to the location X that follows a write by another processor P2 to X, with no other writes to X made by any processor occurring between the two accesses and with the read and write being sufficiently separated, X must always return the value written by P2. Propagating the writes to the shared memory location ensures that all the caches have a coherent view of memory. If P1 reads the old value of X, even after the write by P2, we can say that the memory is incoherent.

There are two main categories of cache coherence mechanisms: *snooping* and *directory-based* [45]. With a snooping coherence mechanism (introduced in 1983 [75]), when specific data is shared by several caches and a processor modifies the value of the shared data, the change must be propagated to other caches which have a copy of the data. With a directory-based coherence mechanism (first applied version introduced in 1990 [57]), a directory component is used to manage caches in place of bus snooping. The data being shared is placed in a common directory that maintains the coherence between caches. Processors must ask permission to the directory before reading or writing an entry.

Cache coherence protocols are the mechanism that ensure data consistency across the whole chip. They apply cache coherence in multiprocessor systems. The protocol must implement the basic requirements for coherence. It can be designed specifically for the target system or application. Protocols, like cache coherence mechanisms, can be classified as snoopy or directory-based. Protocols can also be split into *write-invalidate* or *write-update* protocols depending on how writes are handled by the cache controller. With a write-invalidate protocol: on a write to a shared cache line, all other copies of the line are invalidated. The opposite behavior is the write-update protocol, where all the

copies of the shared line are updated on a write. The write-invalidate protocol offers a lower traffic overhead since only an invalidation message is sent, and no data transfer occurs unless another core later requests the data. On the other hand, write-update protocols offer a lower read latency for shared data. Since other cores already have the updated data, they avoid costly re-fetch operations. Today, write-invalidate protocols are the most widely used in modern cache-coherence implementations due to its better scalability in system with many cores. It minimizes unnecessary data movement, which is critical in avoiding network congestion.

Snoopy protocols are usually faster than directory-based protocols. However, they have a limited scalability. As each request needs to be broadcasted to all nodes in the system, the size of the bus and its bandwidth needs to grow as systems become larger. This leads to a wide use of directory-based protocols that are advantageous when multicore systems grow in number of cores.

As we will see in later part of this thesis, cache coherence protocols enforce coherence through state machines for each cache and directory. Many cache-coherence protocols stem from the most fundamental three states *M, S, I* (Modified, Shared, Invalid). The *Modified* state means that the block has been modified in the cache. The data is inconsistent with the main memory. The *Shared* state means that the block is not modified and exists in a read-only state in at least one cache. The *Invalid* state means that the block is either not present in the cache or has been marked as invalid by a request. The up-to-date data must be fetched from main memory or a cache that holds it. These states enable multiple cores to simultaneously hold a cache line in read-only (Shared), or to suggest that a single core holds the cache line in read-write (Modified) and the other cores hold the line in an outdated state (Invalid). The *MOESI* protocol extend the MSI protocol with the *Owned* and *Exclusive* states for optimization purposes.

The Figure 2.3 shows how cache-coherence solves the problem highlighted in Figure 2.2. In the first 3 steps, core 0 and core 1 read the data from memory. In step 4, core 0 wants to write the data. A write request is sent to the directory managing the cache coherence. The directory will then invalidate the copy of the cache line in core 1 (step 5). In step 6, the directory receives the invalidation acknowledgment from core 1. The directory then sends an acknowledgment to core 0 (step 7), which can now write the cache line. In step 8, core 1 reads the data again. Since its copy of the cache line is in *invalid* (i.e., I) state, the controller sends a read request to the directory. In this example, upon receiving the read request, the directory will prompt the owner of the cache line to forward it to the requester. This is what happens in step 9 with the directory sending a snoop forward (i.e., SnpFwd) to core 0. In step 10, core 0 provides the up-to-date data to core 1. Core 1 then goes through the steps 4, 5, 6 and 7 to write the cache line. Finally,

step 11 shows the final state of the system, with core 1 owning the last version of the cache line. Thanks to the cache coherency, the data is not erroneous.

Multiple cache coherence protocols have been developed, such as MSI, MESI [69], MOESI, Firefly [87], and Dragon [9] protocol, among others. In 2011, Arm proposed the AMBA 4 ACE for handling coherency in modern system-on-chips (SoCs). Then, Arm developed the AMBA CHI (Coherent Hub Interface) specification, which defines the protocol for fully coherent processors. This protocol enables cache coherence in modern multicore systems. In Section 2.3 we describe the CHI protocol in more details.



Figure 2.3: Cache-coherence solving data consistency issues in a simple 2-core architecture.

2.2 Coherent chip-to-chip interconnects

We previously went over cache-coherence principle. We have mentioned that on top of having cache hierarchies, modern high-performance processors are multicore architectures. In order to connect the different components of the processor, Network-on-Chip (NoC) are used. In response to the design challenges of the early 2000s, network-on-Chip were proposed as a solution by the System-on-Chip (SoC) community [14, 30, 76, 80]. The term Network-on-Chip appeared when it was proposed as a solution to allow for the constant growth in performance [42]. Then, network-on-chip research and development took off and became very active. A survey can be found here [19] as well as a more recent survey here [46]. Network-on-chips are network-based communication systems between modules of a system-on-chip.

In this section, we will cover the important concepts of network-on-chips. Then, we will go over the specificities of chiplet-based architectures. We will explain why chiplet-based architecture came to be and what are their advantages. Finally, we will describe a subset of chiplet-based architecture where chiplets are extending their cache-coherence between them. These architectures are the focus of this work.

2.2.1 Network-on-chip concepts

In order to describe network-on-chip protocols, we will mention terms from the *OSI* protocol (e.g., Physical layer, Network layer, etc.). The Open Systems Interconnection (OSI) model from the International Organization for Standardization (ISO) provides a common basis for the coordination of standard development of interconnections between systems. In the OSI reference model, communication between systems is split into seven different *layers*. For example, the Physical layer, Data Link layer, Network layer, etc. Table 2.4 shows the list of OSI standard layers from, topmost, the application layer, to the physical layer, responsible for the physical sending mechanisms. The layers each describe a part of the functionality used from the physical implementation of transmitting bits across a communication medium (e.g., wire) to the high-level data of an application. Each layer in the OSI model has well-defined functions, and the methods of each layer communicate and interact with those of the layers immediately above and below as appropriate.

As previously stated, in the early 2000s, there was a general trend among designer to provide efficient and standardized communication infrastructure for connecting multiple resources on the chip [55]. Thus, the trend also manifested itself in adapting the layered approach of the OSI reference model towards on-chip communication. Following is a description of the most common network-on-chip function

layers. Figure 2.4 identifies the components of a SoC regarding the layers of the OSI standard.

- The *System layer* corresponds to the processing cores and the application.
- The *interface layer* decouples the cores from the network and controls the process of sending and receiving packets.
- The *network layer* defines how a packet is transmitted over the network from a sender to a receiver (i.e., routing). It is dependent on the interconnect architecture and topology.
- The *physical layer* (or *link layer*) represents the number and length of physical wires connecting the devices.

OSI Layers	SoC/NoC Layers	SoC Components
Application / Presentation	System	Cores
Session / Transport	Interface	Network Adapters
Network	Network	Routers
Link / Physical	Physical	Wires

Figure 2.4: Relationship between the OSI reference model and SoC/NoC layers.

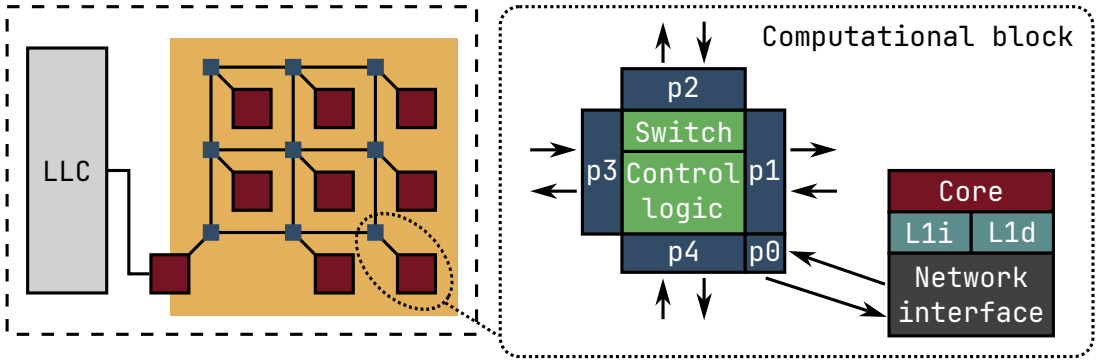


Figure 2.5: Network-on-chip in a mesh topology. Right side details the router.

We will now cover the important terminology for network-on-chips and interconnects. Figure 2.5 shows a NoC diagram with more details about the routers on the right side. The *topology* describes the network's physical layout, dictating how nodes, such as processing elements, are interconnected. The NoC in Figure 2.5 is arranged in a (2D) mesh topology. Data is transmitted through *routers*, which forward *packets*. Each packet can be divided into smaller units called *flits* (flow control unit or flow control digit), to enable for more efficient routing. Flits are atomic pieces (i.e., smallest size of transmission packet). *Latency* refers to the time it takes for a packet to travel from

source to destination, while *bandwidth* defines the amount of data that can be transferred per unit time. *Quality-of-Service* (QoS) mechanisms ensure reliable data delivery. *Routing algorithms* determine the path packets follow through the network, while *flow control* manages the allocation of resources like buffers and links, preventing overflow. When the network is heavily used (i.e., *congestion*) it can lead to increased delays. Finally, *deadlocks* are states where packets are permanently blocked.

2.2.2 Chiplet architectures

In this section, we will provide context and description of chiplet-based architectures. Departing from traditional architectures, heterogeneous architectures make use of different kinds of processing elements to make the architecture more efficient. However, despite offering performances advantage, they come with tradeoffs. For example, the accelerators are always present on the chip even when not in use. To address the challenges of tightly integrated heterogeneous architectures, the industry is turning to *chiplet-based* architectures. Chiplets are smaller integrated circuits that contain a defined subset of functionality. They are usually combined with other chiplets to form a complete System-in-Package (SiP). Chiplets offer a modular approach to SoC design, where different functional blocks are fabricated as separate dies and then integrated on the same die or System-in-Package. This approach provides several advantages, including improved yields, as smaller dies are less prone to manufacturing defects. It also allows for flexibility, where different process nodes can be used for various components, optimizing performance and cost. Chiplet-based architectures can be scaled up or down more easily by adding or removing chiplets. Research has shown that chiplet-based architectures can reduce overall chip costs [68].

The benefits of chiplet-based architectures extend beyond cost reduction and yield. It enables the integration of components from different vendors and for diverse applications into a single SiP [54]. This flexibility enables innovation and allows for customized solutions for specific applications. The potential of this approach has caused industry leaders to actively develop and promote their preferred solution for chiplet integration. Examples include the Universal Chiplet Interconnect Express (UCIe) [82] and the Bunch of Wires (BoW) interface [7]. However, the industry is yet to converge on a single, adopted standard, leaving uncertainties about which solution will emerge as standard.

A particularly promising chiplet-based approach is when cache-coherency is extended between chiplets. This mean that the chiplets of the complete SiP are working on a unified memory region. The industry has recently pushed for new coherent interconnect standards designed to handle coherent chiplet architectures. They are usually

compatible with PCIe (see Section 5.9.1) physicals or SiP interconnects. Examples are the Cache Coherent Interconnect for Accelerators (CCIX) [27], the Compute Express Link (CXL) [89] and OpenCAPI [85]. Each of these protocols builds upon existing physicals while introducing coherency support at the transaction layer.

Figure 2.6 shows a high-level architecture of two dual-core chips that are interconnected. Each CPU has private L1s and L2. Each chip houses a shared L3 (i.e., last level cache) and a memory controller managing a unique memory range.

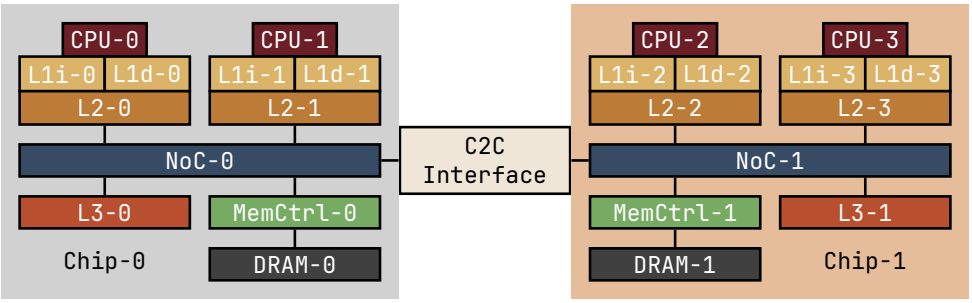


Figure 2.6: High-level architecture with two dual-core interconnected chips with private L1s, private L2s, shared L3.

Among these emerging standards, this thesis will focus on CCIX. Given the uncertainty about which interconnect standard will become dominant, our goal is not to implement a specific protocol (e.g., CCIX). Instead, we aim to provide a reference that can be adapted to any protocol. Moreover, we chose CCIX as we later use a real hardware platform to calibrate our model using the CCIX protocol (see Section 5.9). The CCIX protocol is explored in further detail in Section 2.3.2, providing more insights into its architecture and features.

2.3 CHI and CCIX protocols

As multicore systems have grown in complexity, new cache coherence protocols have emerged. CHI is a high-performance coherence protocol developed by Arm. It allows to maintain coherence in many-core NoCs. As we have seen in Section 1.1.1, because of the advantages that they propose, coherent multi-chip architectures are gaining traction. In these architectures, multiple chiplets are interconnected together, extending cache coherence and working on a unified memory view. To allow these architectures, multiple chiplets of different manufacturers must be designed around a single interconnect standard. Among the currently competing standards, we choose to study CCIX because we had access to the specification. Given the uncertainty about which chip-to-chip interconnect standard will become dominant, our goal is not to implement a specific protocol. However, we study CCIX to better understand the principles of chip-to-chip interconnection.

This thesis will often refer to the CHI protocol and CCIX protocol. This section describes in more details both protocols and gives a practical use case example of a chip-to-chip memory access, following the study of the reference of both protocols.

2.3.1 CHI protocol

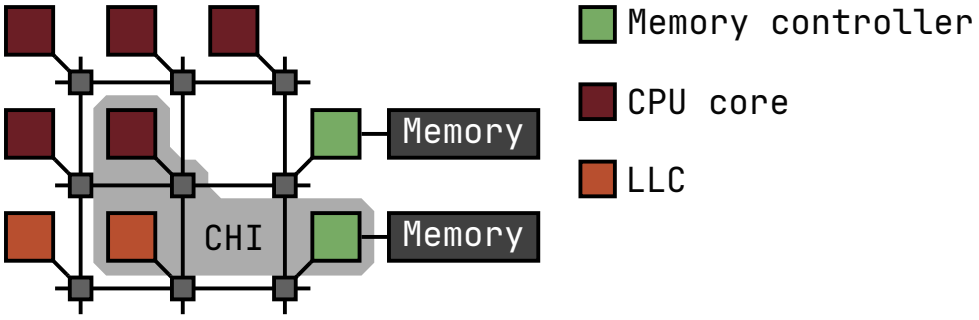


Figure 2.7: Network-on-chip in a mesh topology, with CPU cores, LLC, memory controllers and a representation of what CHI implementations define.

The CHI (Coherent Hub Interface) architecture is a scalable, coherent hub interface and on-chip interconnect used by multiple components. It allows to connect, in flexible topologies, multiple components. For example, standalone processors, graphic processors, memory controllers, PCIe subsystems and interconnects. Arm released the Advanced Microcontroller Bus Architecture in 1996 [35]. It was, at that time, facilitating the development of multiprocessor designs with large numbers of controller and peripherals. The scope of AMBA has, since then, gone beyond microcontroller devices. In 2013, the AMBA CHI specification was introduced with features to reduce congestion and allow scalability while maintaining performance for the growing number of components. Figure 2.7 shows the scope of the CHI implementation. We can see that multiple different nodes (e.g., memory controller, CPU core, LLC, etc.) are part of the same network. CHI provide the functionality for these notes to work on a unified memory view. Implementation like Arm Neoverse CMN-700 (Coherent Mesh Network) make use of the CHI protocol.

The CHI protocol enables modular designs, from small edge applications to large High-Performance Computing (HPC) applications. It uses a packet-based communication meaning that cache lines are encapsulated in a short message of fixed format, called *packets* that are transmitted over the network. Apart from the data, the packet contains a header and relevant metadata for routing. CHI's functionality is grouped into three main layers: the protocol, network, and link layer.

- **The protocol layer** is the topmost layer of the CHI architecture. Its function is to generate and process requests for the protocol nodes. It defines the transaction flow for each request type. It also defines the permitted cache state transitions.

- **The network layer** encapsulates the protocol layer's message. It also Determines the source and destination node IDs required to route the packet over the interconnect.

- **The link layer** provides flow control between network devices. It also manages the network in order to avoid deadlocks.

A *Transaction* carries out a single operation. Typically, transactions are either read or write operations to memory. Then, the protocol layer handles *messages*. Examples are requests, data responses, snoop requests. A data response can be made from numerous *packets*. Packets are used over the interconnect between endpoints. Each packet contains routing information such as destination ID, source ID, allowing routing of the message and responses over the interconnect.

The CHI protocol has three main node types. The *Request node*, *Home node* and *Subordinate node*.

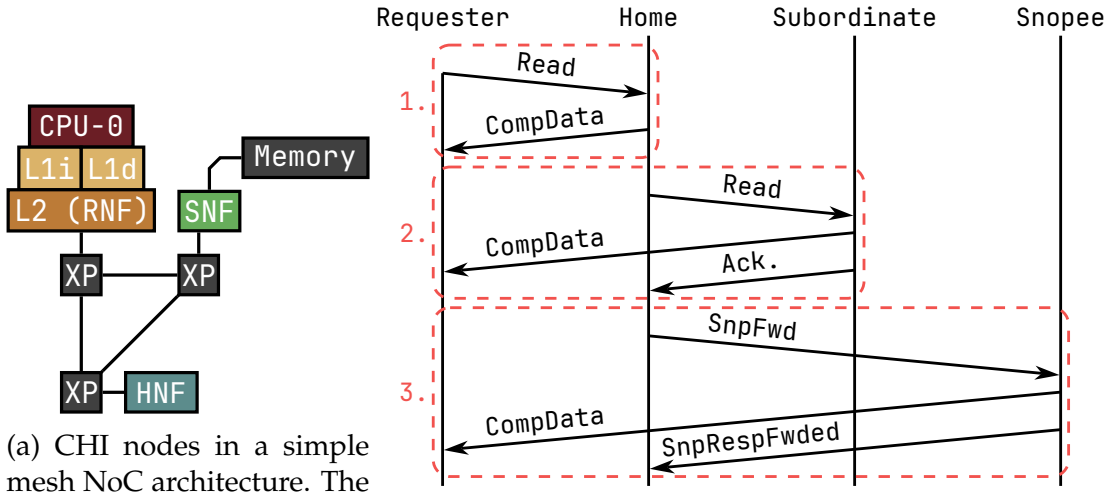
- **Request Node (RN)**: generates protocol transactions, including reads and writes, to the interconnect. For example, a L2 controller.

- **Home Node (HN)**: node within the interconnect that receives protocol transactions from Request Nodes, completes the coherency actions, and returns a response.

- **Subordinate Node (SN)**: node that receives a request from a Home Node, complete the required actions and sends a response. These nodes are often used for off-chip memory accesses (e.g., DRAM accesses).

Figure 2.8a shows CHI nodes in a simple architecture with a mesh NoC. The L2 is a request node and issues its requests to the HN. When the HN needs to access main memory, it issues requests to the SN. Despite the figure showing a mesh topology, the CHI protocol is topology-independent and can adapt to ring, crossbar among others.

Figure 2.8b shows three possible transactions flows for allocating read transaction. The transaction starts with the Requester issuing a read request to the Home Node. This initial request can be for example: ReadClean, ReadShared, ReadUnique, etc. In the first case, the data is cached by the Home Node and returned in a combined response and read data (i.e., *CompData*). In the second case, the data is not cached by the home. The home node sends a downstream request to the Subordinate. The Subordinate returns the combined response and read data to the Requester and sends a *receipt* or *Acknowledgment* to the Home Node (e.g., CompAck). In the third case, the Home Node requests a snoopee (e.g., a cache controller) to forward the data (SnpFwd)



(a) CHI nodes in a simple mesh NoC architecture. The L2 is a request node.

(b) Different outcomes for a read request.

Figure 2.8: Example architecture with simple CHI nodes and read flow diagram.

to the Requester. This option is used when the Home Node does not cache the line but knows that a snoopee (or sharer) has a copy. The snoopee returns the CompData to the Requester and provides a snoop response to the Home Node. This last step happens when the snoopee does not provide a copy of the data to the Home Node.

2.3.2 CCIX protocol

This thesis will mention the CCIX protocol on multiple occasions. The Cache-Coherent Interconnect for Accelerators (CCIX) is an open cache coherent interconnect architecture developed by a consortium of companies. Its base specification was released on June 2018. It enables data sharing between a processor, accelerators and external attached memory. It extends the traditional processors-to-processor and processor-to-memory data sharing to include processor-to-accelerator and accelerator-to-accelerator. Moreover, it enables the data sharing between devices without software intervention (OS or driver) to manage data movement. Cache coherency is automatically maintained at all times between the central processing unit (processor) and other accelerators. CCIX supports signaling rates between 16 GT/s (Giga Transfers per Second) and 25 GT/s per link, when operating over PCIe physicals. CCIX is built upon a layered protocol stack that extends PCIe.

- First is the protocol layer. This layer is responsible for the coherency protocol, including memory read and write flows.
- Second is the CCIX link layer. This layer is responsible for formatting the CCIX messages for the target transport system (in most cases PCIe).
- Third is the CCIX and PCIe transaction layer, which are in charge of handling

their respective packets. CCIX can use standard PCIe packets or CCIX optimized packets that have a reduced overhead. CCIX uses the standard PCIe data-link layer, which performs a series of checking to ensure Quality-of-Service (QoS).

- Finally, the CCIX protocol uses the PCIe physical layer.

In the interconnect landscape, we see different types of new and already-existing solutions. First, The Intel Ultra Path Interconnect (UPI) is a low-latency coherent interconnect for scalable multiprocessor with a shared address space. It uses a directory-based coherency protocol. A combined caching and home agent (CHA) handles resolution of coherency across multiple processors as well as snoop requests from processor cores and local and remote agents. It is a proprietary solution from Intel. Another protocol is The Universal Chiplet Interconnect Express (UCIe). It is an open industry standard designed to enable seamless, high-performance interconnection between chiplets within a package. It provides a standardized physical layer, protocol stack and software interface to allow chiplets to communicate. It does not support cache coherent chiplets. Unlike symmetric coherency protocols like UPI or CCIX that require all agents to implement a home agent, CXL implements an asymmetric *CXL.cache* protocol that relies on the host agent to maintain coherency between CPU and accelerator. This greatly simplifies the accelerator design since it does not need to orchestrate coherency and instead focuses on providing the core functionality of the device and the simple set of abstract commands (e.g., reads, writebacks, streaming writes, snoops). In addition, the host is responsible for managing coherency for all device-attached memory that is exposed to the host using *CXL.mem*.

Today's chip designers are facing a variety of interconnect protocols that may appear similar at first glance but serve distinct purposes in modern chiplet-based architectures. While CXL has gained significant attention for its asymmetric approach that simplifies accelerator design by centralizing coherency management in the host processor, symmetric protocols like Intel's UPI and the newer CCIX standard will likely play increasingly big part as we move deeper into the multi-chiplet era. UPI serves as a specialized processor-to-processor interconnect, whereas UCIe provides a broader, standardized foundation for inter-chiplet communication within a package. Looking ahead, as system design increasingly embraces disaggregation and chiplet-based architectures, symmetric protocols that enable distributed coherency management will become essential for building more flexible and scalable systems, complementing solutions like CXL.

2.4 Summary

This chapter introduced the fundamental concepts and technologies critical to modern multicore and multi-chip system architectures. First, we went over why modern processors use caches and what they are. We examined cache coherence mechanisms, analyzing protocols like MSI, MESI, and CHI that ensure data consistency across multiple cores. The discussion traced the evolution from traditional monolithic designs to heterogeneous and chiplet-based architectures, driven by the limitations of transistor scaling. We explained the main mechanisms enabling efficient data sharing in contemporary processors. Finally, we looked at two protocols of interest, CHI and CCIX, and gave context on the interconnect protocol landscape, as well as why we think that symmetrical coherent protocols might prevail in the future.

CHAPTER 3

THE GEM5 SIMULATION FRAMEWORK

In this chapter, we detail gem5, the simulation framework that serves as the foundation for the experimental work in this thesis. We begin by examining different simulation platforms and methodologies in computer architecture research, highlighting their respective strengths and trade-offs. We then expand into gem5's architecture, focusing particularly on its memory system simulation capabilities through the Ruby subsystem and the SLICC domain-specific language for implementing cache coherence protocols. Finally, we discuss essential debugging methodologies and tools available in gem5, which are crucial for developing and validating new components and models and have been used extensively during this project.

3.1 Computer architecture simulation platforms

Simulation stands as the de-facto standard methodology for performance modeling in computer architecture research [83]. While analytical models exist, they often fall short in accurately evaluating architectures and micro-architectural designs. This is due to the rising complexity of modern processors, where numerous configurations and implementation details can significantly impact performances. Thus, computer architects rely on simulation-based approaches to assess various designs alternatives and analyze both the performances and power consumption of different models of different parts of modern processors. This heavy use of simulation is reflected in the computer architecture research community, where the majority of published works employs simulators as main evaluation methodology. Multiple computer architecture simulators support various Instruction Set Architectures (ISAs), configuration granularity, and are based on different simulation models. In this section, we discuss the simulation platforms choices and motivate our choice for this thesis.

Simulation platforms can be categorized by their level of detail. We make a distinction between *functional*, *timing*, and *timing and functional* simulators.

- Functional simulators are emulating the target processor's ISA. They focus solely on replicating the architectural behavior without implementing the underlying micro-architectural details. This level of simulation offers several advantages: it enables fast testing of new instruction sets during development, facilitates the analysis of software execution (e.g., memory access patterns), and provides faster simulation speeds compared to other simulator categories [8]. However, the trade-off for this improved performance is the inability to track detailed micro-architectural parameters during the program execution, as these simulators are abstracting the implementation specifics. This limitation comes from their fundamental design choice to emulate only the architectural functionality.

- Timing simulators are used for detailed modeling of the processor's performances. They focus on simulating the micro-architectural aspects of a processor. These simulators generate detailed statistics about the target's behavior. For example, the total of instructions executed, the instructions per cycle (IPC), the execution time, among others. Among timing simulators, we can make the distinction between three different level of simulation granularity and methodology: *cycle-level* and *event-driven*.

  - Cycle-level simulators operate by simulating the target operations for each cycle. As opposed to RTL implementation (i.e., *cycle-accurate* simulators), the cycle-level simulator does not model every single detail about the target architecture. Cycle-level simulation is slow compared to other simulation methods.

  - Event-driven simulators model processor behavior by tracking and processing events that occur at specific points in the simulated time, rather than simulating the system state at every cycle. These simulators maintain an event queue and advance simulation time from one event to the next, processing only the necessary state changes at each event occurrence. This approach can be more efficient than cycle-level simulation, especially for systems where events happen more sporadically, as it skips idle cycles where no changes occur.

- Timing and functional simulator are, in most cases, functional simulators that have timing capabilities. These capabilities can follow different coupling strategies. For example, *execute-in-execute* is a notable approach where instruction execution is synchronized with the pipeline's execute stage. This type of technique provides precise modeling of instruction behavior at the cost of complexity. The gem5 [59] simulator is a simulator that uses this technique. The coupling between

functional and timing components can be implemented in various ways, leading to different simulator architectures.

