The Effect of an Optical Network On-Chip on the Performance of Chip Multiprocessors

Anouk Van Laer

A thesis submitted to the University College London (UCL) for the degree of Doctor of Philosophy

Optical Networks Group
Department of Electronic and Electrical Engineering
University College London (UCL)

August 2016

I, Anouk Van Laer, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated.

To my father ***Patrick Van Laer***
– The kindest and smartest man I knew

Abstract

OPTICAL networks on-chip (ONoC) have been proposed to reduce power consumption and increase bandwidth density in high performance chip multiprocessors (CMP), compared to electrical NoCs. However, as buffering in an ONoC is not viable, the end-to-end message path needs to be acquired in advance during which the message is buffered at the network ingress. This waiting latency is therefore a combination of path setup latency and contention and forms a significant part of the total message latency.

Many proposed ONoCs, such as Single Writer, Multiple Reader (SWMR), avoid path setup latency at the expense of increased optical components. In contrast, this thesis investigates a simple circuit-switched ONoC with lower component count where nodes need to request a channel before transmission. To hide the path setup latency, a coherence-based message predictor is proposed, to setup circuits before message arrival.

Firstly, the effect of latency and bandwidth on application performance is thoroughly investigated using full-system simulations of shared memory CMPs. It is shown that the latency of an ideal NoC affects the CMP performance more than the NoC bandwidth. Increasing the number of wavelengths per channel decreases the serialisation latency and improves the performance of both ONoC types. With 2 or more wavelengths modulating at 25 Gbit/s , the ONoCs will outperform a conventional electrical mesh (maximal speedup of 20%). The SWMR ONoC outperforms the circuit-switched ONoC.

Next coherence-based prediction techniques are proposed to reduce the waiting latency. The ideal coherence-based predictor reduces the waiting latency by 42%. A more streamlined predictor (smaller than a L1 cache) reduces the waiting latency by 31%. Without prediction, the message latency in the circuit-switched ONoC is 11% larger than in the SWMR ONoC. Applying the realistic predictor reverses this: the message latency in the SWMR ONoC is now 18% larger than the predictive circuit-switched ONoC.

Acknowledgements

NOW comes the hardest part of this thesis to write because how can I properly I thank everybody who supported and helped me throughout these years? Most of all though I want to thank my supervisor Dr Philip Watts . I have learned so much from him and his continuous support meant the world to me. He taught me so much, not only about the subject but also on being a good researcher and pushed me when I needed it most. It was an honour and a pleasure being his first PhD student. I also would like to thank Dr Timothy Jones at the University of Cambridge, without whom I would not have been able to tackle the gem5 simulator. He never hesitated when I needed help, be it with gem5 or more conceptual issues. It was a pleasure working with Ridwan Madarbux and discussing networks on-chip together. Every single person in the ONG group made my time in London a wonderful experience and supported me in various ways during the difficult months at the end of my PhD. I would also like to thank Dr Jose Mendinueta for the thesis template. I especially want to thank Professor Polina Bayvel for her kind words when I needed it most. They were truly appreciated. I would also like to thank my friends for keeping me (more or less) sane during my time as a PhD student. I would neither be where I am right now, be as content where I am right now or be the person I am right now, without my boyfriend Federico. He continuously encourages me to reach for more than I initially aim for and supports me in whatever I do. I would also be nowhere without my grandparents, aunts, uncles and cousins. They have always supported me from afar and were always ready to throw family parties whenever I went back home to Belgium.

Above all, I want to thank my sister, my mom and my dad. Their love is everything to me. They never batted an eyelid when I said I would go on exchange to Sweden or move to London for my PhD. I know I can run, just like in kindergarten, because they are always there to catch me with open arms when I return.

I never imagined I would write these acknowledgements without my dad being there to read them. This thesis is for him.