When looking at simulation platforms, we can also distinguish between two main approaches in regard to handling multithreaded applications: *full-system* simulation and *application-level* simulation. Full-system simulator have the capability to boot an unmodified OS to schedule threads and map them into different processing cores. A full-system simulator simulates all needed I/O devices, memory and network connections needed to boot the OS. This approach is beneficial for architecture exploration but usually comes with a heavy simulation speed tradeoff compared to trace-based simulation. For example, gem5 is a full-system simulator, and has the ability to boot Linux and Android operating systems. On the other hand, application-level simulators run only target application instead of simulating the whole OS.

Simulation platform inputs can also take different forms. First, *trace-driven* simulators operate by processing pre-recorded sequences of execution events from a program (as opposed to executing the program instructions). Traces usually contain detailed information about the instructions, memory address and/or other relevant architectural events captured during a simulated or real execution. This approach offers several advantages: it can easily be reproduced, it is faster since it removes the need for instruction emulation and facilitates the focused analysis of specific behavior (highlighted by our use in Section 5.5). However, input trace files can sometimes become huge and may lead to slower simulations [32]. Trace-driven simulation has limitations, particularly in modeling systems or programs with execution that depends on runtime conditions, since the trace represents only one possible execution path. Thus, they should be avoided for timing-dependent systems and multicore systems [37]. Second, and as opposed to trace-driven simulation, *execution-driven* simulators use binaries of benchmark applications. Unlike trace-driven approaches, these simulators can accurately model the complete system behavior. As such, execution-driven simulation can capture complex interaction in modern processors such as cache coherency or branch prediction. While this approach offers better flexibility and accuracy, it often results in slower simulation speeds.

For this project, we choose gem5 as target simulation platform. In the following section, we discuss the reasons of this choice, and we give a description of the key aspects of the gem5 simulator.

3.2 Overview of gem5

gem5 [59] is a flexible and highly configurable full-system event-driven framework, which is extensively used in both academia and industry. Although gem5 is an event-driven simulator, it can keep track of events on a cycle-to-cycle basis, which makes its accuracy comparable to a cycle-level simulator. The gem5 simulator is the result of the combination of two open-source simulation projects: M5 [18], from which gem5 gets its CPU models from, and GEMS [61], which provided a highly detailed memory system. gem5 supports four main CPU models: *AtomicSimple*, *TimingSimple* that are single-cycle CPU models without pipelined structures, and *Minor* (in-order) and *O3* (out-of-order). TimingSimple models the timing of memory accesses, but AtomicSimple does not. Instead, AtomicSimple uses atomic memory accesses and uses latency estimates from the atomic accesses to estimate overall cache access times. Moreover, gem5 has a kernel-based virtual machine CPU model (KVM). This model allows, when running the simulation in full-system mode, to run the simulated application on the host's hardware, significantly improving simulation speeds. gem5 has the following additional capabilities:

- Two **memory systems**: First is the gem5 *classic* memory. This is an easily configurable fast memory system. Second is the *Ruby* subsystem. Ruby allows to configure the caches, network and protocols with great details, but is more complex to set up.

- Two **simulation modes**: First is *System-call Emulation* (i.e., syscall emulation) (SE) which emulates system services, eliminating the need to boot an operating system. This is the fastest simulation mode. Second is the *Full System* (FS) mode, which contains a bare-metal environment, and allows to boot an unmodified OS to run applications.

- ISA support: ALPHA, MIPS, SPARC, Power, ARM and x86.

- OS support: Linux and Android.

These different capabilities make gem5 a very flexible simulation platform ranging from fast and low precision to slow and very accurate simulations. Figure 3.1, from [17] shows the spectrum of precision and simulation speeds depending on the models and modes used. However, all the combinations showed are not possible. gem5 is a very active project with multiple research projects adding to the existing simulator. Moreover, gem5 also sees contribution from private companies like a GPU model [13, 39] added by AMD and the Arm AMBA CHI protocol added by ARM.

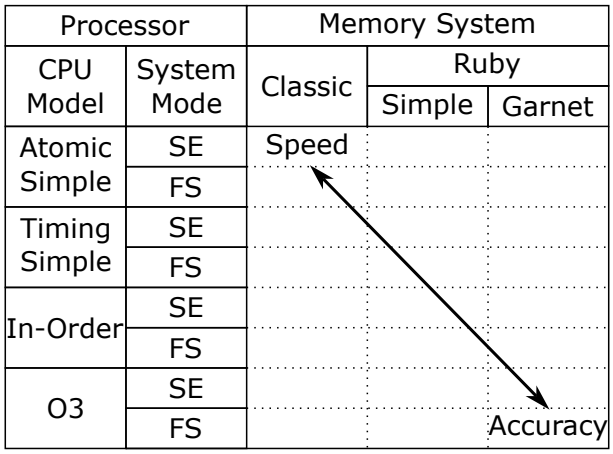


Figure 3.1: gem5 precision depending on models and modes. Source: [17]

3.3 Simulation workflow in gem5

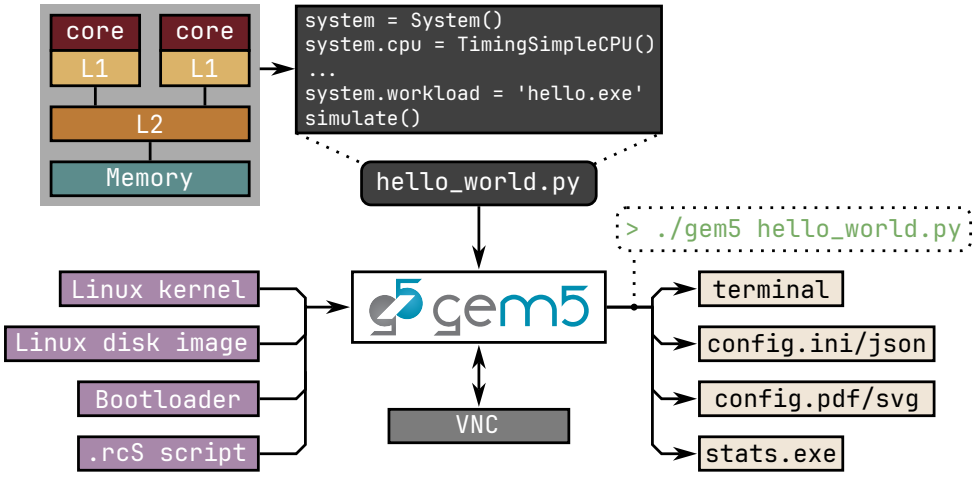


Figure 3.2: gem5 simulation input, outputs and runtime interface.

gem5's architecture is based around a dual-language implementation that leverages both Python and C++, with an event-driven approach. At its foundation, the simulator employs *SimObjects*, a unified abstraction that exists as both C++ and Python objects. This dual representation enables a very flexible and expressive architecture description through Python. The simulator C++ kernel is fast and compiled to target specific architectures (e.g., ARM, x86, etc.) and can be configured with different debug flags and optimization settings.

Figure 3.2 shows the gem5 simulation flow, with the input files, output files and in-simulation interfaces. On the left side, in yellow, are the input files necessary to run full-system simulations. They include all the components to boot the OS. The `rcS` script is used to describe the set of commands that are executed in the simulated machine once the OS is booted. On the top part, we can see the python script that describes the simulated architecture. This script is then called on the right side when running

the gem5 binary. On the right side, we can see a part of the many output files of the gem5 simulator. `stats.txt` is the main output statistics file. `config` files describe, in different formats, the simulated architecture in great details. Finally, gem5 generates graphic files of the simulated architecture. Graphic files are a good way to quickly verify if the architecture is correct, although the shown connections between models are sometimes not very accurate.

The simulation workflow in gem5 follows a structured process, as illustrated by Figure 3.2. The workflow begins with the user selecting their target system configuration (for example, a dual-core processor with a two-level cache hierarchy, depicted on Figure 3.2′s top-left side). Following this selection, the user develops a Python configuration script that describes the target system architecture by creating and connecting SimObjects, gem5 modeling components. These SimObjects are highly configurable, allowing users to fine-tune their behaviors through various parameters adjustments via Python objects attributes. Apart from system configuration, this script serves as the primary interface for simulation control, allowing operations such as simulation initialization, termination, checkpointing, and others. Then, the simulation process is instantiated when the gem5 binary, functioning as a Python interpreter, processes the configuration script. This phase instantiates the described system components and processes the simulation according to the script commands. Upon completion, the simulation generates two main types of outputs: the simulated application's result (i.e., standard output or full-system serial terminal output) and statistical data collected from the SimObjects.

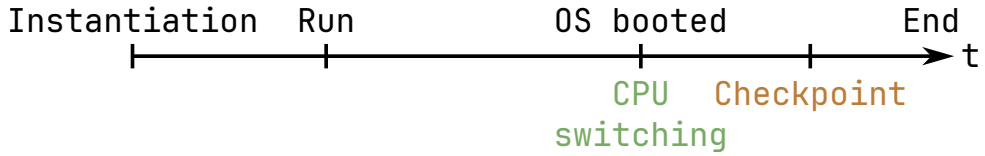


Figure 3.3: gem5 full-system simulation timeline with optional CPU switching and checkpointing.

Figure 3.3 shows the timeline of a full-system simulation in gem5. First, objects are instantiated from the python script. Then, the simulation starts. This is a full-system simulation, so the first simulation phase is dedicated to booting the OS. Different options exist in order to mitigate this very time-consuming step. *CPU switching* consists in using a faster but less precise CPU model for the first parts of the simulation (i.e., booting) and switch to a slower but precise CPU model when entering the Region of Interest (ROI). *Checkpointing* allows the user to create snapshots of the simulation to later-on resume to. This is especially useful when running extensive benchmark applications and debugging an error that happens late in the run.

The Figure 3.3 shows a timeline with an abstract time unit. However, in gem5, there are two main units of time: tick and cycle. *Ticks* are the main unit of time in gem5. They are the smallest unit of simulated time. It represents the granularity at which gem5 models the passage of time in the simulation. All events in gem5, such as memory accesses or processor cycles, are scheduled and executed at specific tick values. The actual duration of one tick in real-world terms depends on the user-assigned value of a tick. By default, gem5 uses $1tick = 1ps(10^{-12}s)$ for high precision, but this can be adjusted for different simulation needs. In the Ruby subsystem of gem5, *cycles* are the fundamental unit of simulated time used to model the behavior of caches, memory controllers and interconnects. Time in Ruby progresses in discrete cycles, which are aligned with the operations of its hardware components. Ruby internally uses cycles for all timing-related operations like scheduling events, tracking message latencies determining when a resource becomes available.

3.4 The Ruby memory subsystem

Amongst gem5 cache systems, *Ruby* puts an emphasis on precision. Ruby models cache coherence protocols with high fidelity. Ruby enables user-defined cache coherence protocols out of the box. When using Ruby, users can also choose between a simple network model and the more detailed Garnet model [2], which offers cycle-level details for the on-chip network. Ruby can be used with any ISA, CPU models or memory controller models. It uses the SLICC domain-specific language to enable users to define new coherence protocols with high fidelity. In vanilla gem5, there are more than 10 protocols including GPU-specific, region-coherence protocols and teaching protocols. Ruby shares the same replacement protocols, the same port system to send requests into and out of the cache system, and the same flexible DRAM controller models as the classic cache models from gem5. This memory subsystem plays a central role in this project as its features enable the development of our modeling methodology. The main advantages of Ruby are as follows:

- It allows to control every minute detail about the cache system and coherence protocol.
- The SLICC language and its compiler are generating a lot of boilerplate code, which allows the user to focus solely on the coherence protocol behavior.

The Ruby subsystem also comes with disadvantages. Following are some of its main drawbacks.

- It adds in complexity of configuration. Its configurability is a strength, but makes

the setup process daunting for new users. Defining a cache hierarchy, coherence protocols, and interconnect requires an in-depth understanding of the system. Moreover, errors in configuration will often lead to simulation crashes or erroneous results.

- There is only a limited and scattered documentation for Ruby. Learning comes from source code and examples, which can be time-consuming. Furthermore, learning SLICC, a domain specific language, for describing cache-coherence protocols is difficult.
- Ruby adds a performance overhead to simulation that are already slow.
- Debugging Ruby is harder compared to the classic memory models of gem5 due to the complex interactions of the coherence protocol and the limited debugging support for SLICC.

3.4.1 Protocol-agnostic components

Some components of Ruby are not dependent on the protocol. They are essential when modifying or developing protocols and will be mentioned further in this thesis. The main components of interest are the *sequencer*, *replacement policies* and *memory controller*. The sequencer's purpose is to inject requests to and receive responses from the memory subsystem. The sequencer injects load, store, atomic and flush requests. Listing 3.1 shows a sample of a generated trace (see Section 3.6.2). This sample shows the first request received by the cache `l1i`. The `event` label shows that the request comes from the sequencer: `AllocSeqRequest` and that the request is a `Load`. Debug flags and traces are described in more detail in Section 3.6. Once the request is serviced by the memory system, the responses are sent back to the sequencer, that interfaces the CPU. Each CPU has a sequencer. The replacement policies of each cache are also independent of the protocols. The two replacement policies that are available are LRU and pseudo-LRU. Finally, the memory controller is responsible for servicing requests that access main memory and is also independent of the protocol used. Certain protocols, like CHI, use a special controller (i.e., Subordinate Node) to communicate with memory controllers.

```
**** REAL SIMULATION ****
500: system.cpus00.l1i: [Cache_Controller 0], Time: 1, state: I, event: AllocSeqRequest,
    addr: 0x4140
500: system.cpus00.l1i: executing AllocateTBE_SeqRequest
500: system.cpus00.l1i: CHI-cache-actions.sm:149: SeqRequest type: IFETCH
500: system.cpus00.l1i: next_state: I
500: system.cpus00.l1i: [Cache_Controller 0], Time: 1, state: I, event: Load, addr: 0
    x4140
500: system.cpus00.l1i: executing Initiate_Request
500: system.cpus00.l1i: executing Initiate_LoadMiss
```

```
500: system.cpus00.l1i: executing Profile_Miss
500: system.cpus00.l1i: executing Pop_ReqRdyQueue
500: system.cpus00.l1i: executing ProcessNextState
500: system.cpus00.l1i: next_state: BUSY_BLKD
```

Listing 3.1: Sequencer load request in a debug trace

In order to avoid the use of full benchmark application, gem5 offers the *traffic generator* component. The traffic generator is a synthetic workload injection component designed to have maximum control over the request and access patterns in the memory subsystem. This tool allows to generate configurable memory requests with control over parameters such as timing, address range, request types (e.g., read, writes, flush, etc.). It enables stress-testing of memory hierarchies and network-on-chips. It also facilitates the detailed evaluation and latency measurements without relying on full application workloads. Moreover, the traffic generator (or *Tgen*, *TrafficGen*) serves critical roles in behavioral and functional debugging by allowing to deliberately generate and analyze problematic memory access patterns. The traffic generator uses custom input traces to inject request into the memory it is connected to. We use them extensively during this project to control the level of complexity of the simulation, debug our simulation and get insight into cache coherence protocol.

3.4.2 Life of a memory request in Ruby

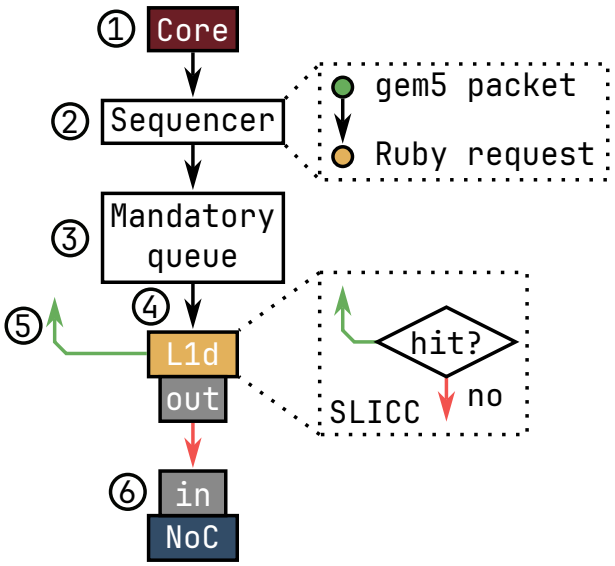


Figure 3.4: Life of a memory request in Ruby, from the core to the network.

As mentioned before, sequencers can inject requests in the cache memories. In this section we will describe the life of a memory request using the Ruby subsystem. Figure 3.4 shows the different steps that a memory request undertakes from its injection by the core (or thread context) to its possible injection in the network model.

- First, in ①, the request is created by a core. The request is a *gem5 packet*.

- In ②, the request is passed on to the sequencer via a *Ruby port*. There is a one-to-one ratio of ruby ports and sequencer to cores (or thread context). The sequencer translates the request into a Ruby message, which is the format interpreted by all SLICC-defined Ruby components.

- In ③, the request is injected in the *mandatory queue*. This queue is connected to a controller.

- In ④ the request is dequeued by the L1d controller. Then, the inbound request goes through the SLICC state machine of the controller. For simplification purposes, we show the internal processing as a hit condition for the cache line. If the cache has the line, it is a hit, and the L1d sends back the data to the sequencer, as depicted in ⑤. Otherwise, if there is a cache miss, the L1d controller will generate a request to the lower level of the hierarchy and send it. The requests are transmitted through *Ruby MessageBuffers*, as shown in ⑥.

- If the request enters the network model, it is then routed using the IDs of each controller, that are generated when the Ruby system is instantiated at the beginning of the simulation. More details about the routing are found in Section 4.2.

3.5 SLICC: cache coherence protocol specification

The SLICC (Specification Language for Implementing Cache Coherence) domain-specific language enables the definition of complex cache-coherence protocols inside Ruby. Cache coherence protocols can be described using FSMs. This technique has been used in the past to debug them [73]. These state machine behaviors are described, in Ruby, through SLICC, which can enforce realistic hardware constraints in its specifications. SLICC handles protocol specification, as well as an integrating tool for various memory model components. To ensure accurate hardware modeling, SLICC allows the implementation of specific limitations. For example, the user can specify a number of cycles for specific transitions, allowing to calibrate their timings. These constraints help maintaining fidelity to real hardware behavior in the simulated environment. The Figure 3.5 shows a simple architecture with a network and caches. Also shown is the state machine described in SLICC, that processes the events from the input queue from the network and sends expected messages back to the network on the right-side queue.

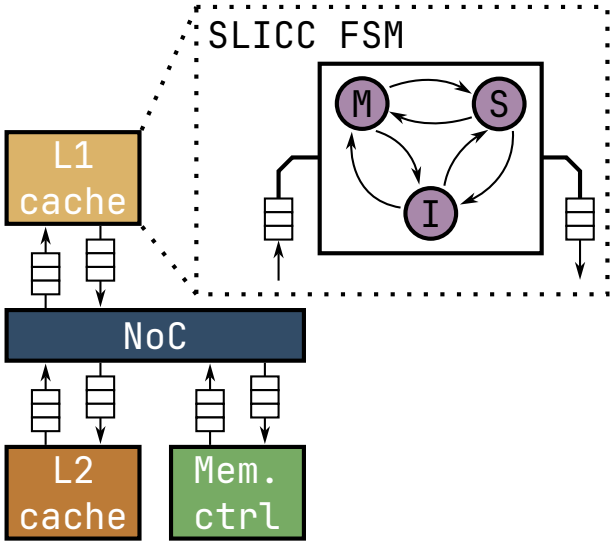


Figure 3.5: Simple architecture with a network-on-chip and caches, as well as a simplified view of the FSM of the L1 cache described in SLICC.

3.5.1 Compilation

As previously mentioned, the gem5 simulator is compiled into a binary from C++. When gem5 is compiled, the SLICC compiler is called first. This compiler, written in Python, will generate C++ code for each controller of the protocol. The compiler takes specific files that describe which controllers are part of the protocol, as input. These files use the `.slicc` format. For example, the `CHI.slicc` file contains, amongst others, the `CHI-cache.sm` file reference, which is the SLICC definition of the cache controller of the CHI protocol. This indicates to the compiler that this file is part of the CHI protocol. The files that constitute a protocol are the definitions of each controller and the message that are exchanged between them. The syntax of SLICC scripts resembles the one of C++, but uses syntactic cues of the compiler. The SLICC and gem5 compilation flow is shown in Figure 3.6. On the left, controllers are defined using the SLICC language. Then, these files are compiled by the SLICC compiler, after being listed as part of the same protocol in the `.slicc` file. The SLICC compiler generates the C++ files as well as additional boilerplate code necessary for any protocol to function. Finally, gem5 is compiled and linked into an executable.

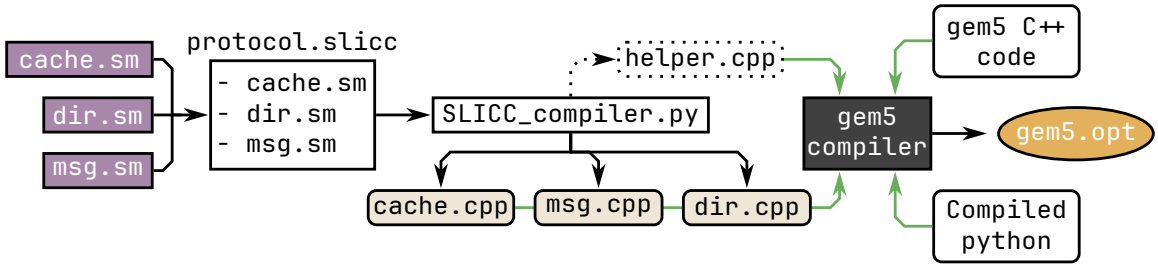


Figure 3.6: SLICC and gem5 compilation flow.

3.5.2 SLICC data types

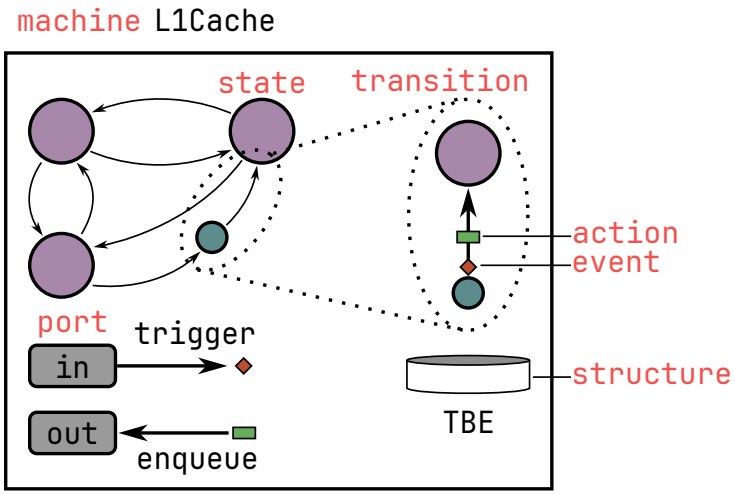


Figure 3.7: SLICC state machine main components with SLICC keywords (in red).

In this section we will go over the different data types that are used to describe state machines in SLICC. The MI protocol, available in vanilla gem5, will be used, for its simplicity, as example. The protocol can be found in `src/mem/ruby/protocol`. The Figure 3.7 shows a SLICC state machine main components and the keywords used when programming them. Each controller (i.e., state machines) are declared with the `machine` data type. Here, users can declare different main variables, inputs and outputs of the controller. Listing 3.2 shows a part of the definition of the cache controller of the MI protocol. The `Cycles` type corresponds to a number of cycles that can be used to delay the sending of messages. Moreover, the `MessageBuffers` describe the inputs and outputs of the controller. These inputs and outputs have attributes that describe their behavior, for example if they are to or from the network.

```
machine(MachineType:L1Cache, "MI Example L1 Cache"):
    Sequencer * sequencer;
    CacheMemory * cacheMemory;
    Cycles cache_response_latency := 12;
    // Network IOs
    MessageBuffer * requestFromCache, network="To", virtual_network="2", vnet_type="
        request";
    MessageBuffer * responseFromCache, network="To", virtual_network="4", vnet_type="
        response";
    [...]
{Rest of the declaration}
```

Listing 3.2: Declaration of the MI cache FSM using SLICC.

An important part of the state machine is its states. The `state_declaration` keyword allows to declare the states that the machine can reach. Listing 3.3 shows a sample of the state declaration of the MI cache. This declaration also specifies attributes for each state. States can be either *stable* or *transient*. A cache line is in a stable state if,

without any other external events, it would remain in that state. Transient states, on the other hand, are multiple states that form a path that needs to be finished in order to reach another stable state. In the Listing 3.3 state I is stable, and state MI is transient.

```
state_declaration(State, desc="Cache states") {
    I, AccessPermission:Invalid, desc="Not Present/Invalid";
    MI, AccessPermission:Busy, desc="Modified, issued PUT";
    [...]
}
```

Listing 3.3: Declaration of the states of the MI cache controller in SLICC.

The events are declared using the `enumeration` keyword. Events trigger transitions from one state to another. Listing 3.3 shows a part of the enumeration of events for the MI cache controller. Events are caused by input messages but can also result from internal processing. They are later used to declare transitions.

```
enumeration(Event, desc="Cache events") {
    // From processor
    Load,   desc="Load request from processor";
    Ifetch, desc="Ifetch request from processor";
    Data,   desc="Data from network";
    [...]
}
```

Listing 3.4: Declaration of the events of the MI cache controller in SLICC.

When describing a controller, the user might need to describe additional structures. For example, the Transaction Buffer Entry (TBE), is a commonly used structure, used to store useful metadata to service a request (e.g., original requester, type, address, etc.). The `structure` keyword is used to declare them. Listing 3.5 shows the declaration of the TBE of the MI cache controller using a structure.

```
// TBE fields
structure(TBE, desc="...") {
    State TBEState,     desc="Transient state";
    DataBlock DataBlk,  desc="Data for the block, required for concurrent writebacks";
}
```

Listing 3.5: Declaration of the TBE using a structure.

Transitions of the state machine are triggered by events. The state machine has inputs and outputs buffers that allows it to communicate with the network, and by extension to the other controllers of the system. Events can be triggered internally, but they are more commonly triggered by input ports. To declare the behavior of the controller when receiving a message in each port, the SLICC `in_port` keyword is used. Listing 3.6 shows the declaration of the `responseNetwork` port input. First, the declaration needs three arguments; the name of the port, the type of message that the port

handles (here *ResponseMsg*) and the name of the queue that this port is feeding. Second, the `if` statement checks if the port is ready, meaning that there is an inbound message in the port. Third, the `peek` method is used. This method is important and is a getter function for the inbound message. It allows to access the input message's fields inside its brackets. As seen in the listing, the address of the message is accessed with `in_msg.addr`. This is possible because of the peek method. Lastly, if the input message is of the right type, the `trigger` method is used to trigger the `Data` event.

```
in_port(responseNetwork_in, ResponseMsg, responseToCache) {
    if (responseNetwork_in.isReady(clockEdge())) {
        peek(responseNerwork_in, ResponseMsg, block_on="addr") {
            Entry cache_entry := getCacheEntry(in_msg.addr);
            TBE tbe := TBEs[in_msg.addr];
            if (in_msg.Type == CoherenceResponseType:DATA) {
                trigger(Event:Data, in_msg.addr, cache_entry, tbe);
            } else {
                error("Unexpected message");
            }
        }
    }
}
```

Listing 3.6: Declaration of the "response from network" port.

In the FSM, events trigger transitions. SLICC provides the `transition` keyword to specify the state machine transitions behavior. Listing 3.7 shows the declaration of two transitions. In this example, the first transition goes from state I to state IM and is triggered by the Store event. When this transition occurs, the actions listed (v_allocate, i_allocateL1CacheBlock, a_issueRequest, etc.) are executed. Actions are executed, in order, atomically, meaning that no other event can happen until all actions have been executed. A set of events and initial states can be provided, as shown in the second transition of Listing 3.7. In this particular example, both initial states IS and IM will transition if there is one of the two events. The final state being omitted, it means that the transition goes back to its initial state (i.e., IS transitions to IS and IM transitions to IM).

```
transition(I, Store, IM) {
    v_allocateTBE;
    i_allocateL1CacheBlock;
    a_issueRequest;
    p_profileMiss;
    m_popMandatoryQueue;
}
transition({IS, IM}, {Fwd_GETX, Inv}) {
    z_stall;
}
```

Listing 3.7: Declaration of transitions using SLICC.

As shown previously, when a transition occurs, a set of actions is executed. In SLICC, they are declared using the `action` keyword. Actions are where the processing happens. Listing 3.8 shows the declaration of an action using SLICC. Action declarations have three arguments. First is the name of the request (e.g., `a_issueRequest`). Second is a shorthand name that is used to generate the HTML table of the protocol. Third is the description of the action. Inside the action, the `enqueue` function is used. This function enqueues a new message in a selected port. In this example, a message of type `RequestMsg` is enqueued in the `requestNetwork_out` output port. This message will be sent after a number of cycles informed in the third argument (i.e. `issue_latency`). The enqueue function implicitly declares the `out_msg` variable, which makes it possible to fill the necessary fields of the outbound message. It works similarly to the peek function but for the output message, as opposed to the input message.

```
action(a_issueRequest, "a", desc="Issues a request") {
    enqueue(requestNetwork_out, RequestMsg, issue_latency) {
        out_msg.addr := address;
        out_msg.Type := CoherenceRequestType:GETX;
        out_msg.Requestor := machineID;
        out_msg.Destination.add(mapAddressToMachine(address, MachineType:Directory));
        out_msg.MessageSize := MessageSizeType:Control;
    }
}
```

Listing 3.8: Declaration of an action using SLICC.

On top of this, SLICC provides additional helper functions. One special function that will be used later in this thesis is the `stall_and_wait` function. This function allows to stall an input port for a certain address. In the meantime, the port can still receive messages for other addresses. This is useful to handle unwanted events and manage serialization. A complicated aspect of this method is that ports need to be awakened with the `wakeUpStalled` method in order to unblock the stalled messages.

3.6 Debugging techniques in gem5

Developing a new controller (i.e., state machine) for an already existing cache coherence protocol using SLICC is a tedious task. The new controller must sit within the protocol and interact with other controllers without disturbing the coherence or interfering with the behavior of the protocol. Moreover, SLICC code can rapidly grow in size and complexity. For example, the simplest MI protocol presented in Section 3.5.2 has a cache controller code of around 500 lines. Our controller implementation presented in Chapter 5 is approximately 3000 lines. Thus, when working with gem5 and

developing new coherence protocols using SLICC, debugging is important. gem5 development requires compile-time and run-time debugging as well as behavior debugging techniques. This section will focus on the debugging aspects of gem5. First, the focus will be on how gem5 functionality like trace injection can be used to facilitate understanding parts of the code. Second, this section will explain trace debugging with output traces and debug flags. Then, this section will make a point about deterministic simulations, important time considerations and checkpointing.

3.6.1 Trace injection

In gem5, CPU models are instantiated and inject memory requests to the memory system through the sequencers (as explained in Section 3.4.1). On top of being able to choose from the available CPU models, the user can also use *traffic generators* (often called *tgens*). These models replace the CPU with a transparent model for trace injection. Tgens will read from a custom trace file and inject the traffic to the controller they are connected to. Listing 3.9 shows the declaration of traffic generators in the Python architecture scripts. The traffic generators are each reading trace files (in the example, read from the `m5out` directory). This method allows to have complete control on timing and type of request. This is useful to generate custom access patterns in the memory system. We use this technique in multiple occasions during this thesis project. First, we use this method to generate special traffic in the memory system, in order to understand the behavior of certain controllers when treating special message types. Second, we use this technique to calibrate a new controller by comparing it with real hardware. In this case, the custom trace injection allowed us to reproduce the same traffic as the one observed on the hardware on the simulation (see Section 5.9). Third, we use it to simplify and control the memory traffic during the development phase (see Section 5.5).

```
tgens=[
    TrafficGen(
        config_file=f"./m5out/lat_mem_rd_{i+1}.cfg",
        progress_check="10s",
    ) for i in range(np)
],
```

Listing 3.9: Declaration of traffic generators in Python.

3.6.2 Trace debugging

```
AccessPermission getAccessPermission(Addr addr) {
    if (respondsTo(addr)) {
        TBE tbe := TBEs[addr];
        if (is_valid(tbe)) {
            DPRINTF(RubySlicc, "%x %s,%s\n", addr, tbe.state,
                Interface_State_to_permission(tbe.state));
```

```
6             return Interface_State_to_permission(tbe.state);
7 [...]
```

Listing 3.10: Declaration of two debug flags inside the SLICC code.

```
build/X86/gem5.opt --debug-flags=RubySlicc --debug-start=420000 configs/example/test.py
```

Listing 3.11: Command line argument to execute gem5 with a debug flag.

For debugging purposes, gem5 includes debug flags. Debug flags are set in the code and allow to print useful information during execution on the terminal. Listing 3.10 shows two debug flags declaration inside SLICC code and Listing 3.11 shows the command line argument to execute gem5 with a selected debug flag. Debug flags have two main arguments. First is the name of the debug flag, second is the content of the printout. As seen in the example, the message can contain variable printouts that are added after the first argument. When executing gem5, it is possible to add the `--debug-flags` and add one or multiple debug flags to be used. As simulation grow, these debug flags can generate too much output and considerably slow down the simulation. It is important to use the `--debug-start` argument, which allows to enter the time at which the debug flags are going to be enabled. gem5 allows the user to add new custom debug flags. In order to not clutter our outputs, we have declared a new debug flag called `RubyC2c` and used it throughout our project.

```
8000: system.cpus0.l2: [Cache_Controller 6], Time: 16, state: I, event: AllocRequest,
    addr: 0x10000000
8000: system.cpus0.l2: executing AllocateTBE_Request
8000: system.cpus0.l2: next_state: I
8000: system.cpus0.l2: [Cache_Controller 6], Time: 16, state: I, event: ReadShared, addr:
     0x10000000
8000: system.cpus0.l2: executing Initiate_Request
8000: system.cpus0.l2: executing Initiate_ReadShared_Miss
8000: system.cpus0.l2: executing Allocate_DirEntry
8000: system.cpus0.l2: executing Profile_Miss
8000: system.cpus0.l2: executing Pop_ReqRdyQueue
8000: system.cpus0.l2: executing ProcessNextState
8000: system.cpus0.l2: next_state: BUSY_BLKD
8000: system.cpus0.l2: [Cache_Controller 6], Time: 16, state: BUSY_BLKD, event:
    TagArrayRead, addr: 0x10000000
8000: system.cpus0.l2: executing Pop_TriggerQueue
8000: system.cpus0.l2: executing TagArrayRead
8000: system.cpus0.l2: executing ProcessNextState_ClearPending
8000: system.cpus0.l2: next_state: BUSY_BLKD
9000: system.cpus0.l2: [Cache_Controller 6], Time: 18, state: BUSY_BLKD, event:
    ReadMissPipe, addr: 0x10000000
9000: system.cpus0.l2: executing Pop_TriggerQueue
9000: system.cpus0.l2: executing ReadMissPipe
9000: system.cpus0.l2: executing ProcessNextState_ClearPending
9000: system.cpus0.l2: next_state: BUSY_BLKD
9000: system.cpus0.l2: [Cache_Controller 6], Time: 18, state: BUSY_BLKD, event:
    SendReadShared, addr: 0x10000000
9000: system.cpus0.l2: executing Pop_TriggerQueue
9000: system.cpus0.l2: executing Send_ReadShared
```

```
9000: system.cpus0.l2: executing Profile_OutgoingStart
9000: system.cpus0.l2: executing ProcessNextState_ClearPending
9000: system.cpus0.l2: next_state: BUSY_INTR
11500: system.ruby.hnf.cntrl: [Cache_Controller 9], Time: 23, state: I, event:
    AllocRequest, addr: 0x10000000
11500: system.ruby.hnf.cntrl: executing AllocateTBE_Request
11500: system.ruby.hnf.cntrl: next_state: I
11500: system.ruby.hnf.cntrl: [Cache_Controller 9], Time: 23, state: I, event: ReadShared
    , addr: 0x10000000
11500: system.ruby.hnf.cntrl: executing Initiate_Request
11500: system.ruby.hnf.cntrl: executing Initiate_ReadShared_Miss
11500: system.ruby.hnf.cntrl: executing Allocate_DirEntry
11500: system.ruby.hnf.cntrl: executing Profile_Miss
11500: system.ruby.hnf.cntrl: executing Pop_ReqRdyQueue
11500: system.ruby.hnf.cntrl: executing ProcessNextState
11500: system.ruby.hnf.cntrl: next_state: BUSY_BLKD
```

Listing 3.12: Sample of a trace generated with the RubyGenerated debug flag.

Among the numerous existing debug flags, two stand out as the most useful to debug SLICC-defined controllers. First is the `RubySlicc` debug flag. This debug flags prints out the resource usage of the SLICC controllers. For example, the number of TBE used allows to detect possible deadlocks. Second, the `RubyGenerated` debug flags prints out all the transitions of the state machines of the system. On top of that, each transition shows the event that triggered it, the initial and final state, the actions that have been executed during the transition and the time (in ticks and cycles, see Section 3.3) at which it happened. By concatenating these printouts in a file, it is possible to understand a problematic access pattern in detail. Listing 3.12 shows a sample of one of the debug trace generated with the RubyGenerated debug flags. This example has been treated with a python script to include only the transitions for the address `0x10000000`. We can see the L2 cache of cpu0 allocating a ReadShared request. The request is missing since the data is present in the cache. The L2 sends the ReadShared down the memory hierarchy, to the HNF. In the end of the example, we can see the HNF receiving and allocating the request 5 cycles (or 2500 ticks) later.

3.6.3 Deterministic simulation

During this thesis, different CPU models and simulation types are used. Particular models like the KVM CPU (see Section 3.2) are not deterministic. This non-deterministic behavior changes the timing of the simulation (tick numbers) each run. This is particularly impacting when setting an enable time for debug flags (see Section 3.6.2). KVM CPU models are mainly used in full-system simulation, in order to boot the OS rapidly. However, it is impossible to precisely set up a debug trace when using these models. As a way to still gain time during the booting of the OS, while retaining a deterministic behavior, the user can use the NONCACHING CPU model.

3.6.4 Time considerations

The gem5 simulator's simulation times can range from short to extremely long depending on test applications, modeled architecture and simulation modes. Simulations using syscall emulation mode are usually manageable (i.e., seconds to minutes). When using full system, simulation times can rapidly become unmanageable. For example, our full-system simulations presented in Section 5.10 are taking upwards to 15 hours to successfully run. When running a long simulation with a new controller in the system, a corner-case access pattern might occur during the execution of the application at the 10th hour. In order to use debug traces to understand the behavior issues that led to the error, the user needs to run the simulation again with the debug flag enabled and with the right debug start parameter to avoid heavy trace files.

In order to avoid multiple simulation runs to debug unsupported access patterns and behavior problems when testing a new controller, the user can use *checkpointing*. Checkpointing allows to record snapshots of the simulation to resume the simulation from. Figure 3.8 shows the time savings of using checkpointing in a full-system simulation timeline. Even without being in a debugging scenario, checkpointing can be used right after the OS has been booted to avoid waiting for the booting each simulation iterations.

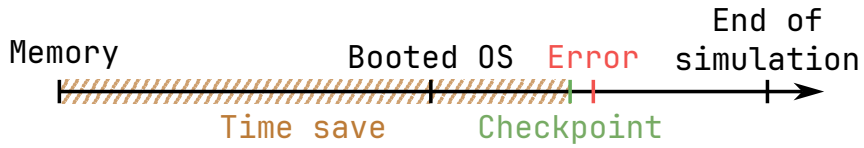


Figure 3.8: Checkpoint time savings in the simulation timeline.

To generate a checkpoint, the caches need to be flushed. This means that the protocol used for the memory subsystem need to support *FLUSH* instructions. However, not all protocols in gem5 support flush instructions, and by extension checkpointing. This is the case for the CHI protocol that is used in this thesis implementation. As such, we do not rely on checkpointing.

Figure 3.9 sums up the different simulation modes available. Out of these simulation modes, our project supports system-call emulation and some full-system options. As previously explained, the proposed solution of this project does not allow the use of checkpointing. In order to reduce the OS booting time, we use CPU switching when running full-system simulations. We can use the KVM CPU models to greatly reduce the booting times (to less than a minute). However, as mentioned, KVM does not allow deterministic simulation times, which makes debugging complicated. When debugging, we use NONCACHING_SIMPLE CPU models instead, which allows to boot the OS in less than 2h.

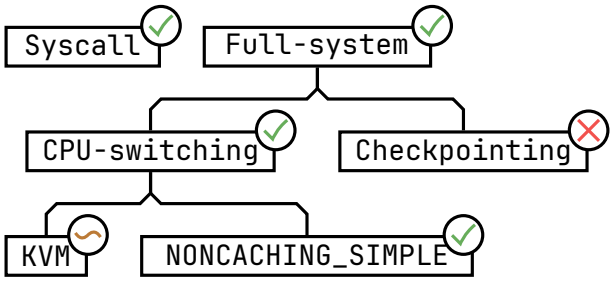


Figure 3.9: Different simulation modes of gem5. Checkmarks show supported simulation modes and red cross shows unsupported modes.

3.7 Summary

This chapter presented gem5, the simulation framework used throughout this thesis. We began by discussing various simulation approaches in computer architecture research and motivated our choice of gem5. We then explored gem5's architecture in detail, particularly focusing on the Ruby memory subsystem and its cache coherence protocol-definition language, SLICC. The discussion covered both the high-level architecture of gem5 and the detailed mechanisms for implementing new controllers and coherence protocols. We examined how Ruby processes memory requests and explained the key components that enable precise modeling of complex memory hierarchies. Finally, we presented various debugging methodologies available in gem5, from trace injection to deterministic simulation techniques, which are essential for developing and validating new architectural components. These tools and methodologies form the technical foundation for the research presented in the following chapters.

Part II

Contributions

CHAPTER 4

EXTENDING CACHE COHERENCE ACROSS MULTI-CHIP ARCHITECTURES

In this chapter, we present the main challenges of simulating a coherent chip-to-chip interconnect protocol. These challenges are simulator-agnostic and should be solved in order to successfully model the target architecture. As a way to first motivate the subsequent chapters, we first begin by reviewing the state of the art and related works in the field of chiplet-based architectures and multi-chip architecture simulation, which also provides context for the challenges we address. Following this overview, we show how routing inside the multicore chip should be handled. Then, the chapter will focus on how to route messages across chips. Third, we present the challenges of maintaining coherence across chips. We give a practical example of an off-chip read request in an architecture combining the previously mentioned CHI and CCIX protocols. Finally, we also explain the main challenges of managing the chip-to-chip link. For ease of explanation and terminology consistency sakes, the rest of the manuscript will follow the CHI semantic when discussing controllers in a NoC (see Section 2.3.1). This chapter is priming Chapter 5 where we solve these challenges with complete implementation of a C2C controller.

4.1 Related work

4.1.1 Chiplet-based architectures

The first aspect of our work expends on chiplet-based architectures, specifically the interconnect linking them together. This new architecture design approach is following the increasing limits of traditional transistor density and speed [52]. In this context, *heterogeneous architectures* have emerged. They combine multiple CPU core types, sometimes even with different Instruction Set Architectures (ISAs) [90], and specialized accelerators. These multiple components (e.g., specialized accelerators, neural processing units, GPUs, etc.) work together to deliver better-optimized performances across

a wider range of applications [47]. Chiplet-based architectures are building upon heterogeneous architectures by leveraging the performance benefits tightly integrated accelerators over external PCIe cards [36, 81], while offering a modular approach to SoC design. Different functional blocks (i.e., chiplets) are fabricated as separate dies and then integrated on the same die or System-in-Package (SiP). Among other advantages, research has shown that chiplet-based architectures can reduce overall chip costs [68]. Moreover, chiplet-based architectures enable the integration of components from different vendors and for diverse applications into a single SiP [54].

The potential of this approach has caused the industry to actively develop and promote their preferred solution for chiplet integration. Examples include the Universal Chiplet Interconnect Express (UCIe) [82] and the Bunch of Wires (BoW) interface [7]. However, the industry is yet to converge on an adopted standard, leaving uncertainties about which solution will be adopted. Recently, and in response to limitations from PCIe to support cache coherency, the industry has been pushing for new *coherent* interconnect standards. These standards aim to maintain coherency between processors and accelerators. Examples are the Cache Coherent Interconnect for Accelerators (CCIX) [27], the Compute Express Link (CXL) [89] and OpenCAPI [85].

Authors of [86] evaluated CCIX against PCIe when interfacing an ARM-based host with two generations of CCIX-enabled FPGAs. The study provided low-level throughput and latency measurements for accesses and address translation. It was found that smaller reads from FPGA to the host benefit significantly from CCIX, with approximately 33% shorter latency compared to PCIe. However, small writes to the host experience a latency roughly 32% higher than PCIe due to the higher coherency overhead associated with CCIX. In this project, we take inspiration from the CCIX protocol because of ease of access, as well as hardware availability. However, as we are unsure which protocol will emerge as standard, we aim at building a flexible solution rather than focusing on a single protocol.

The future of chiplet-based design lies in further refining interconnection technologies for inter-chiplet communication. [29] presents a new active interposer technology for chiplet-based advanced 3D architectures. The authors have developed a process for fabricating active interposers, allowing for the stacking of chiplets in a 3D configuration. As such, our work is complementary to this push in chiplet interconnect research, but focusing on the modeling aspect, providing a solid modeling framework for this type of research work.

4.1.2 Computer architecture simulation

The second aspect of our work relates to computer architecture simulation. The scientific community has been actively developing open-source simulation platforms [5]. They allow architecture exploration and performance predictions for a wide variety of applications from edge devices to cloud many-core systems. Computer architecture simulation and computer assisted design have become an essential part of the chip design process, as architectures have rapidly gained in complexity. For example, authors of [67, 95] are pushing for automatic generation of cache-coherence protocols for many-core processors and easier automatic Hardware-Level Synthesis (HLS) code generation. This body of work is part of the larger push for easier design and analysis of modern architectures.

Simulation platforms can be differentiated with many different aspects. One of such aspects is the mode of simulation, namely functional and/or timing. Cache simulators can support functional-only simulation, meaning that they do not model the time. They use a statistical model to evaluate performances based on the frequency of events like hit or misses. These simulators are faster. On the contrary, timing simulator model CPUs, interconnect, memory hierarchies, etc. They are beneficial to evaluate the performances trade-offs of complete architectures when running applications. We refer to the survey of Brais et al. [20]. We see that most cache-only simulators can run functional simulation. For example, Cachegrind [65], CASPER [48], vCSIMx86 [51]. However, in order to model our target multi-chip architecture, we need flexibility in the configuration of the cache subsystem and interconnect. More importantly, we need to be able to control the coherence protocol at the message level. For this reason, we turn ourselves toward full-system simulators such as MacSim [53], gem5 [59], Sniper [24], etc.

We focus on the simulators that have the right tradeoff between simulation speed and precision, that allow full-system simulation (booting of an unmodified OS to run the target application benchmarks) or have the level of detail and flexibility required to allow cache-coherence protocol research [41, 59, 71, 78, 88]. The main drawback of these simulation platforms is that they are all built for traditional monolithic networks and lack the additional features for multi-chip simulation. However, within the simulation ecosystem, we identify two areas of related work that are particularly relevant. On the one hand, cycle-accurate NoC simulators such as BookSim2 [49], Noxim [26], Nostrum [60], and Garnet [2], originally designed for NoC simulations in monolithic chips, have been extended for multi-chiplet designs [96]. These extensions are useful for conducting simulations and examining various network topologies and routing

algorithms for given traffic patterns. Our work is complementary, as we focus on providing the functionality to extend coherency across multiple coherent chips and thus produce realistic traffic for these NoC simulators.

On the other hand, cycle-accurate full-system simulators like gem5, with its Ruby CHI memory system, can accurately model monolithic many-core architectures [74]. Some methods coordinate multiple gem5 instances to simulate interconnected chips [21, 62, 79], but this setup requires each instance to run its own OS. In particular, authors of [79] are using gem5's Ruby for its robust CPU models and cache coherence protocols, while SystemC TLM models the die-to-die interface, allowing for flexible co-simulation. This methodology is particularly suited for analyzing multi-chip architectures. However, the framework doesn't allow the full-system simulation of multiple chiplets working under a single OS. Our work provides a simulation framework for coherent chiplet-based architectures.

Authors of [39] investigate the accuracy of GPU simulations using an intermediate language (IL) instead of the machineISA. They demonstrate significant discrepancies between IL and machine-level simulations, showing that ILs abstract away crucial micro-architectural details impacting performance predictions. Ultimately, they demonstrate a 33% error increase when using IL simulations for runtime prediction, indicating the limitations of IL and the necessity of low-level ISA simulations to obtain reliable GPU performance data. This work relies on gem5's GPU model, released by AMD [13] and the open-source ROCm software stack. Our work has used this GPU in instances to study the use of multiple network models in one modeled architecture. Furthermore, our work adds on top of this GPU model, allowing the two modeled chips to be connected.

Recent work by the authors of [66] introduces a simulation framework developed to facilitate the design exploration of multi-chip many-core systems, particularly for communication-intensive applications. This work focuses on scalability. However, the framework is designed for architectures with software-managed coherence and distributed memory. Our work is complementary in that it focuses on hardware-based coherence and on enabling lower-level design and exploration of inter-chip coherence protocols.

Other approaches, such as OpenPiton [12], use a similar strategy but incorporate RTL for faster FPGA emulation. In contrast, our work uniquely enables the simulation of systems consisting of multiple coherent chips running under a single OS. OpenPiton is part of a group of RTL generators. These platforms aim at providing a library of components to build routers and NoCs. The RTL is then used for modeling using

RTL synthesis and analysis tools. OpenPiton and other such RTL generators use verilog [94], while others use VHDL [50], BSV [56, 70] or Chisel [34]. As we believe that RTL generation is a good solution to overcome the complexity of RTL simulation set up, simulation time remain very high. Our work strives to strike a tradeoff with better simulation times while retaining cycle-accurate precision and full control over the coherence protocol.

Authors of [23] introduce LDBR (Load-Balanced Deadlock-Free Routing), a novel routing strategy designed to improve performance in chiplet-based systems. The authors address the limitations of existing chiplet interconnect methods, which prioritize correctness (deadlock prevention) over load balancing. LDBR aims to achieve both by balancing traffic across vertical links connecting chiplets and interposer. The validation of this work has been done using gem5 and the integrated Garnet NoC simulator. It is unclear how the simulation has been set up and how the simulated architecture looks in Garnet's semantic. We can assume that the simulated architecture uses a single Garnet network model. However, the focus of this work is to compare different algorithms in specific baseline topologies. As such, using a single network model to simulate chiplets can work. Our work aims at providing better simulation environment for these types of project, allowing more configurability.

4.2 Intra-chip message routing strategies

In this section, we present a detailed examination of intra-chip message routing. We further discuss the criteria and mechanisms utilized to determine message destinations for various message categories. When a CPU issues a request and the request misses in its private caches, the request is sent down the memory hierarchy, through the NoC. In order to route the message to its destination, its header contains additional metadata. An important part of this header is the destination of the message. Each requester of the system is assigned several *downstream clients*. Each one of these clients is a potential destination for requests emitted by the requester (e.g., next cache level in the hierarchy, HNF, SNF). Downstream clients are set during the booting, by the discovery algorithm. Each controller that receives requests manages, or is assigned to, an address range. Figure 4.1 shows a mesh network-on-chip with a L2, LLC, Dir (Directory) and MemCtrl (Memory Controller). The directory is a downstream client for the L2 and LLC and manages the memory range [10KB, 20KB]. The memory controller is a downstream client for the directory and manages the same memory range. The L2 cache issues a read request to an address in [10KB, 20KB], the destination of the request is the downstream client managing this request, i.e., the directory. The directory determines that to service this request, DRAM must be accessed. The request to the

memory controller is sent with a destination set with the downstream client managing the address.

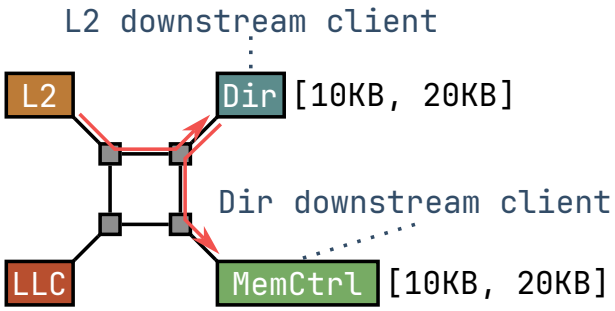


Figure 4.1: Diagram of a mesh network-on-chip with a request from the L2 cache, address ranges and downstream clients.

Routing and message transmission varies with the type of message. Data can be routed using multiple messages when the payload is bigger than the set size of a data message. In the CHI implementation of gem5, data payloads are sent in two messages. When sending snoop messages, the snoop must be routed to the destination within the network-on-chip. This destination is usually a cache controller that shares a cache line, meaning that the line is cached in its cache. The directory, when the need to send a snoop message arises, will fetch the routing destination in a directory entry. This directory entry stores all the sharers of a specific cache line (i.e., address). This information is then used to route the snoop message to destination. Specific snoop requests necessitate *forwarding*. This means that the controller (snoopee) must forward responses to a third controller. For example, in Figure 4.1, when the directory receives the read request, if the LLC cache has a copy of the cache line, the directory will send a snoop request to the LLC controller to ask for the forwarding of the cache line to the L2 controller (original requester). This snoop message will then contain additional routing information: its destination (the LLC) and the forwarding destination (L2).

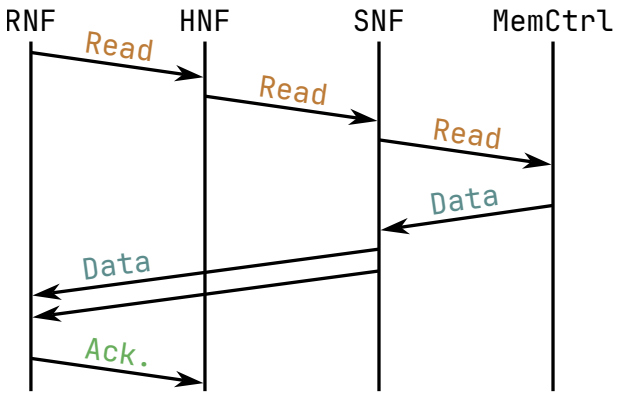


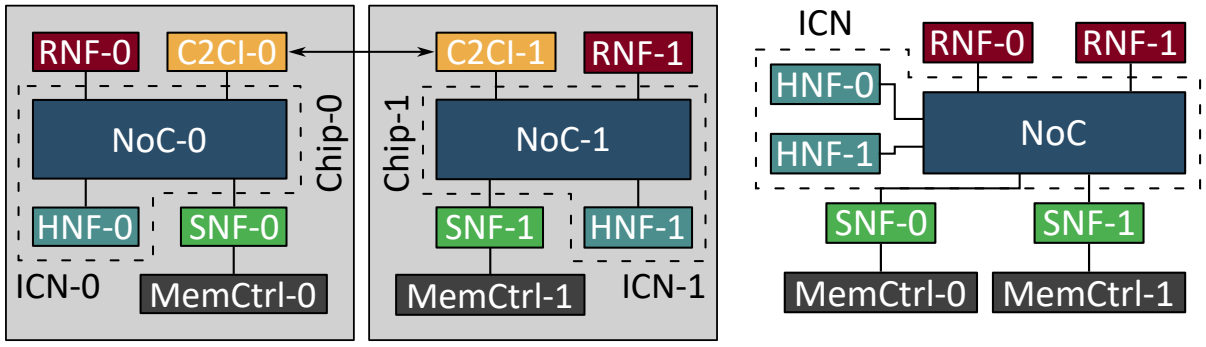
Figure 4.2: Flow diagram of a read request with response data and acknowledgment.