Contents

List of Figures

List of Tables

Acronyms & Abbreviations

CMOS	Complementary Metal Oxide Semiconductor logic
CMP	Chip Multiprocessor
CPI	Cycles per Instruction
CPU	Centralised Processing Unit
DVFS	Dynamic Voltage and Frequency Scaling
DRAM	Dynamic Random Access Memory
FLOP	Floating Point Operation
FSM	Finite State Machine
I	Number of Useful Instructions
ILP	Instruction Level Parallelism
IPC	Instructions per Cycle
ISA	Instruction Set Architecture
ITRS	International Technology Roadmap for Silicon
L1	First Level Cache
L2	Second Level Cache
L3	Third Level Cache
LLC	Last Level Cache
LUT	Look-up Table
LRU	Least Recently Used
MESI	Modified Exclusive Shared Invalid
MESIF	Modified Exclusive Shared Invalid Forward
MMI	Multi Mode Interference
MOESI	Modified Owner Exclusive Shared Invalid
MOSFET	Metal Oxide Semiconductor Field Effect Transistor
MWSR	Multiple Writer, Single Reader

N	Number of Cores per CMP
NACK	Negative Acknowledgement
NoC	Network On-Chip
NUMA	Non-Uniform Memory Access
OCIN	On-Chip Interconnection Network
OCN	On-Chip Network
OOK	On-Off Keying
Pt2Pt	Point-to-Point
QoS	Quality of Service
SEM	Scanning Electron Microscopy
SERDES	Serialisation/Deserialisation
SOI	Silicon-on-Insulator
SRAM	Static Random Access Memory
SWMR	Single Writer, Multiple Reader
TDM	Time Division Multiplexing
TDP	Thermal Design Power
TLP	Thread Level Parallelism
UMA	Uniform Memory Access
VLSI	Very Large Scale Integration
WDM	Wavelength Division Multiplexing

1 Introduction

THE ever increasing computational power we have at our disposal fueled the incredible scientific progress we have seen in the last decades. We are now able to send rovers to Mars, communicate instantly with people at the other side of the globe and track a single molecule moving in a living cell using image processing. If the transport industry would have improved at the same rate as the computing industry, it would be possible to travel from London to New York in about a second and would cost less than a dollar [1]. This analogy is limited but it does illustrate the impressive growth in computational performance.

This growth in computational power has been sustained by a steady increase in the performance of microprocessors over the last decades, as a result of continuous research and development in this area. Until 2000 when IBM released the first on-chip multiprocessor [2, Chapter 4.2], microprocessors consisted of a single processor core. As there were growing concerns about the power density in these microprocessors known as uniprocessors and a diminishing return in performance, there was a paradigm shift towards multiprocessors: systems where multiple processor cores work jointly on a program. Because of the continuous scaling down of the dimensions of Complementary Metal Oxide Semiconductor logic (CMOS) technology, it was possible to increase the number of processor cores on a single die, leading to the birth of the Chip Multiprocessor (CMP).

The core count in CMPs has been increasing ever since. Bohnenstiel et al. presented the first 1000 core processor array in June 2016 [3] and the highest core count in commercially available CMPs is 72 processor cores. Intel will release the Xeon Phi series later this year, containing 72 cores [4] and Tilera[1] released the Tile-Gx72 in 2013 which also contains 72 cores [5]. The rise in core count puts more pressure on the communication fabric which facilitates the communication between the various processor cores, the cache hierarchy and main memory. Initially communication was handled by bus-based systems but as they do not scale well, the use of a Network On-Chip (NoC) was proposed. The most commonly used NoCs are meshes (as is the case in the Tile-Gx72 [5]), crossbars (SPARC T5 [6]) and rings (for example, Xeon Phi [7]). Of these three network types, the mesh network is seen as the most viable option at high core counts.

The NoC is a vital part of the CMP as it handles all communication between the processor cores. The links in the NoC span the complete chip (global interconnect) and are conventionally formed by electrical wires. However there has been debate on the efficacy of electrical signalling on these global interconnects. The use of optics has been proposed as they could offer lower power consumption and higher bit rates [8]–[10]. However, the effect of an optical NoC on the computational performance of the CMP has never been quantified whilst this is an essential factor to consider before moving to optical NoCs. This thesis will, therefore, look at how an optical NoC affects the performance of a CMP, from the perspective of the programmer. This forms the first part of this study: can an optical NoC outperform an electrical mesh?