When a request is treated, a number of response messages can be exchanged between controllers. For example, when a RNF issues a read shared and receives the read data, it will issue a `CompAck` acknowledgement message to the HNF that has serviced the request. Thus, controllers need to store the ID of the original requester to be able to send back responses. Figure 4.2 shows a flow diagram of a read request serviced in a network-on-chip. We can see that the SNF sends the data back in 2 messages. In order for the SNF to fill the destination field of the data messages properly, the HNF must provide the ID of the controller the data needs to be forwarded to (i.e., the RNF). The SNF will then store this forward ID while waiting for the data from the memory controller. The ID is then used when sending back the data provided by the memory controller.

When modeling a chip-to-chip coherent interconnect protocol, it is important to understand the internal routing of the chip in order to properly transmit messages and interface the chips. The controller that acts as a gateway to other chips must have an identification for the local network. Moreover, the controller must also have downstream clients to send off-chip requests to. As such, the simulator must provide a unique identifier for each controller of the whole multi-chip architecture. This is the case when working with gem5's Ruby environment.

4.3 Inter-chip message routing strategies

After showing how routing is managed within a network-on-chip, we will, in this section, go over the routing of messages across chips. In this section, we will use the Arm CHI semantic for network modules and protocol (see Section 2.3.1).



(a) (b)

Figure 4.3: High-level C2C (a) and reference (b) architecture with CHI components and C2CIs.

In the following examples, the Chip-to-Chip Interface (C2CI) component refers to a new controller that we have added to the gem5 simulator, along with modifications of the CHI protocol. This extension to the simulator is explained in Chapter 5. The purpose of this new controller is to allow the communication between the two chips. This controller is inspired from the CCIX protocol C2C interface (see Section 4.3.1).

To demonstrate the key differences between a standard on-chip local memory request and one that requires traversing the chip-to-chip interface to access memory on another chip, we refer to Figure 4.3 and the following examples.

When Chip-0 is not connected to Chip-1, it operates following the standard CHI protocol. Thus, when RNF-0 initiates a local memory request, NoC-0 routes the request to HNF-0, which is responsible for managing memory coherency within Chip-0. HNF-0 monitors all caches that store a copy of the requested memory block (i.e., sharers). If the data is not found in any cache, HNF-0 requests SNF-0 to retrieve the data from Chip-0's main memory and to send it to RNF-0. Instead, if the data is present in a local cache, HNF-0 issues a snoop request (e.g., `SnpSharedFwd`) to that cache, which provides RNF-0 with the most up-to-date version of the data. Finally, HNF-0 updates the state of the block after receiving the corresponding acknowledgment messages.

When Chip-0 is connected to Chip-1 and RNF-0 initiates a memory request to a block allocated to Chip-1's main memory, new challenges arise. NoC-0 should check the address range and route all remote memory accesses to the chip-to-chip interface C2CI-0, instead of the local HNF. To achieve this, the local C2CI must be the downstream client of all requesters for the aggregated address ranges that are off-chip.

C2CI-0 then forwards the request to C2CI-1, as it belongs to the chip housing the HNF managing the address of the request. C2CI-1 initiates a new request that NoC-1 routes to HNF-1. In this case, HNF-1 must monitor all caches that may hold a copy of the requested block, both locally on Chip-1 but also remotely on Chip-0. When sharers are remote, HNF-1 must issue snoops that need to be forwarded by the C2CI-1 to Chip-0 and vice versa for the corresponding acknowledgment.

The CHI NoC only routes messages to local components. Thus, for remote accesses, two C2CIs must coordinate to function as a gateway. In the previous example, C2CI-0 serves as the destination of the request in the local NoC-0, while the C2CI-1 is the initiator of the forwarded request in the remote NoC-1. To achieve this, NoC routing tables must be updated to direct remote accesses to the local C2CI, and C2CIs need to incorporate both responder and initiator functionality.

In the previous section, we have explained how requests are routed to their destination within a chip. We have seen that the initial requester of the request is added to the metadata of the message in order to route back the response messages. However, when a request goes off-chip, new metadata is needed to fully service the request. When a request is sent through a C2C interface, the ID of the request's initiator is saved to route back any data / response message.

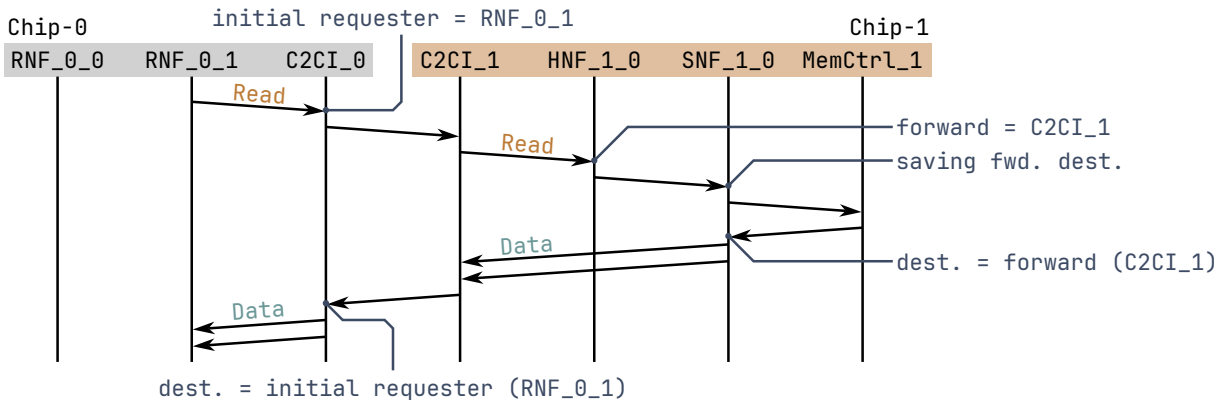


Figure 4.4: Flow diagram of an off-chip read example with useful routing and forwarding information.

In order to properly route back responses to the initial requester, metadata needs to be tracked by controllers. In a chip-to-chip scenario in particular, the controller that acts as a chip-to-chip gateway needs to track in-flight requests for each cache line. Tables are dedicated to storing metadata like the initial requester, the type of initial request, the expected number of responses, etc. Figure 4.4 shows an example use case of different routing information saved in controllers in the context of an off-chip read request. The initiator of the request (RNF_0_1) is saved by the C2C gateway controller to later route back the response data. Moreover, when the HNF_1_0 issues the read request to the SNF_1_0 (because the data is not shared by other controller, it has to be fetched from main memory) the *forward* field of the message is filled with the initial requester (C2CI-1). This field is then used by the SNF_1_0 to directly forward the data to the original requester after receiving it from the memory controller.

In order to properly handle the routing in a coherent chip-to-chip architecture, tables to store additional routing metadata, as well as additional fields in messages are needed. Moreover, when modeling such architectures, the simulation platform must allow enough flexibility to allow the configuration of such parameters in the different controllers of the network.

4.3.1 Chip-to-chip memory access example

In this section, we present an example of a chip-to-chip read access. Figure 4.5 shows an architecture with two chips. Each chip is using the CHI protocol to communicate between local controllers. The two chips are interconnected by a CCIX protocol, using PCIe physicals. Following is a step-by-step description of how a read request to DRAM-1 is serviced.

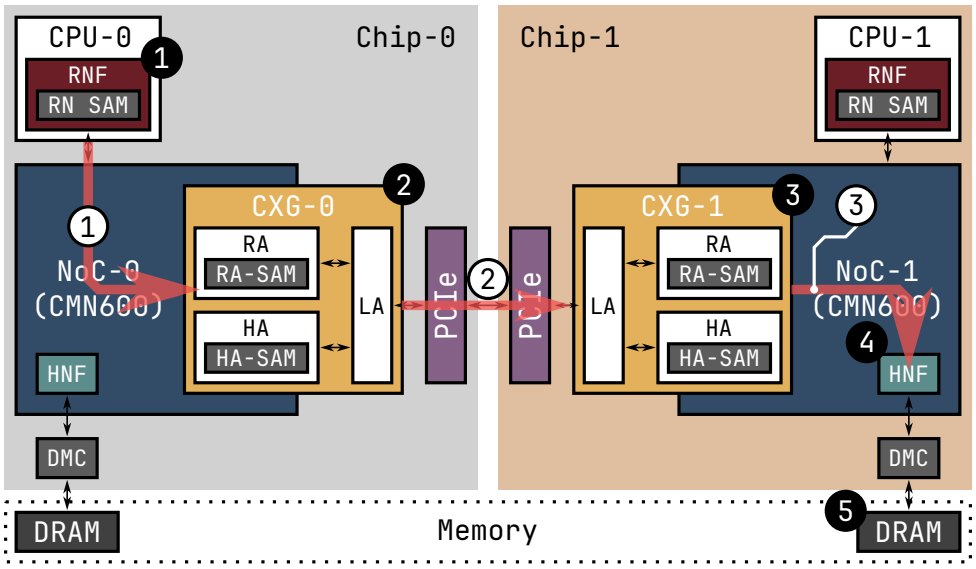


Figure 4.5: High level architecture of two chips with CHI NoCs, interconnected through CCIX with an arrow representing access to DRAM-1. CXG refers to CCIX Gateway, interfacing the chips.

1. CPU-0 issues a READ request to a memory address `0x8000`, allocated in DRAM-1 ❺

2. The READ request misses CPU-0 local caches

3. RN-F ❶ (connected to a crosspoint of chip-0's CMN NoC), generates a corresponding READ CHI request. Three main fields of interest are filled in the request. The source of the request is the *NodeID* of the RN-F ❶, the target of the request is the *Node ID* of the CXG-0 interface. This target is found using an internal routing table. The *LDID* is the Logical ID, which here is the RN-F in chip-0.

4. Chip-0's CMN NoC routes the CHI request to destination ①

5. The Request Agent (RA) in CXG-0 ❷ generates a corresponding CCIX request. Three main fields are filled. First the source of the request is set, which corresponds to the ID of RN-F ❶. Second, the target of the request is set and corresponds to the HA (Home Agent) of the CXG-1 interface ❸. This destination is

found using an internal routing table. Finally, the transaction ID is filled, in order to track the request.

6. The CCIX network routes the CCIX request to destination ②

7. The Home Agent (HA) in CXG-1 ❸ generates a corresponding CHI request. Similarly to the first step, the three fields are filled in the message. First, the source of the request is the Node ID of CXG-1's HA ❸. Then, the target of the request is the Node ID of the HN-F ❹. Finally, the Logical ID corresponds to the remote RN-F ❶.

8. Chip-1's CMN network routes the CHI request to destination ③

9. HN-F ❹ processes the READ request as follows. First, if the System-Level Cache (SLC) has the cache line (i.e., cache hit), the RN-f ❶ is added to the sharers list in the Snoop Filter (SF) ❹ and the DATA is sent to the RN-F. If the SLC does not have the cache line (i.e., cache miss) the HN-F now checks the snoop filter to determine if a cache has the CL. If a cache has the line (i.e., snoop filter hit), the HN-F sends a snoop to the RN-F sharing the requested address, which then transmit the DATA to the original requester RN-F ❶ and sends DATA (or not) and response to the HN-F ❹. The DATA is stored in the SLC of the HN-F ❹ and the RN-F ❶ is added to the list of sharers of the SF. However, if no caches have the line (i.e., snoop filter miss), the DATA is read from DRAM-1 ❺ using a Subordinate Node (SN-F). The SN-F forwards the DATA to the RN-F ❶ and a response DATA (or not) to the HN-F ❹. Finally, the RN-F is added to the list of sharers in the SF.

4.4 Maintaining coherence across chips

In this section, we will go over the functionality that allows to extend the on-chip local cache coherency to multiple chips. We will explain how tracking sharers of a cache line is handled. Then, we will show how the chip-to-chip gateway (or interface) controller might treat a coherence snoop differently depending on its origin.

In order to maintain coherence across chips, each chip must house a mechanism that tracks the caches in possession of a cache line. This mechanism ensures that when a cache modifies the cache line, the changes are propagated to the other caches (or the other caches get their copies of the cache line invalidated). This is paramount in order for each cache to work with an up-to-date version of the cache line. For example, in Figure 4.3a, Chip-1 must be able to track every cache line of MemCtrl-1's memory range that is cached by RNF-0. Thus, each HNF must track all caches storing a copy of

a specific memory block to maintain data consistency across the system. When a cache modifies a shared memory block, the HNF identifies all caches holding copies of that block and issues snoop requests to either invalidate or update their copies. Although the CHI specifications provides the functionality, the gem5 implementation does not. To extend CHI coherency across multiple chips, the HNF needs to be extended to track sharers located off-chip.

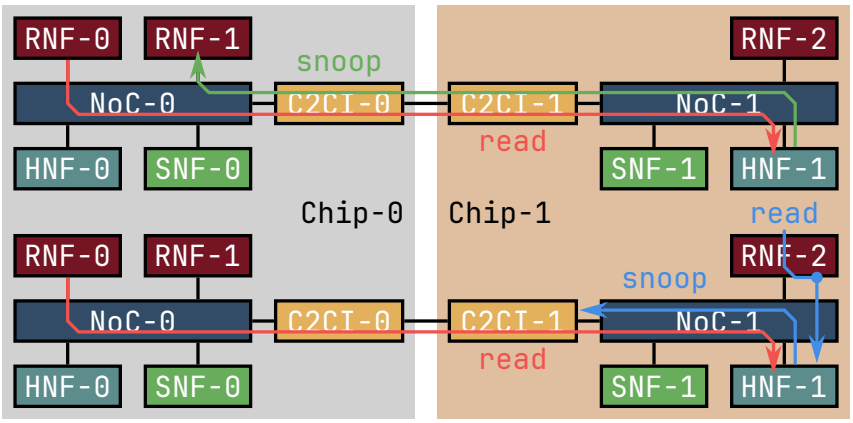


Figure 4.6: Different interpretation of a snoop from a C2CI depending on its origin.

The interaction between the C2C link controllers and local chips is important. During the development of our implementation, we find instances where we need to extend functionality to have insight into this aspect. For certain transactions, the behavior of the C2CI must differ based on whether the initiator is local or remote. Figure 4.6 shows an example problematic scenario. In the first part, RNF-0 sends a read request to a memory region managed by HNF-1. RNF-1 has the data cached. Thus, HNF-1 sends a SnpSharedFwd request to RNF-1, that reaches the C2CI-1. This snoop request is part of the servicing of the initial request from RNF-0. However, in the second case, while the RNF-0 sends an identical request to HNF-1, RNF-2 sends a read request to HNF-1 for the same address. The C2CI-1 receives an identical snoop, but this snoop is not part of the servicing of the initial request from RNF-0 (red arrow). Without additional mechanism and information, the interpretation of this message by the C2CI can be erroneous. To enable this differentiation, new metadata must be added to the CHI messages to differentiate between local and remote initiators.

4.5 Inter-chip link management for coherence

In this section, we will show how managing the link between multiple chips brings its own challenges. First, we will go over what are concurrency issues and what solutions can be used. Then, we will explain how multiple in-flight transactions are handled at the C2C link.

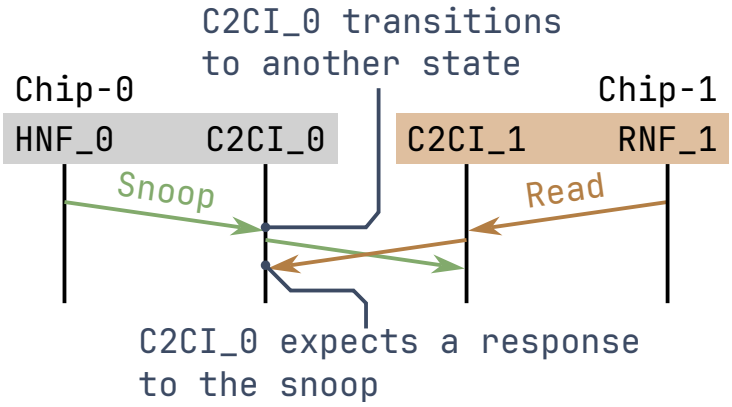


Figure 4.7: Flow diagram with two requests arriving at the same time to the two C2C link interfaces and causing a deadlock.

When modeling a chip-to-chip architecture, the model must implement concurrency-based deadlocks avoidance mechanisms. When two requests to the same cache line are traversing the C2C link, the controller interfacing the C2C link must be able to handle the new, unexpected request. It is possible to stall any network-side requests on both sides of the C2C link after the two interface controllers have transitioned to any other state than a *ready* state. However, the timing at which each controller receives a request when in a ready state can cause deadlocks. Figure 4.7 illustrates the problem with a simple flow diagram. The C2CI-0 is set up to treat the snoop request, thus expects a response, but receives a read request. In a working scenario, the C2CI-1 would stall the inbound yellow read request in order to treat the green snoop request. This can happen only when the C2CI-1 is aware that the snoop request is being serviced. However, in this situation, none of the C2C interface controller can stall the inbound network-side request because they are not aware of the request being serviced on the other side of the link.

In order to ensure proper routing and complete servicing of each request, the C2C interface must keep track of multiple in-flight transactions that are currently being processed by storing the necessary metadata (e.g., source, destination, type of transaction, etc.) locally in Transaction Buffer Entries (TBEs). This is straightforward when transactions operate on different addresses. However, it becomes significantly more complex when transactions operate on the same address. In such cases, the address is no longer sufficient to identify individual transactions, and more complex mechanisms are needed. One alternative is to limit the number of C2C in-flight transactions to the same address.

4.6 Summary

This chapter presented the fundamental challenges and considerations in simulating coherent chip-to-chip interconnect protocols. We began by examining how messages are routed within individual chips, focusing on the role of downstream clients, address ranges, and the handling of response messages. We then explored the complexities of extending this routing across multiple chips, introducing the concept of chip-to-chip interfaces (C2CIs) and their role in managing inter-chip communication. The discussion covered both the architectural requirements, and the additional metadata needed for proper message routing across chips. We examined the challenges of maintaining coherence in multi-chip systems, particularly focusing on how to track cache line sharers across chips and handle different types of coherence messages. Finally, we presented the challenges in managing chip-to-chip links, addressing concurrency issues and the complexities of handling multiple in-flight transactions. These concepts represent the foundation for the implementation details presented in the following chapter, where we provide concrete solutions through an implementation.

CHAPTER 5

FULL-SYSTEM SIMULATION OF CACHE-COHERENT INTERCONNECTS

Previously, we have described the gem5 simulator and our motivations when choosing our simulation platform of choice. We have also presented the challenges of extending cache coherency within and across chips. In this chapter, we focus on the implementation of a coherent chip-to-chip interconnect model inside the gem5 simulator. This shows the different solutions that we have implemented to reach a working model. First, we motivate the use of gem5, and the main design methodology used. Second, we present the Chip-to-Chip Interface (C2CI) controller, which allows to connect two networks in a simulation and extend their coherency. Third, we show the port and internal structure of the controller, also going over solutions to code scalability. Fourth, we explain how we solved a port instantiation problem by extending the gem5 simulator. Fifth, we go over our iterative development approach. Sixth, we present our solutions to congestion and link management. Finally, we focus on the calibration of the model using a real hardware platform.

5.1 Motivations

As seen in Section 1.1, High-Performance Computing (HPC) is transitioning from traditional monolithic chip designs to chiplet-based System-On-Chip architectures. Accordingly, adequate simulation technology is required to tackle the new challenges in designing, evaluating, and optimizing these systems. Today's simulation platforms are designed for multicore systems, do not allow multiple independent chips to extend their coherency under a single OS environment, and are thus not adapted to new emerging architectures. Our goal is to address the limits of modern simulators in simulating modern coherent interconnect protocols.

We develop the model using gem5. As previously stated, gem5 is an open-source event-driven simulator that allows full-system simulation. Full-system capability is primordial for multithreaded application execution, as well as adding flexibility of

controlling what the OS considers as a processor. Moreover, gem5 houses the Ruby subsystem, a highly configurable cache-coherence and memory system simulator. This allows a very high level of flexibility and detail over the cache-coherence protocol. Finally, gem5 is an active and well-maintained open-source project that receives industrial participation (e.g., ARM, AMD). Gutierrez *et al.* [40] and Butko *et al.* [22] have evaluated gem5's accuracy for the ARM ISA, against real hardware. Gutierrez *et al.* showed an average inaccuracy of 13% for SPEC CPU2006 benchmarks. Butko *et al.* studied gem5's accuracy for multicore simulation. Walker *et al.* [91] validated gem5 on ARM architectures, with an inaccuracy varying from 1.39% to 17.94%. Akram *et al.* [4] validated the gem5 simulator for x86 architectures and, after detailed configuration to match Intel's Core-i7 Haswell micro-architecture, matched real hardware results with less than 6% mean error rate for different control, memory, dependency and execution micro-benchmarks.

Our design approach is to develop a new controller (i.e., state machine) to extend the CHI protocol inside the Ruby subsystem of gem5. Ruby allows to declare components using SLICC. SLICC offers numerous advantages, making the description of the state machine easier, at the price of added complexity for debugging. SLICC also brings difficulties when bridging two networks as it has been designed for monolithic NoC protocols. We choose to extend the CHI protocol because it is a state-of-the-art solution for high-performance, scalable, multicore systems. Moreover, we later discuss the calibration of the model using a real hardware testbed. This testbed is using the Arm CHI protocol and the CCIX protocol, which also motivates our choice.

5.2 Implementing the C2CI in SLICC

The Chip-to-Chip Interface (C2CI) is a new controller developed to bridge multiple network models within a single Ruby environment, allowing to extend the coherency between them. To develop this controller, we use the SLICC language in Ruby. All SLICC-defined controllers are built as state machines. They have *states, events, transitions*, etc. Moreover, each controller also has a set of ports in order to communicate with various other components. In this section, we go over the different parts of the C2CI and show how they have been described using SLICC.

5.2.1 Ports

In this section, we describe the port structure of the Chip-to-Chip Interface (C2CI) controller. The C2CI is modeled as a state machine using the SLICC language. This state machine has a set of inputs, which receive messages from other components, and a set

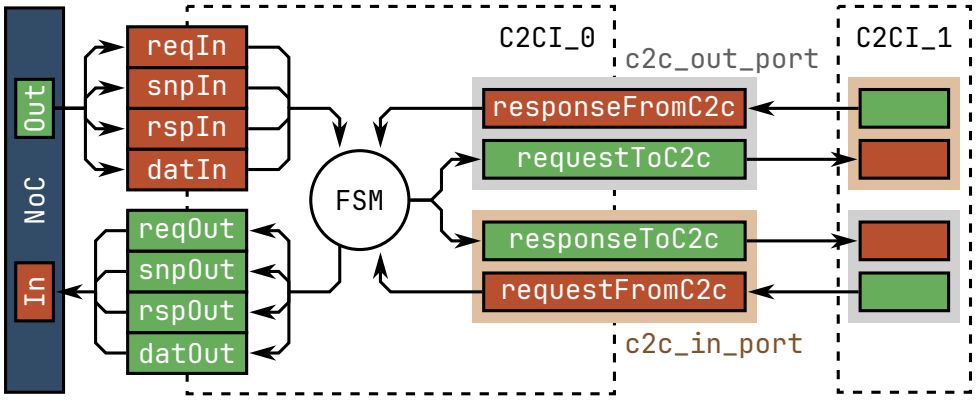


Figure 5.1: Port structure of the C2CI.

of output ports, which the state machine uses to communicate with and control other parts of the system. When the state machine transitions between states, it can execute actions. These actions might involve reading messages from the input ports, processing them and generating messages on the output ports. This state machine Figure 5.1 shows the port structure of the C2CI. On the left side, ports are following the CHI semantic. The *reqIn*, *snpIn*, *rspIn* and *datIn* allow to receive messages from the network, meanwhile the *reqOut*, *snpOut*, *rspOut* and *datOut* allows the C2CI to send messages to the network. Ports have different names in order to concurrently send and receive different message types. On the right side are ports used to communicate between C2CIs. The *requestToC2c* and *responseToC2c* allows the C2CI to send requests and responses to the C2C link. The *responseFromC2c* and *requestFromC2c* allows to receive requests and responses from the C2C link. Listing 5.1 shows the SLICC port declaration of the C2CI. We can see the network-side interface, as well as the C2C-side.

```
1  // Communicating with the network
2  MessageBuffer * reqOut,   network="To", virtual_network="0", vnet_type="none";
3  MessageBuffer * snpOut,   network="To", virtual_network="1", vnet_type="none";
4  MessageBuffer * rspOut,   network="To", virtual_network="2", vnet_type="none";
5  MessageBuffer * datOut,   network="To", virtual_network="3", vnet_type="response";
6  MessageBuffer * reqIn,   network="From", virtual_network="0", vnet_type="none";
7  MessageBuffer * snpIn,   network="From", virtual_network="1", vnet_type="none";
8  MessageBuffer * rspIn,   network="From", virtual_network="2", vnet_type="none";
9  MessageBuffer * datIn,   network="From", virtual_network="3", vnet_type="response";
10 // C2c out port
11 MessageBuffer * requestToC2c;
12 MessageBuffer * responseFromC2c;
13 // C2c in port
14 MessageBuffer * requestFromC2c;
15 MessageBuffer * responseToC2c;
```

Listing 5.1: SLICC C2CI port declaration.

As shown in Figure 5.1, C2C-side ports are laced in pairs. The *c2c_out_port*, and similarly the *c2c_in_port*, houses one input and one output port. This is because the

C2CI uses a different type of port on its C2C-side. The ports that are used to communicate on the network-side are *Ruby Message Buffers* and are traditionally used between SLICC defined controllers and network models. The C2C-side ports are *gem5 classic ports*. These ports are traditionally used outside of Ruby and to communicate with classic gem5 models (e.g., Memory Controller). Section 5.3 explains why we opt for this solution. gem5 classic ports are bidirectional. Thus, we use two Ruby Message Buffers per port.

As seen in the Background Section 3.5.2, when input ports hold a message, they can trigger events that fire a transition.

5.2.2 Actions

When input ports or internal queues (see Section 5.2.4) trigger events, the state machine can transition from one state to another. These transitions execute *actions*, which are small, discrete blocks that perform specific tasks as part of the state transition. Actions might involve internal variables, sending messages to other components, triggering internal events or modifying the behavior of the state machine itself. The Listing 5.2 is an execution trace sample that shows a state transition in the C2CI controller. The trace indicates that the controller is in the state WRITE and receives a `Trigger_SensToNet_Request` event. In response to this event, the controller transitions and executes a set of actions: `sentToNet_Request` and `popTriggerQueue`.

```
2014500: system.ruby.interface0.cntrl: [Interface_Controller 0], Time: 4029, state: WRITE
    , event: Trigger_SendToNet_Request, addr: 0x10000000
2014500: system.ruby.interface0.cntrl: executing sendToNet_Request
2014500: system.ruby.interface0.cntrl: executing popTriggerQueue
2014500: system.ruby.interface0.cntrl: next_state: WRITE
```

Listing 5.2: Execution trace sample showing a transition.

In most cases, and in multiple other controller's implementations, actions are tailored for each message types. For example, the `sendToNet_CompAck` action sends a CompAck message to the local NoC. However, in order to keep the code scalable and to a reasonable length (approximately 3000 lines) we use two main mechanisms. First, we use generalized actions to handle different types of requests. For example, *sendToNet_Data* sends a data message to the network. The fields of each message are filled with information from the TBE. Second, we sort the requests into different FSMs. Some requests are treated the same way and can be executed using the same FSM. For example, *ReadShared* is treated the same way as *ReadUnique*.