Maximising the power consumption and latency benefits of optical NoCs requires uninterrupted source to destination transmission: compact optical buffers are not viable and buffering in the electrical domain is not efficient[2]. This is in stark contrast to electrical NoCs which largely rely on intermediate buffering. In the the absence of optical buffers, the complete path from source to destination has to be setup before the actual message transmission can start. This path setup process has an associated latency. The message latency in an optical NoC therefore consists, broadly speaking, of the serialisation latency and the path setup latency. Various optical NoC types have been proposed to avoid the path setup latency, the most viable of them being the Single Writer, Multiple Reader (SWMR) scheme. In a SWMR network, every node writes to a dedicated channel which is broadcast to all other nodes. This completely avoids path setup at the cost of a high number of optical components. This thesis, however, proposes the use of a circuit-switched optical NoC in which messages have to request an optical path before transmission can start. This NoC has a lower component count, at the cost of an increased message latency due to the path setup process. However, I

[1]Tilera was acquired by EZchip Semiconductor in 2014, which in its turn has been acquired by Mellanox Tecnologies in 2016.

[2]These and other assumptions in this introduction will be discussed in more detail in Section 2.2.3

will argue that this simpler NoC can outperform the more complex SWMR network by using message prediction techniques. Messages in the NoC are not random as they are determined by the coherence protocol, which is the set of rules designed to keep the view of memory coherent and consistent across all processor cores. By using knowledge of the coherence protocol, the proposed techniques attempt to predict upcoming messages and use that knowledge to set up the required optical paths in advance, thereby hiding the path setup latency. This is the second question to be answered by this thesis: can the use of coherence-based techniques help lower the latency of a circuit-switched NoC to that of a SWMR NoC?

1.1 Thesis overview

Chapter 2 will further sketch the background of this thesis. Section 2.1 will discuss CMPs in more detail, with a focus on the cache hierarchy and coherence protocol as these affect the traffic in the NoC. In Section 2.2.1 the evolution from bus-based systems to electrical NoCs will be described, followed by the motivation behind optical NoCs in Section 2.2.2. The existing proposals for optical NoCs and the optical components will be discussed in Section 2.2.3. This chapter will then continue with an overview of holistic methods used for NoC optimisation in Section 2.2.4, after which the the system architecture and NoCs, as assumed in this thesis, will be illuminated in Section 2.3.1 and Section 2.3.2 respectively. At the end of this chapter the research questions that this thesis aims at answering will be discussed in more detail in Section 2.4.

As the research questions revolve around the concept of performance of a CMP, Chapter 3 will discuss how the performance of a CMP should be measured. Section 3.1 will give an overview on how performance can be evaluated for multithreaded workloads. Following that, Section 3.2 will discuss what is needed to actually measure performance: the workloads used in the thesis will be discussed and assessed (Section 3.2.1), as will the simulator of choice (Section 3.2.2). Full-system simulations as used in this thesis are costly in terms of implementation time and simulation time. Therefore, the need for full-system simulations will be justified in Section 3.2.3.

Chapter 4 will then compare the performance of electrical and optical networks in the context of CMPs. This chapter will start in Section 4.1 by assessing the effects of the latency and bandwidth of an ideal NoC on the overall performance in order to better understand the impact of an optical NoC. After describing the implementation of the electrical mesh and optical NoCs in the simulator (Section 4.2.1–Section 4.2.2), the effect of an optical NoC on performance compared to an electrical mesh will be quantified and discussed in Section 4.2.3.

Chapter 5 will then propose various methods that can be used to improve the performance of switched optical networks. The methodology will be highlighted in

Section 5.1. A first speculative technique will be introduced and its results quantified in Section 5.2. The same will be done in Section 5.3 for an ideal coherence-based predictor. Section 5.4 will then discuss various ways of making the ideal predictor more realistic.