5.2.3 States

The C2CI controller's state machine uses states to represent different operational phases of requests processing, rather than cache coherence states. These states reflect the internal processing stages of the interface controller as it handles messages and coordinates communication between different system components.

Each state in the FSM represents a specific waiting or processing condition. For instance, when a request arrives and a TBE is allocated, the state machine may transition to states like `READING_MEM`, where it waits for specific responses or conditions to be met before proceeding. The state machine continues to transition through various states as it processes the request, executes necessary actions, and coordinates with other system components. Once a request has been fully serviced, the FSM transitions back to its default `READY` state, at which point it deallocates the TBE (see Section 5.2.7), indicating the successful servicing of the request. This return to the `READY` state with the TBE deallocation is crucial as it frees up resources for handling subsequent requests and maintains the efficient operation of the C2CI controller.

States in the C2CI are independent for each cache line address that are being serviced. For example, the C2CI can be in state `READING_MEM` for address `0xadd` and state `READY` for address `0x10000`. Subsequently, states are inherently stored in the TBE of the controller by the SLICC generated code (see Section 3.5.1). When a TBE is deallocated for an address and the C2CI receives a message for this address, the FSM will default to the default `READY` state. Listing 5.3 shows the `getState()` method, that is generated from the SLICC code to C++. This method is used to get the state linked to a specific address. We can see that if the TBE is not allocated, the method defaults to the `READY` state.

```
Interface_State
Interface_Controller::getState(Interface_TBE* param_tbe, const Addr& param_addr)
{
    if ((param_tbe != NULL)) {
        #ifndef NDEBUG
        if (!((*param_tbe).m_addr == param_addr))) {
            panic("Runtime Error at CHI-interface.sm:912: %s.\n", "assert failure");
        }
        #endif
        ;
        return (*param_tbe).m_state;
    } else {
        return Interface_State_READY;
    }
}
```

Listing 5.3: C2CI getState method in the generated C++ code.

5.2.4 Internal queues

The C2CI controller employs a set of internal queues to manage message flow and control internal behavior. These queues serve two primary purposes: message stalling and request handling. The *stalled* and *stalledSnoops* queues are used to temporarily hold messages when they cannot be processed immediately, providing a buffering mechanism to handle system congestion. They are used in tandem with stalling and awakening functions of SLICC (see Section 3.5.2). For requests that are ready to be processed, the *reqRdy* and *c2cReqRdy* queues are utilized to hold the message and trigger the right event, once the availability of TBE slots has been confirmed.

A special queue, the *triggerQueue*, plays a unique role in the C2CI controller's operation. This queue manages trigger messages (*triggerMsg*), which are internal control messages that carry events to be triggered after a specific delay. When a trigger message is enqueued, it remains in the trigger queue for the specified number of cycles before its associated event is triggered, enabling precise timing control of internal state machine events. Listing 5.4 shows the *trigger_SendToC2c_Request* action that uses the trigger queue in order to trigger an event called *Trigger_SendToC2c_Request*. The trigger message holds the event in the *out_msg.event* field and is triggered after *c2c_link_latency* number of cycles.

```
1 action(trigger_SendToC2c_Request, "ts2cr", desc="Trigger sendToC2c_Request") {
2     assert(is_valid(tbe));
3     enqueue(triggerOutPort, TriggerMsg, c2c_link_latency) {
4         out_msg.addr := address;
5         out_msg.event := Event:Trigger_SendToC2c_Request;
6     }
7 }
```

Listing 5.4: Action using the triggerQueue.

5.2.5 Events

The C2CI controller's state machine is driven by a variety of events that trigger transitions between different states. These events originate from three main sources: the input ports, the internal queues, and the specialized trigger queue. Moreover, events can also be directly triggered in the SLICC code using the `trigger()` method. The input ports (see Section 5.2.1) receive messages from the network and the C2C link, generating events such as `Net_<MessageType>` and `C2c_<MessageType>` based on the type of message received. These events signal the arrival of new requests or responses that need to be processed by the state machine.

In addition to the input port events, the internal queues (see Section 5.2.4) also generate events when certain conditions are met. For example, when a request arrives

in the input port, the `checkAllocTBE` will perform a series of checks to determine if the request can or need to be treated, after which the message is placed in the `reqRdy` internal queue. This queue will then trigger an event linked to the type of request. This mechanism allows to only accept requests that can be treated and *Retry* requests that cannot.

Finally, the trigger queue manages a special type of event called trigger event, such as the `Trigger_SentToNet_Request` event. These events are enqueued with a specified delay and triggered after the delay. Listing 5.5 shows a sampled version of the event declaration of the C2CI controller. We can see first the events that allocate a TBE for inbound requests. For each message types, we can see a dedicated event depending on its origin (i.e., C2C link or local network).

```
enumeration(Event, desc="Memory events") {
    // Checks if a request can allocate a TBE be moved to reqRdy
    CheckAllocTBE, desc="";
    CheckAllocTBE_WithCredit, desc="";
    // Checks if there is a request in the requestFromC2c
    CheckC2cReqAllocTBE, desc="";
    // Checks if the inbound request is expected
    CheckC2cExpectedReq, desc="";
    // Checks if we can allocate the request from the stalled queue
    CheckStalledAllocTBE, desc="";
    // Requests
    Net_ReadShared, desc="Network side read shared";
    C2c_ReadShared, desc="";
    Net_ReadOnce, desc="Network side read once";
    C2c_ReadOnce, desc="";
    [...]
    // Snoops
    Net_SnpCleanInvalid, desc="Network side snp invalid";
    C2c_SnpCleanInvalid, desc="c2c side snp invalid";
    [...]
    // Data
    Net_CompData_UC, desc="Response data from net of type UC";
    C2c_CompData_UC, desc="Response data from c2c of type UC";
    Net_CompData_I, desc="Response data from net of type I";
    C2c_CompData_I, desc="Response data from c2c of type I";
    [...]
    // Responses
    Net_CompAck, desc="";
    C2c_CompAck, desc="";
    Net_Comp_I, desc="";
    C2c_Comp_I, desc="";
    [...]
    // Internal event triggers
    Trigger_Send, desc="Used to loop and send multiple data packets";
    Trigger_SendDone, desc="";
    Trigger_SendSnpResp, desc="Used as serialisation tool to delay the snp resp";
    Trigger_ReceiveDataDone, desc="Received all read data";
    [...]
    // Priority snoop triggers
    Trigger_Finalize, desc="";
    Trigger_PendingRequest_BlkdReq, desc="";
    [...]
```

```
43      // ReqAck triggers
44      Trigger_InitiateReqAck, desc="Initiates the request acknowledgement";
45      Trigger_SendToC2c_ReqAck, desc="Trigger the sending of the ReqAck to c2c";
46  }
```

Listing 5.5: Sample of the event declaration of the C2CI controller.

5.2.6 Transitions

The C2CI controller's state machine is defined with a set of transitions that govern how the system responds to events. These transitions describe the specific state changes that occur when an event is triggered, and the actions that are executed as part of those state changes.

Each transition in the state machine is defined with a starting state, a set of triggering events, and a destination state. Instead of defining a set of transitions for every type of message, we sort each request messages in different categories. Different requests can have the same behavior in that they expect the same type of responses. For example, a `ReadShared` and `ReadUnique` both wait for response data. We sort requests in four main FSMs with the following transitions:

- `READY` → `WAIT_COMP`
  - Waits for completion responses from the C2C (e.g., `C2c_Comp_I` for invalidation).
  - Can transition to further states to write requests.
- `READY` → `BLKD_REQ`
  - Waits for response data from the Network.
  - Can transition to further states when receiving a snoop, which has priority.
- `READY` → `READING_MEM`
  - Waits for response data from the C2C link.
  - When receiving the data, transitions to `SENDING_NET_DATA` and sends the response data to the local network requester, separated in multiple messages.
- `READY` → `BLKD_SNP`
  - Waits for snoop responses (e.g., `Net_SnpResp_I`) and/or snoop response data (e.g., `Net_SnpRespData_I_Fwded_SC` which means network-side Snoop Response Data Invalid Forwarded Shared Clean) or simply response data

(e.g., `Net_CompData_UC` which means network-side Completion Data Unique Clean).

The CHI implementation of gem5 makes the distinction between snoop requests and standard requests (e.g., ReadShared, WriteUnique, etc.). These four transitions represent the beginning of the FSM of all the requests and snoop requests of the system. They all start at the READY state (see Section 5.2.3). After transitioning to a *request state*, the FSM can transition again depending on the nature of the response.

When issuing a `ReadShared`, the HNF that receives the request will check if there are sharers for the specific cache line. If there are sharers for the cache line, and instead of prompting a subordinate node (SNF) for the data, the HNF will send a `SnpSharedFwd` (i.e., Snoop Shared Forward) request. This snoop asks the sharer of the cache line to provide the data to the original requester. After a network-side `ReadShared` reaches a C2CI, it will transition to the `READING_MEM` state. In this state, the C2CI can either transition to a state where it sends the received C2C-side data to the network (i.e., `SENDING_NET_DATA`), or transition to a state where it handles a `SnpSharedFwd` (i.e., `SNP_FWD_LOCAL_DATA`). At the time of writing this thesis, the C2CI contains 103 different transitions.

5.2.7 Structures

The C2CI controller's internal logic is built upon several key data structures defined in the SLICC code. These structures encapsulate critical information and provide built-in methods to manipulate and manage the state of the interface controller. One of the primary structures is the `ExpectedMap`, which is used to track the expected responses for each outstanding request. When a request is initiated, the ExpectedMap is set to monitor the expected responses, allowing the controller to properly handle and correlate incoming messages. The `ExpectedMap` provides methods such as `clear`, `addExpectedRespType`, and `hadExpected` to facilitate the management of these expected responses.

The `ExpectedMap` is a structure that has been added to gem5 with the Arm CHI protocol. We added a new structure to the protocol called `ExpectedReqMap`, which is used to check if an incoming request is expected by the C2CI controller. This structure helps the controller to determine whether a request should be processed or rejected based on the current state of the system. This structure is used by the *Handshake* mechanism of the C2CI controller (see Section 5.6).

The Table Buffer Entry (TBE) is the central data structure within the C2CI controller. It stores all the relevant metadata for a transaction, including the address, requester, request type, and the current state of the FSM for a specific cache line. The TBE is essential for maintaining the context and progress of each request being serviced by the controller. To manage the TBEs, the C2CI controller utilizes the *TBEStorage* which provides methods to track the available TBEs, allocate new ones, and check resource utilization.

Finally, the `TriggerMsg` structure represents a unique message type used to hold and trigger internal events within the C2CI controller. This structure enables the controller to schedule and execute delayed actions, facilitate the coordination of complex sequences of operations (see Section 5.2.4).

5.3 Addressing port instantiation challenges

We implement the new cache-coherent chip-to-chip interconnect model called c2c-gem5 within gem5. However, implementing c2c-gem5 withing Ruby is not straightforward, as Ruby does not natively support multiple interconnected NoCs. Ruby was originally designed to model monolithic chips including a single NoC model. But modeling multiple chips in Ruby requires multiple NoCs. As mentioned previously, recent work [39] removed the single NoC limitation, enabling the use of multiple NoCs to model separate processor and GPU networks in the same Ruby environment. However, connecting multiple NoCs and extending their coherence is still problematic: Ruby makes implicit assumptions about how its controllers are connected to the NoC.

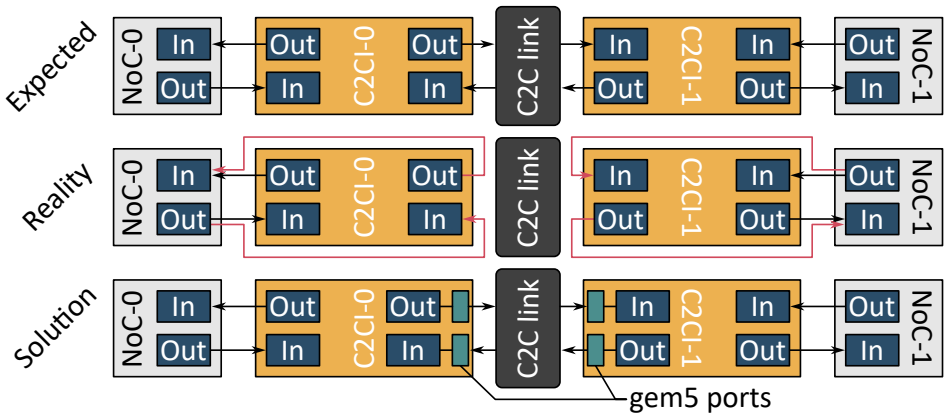


Figure 5.2: Port connection problem when using Ruby ports between two controllers of different networks.

Figure 5.2 illustrates the problem. The top subfigure shows the expected configuration when connecting two Ruby controllers of different networks (e.g., C2CI-0 from NoC-0 and C2CI-1 from NoC-1). However, as shown in the middle subfigure, Ruby

automatically connects all the ports of a controller to its corresponding network, including the C2C-side ports that we intend to connect to the C2C link. To address this issue, we use gem5 ports instead of Ruby ports to connect the C2CI to the C2C link, as shown in the bottom subfigure. In order to use gem5 ports, we extend the gem5 simulator by adding new ports to all controllers. These changes are explained in the following subsections.

5.3.1 Ruby message buffers and gem5 classic ports

In gem5 communication between different simulation components is done through two main port types: Ruby *MessageBuffer* and gem5 *classic ports*. In the Ruby memory system, MessageBuffers serve as communication interfaces between various components of the cache coherence protocol. These buffers are instantiated in the SLICC generated code and provide flexible communication channels with two primary operational modes: ordered and unordered buffering. MessageBuffers transport messages between different controllers of the Ruby system but also allow to account for communication latencies by scheduling enqueue and dequeue operations.

Within the broader gem5 simulator, classic ports provide a standardized communication mechanism for models outside the Ruby memory system. In Ruby, all controllers are built by inheriting from the *AbstractController* class, which offers an optional gem5 classic port for inter-component communication. For example, using the CHI protocol, the SNF will connect to the memory controller, a gem5 classic model, through the `memory_out_port`, a gem5 classic port provided by the AbstractController parent class.

In gem5, *packets* represent the fundamental unit of communication between different models, providing a standardized mechanism for transferring memory requests, data and responses. Each packet is characterized by its *Command* and, by extension, its *CommandInfo* attributes. They describe the packet's purpose and behavior. These class parameters encode essential information such as whether the packet represent a read, write, invalidate, or if the packet is a response or request.

As previously mentioned, the AbstractController parent class is inherited by all SLICC-defined Ruby controllers. This allows certain controllers to use a gem5 classic port to "get out" of Ruby and exchange with models outside of Ruby. The AbstractController hosts the functionality responsible for the translation from Ruby messages to gem5 packets. As explained in Section 3.5.2, two Ruby message buffers are used to create a single bidirectional gem5 classic port.

Figure 5.3 shows some internal components of the SNF controller of the CHI protocol. On the left, we can see the network-side input and output ports. These ports are internally feeding to and from the state machine. Then, internal ports called `requestToMemory` and `responseFromMemory` are used to communicate with the *MemCtrl* model. This model is a classic gem5 model that lives outside of Ruby. The `memory_out_port`, provided by the AbstractController parent class, translates to and from Ruby messages into and from gem5 packets. Finally, AbstractController calls the MemCtrl `in_port` methods to receive the gem5 packets. This system effectively allows any SLICC-defined controllers to have access to the `memory_out_port` to communicate with outside models.

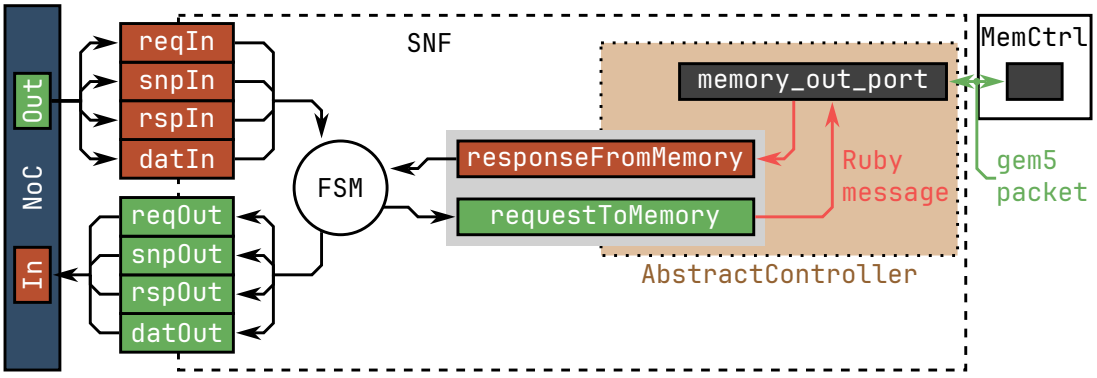


Figure 5.3: SNF controller with I/O network-side ports and internal Ruby message buffers, themselves connected to the memory_out_port gem5 port, provided by the AbstractController parent class.

5.3.2 AbstractController extension

In order to connect the two networks together using two gateway controllers (i.e., the C2CIs), and to avoid the port instantiation issue, we decide to use gem5 classic ports. We use gem5 ports instead of Ruby ports to connect the C2CI to the C2C link, as shown in the bottom subfigure of Figure 5.2. So as to use new gem5 ports in our SLICC-defined controller, we need to extend the AbstractController parent class. We add new ports that can be used by the C2CIs without modifying the already-existing `memory_out_port`, which could compromise already implemented protocols. The two ports are called `c2c_out_port` and `c2c_in_port`. Each gem5 port is bidirectional and uses two Ruby message buffers in order to communicate to and from the C2CIs. These four ports are named `requestToC2c`, `responseFromC2c`, `requestFromC2c` and `responseToC2c`. We add new functionality in AbstractController to send and receive messages using the new ports, as well as creating new Ruby messages from gem5 packets and creating packets from Ruby messages. Figure 5.4 shows the connection from C2CI-0 to C2CI-1 with network-side and C2C-side ports. When observing the interface0 of the figure we can see, on the left-side, the network-side ports. Then, the

simplified FSM block is connected to Ruby queues (e.g., requestToC2c) used to send and receive message to and from gem5 classic ports. The figure also highlights the detail of the two new gem5 ports. For example, the `serviceReqToC2cQueue` method takes the Ruby message from `requestToC2cQueue`, creates a new corresponding gem5 packet, and calls `sendTimingReq` from the `c2c_in_port` in order to send the packet.

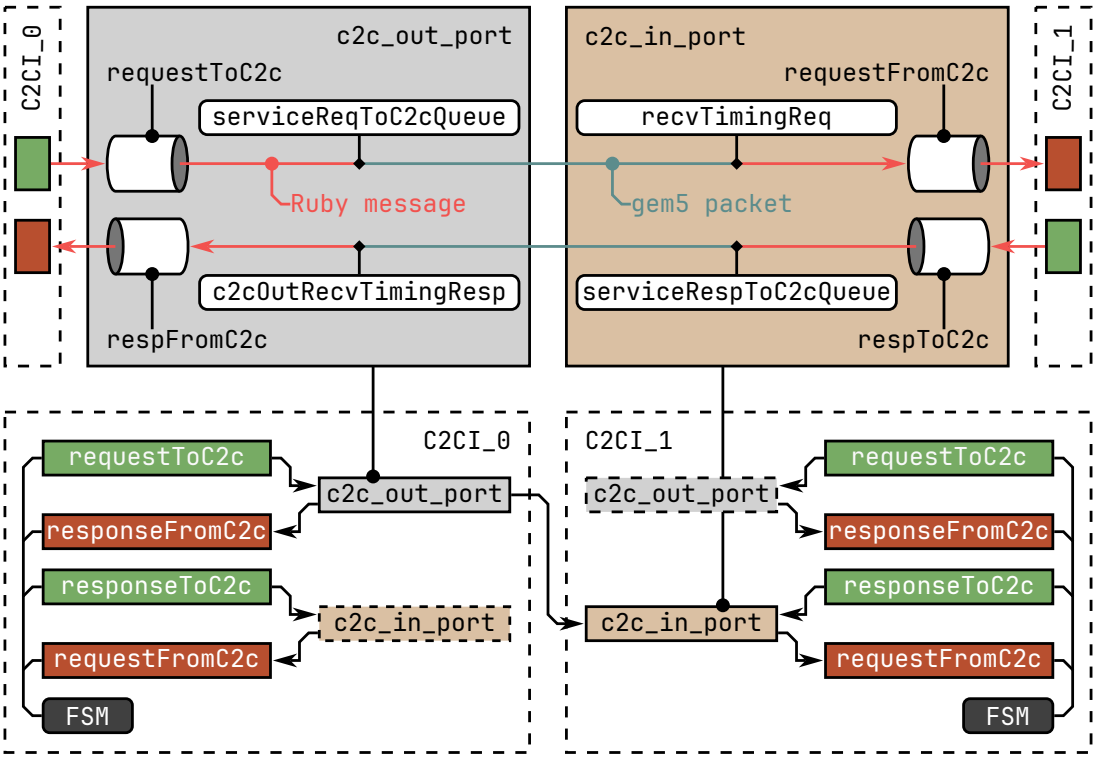


Figure 5.4: C2CI-0 to C2CI-1 communication link with C2C-side ports. Detailed view of gem5 classic ports (i.e., `c2c_out_port` and `c2c_in_port`) structure with method names.

5.3.3 SLICC compiler extension

As seen in Figure 5.4, the SLICC-defined controller's C++ code uses special methods to access Ruby message buffer queues (e.g., `getReqToC2cQueue`). However, the SLICC compiler, before our modifications, assumed only one gem5 classic port (i.e., `memory_out_port`). Thus, the getter function for the queues as well as other helper functions weren't available in the compiled code (from SLICC to C++) for our new ports. This prompted the need to extend the SLICC compiler. With these modifications, the generated C++ code includes additional methods to handle the new `c2c_in_port` and `c2c_out_port`. Most modifications have been done to the `StateMachine.py` file. This file generates, from the SLICC code, the main C++ files for the controller model. We add the `getReqToC2cQueue()`, `getRespFromC2cQueue()`, `getRespToC2cQueue` and `getReqFromC2cQueue()` methods to be written by the compiler.

Listing 5.6 shows a sample of the SLICC compiler with modifications to the writing of the `wakeup` function of the controllers. We can see that if the `ident` parameter is "Interface" (e.g., the name of the chip-to-chip interface model), the compiler will write different code, that includes the new declared getter functions. On the other hand, if the compiler is not writing the interface's code, the wakeup function will stay as before (using the memory_out_port). This shows the type of extension that has been added to the SLICC compiler in order to match the new changes in the AbstractController parent class. These changes are working together to generate the proper code that, in finality, allows two networks to communicate through Ruby controllers (namely, the C2CI gateway controller).

```
if self.ident == "Interface":
    code(
        """
        namespace gem5
            {
            namespace ruby
                {
                void
                ${ident}_Controller::wakeup()
                    {
                    if (getReqToC2cQueue() && getReqToC2cQueue()->isReady(clockEdge())) {
                        serviceReqToC2cQueue();
                    }
                    if (getRespToC2cQueue() && getRespToC2cQueue()->isReady(clockEdge()))
                         {
                        serviceRespToC2cQueue();
                    }
                [...]
        """
    )
else:
    code(
        """
        namespace gem5
            {
            namespace ruby
                {
                void
        ${ident}_Controller::wakeup()
            {
            if (getMemReqQueue() && getMemReqQueue()->isReady(clockEdge())) {
                serviceMemoryQueue();
            }
        [...]
        """
    )
```

Listing 5.6: SLICC compiler modifications to program the wakeup function differently.

5.3.4 Instantiation and results

We develop two new ports in the AbstractController, parent class to all SLICC-defined Ruby controllers. These new ports allow to connect two Ruby controllers, attached to different NoCs, together. This connection itself brings new architecture exploration opportunities for the simulator. The ports use an efficient system that translates to and from Ruby messages to and from gem5 packets. By using these ports, we can effectively instantiate two C2CI controllers in the same Ruby system, each connected to their respective NoC models, connect them together, and transmit messages from one network model to the other. This addition, that necessitated the extension of the C++ gem5 kernel as well as the SLICC compiler, represents an important first step towards modeling coherent multi-chip architectures in gem5.

5.4 Integrating KVM with dual networks

During the development and testing phase of our model, we often get limited by simulation times. Simulations, when using gem5 in full-system mode, can regularly reach tens of hours. In order to reduce simulation times, we use CPU switching (see Section 3.3). We want to use KVM CPU models (see Section 3.2), a fast model using the underlying host hardware to run, to boot the OS. KVM uses *functional* memory requests. This means that the requests are bypassing the cache subsystem and are sent directly to the memory controller. This allows the model to execute the required memory transfers without the unnecessary timing considerations. Unfortunately, the functional protocol of gem5 does not natively support multiple networks. Since the protocol is bypassing the cache memories and network-on-chip, there is no routing of the request. Instead, the protocol accesses directly the memory controller. The protocol needs the ID of the memory controller and uses a table to do so. However, this table is not initialized properly. This causes requests from KVM CPU models to not be able to reach the other chip.

In order for us to use KVM CPU models to boot the OS, we add a simple hack that, depending on the address accessed, returns the ID of the correct memory controller. Knowing that the memory ranges of the two controllers are non-interleaved and as follows: [0,1.5 GB) for the first memory controller and [1.5 GB, 3 GB) for the second. Listing 5.7 shows the simple modification done to the `RubyPort.cc`. The ID of the memory controller is selected depending on the accessed address. This solution is simple and was used within our development time constraint. However, a cleaner solution will be implemented in further developments.

```
if (id.getType() == MachineType_Memory && id.getNum() == 2) {
    if (pkt->getAddr() <= 1610612736) {
        id.num = 0;
    } else {
        id.num = 1;
    }
}
```

Listing 5.7: Hack to select the correct memory controller.

5.5 Iterative development and validation

In this section we will go over the different phases that the C2CI controller went through during its development. This section highlights the iterative approach adopted as a way to maintain manageable complexity throughout the whole development process. First this section will focus on the first architecture that we have called "memory expansion" architecture. Then we will go over the features of the second development milestone, which corresponds to the modeling of the C2C with two C2CI controllers. Finally, we will show how the current state of c2c-gem5 was developed. During all phases of development, we have used, in parallel, different simulations of simpler architectures (i.e., without any C2C link), in order to understand how the CHI protocol implementation handles each access patterns.

5.5.1 Memory expansion architecture

In the first step of the development, we realized the first dual-network simulation. In this step, we still didn't solve the port instantiation problem (Section 5.3). The main objective was to validate the simulator when using two interconnected network models. To minimize the potential for errors, we aim to reuse as much as possible from the vanilla CHI protocol in Ruby. Accordingly, we chose to use an SNF (see Section 2.3.1) to model the C2C interface controller of the chip emitting the remote read request (chip-0 in Figure 5.5).