Chapter 6 will then conclude this thesis and discuss the future work.

1.2 Original contributions

The main original contributions in this thesis are summarised as follows:

- Demonstration of the effect of the link latency and bandwidth of links in the NoC on overall performance using the full-system, cycle accurate simulator gem5. By adding non-determinism to these simulations, the effect of space-variability (see Section 3.1.2) on performance was also demonstrated. These experiments are described in Section 4.1.
- Demonstration of the need for full-system cycle-accurate simulation to accurately determine the effect of changes to the NoC. These experiments are described in Section 3.2.3.
- Demonstration of the effect of an optical NoC on the overall performance of CMP using full-system cycle accurate simulations.
    - This was achieved by implementing a model representing an optical crossbar-based network in the existing simulator gem5. The model is described in Section 4.2.2. This resulted in, to the best of my knowledge, the first publication of a full-system, cycle accurate simulation of the effect of an optical NoC on performance by Van Laer et al. [11] .
    - This work was extended by implementing a model for a SWMR based optical NoC, a commonly proposed topology in the optical NoC topology. The model is described in Section 4.2.2. This allows for a comparison in terms of CMP performance between the optical crossbar used in this thesis and the SWMR network. This is discussed in Section 4.2.
- Demonstration on how the use of prediction techniques that are based on knowledge of the coherence protocol can be used to reduce the latency of a switched NoC (be it optical or electrical). These techniques are discussed in Chapter 5 and were published by Van Laer et al. [12] .

1.3 List of publications

The following list gives a chronological overview of the publications that originated from the work described in this document:

- M. Madarbux, **A. Van Laer**, P. M. Watts and T. Jones, "Energy Efficient And Low Latency Interconnection Network For Multicast Invalidates In Shared Memory Systems" in *Proceedings of the 1st International Workshop on Advanced Interconnect Solutions and Technologies for Emerging Computing Systems at HIPEAC 2016*, HiPEAC, Prague, Czech Republic, pg 1 – 6, 2016
- **A. Van Laer**, C. Ellawala, M. Madarbux, P. M Watts, T. M. Jones, "Coherence Based Message Prediction for Optically Interconnected Chip Multiprocessors" in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, EDAA, Grenoble, France, pg 613–616, 2015
- M. Madarbux, **A. Van Laer**, P. M. Watts and T. Jones, "Towards Zero Latency Photonic Switching in Shared Memory Networks" in *Concurrency and Computation: Practice and Experience, Vol. 26*, pg 2551 – 2566, 2014
- **A. Van Laer**, M. Madarbux, P. M. Watts and T. Jones, "Towards Zero Latency Photonic Switching in Shared Memory Networks" in *Workshop on Silicon Photonics at HIPEAC 2014*, HiPEAC, Vienna, Austria, pg 1 – 8, 2014
- M. Madarbux, **A. Van Laer** and P. M. Watts, "Low Latency Scheduling Algorithm for Shared Memory Communication over Optical Networks" in *High-Performance Interconnects (HOTI), 2013 IEEE 21th Annual Symposium on,*, San Jose, USA, IEEE, 2013
- **A. Van Laer**, T. Jones and P. M. Watts, "Full System Simulation of Optically Interconnected Chip Multiprocessors using gem5" in *Optical Fiber Communication Conference/National Fiber Optic Engineers Conference 2013*, Anaheim, USA, Optical Society of America, 2013

2
Background

THIS chapter aims at describing the system being studied in this thesis, with a focus on all components that will affect in one way or another the communication between the processor cores.

Firstly, in Section 2.1, the evolution to chip multiprocessors will be sketched. This will allow for a better understanding of the communication requirements of a Chip Multiprocessor (CMP). Secondly, this chapter will take a closer look at the communication fabric of a CMP in Section 2.2 and describe the need for a Network On-Chip (NoC) to connect all nodes on a CMP. Section 2.2.2 will compare electrical and optical interconnect. After discussing the various optical components needed in an optical NoC, the most prominent optical NoC proposals will be reviewed. The last section of this chapter, Section 2.2.4 will review some holistic methods aimed at improving the NoC performance and/or efficiency by using information from components technically unrelated to the NoC such as the coherence protocol.

2.1 Introduction to Chip Multiprocessors

2.1.1 The transition from uniprocessors to chip multiprocessors

Microprocessors initially consisted of a single Centralised Processing Unit (CPU), working on one single stream of instructions and retrieving data from a single monolithic memory (Von Neumann programming model) [13, Chapter 1]. One way of improving performance employed was reducing the memory access time. The latency of accessing Dynamic Random Access Memory (DRAM) main memory (10 ns to 100 ns) [14] is high compared to the average processor clock cycle ($\leq$1 ns). This increasing gap between processor speed and memory access time is also known as the memory wall [15]. To circumvent this high main memory access time, a memory hierarchy is installed. Memory operations in the CPU are performed on words, a chunk of data which is 64 bits long (in a 64-bit processor). Every word is located in a specific place in the memory and as such has a memory address associated. Frequently used memory addresses should be kept close to the CPU instead of fetching them from memory every time.

However, faster memory technologies such as Static Random Access Memory (SRAM) are expensive, both in terms of actual cost and area. To give an estimate of the price difference, in 2008, the price of SRAM was around \$2000 – \$5000 per GB. DRAM on the other hand was only \$20 – \$75 per GB [1, Chapter 5.1]. The area cost of SRAM is higher as the density of DRAM is 10–20 greater and as such, DRAM can store more bits per cm^2 [16].

To circumvent these issues, a memory hierarchy with different levels was installed - the closer to the core, the faster and smaller the memory. The organisation of these caches is of utmost importance. To make handling and transferring words more efficient, multiple words are grouped into one cache block (also called cache line). Cache lines are efficient, not only because they reduce the handling overhead, but also because of spatial locality. When the CPU requests a memory address, it will very likely perform some operations on nearby addresses as well. This phenomena is called spatial locality. Because a cache line contains neighbouring addresses, the first request to a memory address will bring in memory addresses that will, in all likelihood, be referenced as well. These subsequent accesses will therefore have a much lower latency (cache access latency versus main memory latency)[1].

Another method of improving performance is exploiting the parallelism between instructions by means of Instruction Level Parallelism (ILP). One kind of processor capable of this is called a superscalar. The instruction stream is constantly being scanned for instructions that can be executed in parallel, using multiple parallel functional units [1, Chapter 4.10]. Another method of increasing performance is increasing the throughput. Throughput is defined as the number of instructions issued per clock cycle. This can be done by pipelining the instructions. Instead of executing the instruction in one cycle, it will be split up in multiple smaller and simpler operations. This way the clock rate can be increased as every stage takes less time. Both pipelining and exploiting ILP cannot be pursued indefinitely. If the pipeline becomes too long, every stage becomes so short that it barely does anything meaningful as even adding two

[1]Another type of locality is temporal locality: a memory address currently in use will most likely be used again in the nearby future. This principle is used in cache replacement policies.