The SNF is normally used to send requests to a memory controller (called MemCtrl in gem5). In this first version, our focus is on the C2C interface of the receiving chip (Chip-1), which replaces the memory controller 1 shown on the left side of Figure 5.5. The C2C interface acts as a memory controller (MemCtrl_1) for the SNF but also acts as a requester (RNF) for Chip-1. We decide to use gem5 classic ports to implement the communication between the C2C interface and the SNF. gem5 classic ports carry gem5 packets and are used to interface with non-Ruby gem5 components, such

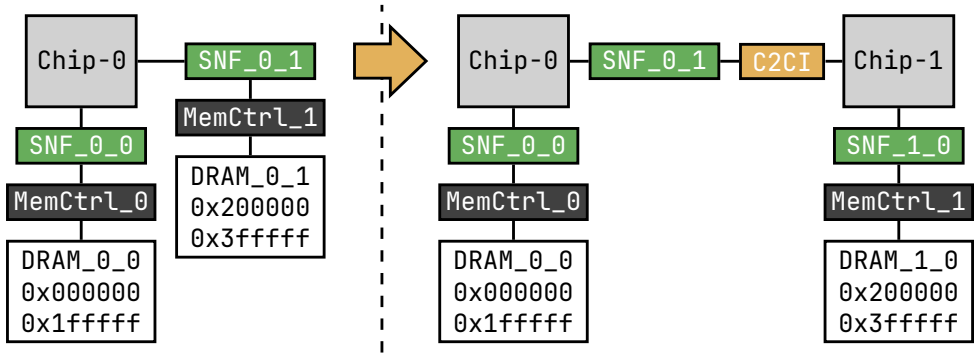


Figure 5.5: Single chip to multi-chip using Chip-0's SNF as request entry point.

as the CPUs and memory controller. For instance, the SNF uses a gem5 classic port called `memory_out_port` to send its requests and receive data responses.

The C2C interface also acts as a request node for the chip-1, issuing read requests. Received request in gem5 packets format must be translated to the MemoryMsg type by the input port, used to fill a TBE (Table Buffer Entry) and generate the right CHI request to be sent in Chip-1. We need to use the MemoryMsg type as a middleman between the gem5 packets and the CHIRequestMsg. It's the type of message that the SLICC FSM sees. For example, the SNF only deals with CHI messages and MemoryMsg, the gem5 packet translation is handled by the gem5 classic ports. Since we are using the SNF and the `memory_out_port` in our current design version, we need to comply with this by using MemoryMsg. Later on, we use the custom C2C Message type and new dedicated ports.

To act as a requester the C2C interface:

- Issues a request message to the right HNF on Chip-1
- Receives the response data directly from the SNF and in multiple messages
- Builds back the full data to be sent back to the SNF in Chip-0
- Sends an acknowledgement message to the HNF on chip when the data has been sent back to Chip-0

In Figure 5.6 we can see the general architecture of the simulation. We have two chips. Among these two chips, we have one RNF. There are two HNFs on Chip-0 that are each managing a memory region. There are two SNF that are in charge of sending requests to the MemCtrl and receiving response data, as well as transferring the data back to the original requester. The SNF_0_1 in charge of memory region [2, 4] sends its requests and gets its responses from the C2C interface. This is a unidirectional architecture where Chip-1 acts as memory expansion.

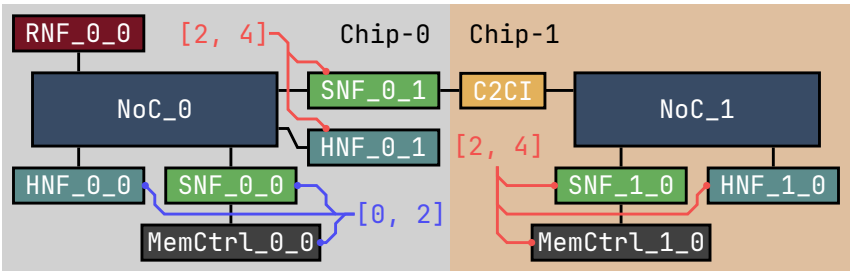


Figure 5.6: High-level architecture of the memory expansion architecture milestone with address ranges.

The C2C interface supports read requests only. These requests are issued by the RNF to the HNF on Chip-1. For addresses in the [2, 4]MB range, the HNF in charge of this range will issue a read request to the SNF connected to the C2C interface. The C2C interface has the following key features:

- Servicing `gem5::packet` request and responses in the MemoryInPort
- Receiving read requests of *MemoryMsg* type
- Allocating and storing request information in a TBE
- Generating a read request of type *CHIRequestMsg* for the Chip-1 network
- Handling the response data that comes in multiple messages
- Creating a single data message out of the different response data packets
- Generating and sending back the data to the requesting SNF in a MemoryMsg
- Generating and sending an acknowledgement message back to the HNF local to Chip-1

Still in order to reduce complexity, we use a traffic generator to generate a single access pattern in the memory system 3.6.1. The traffic generator injects traffic from a custom trace into the RNF_0_0. The trace generated for this test is a set of read requests to random addresses between 1MB and 4MB. Since the address range on Chip-0 is [0, 2MB] and the address range of chip-1 is [2MB, 4MB], we are effectively targeting both local and off-chip memory ranges.

When running the simulation, we use the *RubyGenerated* debug flag. This debug flag allows us to generate a trace of all the transitions that are taken by each controller of the architecture for each address. This trace also shows the different actions that are being executed with each transition. The end-of-simulation printout shows the message "Exiting @ tick 38695500 because exiting with last active thread context" meaning

that the simulation ran without functional errors. Figure 5.7 is a diagram of the transaction messages sent to complete this read request. We can see how chip-1 acts as a DRAM for chip-0.

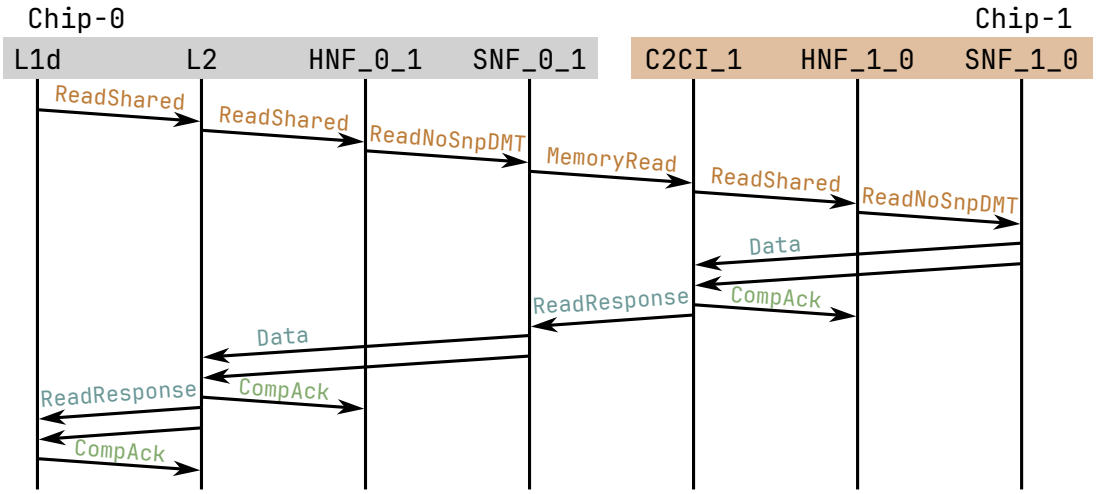


Figure 5.7: Read request transaction with times (cycles).

5.5.2 C2CI on both sides of the link

In the perspective of enabling a full C2C simulation we needed to extend the C2C interface model. The first step is to stop using the SNF as part of the C2C connection. The SNF is a first step solution but does not have the features to handle coherence traffic nor diverse request and response types. We will extend the interface to allow an interface-interface C2C connection, as seen in Figure 5.8. This will also allow us to declare and use a new type of message between the two C2C interfaces. The currently used MemoryMsg is sufficient now but cannot be modified to handle more C2C-tailored traffic.

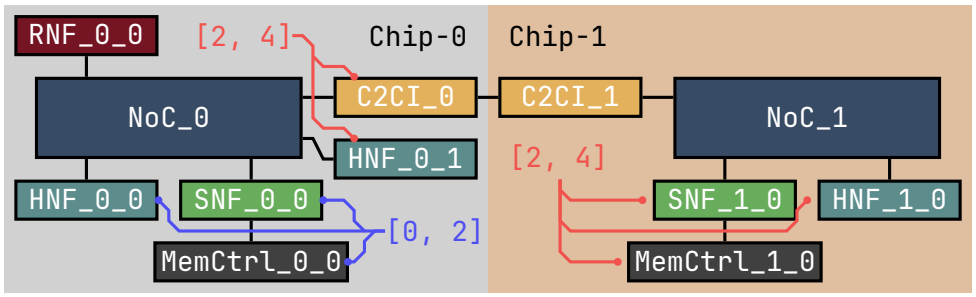


Figure 5.8: Interface-interface high-level architeture with address ranges.

In this development phase, we wanted to have more control over the link between the C2C interfaces in order to send custom C2C messages. Since we have now control over the traffic sent from chip-0 (since we are using a C2C interface instead of a SNF) we add the two new ports, namely c2c_out_port and c2c_in_port, to the simulator (as described in Section 5.3). Then, we add the SNF functionality to the C2C interface's code. This allows the C2C interface to be instantiated in place of the previously

used SNF to validate that the two new ports were working. After validation of the new ports, we add the new C2C message type (as described in Section 5.7.3). Finally, we implement support for write transaction and, after extending the C2C controller and modifying the injected trace to inject write requests, we get to a new working simulation state.

5.5.3 Coherent C2C interface

The last main development milestone until we reached the current state of the model is called "coherent c2c interface". This step implies the addition of a requester node on chip-1. Thus, the coherence must now be maintained between the two chips. Figure 5.9 shows the architecture with the added RNF1_0. We note that the HNF0_1 of the previous architecture (Figure 5.8) has been removed. This is because now, the C2C interface 0 is addressed directly by the RNF0_0 instead of acting like a SNF. With this architecture, the coherence must be maintained between RNF0_0 and RNF1_0. This prompted the extension of the HNF as well as the other additions to the model later explained in this Thesis.

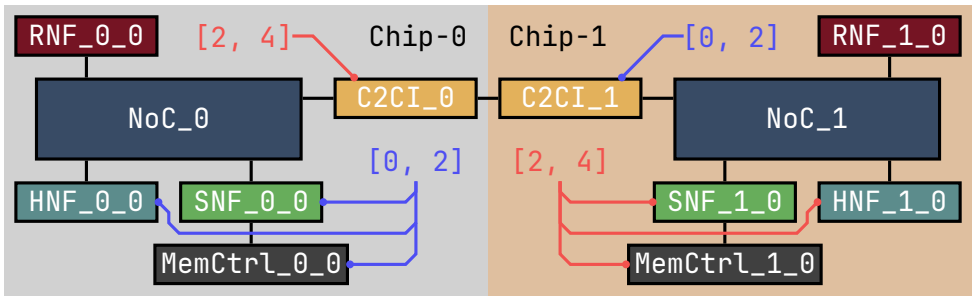


Figure 5.9: Fully coherent chip-to-chip architecture.

When starting this development phase, we kept using the traffic generators. This allowed us to gradually add features while controlling all the run parameters, especially memory access patterns. When we reached a working state with very simple read and write request injections, we switched to using a simple benchmark application (i.e., STREAM benchmark) in syscall emulation mode. We then started to implement a more diverse cast of access patterns and message types that could be handled by the C2C interface controller. Once the STREAM benchmark was working with this architecture, we started to work with full-system simulation. We needed to discover new problematic access patterns in order to extend the C2C interface controller to handle them. The Ruby environment has a random traffic tester component that can prove useful for this task. However, in the objective of running full-system simulation, we decided to use the booting of and OS (Ubuntu Linux 18.04) as workload. The booting of the OS is generating a lot of access to memory and allowed us to greatly increase the robustness of the model.

In this section, we have presented the different steps to reach the current development phase of the model. We believe that controlling the complexity by starting with the simplest simulation setup possible is a valuable approach. Moreover, we leverage the simulator's tools like traffic generators and debug flags. The simulation results have shown the capability of the C2C interface controller to handle syscall emulation simulation running benchmark applications to completion. Most importantly, it demonstrates that our design approach satisfies all gem5 and Ruby constraints and showcases our expertise of the key constructs that are necessary to achieve a fully-fledged C2C simulation in gem5.

5.6 Inter-chip handshake mechanism

After reaching a point where *c2c-gem5* could handle full-system simulation with a fully coherent architecture, we started to experiment with more complex multithreaded benchmark applications. Complex memory access patterns were crashing the simulations, most frequently with *invalid transition* issues, where an event is triggered but does not correspond to any possible transitions. We implement a handshake mechanism as a way to avoid some of these concurrency issues.

5.6.1 Motivations

The C2CI must keep track of multiple in-flight transactions that are currently being processed by storing the necessary metadata (e.g., source, destination, type of transaction, etc.) locally in Transaction Buffer Entries (TBEs). This is straightforward when transactions operate on different addresses. However, it becomes significantly more complex when transactions operate on the same address. Figure 5.10 shows a flow schematic of two requests to the same address and from local controllers, at the same time, to their local C2CI. The C2CI-0, upon sending the SnpSharedFwd request, transitions to a blocked state, waiting for a response and / or data message. Since neither C2CI controller is aware of remote requests, both proceed with their request. This creates a deadlock in the FSM. One solution to this problem is to allow multiple in-flight transaction for the same address. However, this solution presents a great development challenge in its complexity and time (Section 6.2 mention this as a part of our future works). Another solution is to add a *handshake* mechanism for request initialization. The purpose of the handshake mechanism is to enable two controllers to acknowledge their readiness to process a request for a specific address.

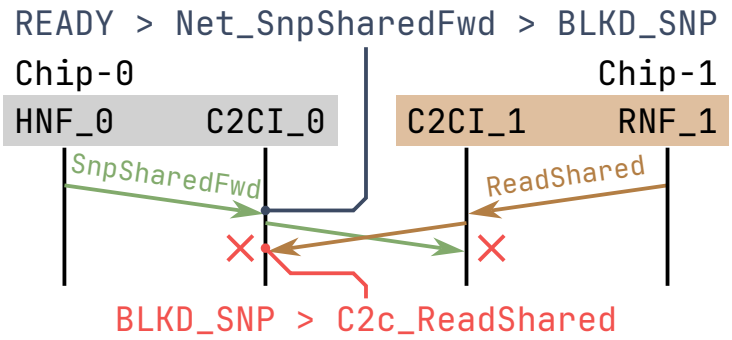


Figure 5.10: Flow diagram of two requests to the same address arriving at the same time to the C2CI and creating a deadlock.

5.6.2 Implementation

We develop a *handshake* mechanism. It is a synchronization technique that ensures reliable communication between multiple components (in our case, the C2CIs). It involves a structured exchange of control messages between a sender (initiator) and receiver to establish and confirm the transfer of message or, in our case, the readiness to service a request. The handshake mechanism is used to solve concurrency issues and solves race conditions for the C2CI controllers.

The handshake mechanism of the C2CI controller is triggered when a request from the network side arrives in the input port of the controller. The C2CI then goes through a series of checks in order to know if it can proceed with the request directly or if it asks for permission to the receiver C2CI. We chose to give priority to the interface that is local to the address of the request. For example, if chip-0 hosts a memory range from address 0 to 50kB, the C2CI local to this chip's NoC will have the priority for the address `0x100` (256) since it is comprised in the [0; 50kB] range. In order for the C2CI to know if an inbound request is within its managed memory region, we add a new method to SLICC called `hasDownstreamClient(Addr addr)`. This method goes through all the machines that are accessible locally to the C2CI. If no potential destination is found for this address, it means that no local controller is managing this address, and subsequently that the C2CI does not manage it. Thus, this means that the C2CI does not have priority for this address. On the other hand, if a local controller is found as destination, it means that the address is managed by a local controller (i.e., HNF), and the C2CI has priority. The method is implemented in the AbstractController parent class and made available in the SLICC code. Listing 5.8 shows a use case of the method in the SLICC code of the C2CI controller. This snippet is used when checking if an inbound request has priority and can be treated directly by the C2CI. If the request can be treated, it is enqueued in the reqRdy internal queue (see Section 5.2.4) for further processing.

```
if (hasDownstreamClient(address)) {
    // address is local, we have priority
    peek(reqInPort, CHIRequestMsg) {
        enqueue(reqRdyOutPort, CHIRequestMsg, 0) {
            out_msg := in_msg;
        }
    }
}
```

Listing 5.8: Use example of the hasDownstreamClient in the SLICC code of the C2CI controller.

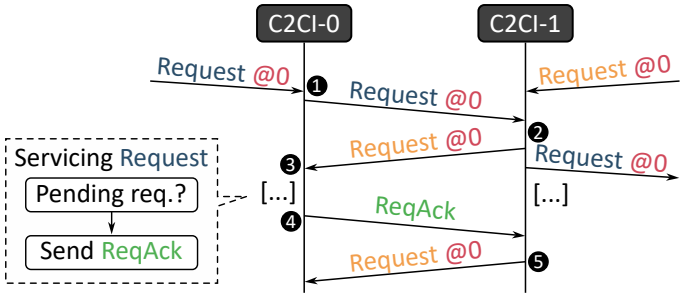


Figure 5.11: Handshake mechanism flow diagram with two requests arriving simultaneously.

Figure 5.11 shows a flow diagram of a message exchange for request acknowledging between two C2C-Interface controllers. In this example, the C2CI-0 is handling the address range containing the address of the request. This means that it has priority and does not need acknowledgment. In ❶ C2CI-0 receives a CHI request. Then, the controller determines if it is granted priority or not. In this example, blue request has priority, so it is sent to the C2C link. In ❷ two things happen. First, C2C-1 checks if the yellow request has priority. It is not the case, so C2CI-1 goes through the handshake process. The yellow request is sent to the C2C link with a special flag. Then, the initial request is enqueued in an internal queue to wait for the acknowledgment. This queue is labeled stalled queue (see Section 5.2.4). Second, the blue request is received by C2CI-1. The controller determines how the request is treated. In this example, the request has priority and is sent to NoC-1. In ❸ C2CI-0 receives the yellow request. The request has the *acknowledgment* flag. Since C2CI-0 is treating the blue request, the yellow request is not treated directly. A TBE flag for the pending request is set. In ❹, the blue request is finalized. C2CI-0 checks if there are pending requests and sends an acknowledgment message accordingly. In ❺, C2CI-1 receives the acknowledgment, dequeues the pending request of the stalled queue and processes it.

5.6.3 Handshake mechanism implementation summary

We develop a novel handshake mechanism that resolves concurrency issues in the Chip-to-Chip Interface (C2CI) controllers by introducing a systematic method for request prioritization and synchronization. By implementing the `hadDownstreamClient()` method and creating a structured message exchange protocol, we enable reliable communication between the two networks via the C2CI controllers. This mechanism provides a robust solution to potential deadlocks scenarios when multiple controllers simultaneously issue requests to the same memory address, offering a time and implementation alternative to allowing multiple in-flight transactions. The handshake mechanism allowed us to solve corner-case problematic access patterns and successfully run benchmark applications.

5.7 Extending CHI for multi-chip systems

The CHI protocol implementation of Ruby has been developed for monolithic networks in mind. This results in the loss of Chip-to-Chip functionality between the AMBA CHI reference and the gem5 implementation. One important aspect is the Home Node (HNF) tracking capabilities. As the HNF manages a specific memory range, it tracks the status of each cache line that is shared in the network. Real life implementations of the HNF have remote tracking capabilities (see the hardware platform in Section 5.9.1). This means that the HNF tracks the status of cache lines shared by local caches as well as cache lines shared in other chips. However, in gem5 and the Ruby implementation, this is not the case. The HNF only tracks cache lines states for caches that are local to the NoC. In addition to the C2CI, we introduce modifications to the HNF and CHI message format of the default gem5 CHI protocol implementation.

5.7.1 HNF extensions

We extend the HNF to track off-chip sharers. Upon completing a local request, the HNF will receive a completion message (e.g., CompAck, snoop response, etc.). The HNF then updates the *dir_sharers* list. This list is a member of the TBE that tracks the sharers of a cache line. After updating the TBE dir_sharers field, the HNF will store this information in the *directory entry* before deallocating the TBE. When a new request is starting its process for the same address, the TBE dir_sharers field is initialized from the previously updated directory entry. This allows the HNF to always have an up-to-date list of all sharers for a specific address. When the HNF needs to send snoops messages to sharers of a line, the dir_sharers field is used to route the snoop to destination. However, this solution is limited when sharers are off-chip. The HNF will track

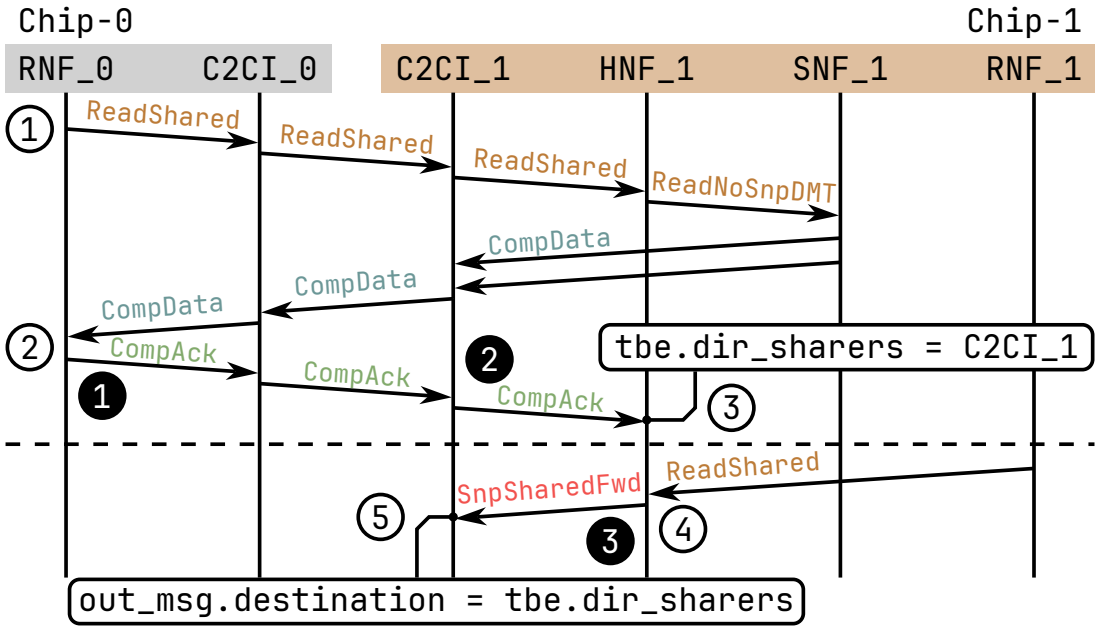


Figure 5.12: Flow diagram showcasing a problematic access pattern requiring off-chip sharer tracking.

the local *responder* as being the sharer of the line. Figure 5.12 shows the flow diagram of a problematic access pattern. This access pattern showcases how the limited off-chip sharer tracking capabilities of the HNF can lead to routing issues.

- In ①, the RNF-0 in Chip-0 issues a read request to an address managed by the HNF-1 on Chip-1.
- In ②, the request is successfully serviced. The RNF-0 sends a `CompAck` message, acknowledging the data reception.
- In ③, the HNF-1, upon receiving the acknowledgement message, updates the dir_sharer field. However, the field is populated by the C2CI-1 *MachineID*, it being the sender of the CompAck message.
- Later, in ④, the RNF-1 issues a read request to the same address. The HNF-1, knowing that a cache is sharing the line, sends a `SnpFwd`. This message asks its recipient to forward the cached data. When the message is sent, the HNF-1 uses the dir_sharers field to route the message. The message is correctly routed to the C2C2-1.
- Unfortunately, past this point, in ⑤, the C2CI-1 does not have sufficient information to route the Snoop message to destination.

The same Figure 5.12 can be used to show our solution. In ❶ the CompAck message has a new c2c_requester field which is filled with the RNF-0 MachineID. This ID

is transmitted off-chip. In ❷ the C2CI-1 populates the field of the c2c_requester field of the local CompAck message. This field is then used to track off-chip sharers in the HNF with a new field called c2c_sharers in both the TBE and dir_entry of the HNF. In ❸, when the HNF-1 issues the SnpFwd snoop message, the c2c_sharer field of the message is populated by the TBE c2c_sharers field. Finally, this information is used by the C2CI-0 to route the snoop request to the RNF-0.

As explained, a new field is used to store the sharer of the cache line in the HNF, and in the C2C and CHI messages carrying it. This field is using a variable type called `MachineID`. Every controller in the Ruby subsystem gets assigned a unique identifier. The MachineID of the controller allows to precisely identify each controller of the system. Then, by using available methods like `add(MachineID id)` we can translate this MachineID to a *NetDest*. The NetDest variable type is a bitmap that is used to route the messages in the network. Bitmap indexes allow to store the destinations as valid or not (i.e., 1 or 0). For example, if the controller with the ID "2" is a valid destination, the bit 2 of the bitmap index will be set at 1. Effectively, this means that the NetDest variable can store multiple destinations.

When a message is sent, the destination of the message is of type NetDest. This means that we can use a MachineID to fill in the destination of a message as follows: `out_msg.Destination.add(subrdinate_controller_id)` where the subordinate controller id is a MachineID. We decide to use MachineID as a variable type to transmit and store the c2c_sharers because each MachineID is unique to the whole Ruby subsystem, whereas the NetDest is only local to a network model (where the target architecture houses two). This offers advantages but comes with a tradeoff. As opposed to MachineID, which stores only one element, NetDest can be used to store multiple destinations. NetDest is used in the vanilla CHI protocol to store the sharers of a line, because it allows to store multiple sharers (e.g., potential destinations of a future snoop request). Moreover, the *enqueue* method of SLICC (see Section 3.5.2) will create and enqueue as many messages as there are destination elements in the NetDest variable.

Thus, each action of the HNF that *enqueues* messages using a NetDest destination field will enqueue as many messages as there are destination in the NetDest variable, automatically filling a unique destination for each message. We have considered using NetDest to store the c2c_sharers but for implementation simplicity sakes, have kept using the MachineID variable type. However, as we advance into the next steps of the development (see Section 6.2) we plan on using NetDest to handle multiple requesters per chip and multi-chip (more than two) architectures.

5.7.2 Message extensions

As previously mentioned, in order to route snoops to their off-chip destination and track the off-chip sharers, we add a new c2c_sharer and c2c_requester field to the CHI message and a c2c_sharers field to the HNF TBE and directory entry.

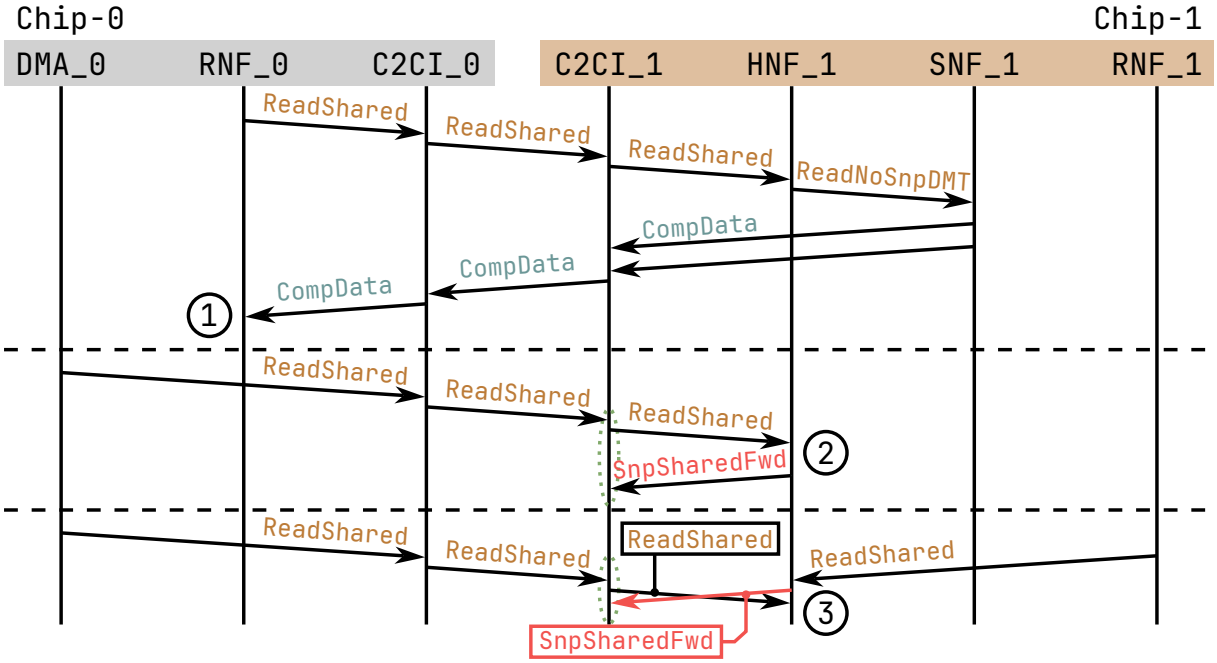


Figure 5.13: Flow diagram showcasing a problematic access pattern requiring a tracking of the HNF and CHI message extension.