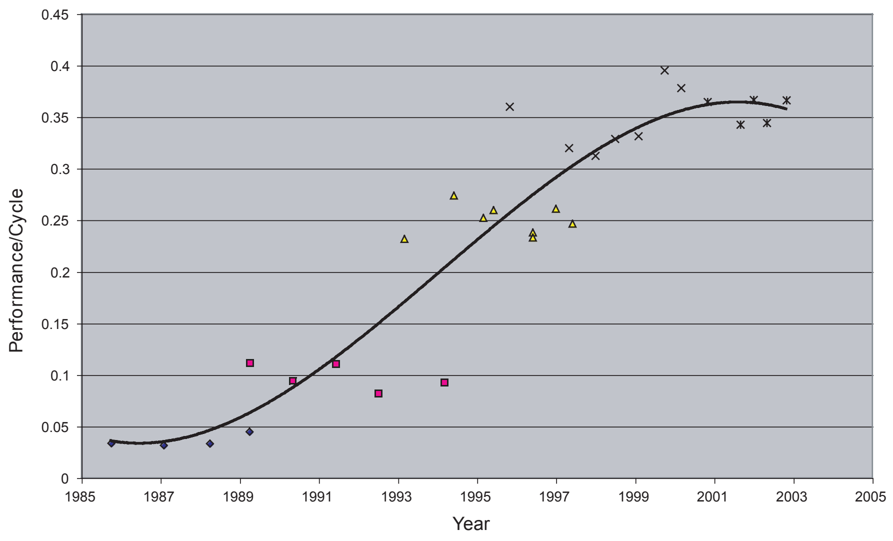


Figure 2.1: Intel processor normalised performance per cycle over time is showed. It shows how performance stays constant before ILP is pursued, followed by an improvement in performance by executing parallel instructions simultaneously. The performance then levels off when the available parallelism has been extracted. Figure taken from [13, Chapter 1]

integers becomes difficult [17]. The additional circuitry needed to find ILP in the instruction stream has to be compensated for by an increase in performance.

There is only a limited amount of ILP to be extracted (between 4 and 10 instructions per cycle [18]), making the logic needed to find it more and more intricate as the complexity of the extraction logic is approximately proportional to the square of the number of instructions that can be executed in parallel [13, Chapter 1], leading to diminishing returns. The resulting processor core becomes so complex that the design and verification of such processor cores becomes increasingly difficult. These issues caused a plateau to be reached in processor performance as shown in Figure 2.1.

However, the change from uniprocessor to multiprocessor came about because of the limited on-chip power dissipation. Although individual transistors are using less power because of scaling, more and more transistors were placed on a chip to accommodate for ILP exploitation and pipelining, leading to an overall increase in power consumption. The increase in clock frequency also contributed to the rising power consumption [17]. Cooling a chip can be done by installing heat sinks, dedicated fans to increase the airflow or liquid cooling. However, these types of cooling systems come at an economical cost and the best cooling solutions are only justified for high end computing systems. This growing power problem combined with the performance plateau led to the advent of a new type of processors: multiprocessors.

In a multiprocessor architecture, multiple processor cores work together. Each processor core is simpler and smaller than the previously used superscalars. The overall throughput will be higher than the throughput of one complex superscalar as the processor work in parallel. The exact increase in throughput will depend on the application [13, Chapter 1.3]. Parallel applications allow the CMP to exploit this, whilst as superscalar processor is limited to ILP present. Because of the growing interest in data-intensive server applications such as design of drugs in the pharmaceutical sector and weather modelling, multiprocessors became more appealing [19, Chapter 1.1.1]. Most of these scientific and engineering applications also require result visualisation, itself an interesting parallel application. Because of continued scaling it became possible to place multiple processor cores on a single die, leading to the advent of the CMP.

2.1.2 Chip multiprocessors

Advantages of CMPs

CMPs have multiple advantages [13, Chapter 1.1]. First of all, as each individual processor core is an exact copy of other cores on the CMP, only one core needs to be designed and can then simply be replicated. This is a big difference with the highly complex design of a superscalar.

The second advantage lies in the type of parallelism they can exploit. As previously stated, superscalars can only exploit parallelism across a few instructions. As the cores on a CMP each have their own instruction and data stream, a CMP can exploit Thread Level Parallelism (TLP), alongside ILP. The processor cores are visible to the programmer as individual entities that can work in parallel. A program can be subdivided into processes and threads. The various processes in a program are completely independent. Far more interesting, from a CMP point of view, are threads. Threads share parts of the code and parts of the memory address space. Threads work in parallel and to exploit this TLP, each one is assigned to a different processor core[2]. To make efficient use of a CMP with N cores, the program running on it needs to be split up in at least N threads. This change in programming model will put an extra burden on the shoulders of programmers as they need to recognise TLP and set up the program accordingly. While the exploitation of TLP can definitely increase performance, doing so increases complexity significantly [1, Chapter 7.1]. A thread scheduler is needed to schedule the threads and possible migrate them across cores in order to ensure a proper load balance across the processor cores [20, Chapter 7.1]. The different threads also need to be synchronised which can introduce race conditions etc. [20, Chapter 3.4.2].

The third advantage is the fact that the processor cores share some resources, such as I/O devices and one or more levels of the cache hierarchy, to make more efficient use of them. By sharing parts of the cache hierarchy, the storage space that is available on chip is used more efficiently as the data will not be replicated.

[2]One processor core can also host multiple threads. When one thread is stalled (because of a memory reference for example), the next thread will take over. This is called coarse-grained multithreading [2, Chapter 3.5] and will not be explored in this work.

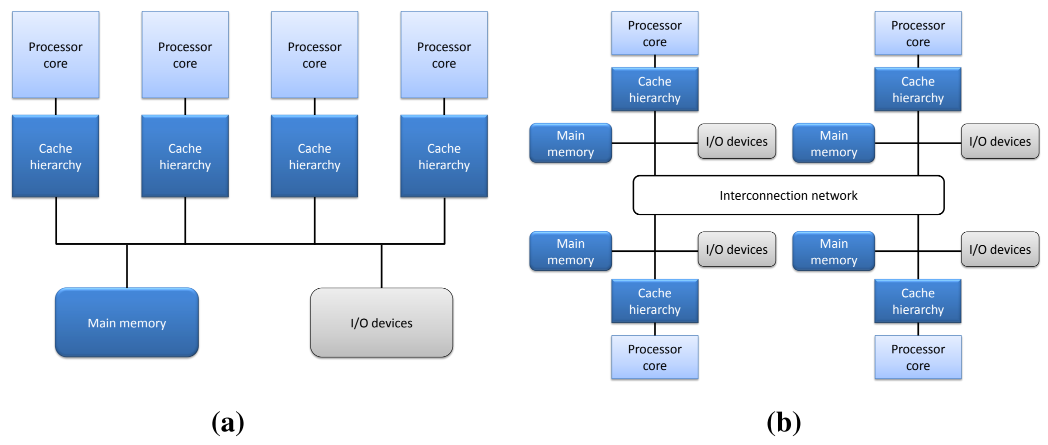


Figure 2.2: Shared memory architectures (after [2, Chapter 4.1]) Centralised shared memory architecture and (b) Distributed shared memory architecture

Classification of CMPs

CMPs can be divided into two classes based upon their memory organisation [2, Chapter 4.1]. In a centralised shared memory architecture (Figure 2.2a), all processor cores share a single centralised memory. Because this creates a bottleneck, this model is only sustainable for a limited number of processor cores [2, Chapter 4.1]. Because all processor cores have the same distance to the memory, this is often called a symmetric shared memory multiprocessor. It can also be classified as an Uniform Memory Access (UMA) architecture because the memory latency is independent from the memory address or requesting processor core. The memory bottleneck can be alleviated by physically distributing the memory across the CMP, thereby creating a distributed-memory multiprocessor (Figure 2.2b). A high bandwidth interconnection network is often used to interconnect all these nodes. The latency to a local memory node will be lower, making a distributed-memory architecture more attractive in terms of memory latencies as well. The communication between the processor cores becomes more complex for reasons discussed later on. In this work, only distributed-memory multiprocessors will be considered.

Processor cores in a distributed-memory CMP can communicate in two manners. Communication can occur through the use of one shared address space. Although the memory is physically distributed, it will act as one memory logically. This implies that processor cores can communicate by writing and reading to the same memory locations. This architecture is also called a distributed shared memory architecture. The memory latency will depend on the distance between a processor core and the part of the memory being addressed, creating a Non-Uniform Memory Access (NUMA). The second possibility is that the distributed memories are logically disjointed as well. There is no common shared address space anymore, every processor core can access a private address space, not accessible to a remote processor core. Communication occurs by explicit message passing. When using a message passing model, the programmer needs to foresee this communication between processor cores, making programming more difficult. This is the advantage of the distributed shared memory architecture. Communication happens implicitly, by writing and reading to and from the same memory addresses, making it easier for the programmer to write parallel code. In this work, only distributed shared memory CMPs will be considered for this reason. When a processor core needs to access a word in a remote part of the memory, this handled in hardware by the coherence protocol. Section 2.1.4 will talk more about the coherence protocol. It is also possible to create one communication model in software and run it on top of hardware supporting the other communication model. Supporting message passing on shared memory hardware is much easier and has less of a penalty than the other way around [2, Appendix H.2], making the shared memory model more general.