To cover a corner-case access pattern, we need to implement a new field in the CHI message. The Figure 5.13 showcases the problem.

- First, in ①, we can see a request being finalized. RNF-0 now has the line cached. Following this, the HNF-1 will now track the RNF-0 as a sharer of the line.
- Second, the Direct Memory Access (DMA) controller DMA-0 issues a read request to the same address. In ②, the HNF-1 will send a snoop back to request RNF-0 to forward the data to the original requester (DMA-0). This scenario is valid.
- However, in ③, we show a scenario that has a similar outcome, the C2CI-1 receives a SnpFwd request after sending the Read request to HNF-1. For C2CI-1, this scenario is indistinguishable from the valid one shown in ②. However, the origin of this snoop message is from a concurrent request issued by RNF-1. Without any modifications, this access pattern will cause routing errors when the data is forwarded.

We add two flags (i.e., boolean values) called `off_chip` and `c2cOrig` to the CHI message. The off_chip field is set to true for all messages sent from a C2CI controller. This value is then tracked by the HNF TBE field similarly named c2cOrig (i.e., C2C origin). When the HNF sends a message, the c2cOrig flag of the message takes the tracked value of the TBE. This allows the C2CI to check if an inbound message is off C2C origin or not, allowing to distinguish between scenarios like shown in Figure 5.13.

5.7.3 C2C message

We define C2cMsg using SLICC to carry requests, snoops, data, and responses over the C2C link. C2cMsg contains metadata serving two primary purposes: first, to facilitate transmission through the C2C link, and second, to enable the generation of appropriate CHI messages for injection into the destination NoC. This also includes CHI-specific fields like *RetToSrc* that affects whether or not a snoop response returns data, or *AllowRetry* if the request can be retried or not. Additionally, C2cMsg includes C2C-specific fields like *C2c_sharers* that indicate the off-chip sharer associated with an address for snoop routing. A Priority field is also included for C2C link access control to indicate that the CHI message is a priority snoop and should be treated, putting the ongoing transaction on hold. This link access control is used to manage the number of in-flight transactions. This message can later be further modified in order to model specific coherent C2C protocols like CCIX.

5.7.4 Extensions summary

We extend the Home Node (HNF) tracking capabilities to address off-chip sharer tracking limitations in gem5 and the CHI Ruby implementations. By introducing new fields such as `c2c_sharers`, `c2c_requester`, `off_chip` and `c2cOrig` in the CHI message format and HNF TBE and directory entry, we enable accurate routing and tracking of cache line sharers across different chips. These modifications resolve critical routing issues in specific access patterns, particularly in scenarios involving concurrent requests and cross-chip interactions. The proposed extension provides the tools needed for maintaining cache coherence in a multi-chip Ruby simulation when using more advanced multithreaded applications.

5.8 Model complexity and development effort

The previous sections described our coherent chip-to-chip interconnect model's functionality and implementation. While these discussions provide insight into the model's

design, they do not fully capture the complexity and effort involved in its development. To provide context about the development effort, we present quantitative metrics about the implementation of *c2c-gem5*.

The scope of the implementation is reflected in the significant codebase [15]. More than 3,000 lines of SLICC code define the core C2CI controller, while hundreds of additional Python and C++ code lines extend the simulator and integrate the model. Moreover, debugging and calibration required long simulation campaigns, where each simulation took up to 15h to complete. The Chip-to-Chip Interface controller's FSM implementation includes:

- **17 Buffers** for internal queues and I/Os
- **22 States**
- **128 Events** that trigger state transitions and actions
- **68 Actions** specifying different operations performed during transitions
- **103 Transitions** allowing the controller to navigate the states in reaction to various events

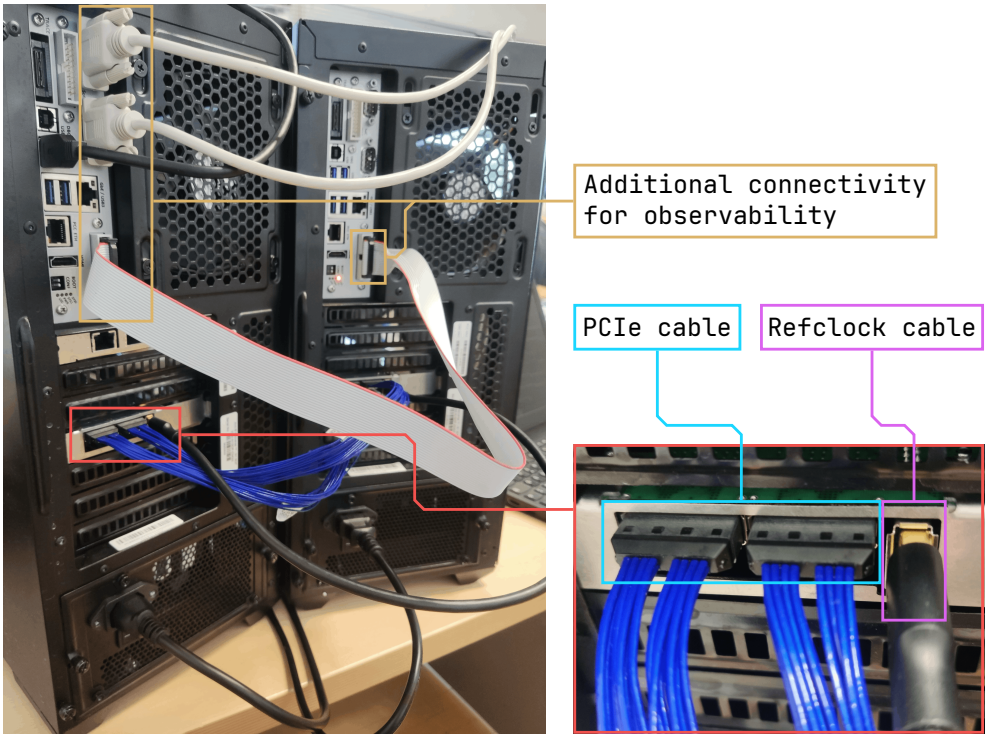
The development history shows over 800 commits, indicating continuous refinement and testing throughout the implementation process. These metrics demonstrate the substantial engineering effort required to create an accurate chip-to-chip interconnect model in an open-source project like gem5.

5.9 Interface calibration

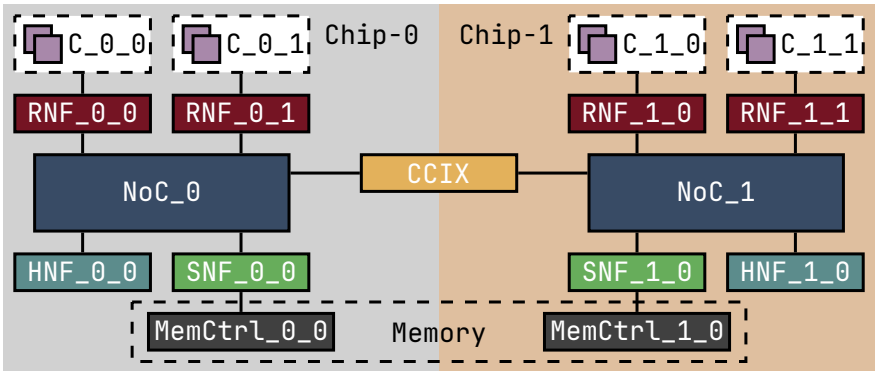
After implementing flow control mechanism like the handshake mechanism and adding new features to both the C2CI and HNF controllers as well as the CHI messages, we can run multithreaded applications in full-system simulation mode with gem5. In order to increase the accuracy of the model we calibrate it using hardware. We calibrate the model targeting DRAM access times.

5.9.1 Hardware platform

We use a real hardware testbed to calibrate the C2C model. This testbed is built around two Quad-core N1 (Arm v8.2a) Software Development Platforms (SDP). These two platforms use the Arm AMBA CMN-600 NoC protocol and are interconnected through a specialized PCIe riser card using the CCIX [27] protocol. Figure 5.14a shows the hardware testbed back panel with connectivity details. The two boards are interconnected



(a) Hardware platform back pannel with connectivity detail.



(b) Hardware platform high-level architecture diagram.

Figure 5.14: Hardware platform picture and architecture diagram.

through the PCIe cable and refclock cable (in blue). The two platforms also house additional connectivity for observability (e.g., serial connections). The Figure 5.14b shows the high-level architecture of the hardware platform with the Arm CHI semantic.

5.9.2 Calibration

We use the real hardware platform described in the previous Section to calibrate the latency of the chip-to-chip link in our model. We design a microbenchmark to measure DRAM access time, on the real hardware, using the following steps:

1. Allocate a large array of 24GiB (larger than the local DRAM capacity to force memory allocation into the remote memory).

```
#define N (1<<28)  // More than half of the memory (2GB = 2^28 int64)
long unsigned *data;
data = malloc((long unsigned)12*N*sizeof(long unsigned));
```

2. Randomly select a position in the array. The predefined reference address (`refe_add`) remains constant, while the second address (`rand_add`) is randomly generated at each iteration. The `gen_rand` function generates a uniformly distributed random number.

```
int unsigned rand_add = gen_rand(((unsigned long)N*12) >> 9, 0) << 9; // aligned to
     64B cache line
int gen_rand(unsigned long hi_num, int low_num){
    return (rand() % (hi_num - low_num)) + low_num;
}
```

3. Read data from the selected position and flush the read data. Here both `data_pp1` and `data_pp2` dereference and access the two addresses. The `flush` macro ensures that the data is flushed from the caches so that the next iteration fetches it from DRAM.

```
for (ind = 0; ind < ((1 << 18)); ind++){
    data_pp1 = ((unsigned long **)*data_pp1);
    data_pp2 = ((unsigned long **)*data_pp2);
    flush(data_pp1);
    flush(data_pp2);
    acc=acc+2;
}
```

4. Go to step 3 and repeat multiple times to reduce measurement noise.

5. Measure the accumulated execution time and divide by the number of reads to obtain DRAM access time. The PAPI (Performance Application Programming Interface) library is used to measure cache and TLB events. During the loop, the events are recorded using `PAPI_start`, `PAPI_read` and `PAPI_stop`.

```
PAPI_start(EventSet);
PAPI_read(EventSet, values);
PAPI_stop(EventSet, values);
```

6. Go to step 2 and repeat for a new address.

7. Compute the execution time per DRAM access.

```
cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("DRAM time (ns): %f\n", (1e9 * cpu_time_used) / (acc));
```

Running this microbenchmark on the real hardware platform allows us to generate a histogram distribution of DRAM access times. The top graph in Figure 5.15

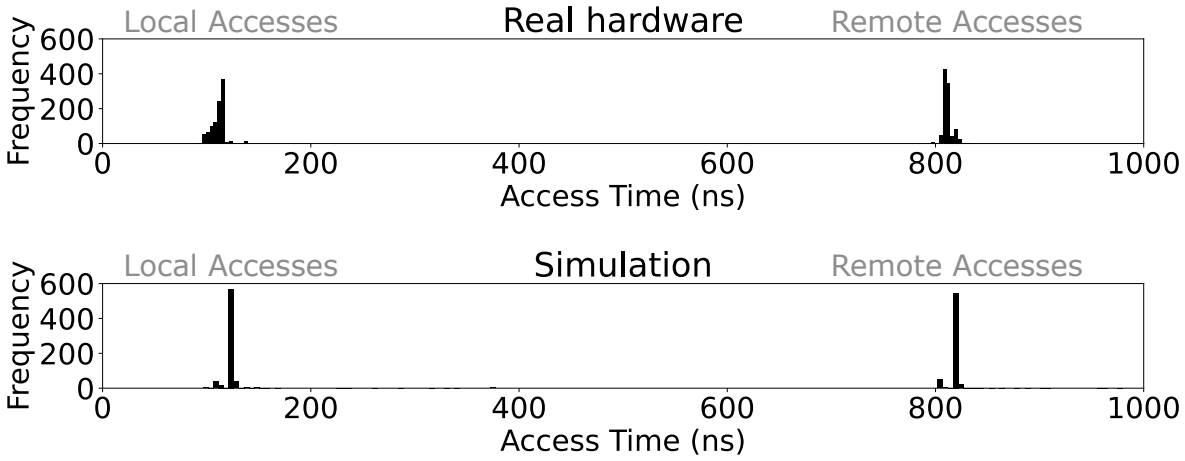


Figure 5.15: Access time distribution on the hardware testbed (top) and the tuned simulation (bottom).

shows the resulting distribution of DRAM access times. Two distinct populations are identifiable: one centered around 110 ns and the other around 810 ns. The first population corresponds to local DRAM accesses, while the second represents off-chip DRAM accesses. We validated this assumption by using the Linux `pagemap` interface to inspect the corresponding physical addresses. We observe an 8× difference between local and off-chip access times.

We use trace injection to reproduce the same experiment with the simulated C2C architecture. Using the gem5 traffic generator component, we can inject custom traffic into the memory subsystem. It allows us to target physical addresses providing full control of the access pattern. Figure 5.16 shows a flow diagram with two different access patterns. This diagram shows the transactions exchanged between Ruby controllers for local and remote DRAM accesses. On the left side, local addresses do not go through the C2C link. As observed, the request is issued by the L1d-0 to the L2-0. Since L2-0 does not have the data, it forwards the request to the HNF-0. The HNF-0 then prompts the local subordinate node for the data. The data is then sent back to the L2-0 and ultimately the L1d-0. All requests take place locally, transmitted with Chip-0's NoC. On the right side, we see the request being routed to Chip-1. When the request reaches L2-0, since the target address is managed by Chip-1, the request is sent to the local C2CI-0. The request is treated similarly between Chip-1's controllers. Then, the data is routed back. The figure also highlights in red the delay added to calibrate the chip-to-chip link, emulating the behavior of the real hardware platform.

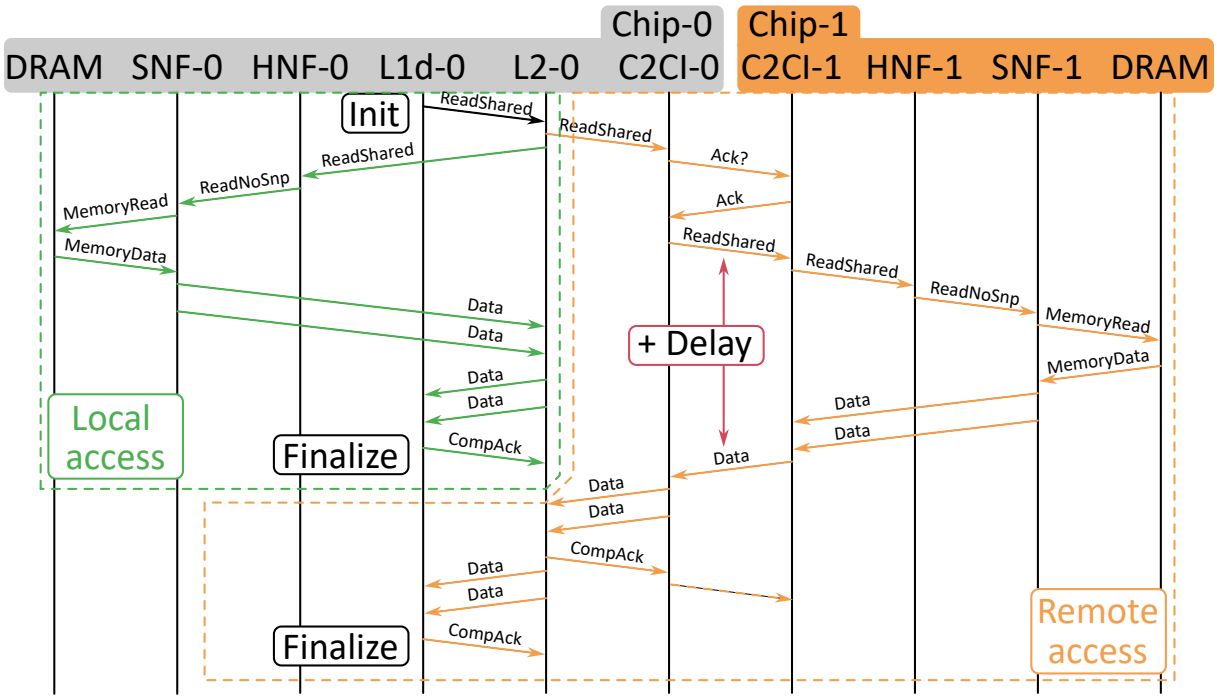


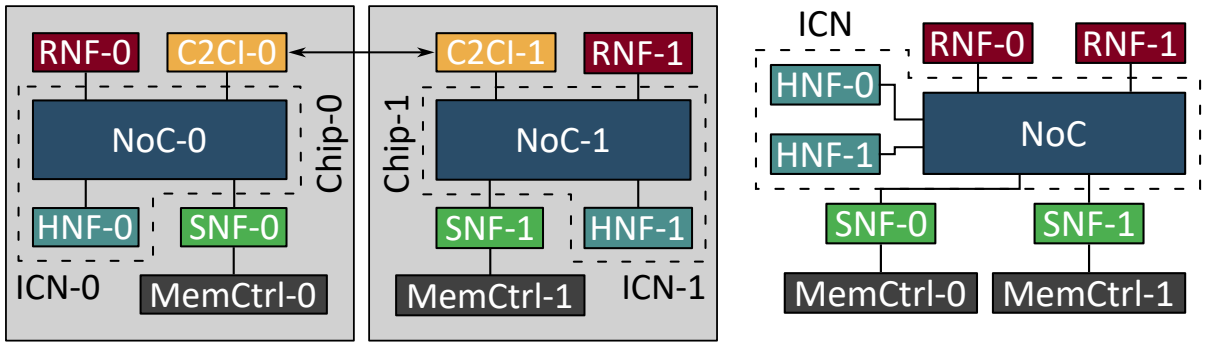
Figure 5.16: Flow diagram with two different access patterns for local and remote access.

5.9.3 Calibration validation and model accuracy

The bottom graph of Figure 5.15 shows the distribution of DRAM access times in the simulated model after calibration. By comparing the two graphs, we see a better matching DRAM access times. We conclude that the simulation model can easily be calibrated to precisely reproduce the access time experienced by the real hardware platform. The calibration results demonstrate the accuracy of our simulation model in reproducing real hardware behavior for chip-to-chip memory accesses. The histograms of Figure 5.15 show two distinct populations in both hardware and simulation environments. Local DRAM accesses clustering around 120ns and remote accesses around 810ns, maintaining the observed 8x latency ratio. Our simulation framework successfully captures this distribution, closely matching the access patterns and latencies measured on the physical N1 SDP. Through the calibration of the C2C link delay parameters, we achieved high fidelity in modeling the interconnect behavior, providing a foundation for further calibration. This validation gives us confidence that the simulation framework can accurately represent real-world chip-to-chip communications patterns, given enough calibration. In Section 5.10 we show results of simulations using this calibrated model.

5.10 Results

5.10.1 Modeled architectures



(a) (b)

Figure 5.17: High-level C2C (left) and reference (right) architecture with CHI components and C2CIs.

We execute gem5 22.1 on a server with Intel Xeon Silver 4214R processor @ 2.4 GHz running CentOS linux 7. We configure c2c-gem5 to model a *two-chip* architecture, as illustrated in Figure 5.17a. Each chip features a CPU with a 64 KB 4-way set-associative L1 data cache and a 1MB 8-way set-associative L2 cache, one HNF, one SNF, and one C2C interface. Each chip also includes a memory controller connected to the SNF. Memory ranges are non-interleaved as follows: [0,1.5 GB) for Chip-0 and [1.5 GB, 3 GB) for Chip-1. Each memory controller includes one channel with 2 ranks, each rank containing 8 banks of DDR3-3200 MT/s. We also model a single-chip *reference* architecture that has the same characteristics but without the C2C link (right side of Figure 5.17). All controllers are assembled around a unique NoC. Both architectures are simulated in full-system mode, running Ubuntu Linux 18.04. The OS boots using non-caching simple CPU models to ensure fully deterministic simulation while reducing boot time by bypassing the cache subsystem. Upon entering the Region-of-Interest (ROI), we switch to the TIMING CPU model, which is more precise and compatible with the Ruby subsystem.

5.10.2 Workloads

We evaluate c2c-gem5 using program from the PARSEC [16] benchmark selecting those that complete the simulation within 15 hours in the reference architecture. PARSEC primarily represents multithreaded application that demand large amount of shared memory.

5.10.3 OS integration

We validate the programmer's view of out multi-chip model. Using full-system simulation, we can boot the OS and run commands like the `lscpu` Linux command to gain insight into the underlying hardware. We execute this command on the simulated platform, and the OS detects a 2-core processor in both the reference and the C2C architecture. When running the same command on the real hardware platform, we observe a consistent result, with the OS detecting an 8-core processor, while both interconnected platform each consist of a 4-core processor.

5.10.4 Simulating multithreaded applications

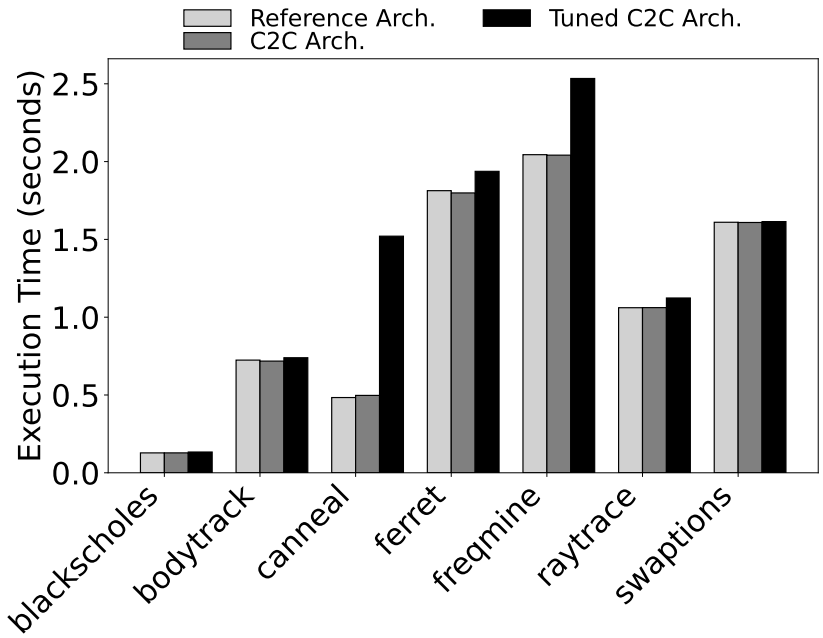


Figure 5.18: Execution times

Figures 5.18, 5.19 and 5.20 shows the results from running PARSEC workloads on the reference and chip-to-chip gem5 architectures described in Section 5.10.1. Figure 5.18 shows the execution times for the reference and the C2C architectures before and after the chip-to-chip link calibration described in Section 5.9. We make two key observations. First, the reference and baseline C2C architectures achieve very similar execution times. The minor differences are due to variations in request ordering caused by slight runtime variations. Second, the calibrated C2C architecture shows a noticeable slowdown in some programs, while others remain unaffected. In particular, `canneal` shows the largest execution time difference, with a 3× increase. Additionally, `freqmine` exhibits 25% increase in execution time. This result highlights the importance of using calibrated models in simulation for accurate performance evaluation.

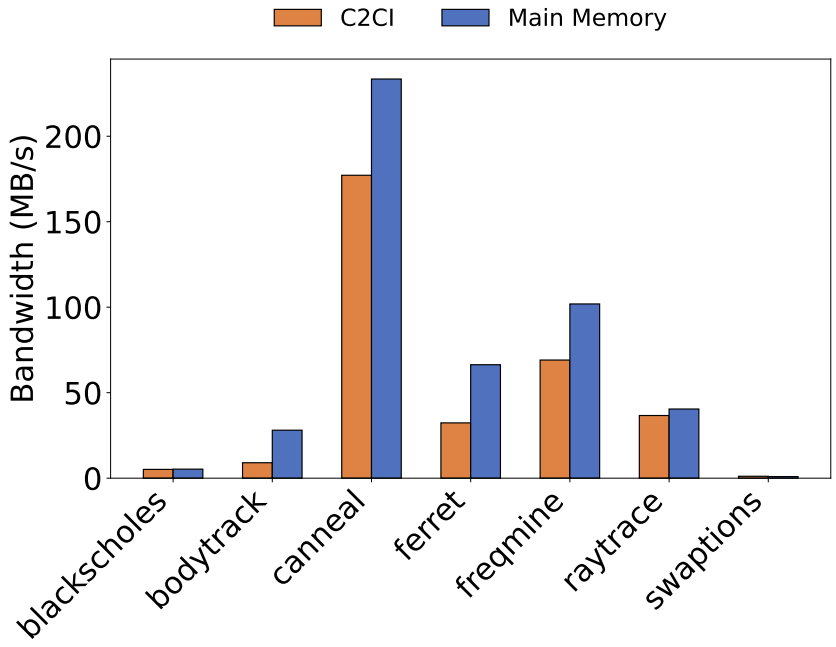


Figure 5.19: Memory vs C2C link bandwidth

Figure 5.19 shows the bandwidths for the C2C link and the main memory (combined for both local and remote main memory controllers) in the calibrated C2C architecture. We make two observations. First, the applications with the highest slowdown also show the highest C2C bandwidth utilization, with the slowdown being directly proportional to the bandwidth utilization. Second, the C2C link bandwidth is typically lower than the aggregated main memory bandwidth but varies depending on the application. *raytrace* shows C2C link bandwidth comparable to the main memory bandwidth, whereas `bodytrack` achieves only a third of the main memory bandwidth.

Figure 5.20 shows the normalized distribution of message types traversing the C2C link. We observe a similar distribution for the applications exhibiting the highest slowdown: `canneal`, `ferret`, `freqmine` and `raytrace` benchmarks. However, the low proportion of snoop requests suggests that the slowdown is not caused by coherence traffic, but rather by the limited C2C link bandwidth. Thus, moving from the modeled PCIe based interconnect to a more integrated chiplet-based interconnect could reduce the performance overheads observed in our experiments.

5.11 Summary

This chapter has detailed the design and implementation of a coherent chip-to-chip interconnect model within the gem5 simulator, enabling the exploration of multi-chip architectures. We addressed the key challenge of bridging independent NoCs within

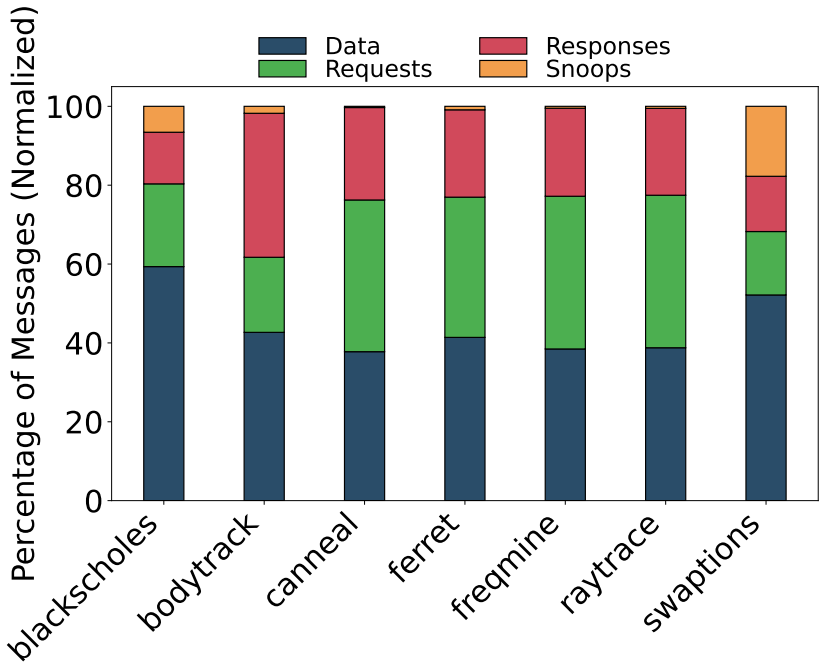


Figure 5.20: Normalized C2C traffic types

Ruby by extending the `AbstractController` class and the SLICC compiler, allowing the use of gem5 classic ports for inter-chip communication. This novel approach enables flexible and accurate modeling of chip-to-chip links. We demonstrated our main development strategy, consisting of controlling every parameter of the simulation while developing each main functionality of the model, effectively controlling the complexity. Furthermore, we introduced a handshake mechanism to manage concurrent requests to shared addresses, ensuring correct serialization and preventing deadlocks. We extended the existing CHI protocol implementation in gem5 to support essential multi-chip functionalities, such as remote sharer tracking, by extending the HNF behavior and CHI message format. Then, we gave some insight on key metrics about the codebase and C2CI controller, in order to showcase the development effort provided during this project. Finally, we calibrated our model against a real hardware platform using a custom microbenchmark, demonstrating the ability to accurately capture the performance characteristics of a real-world chip-to-chip interconnect. This calibrated model, validated through full-system simulation of PARSEC benchmarks, provides a powerful tool for evaluating the performance implications of different multi-chip architectures and interconnect designs.

CHAPTER 6

CONCLUSION AND FUTURE DIRECTIONS

In summary, this thesis tackles the modeling of coherent chip-to-chip interconnections. We provide insight into the gem5 simulator, a methodology to extend cache coherency across chip, and a complete implementation of a coherent chip-to-chip interconnect model. The proposed modeling methodology offers a first step towards complete chiplet-based architecture exploration by allowing to extend coherency between chips under a unified OS environment.

In this chapter, we summarize the key contributions of this thesis, we provide insight for future work and end with concluding remarks.

6.1 Summary of contributions

This thesis provides a new methodology to model novel coherent chip-to-chip interconnects. In this section, we summarize the two main contributions of this project.

6.1.1 Extending coherency across chips

The first contribution of this thesis is an in-depth analysis of coherent chip-to-chip modeling. We provide insights on the main challenges faced when modeling coherent interconnect protocols.

We provide extended details about the routing of messages inside a traditional chip using the CHI protocol. Then, we talk about the requirements and challenges of extending coherency across chips. We find that the simulation platform must allow two network models in a single simulation instance. We also go over the simulation platform's OS usage and requirements. Then, we go over the implementation requirements. We show that new metadata, in both controllers and messages transmitted, is needed in order to treat corner case access patterns between the networks. We also

show that concurrency issues can lead to deadlocks, and that link management and flow control mechanisms at the interface between two chips is needed.

6.1.2 Full-system simulation of cache-coherent interconnects

The second contribution of this thesis is a complete implementation and solution to model coherent chip-to-chip architectures. We extend the widely used gem5 simulator. First, we extend the C++ kernel of the simulator to allow multiple network models to be instantiated and connected together in one simulator instance. Moreover, we extend the CHI coherence protocol in order for it to handle multi-chip coherency. This means extending the HNF controller and the CHI message structure. Finally, we develop a new Ruby controller called Chip-to-Chip Interface (C2CI) that allows coherent communication between two chips using a new C2C message and newly developed C2C ports. We expand key metrics of the codebase and the C2CI controller, showcasing the development efforts. We then provide a strategy to accurately calibrate the model using a real hardware testbed. We validate the model using PARSEC benchmark applications, showing key metrics extracted from simulations.

The development of *c2c-gem5* was not a linear process and was punctuated by challenges and breakthroughs. The initial phase of this project required more than six months dedicated to pinpoint the correct methodology for the model implementation. This initial phase was key to understand the complexities of ensuring correct cache-coherence between multiple simulated networks. The learning process was deeply tied to the debugging efforts, as multiple errors occurred because of unanticipated conflicts, highlighting the non-trivial nature of working with open-source projects like gem5.

The main development and debugging phase lasted on year. Among multiple roadblocks that sometimes took multiple weeks to understand and solve, key technical challenges included:

- Extending the SLICC compiler functionality
- Debugging KVM CPU models for fast OS boot support
- Resolving Ruby's port handling limitations

This work resulted in a contribution to an international conference paper: Luis Bertran Alvarez et al. "c2c-gem5: Full System Simulation of Cache-Coherent Chip-to-Chip Interconnects". In: *29th IEEE International Conference on Design, Automation & Test in Europe (DATE)*. 2025. The source code and the data used in this work are open-source and available on GitLab [15].

In addition, we plan to submit an extended version of this work to a journal in the upcoming months. This extension will include a solution for c2c-gem5 and the CHI protocol implementation to handle multiple chips and multiple requesters in each chip, as explained further in this Section. This addition to this contribution is still a work in progress.

6.2 Future work

In this section, we provide directions for future work based on the contributions of this thesis. We have presented an implementation of a chip-to-chip interconnect model. This model represents a first step forward in modeling coherent chiplet-based architectures and coherent heterogeneous architectures. However, we believe that there are key features that would allow to reinforce the foundation of the model, making it more easily used and adopted.

First, the current version of the model only supports one requester node (with fully coherent caching) per chip. This limits architecture flexibility for each chip. The first objective for the future of the model is to extend the C2CI and the HNF in order to handle multiple requesters (i.e., RNFs) per chip. For this, the off-chip cache line sharer mechanism needs to be modified. Right now, the variable type used to track off-chip sharers is `MachineID`. This type does not allow to store multiple items, therefore does not allow to track multiple off-chip sharers. Moreover, when sending snoop requests to the sharers of a cache line, the HNF needs to send one snoop per cache that has a copy of the line. This needs to be implemented as well. This extension work is already ongoing.

Second, the current version of the model only supports two chips: two network-on-chip models with one C2CI per NoC. This also limits the capabilities of the solution to model more complex chiplet-based architectures. In order to support multiple chips in one simulation and one Ruby instance, the C2CI model as well as the HNF controller need to be extended. First, the C2CI now need to be able to properly route the C2C message to destination. This can be achieved by adding a new component between all C2CIs. We envision a *C2C Link Switch* component. Moreover, the HNF now needs to track additional cache line status information. We believe that a new external structure can be added to the SLICC code. This structure would allow the storage and management of multiple `NetDest` (i.e., Network Destinations). Each NetDest object would hold the sharers of one chip. Figure 6.1 shows a high-level architecture with 3 chips. Each chip has 2 RNFs. We believe that making this architecture possible in gem5 is a logical next step in extending the model's functionalities. Both this extension

and the previously mentioned multi-RNF extension are work in progress. We expect to ship these features in the upcoming months.

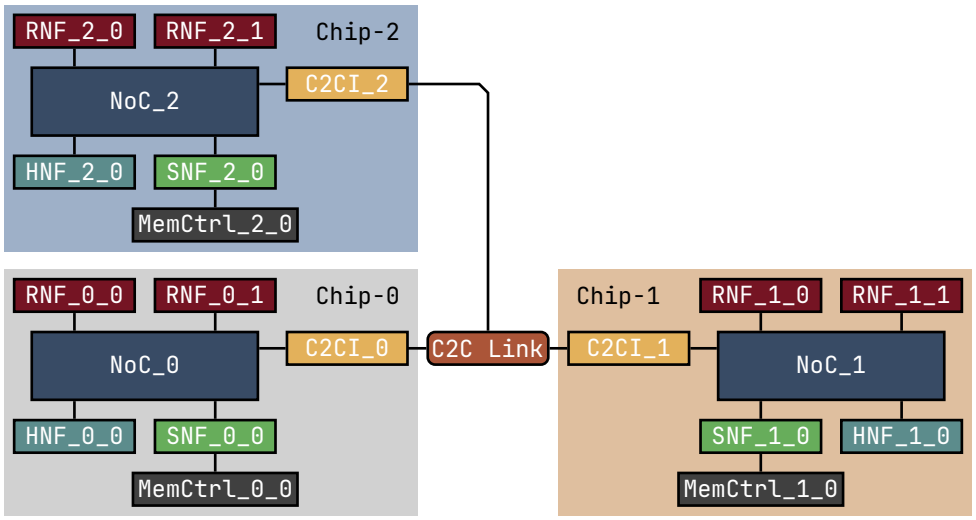


Figure 6.1: High-level architecture diagram with two RNF per chip and 3 chips.

Third, the current version of the model supports one in-flight transaction for a specific address, excepted on select cases. This limits the flexibility of the model. We believe that another key next objective of this project is to allow the model to handle a user-set number of in-flight transactions for a specific address. This means that the C2CI would be able to track ongoing transactions and states. In order to achieve this, it is necessary to be able to save the state of the FSM for a specific transaction, in order to restore the state later on. This technique is already use in the HNF code. States and actions are saved in specific fields.

Fourth, once the model features are satisfactory, we would like to implement a specific coherent interconnect protocol (e.g., CCIX, UCIe). This will allow to compare the model to real implementations using further metrics and more heavily calibrate the model as well as testing its limits.

Finally, we believe that contributing to gem5 by pushing the model to the main version would allow other researchers to modify and build upon our work. This would necessitate that the aforementioned feature be implemented and working. Moreover, the current version of the model is built using the version 22.1 of gem5. In order to contribute, the model needs to be ported to the latest stable version of gem5.

6.3 Concluding remarks

In summary, this thesis has addressed the need for accurate and flexible modeling of coherent chip-to-chip interconnects. We have provided a methodology built on the

gem5 simulator, a complete implementation, and insights into the challenges associated with the design and simulation of interconnect models. Our work started with the analysis of the challenges of extending cache coherency from a single network-on-chip to multiple chips. This first contribution shows that the persistence of routing information and state tracking in different controller is of utmost importance in order to route snoop and responses messages and complete requests. Moreover, we show that to model multi-chip coherence, simulation platforms must allow two network-on-chip models and must allow control over the OS environment. Then, we have developed a solution, c2c-gem5, which provides a comprehensive, expendable and flexible framework for researchers to model multi-chip architectures within a single OS environment. We leverage the flexibility of the SLICC language to develop a new controller and extended existing structures of gem5, adding a new message type and ports and have created a simulation environment that captures the characteristics of chiplet-based architectures. We have also provided a calibration methodology, ensuring that out simulation results are matching real hardware metrics.

Through this work, we have taken a step forward in enabling the research community to explore and optimize the next generation of chiplet-base architectures. We believe that the methodologies and insights that we developed in this thesis will be valuable to both the architecture research and industry communities. We hope that out contribution will aid in the design and development of more efficient and powerful computing platforms.

BIBLIOGRAPHY

[1] Kenneth Aamodt et al. "The ALICE experiment at the CERN LHC". In: *Journal of Instrumentation* 3.08 (2008), S08002.

[2] Niket Agarwal et al. "GARNET: A detailed on-chip network model inside a full-system simulator". In: *International symposium on performance analysis of systems and software (ISPASS)*. 2009, pp. 33–42.

[3] Vinayak Agrawal et al. "Level 4 Autonomous Driving SoC, leveraging chiplet, advanced package and UCIe". In: *2023 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE. 2023, pp. 9–14.

[4] Ayaz Akram and L. Sawalha. "Validation of the gem5 Simulator for x86 Architectures". In: *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)* (2019), pp. 53–58. DOI: 10.1109/PMBS49563.2019.00012.

[5] Ayaz Akram and Lina Sawalha. "A survey of computer architecture simulation techniques and tools". In: *Ieee Access* 7 (2019), pp. 78120–78145.

[6] Anton Shilov. *Intel delays mass production of 10nm cpus to 2019*. Accessed: 14-10-2024. 2018. URL: https://www.anandtech.com/show/12693/intel-delays-mass-production-of-10-nm-cpus-to-2019.

[7] Shahab Ardalan et al. "Bunch of wires: An open die-to-die interface". In: *2020 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE. 2020, pp. 9–16.

[8] Eduardo Argollo et al. "COTSon: infrastructure for full system simulation". In: *ACM SIGOPS Oper. Syst. Rev.* 43 (2009), pp. 52–61. DOI: 10.1145/1496909.1496921.

[9] Russell R Atkinson and Edward M McCreight. "The dragon processor". In: *ACM SIGOPS Operating Systems Review* 21.4 (1987), pp. 65–69.

[10] Todd Austin, Eric Larson, and Dan Ernst. "SimpleScalar: An infrastructure for computer system modeling". In: *Computer* 35.2 (2002), pp. 59–67.

[11] John Backus. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs". In: *Communications of the ACM* 21.8 (1978), pp. 613–641.

[12] Jonathan Balkind et al. "OpenPiton: An open source manycore research framework". In: *ACM SIGPLAN Notices* 51.4 (2016), pp. 217–232.

[13] Bradford M Beckmann and Anthony Gutierrez. "The AMD gem5 APU simulator: Modeling heterogeneous systems in gem5". In: *Tutorial at the International Symposium on Microarchitecture (MICRO)*. 2015.

[14] Luca Benini and Giovanni De Micheli. "Networks on chips: A new SoC paradigm". In: *computer* 35.1 (2002), pp. 70–78.

[15] Luis Bertran Alvarez and David Novo. *c2c-gem5: Full System Simulation of Cache-Coherent Chip-to-Chip Interconnects*. Accessed: 2025-01-17. 2025. URL: https://gite.lirmm.fr/adac/c2c-gem5%5C#.

[16] Christian Bienia et al. "The PARSEC benchmark suite: Characterization and architectural implications". In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 2008, pp. 72–81.

[17] Nathan Binkert et al. "The gem5 simulator". In: *ACM SIGARCH computer architecture news* 39.2 (2011), pp. 1–7.

[18] Nathan L Binkert et al. "The M5 simulator: Modeling networked systems". In: *Ieee micro* 26.4 (2006), pp. 52–60.

[19] T. Bjerregaard and S. Mahadevan. "A survey of research and practices of Network-on-chip". In: *ACM Comput. Surv.* 38 (2006), p. 1. DOI: 10.1145/1132952.1132953.

[20] Hadi Brais, Rajshekar Kalayappan, and Preeti Ranjan Panda. "A survey of cache simulators". In: *ACM Computing Surveys (CSUR)* 53.1 (2020), pp. 1–32.

[21] Andreas Brokalakis et al. "Cossim: An open-source integrated solution to address the simulator gap for systems of systems". In: *2018 21st Euromicro Conference on Digital System Design (DSD)*. IEEE. 2018, pp. 115–120.

[22] Anastasiia Butko et al. "Accuracy evaluation of gem5 simulator system". In: *7th International workshop on reconfigurable and communication-centric systems-on-chip (ReCoSoC)*. IEEE. 2012, pp. 1–7.

[23] Zhipeng Cao et al. "LBDR: A load-balanced deadlock-free routing strategy for chiplet systems". In: *Integration* 96 (2024), p. 102149.

[24] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 2011, pp. 1–12.

[25] Davide Castelvecchi. "Mathematicians welcome computer-assisted proof in 'grand unification'theory". In: *Nature* 595.7865 (2021), pp. 18–19.

[26] Vincenzo Catania et al. "Noxim: An open, extensible and cycle-accurate network on chip simulator". In: *International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. 2015, pp. 162–163.

[27] CCIX Consortium. *CCIX Consortium: Cache Coherent Interconnect for Accelerators (CCIX)*. Accessed: 2024-09-09. 2024. URL: https://www.ccixconsortium.com/.

[28] Ori Chalak et al. "ScaleSimulator: A Fast and Cycle-Accurate Parallel Simulator for Architectural Exploration". In: *Proceedings of the 10th EAI International Conference on Simulation Tools and Techniques*. 2017, pp. 32–41.

[29] Perceval Coudrain et al. "Active interposer technology for chiplet-based advanced 3D system architectures". In: *2019 IEEE 69th Electronic Components and Technology Conference (ECTC)*. IEEE. 2019, pp. 569–578.

[30] William J Dally and Brian Towles. "Route packets, not wires: on-chip inteconnection networks". In: *Proceedings of the 38th annual design automation conference*. 2001, pp. 684–689.

[31] Robert H Dennard et al. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of solid-state circuits* 9.5 (1974), pp. 256–268.

[32] Lieven Eeckhout. *Computer architecture performance evaluation methods*. Morgan & Claypool Publishers, 2010.

[33] Hadi Esmaeilzadeh et al. "Dark silicon and the end of multicore scaling". In: *Proceedings of the 38th annual international symposium on Computer architecture*. 2011, pp. 365–376.

[34] Farzad Fatollahi-Fard et al. "OpenSoC Fabric: On-chip network generator". In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2016, pp. 194–203.

[35] David Flynn. "AMBA: enabling reusable on-chip designs". In: *IEEE micro* 17.4 (1997), pp. 20–27.

[36] Davide Giri, Paolo Mantovani, and Luca P Carloni. "Accelerators and coherence: An SoC perspective". In: *IEEE Micro* 38.6 (2018), pp. 36–45.

[37] Stephen R Goldschmidt and John L Hennessy. "The accuracy of trace-driven simulations of multiprocessors". In: *ACM SIGMETRICS Performance Evaluation Review* 21.1 (1993), pp. 146–157.

[38] Xuan Guo and Robert Mullins. "Accelerate cycle-level full-system simulation of multi-core RISC-V systems with binary translation". In: *arXiv preprint arXiv:2005.11357* (2020).

[39] Anthony Gutierrez et al. "Lost in abstraction: Pitfalls of analyzing GPUs at the intermediate language level". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2018, pp. 608–619.

[40] Anthony Gutierrez et al. "Sources of error in full-system simulation". In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2014, pp. 13–22.

[41] Nikolaos Hardavellas et al. "Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture". In: *ACM SIGMETRICS Performance Evaluation Review* 31.4 (2004), pp. 31–34.

[42] Ahmed Hemani et al. "Network on chip: An architecture for billion transistor era". In: *Proceeding of the IEEE NorChip Conference*. Vol. 31. 20. sn. 2000.

[43] Margriet MWB Hendriks et al. "Data-processing strategies for metabolomics studies". In: *TrAC Trends in Analytical Chemistry* 30.10 (2011), pp. 1685–1698.

[44] John L Hennessy and David A Patterson. "A new golden age for computer architecture". In: *Communications of the ACM* 62.2 (2019), pp. 48–60.

[45] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[46] Salma Hesham et al. "Survey on Real-Time Networks-on-Chip". In: *IEEE Transactions on Parallel and Distributed Systems* 28 (2017), pp. 1500–1517. DOI: 10.1109/TPDS.2016.2623619.

[47] Mark D Hill and Vijay Janapa Reddi. "Accelerator-level parallelism". In: *Communications of the ACM* 64.12 (2021), pp. 36–38.

[48] Ravi Iyer. "On modeling and analyzing cache hierarchies using CASPER". In: *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003.* IEEE. 2003, pp. 182–187.

[49] Nan Jiang et al. "A detailed and flexible cycle-accurate network-on-chip simulator". In: *2013 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE. 2013, pp. 86–96.

[50] Jan Moritz Joseph et al. "Ratatoskr: An open-source framework for in-depth power, performance, and area analysis and optimization in 3d nocs". In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 32.1 (2021), pp. 1–21.

[51] Hui Kang and Jennifer L Wong. "vcsimx86: a cache simulation framework for x86 virtualization hosts". In: *Stony Brook University* (2013).

[52] Hassan N Khan, David A Hounshell, and Erica RH Fuchs. "Science and research policy at the end of Moore's law". In: *Nature Electronics* 1.1 (2018), pp. 14–21.

[53] Hyesoon Kim et al. "Macsim: A cpu-gpu heterogeneous simulation framework user guide". In: *Georgia Institute of Technology* (2012), pp. 1–57.

[54] Jinwoo Kim et al. "Architecture, chip, and package codesign flow for interposer-based 2.5-D chiplet integration enabling heterogeneous IP reuse". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28.11 (2020), pp. 2424–2437.

[55] Shashi Kumar et al. "A network on chip architecture and design methodology". In: *Proceedings IEEE Computer Society Annual Symposium on VLSI. New Paradigms for VLSI Systems Design. ISVLSI 2002*. IEEE. 2002, pp. 117–124.

[56] Hyoukjun Kwon and Tushar Krishna. "OpenSMART: Single-cycle multi-hop NoC generator in BSV and Chisel". In: *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2017, pp. 195–204.

[57] Daniel Lenoski et al. “The directory-based cache coherence protocol for the DASH multiprocessor”. In: *ACM SIGARCH Computer Architecture News* 18.2SI (1990), pp. 148–159.

[58] Tao Li et al. “Chiplet heterogeneous integration technology—Status and challenges”. In: *Electronics* 9.4 (2020), p. 670.

[59] Jason Lowe-Power et al. “The gem5 simulator: Version 20.0+”. In: *arXiv preprint arXiv:2007.03152* (2020).

[60] Zhonghai Lu et al. “NNSE: Nostrum network-on-chip simulation environment”. In: *Swedish System-on-Chip Conference (SSoCC)*. 2005.

[61] Milo MK Martin et al. “Multifacet’s general execution-driven multiprocessor simulator (GEMS) toolset”. In: *ACM SIGARCH Computer Architecture News* 33.4 (2005), pp. 92–99.

[62] Alian Mohammad et al. “dist-gem5: Distributed simulation of computer clusters”. In: *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2017, pp. 153–162.

[63] Gordon E Moore et al. *Cramming more components onto integrated circuits*. 1965.

[64] Samuel Naffziger et al. “Pioneering chiplet technology and design for the amd epyc™ and ryzen™ processor families: Industrial product”. In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2021, pp. 57–70.

[65] Nicholas Nethercote and Julian Seward. “Valgrind: a framework for heavy-weight dynamic binary instrumentation”. In: *ACM Sigplan notices* 42.6 (2007), pp. 89–100.

[66] Marcelo Orenes-Vera et al. “Muchisim: A simulation framework for design exploration of multi-chip manycore systems”. In: *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2024, pp. 48–60.

[67] Nicolai Oswald, Vijay Nagarajan, and Daniel J Sorin. “HieraGen: Automated generation of concurrent, hierarchical cache coherence protocols”. In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020, pp. 888–899.

[68] Saptadeep Pal et al. “Design space exploration for chiplet-assembly-based processors”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28.4 (2020), pp. 1062–1073.

[69] Mark S Papamarcos and Janak H Patel. “A low-overhead coherence solution for multiprocessors with private cache memories”. In: *Proceedings of the 11th annual international symposium on Computer architecture*. 1984, pp. 348–354.

[70] Michael K Papamichael and James C Hoe. “CONNECT: re-examining conventional wisdom for designing nocs in the context of FPGAs”. In: *Proceedings of*

the ACM/SIGDA international symposium on Field Programmable Gate Arrays. 2012, pp. 37–46.

[71] Avadh Patel, Furat Afram, and Kanad Ghose. "Marss-x86: A qemu-based microarchitectural and systems simulator for x86 multicore processors". In: *1st International Qemu Users' Forum*. Citeseer. 2011, pp. 29–30.

[72] David A Patterson and John L Hennessy. *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann, 2016.

[73] Fong Pong and M. Dubois. "A New Approach for the Verification of Cache Coherence Protocols". In: *IEEE Trans. Parallel Distributed Syst.* 6 (1995), pp. 773–787. DOI: 10.1109/71.406955.

[74] Antoni Portero et al. "COMPESCE: A Co-design Approach for Memory Subsystem Performance Analysis in HPC Many-Cores". In: *International Conference on Architecture of Computing Systems*. Springer. 2023, pp. 105–119.

[75] C Ravishanicar and James R Goodman. "Cache implementation for multiple microprocessors". In: *Proc. 26th IEEE Comput. Soc. Int. Conf.* 1983, pp. 1–5.

[76] Edwin Rijpkema, Kees Goossens, and Paul Wielage. "A router architecture for networks on silicon". In: *Proceedings of Progress* 2 (2001).

[77] Felipe Sampaio et al. "Approximation-aware multi-level cells STT-RAM cache architecture". In: *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE. 2015, pp. 79–88.

[78] Daniel Sanchez and Christos Kozyrakis. "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems". In: *ACM SIGARCH Computer architecture news* 41.3 (2013), pp. 475–486.

[79] Fabian Schätzle et al. "Modeling methodology for multi-die chip design based on gem5/SystemC co-simulation". In: *Proceedings of the 16th Workshop on Rapid Simulation and Performance Evaluation for Design*. 2024, pp. 35–41.

[80] Marco Sgroi et al. "Addressing the system-on-a-chip interconnect woes through communication-based design". In: *Proceedings of the 38th annual design automation conference*. 2001, pp. 667–672.

[81] Yakun Sophia Shao et al. "Co-designing accelerators and SoC interfaces using gem5-Aladdin". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2016, pp. 1–12.

[82] Debendra Das Sharma et al. "Universal chiplet interconnect express (UCIe): An open industry standard for innovations with chiplets at package level". In: *IEEE Transactions on Components, Packaging and Manufacturing Technology* 12.9 (2022), pp. 1423–1431.

[83] Kevin Skadron et al. "Challenges in computer architecture evaluation". In: *Computer* 36.8 (2003), pp. 30–36.

[84] Alan Jay Smith. "Cache memories". In: *ACM Computing Surveys (CSUR)* 14.3 (1982), pp. 473–530.

[85] Jeffrey Stuecheli et al. "IBM POWER9 opens up a new era of acceleration enablement: OpenCAPI". In: *IBM Journal of Research and Development* 62.4/5 (2018), pp. 8–1.

[86] Sajjad Tamimi et al. "An evaluation of using CCIX for cache-coherent host-FPGA interfacing". In: *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2022, pp. 1–9.

[87] Charles P Thacker and Lawrence C Stewart. "Firefly: a multiprocessor workstation". In: *ACM SIGARCH Computer Architecture News* 15.5 (1987), pp. 164–172.

[88] Rafael Ubal et al. "Multi2Sim: A simulation framework for CPU-GPU computing". In: *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. 2012, pp. 335–344.

[89] Stephen Van Doren. "Hoti 2019: Compute express link". In: *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE. 2019, pp. 18–18.

[90] Ashish Venkat and Dean M Tullsen. "Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor". In: *ACM SIGARCH Computer Architecture News* 42.3 (2014), pp. 121–132.

[91] Matthew Walker et al. "Hardware-validated CPU performance and energy modelling". In: *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2018, pp. 44–53.

[92] Charles Weber. "Yield learning and the sources of profitability in semiconductor manufacturing and process development". In: *IEEE Transactions on Semiconductor Manufacturing* 17.4 (2004), pp. 590–596.

[93] Wm A Wulf and Sally A McKee. "Hitting the memory wall: Implications of the obvious". In: *ACM SIGARCH computer architecture news* 23.1 (1995), pp. 20–24.

[94] Shaolin Xie and Michael Bedford Taylor. "The basejump manycore accelerator network". In: *arXiv preprint arXiv:1808.00650* (2018).

[95] Qian Zhang et al. "Heterogen: transpiling c to heterogeneous hls code with automated test generation and program repair". In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2022, pp. 1017–1029.

[96] Haocong Zhi et al. "A methodology for simulating multi-chiplet systems using open-source simulators". In: *Proceedings of the Eight Annual ACM International Conference on Nanoscale Computing and Communication*. 2021, pp. 1–6.