As mentioned in the previous section, some levels of the cache hierarchy will be shared among the processor cores. Such a cache could potentially become a bottleneck, as it will be addressed by all processor cores. To prevent this, a shared cache can be logically shared, physically distributed. Every processor core gets grouped together with its private cache levels and a slice of the distributed shared cache levels. This forms a tile. This also means only one such tile needs to be designed, for it to then be repeated across the CMP. Tiled architectures thus only need a short design cycle [21]. An example of a tile is given in Figure 2.3. In this thesis, the CMP consists of such tiles. Every processor core has a private first level cache, the First Level Cache (L1). The L1 will be subdivided into two units: one to store instructions (L1-I) and the other one for the storage of data (L1-D). All processor cores share the second level cache (Second Level Cache (L2)). To physically distribute the L2, every L2 bank[3]Caches can be divided into banks or slices, which can be simultaneous accessed, rather than a monolithic cache which can only handle one access at a time [2, Chapter 5.1] will be assigned to a tile. A tile therefore consists of a processor core, connected to its private L1 and one bank of the shared L2. Both the L1 and the L2 share a network interface. This network interface is used to communicate with other parts of the cache hierarchy.

2.1.3 Introduction to cache organisation

The organisation of the caches will affect the amount of traffic being directed to the NoC. This is not explored further in this thesis. Concepts used in the organisation and more specifically the indexing of the caches will be used in Chapter 5 and will therefore be introduced here. In a direct-mapped cache every cache line has its own dedicated block frame in the cache [2, Appendix C]. In a cache which is organised in a set-associative manner [2, Appendix C], multiple cache lines can map onto a set of

[3].

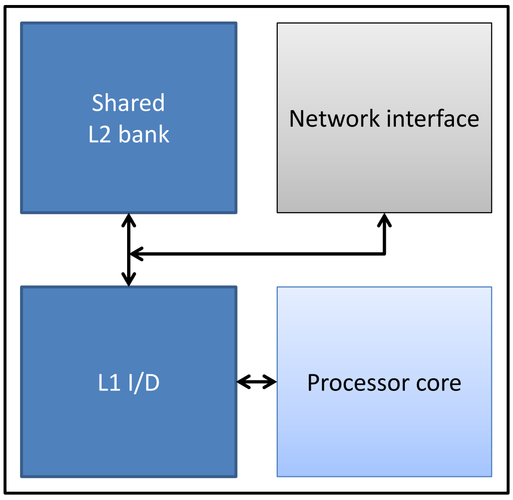


Figure 2.3: A tile in a distributed shared memory architecture (after [22, Chapter 2])

block frame . The lookup process in a set-associative cache therefore consists of two steps. First, after determining the cache line to which this memory address belongs, a subset of the cache line address bits (called the *index*) is used to determine the set onto which this cache line maps. A cache line can reside everywhere in the set so in the second step all entries in the set need to be checked to see if the cache line is already present in the cache. This is done by comparing the remaining address bits (called the *tag*) with the tags of the cache entries. A set-associative cache will therefore be larger in storage size than a direct-mapped cache (with the same number of entries) as it needs to keep the tags of all cache lines as meta-data. A X-way associative cache is a set-associative cache where every set contains X cache block frames. When increasing the associativity X, the cache increases in size (longer tag as there are less sets) and becomes slower (X slots need to be checked during the lookup phase) but the mapping process becomes more effective (less conflicts).

A fully-associative cache is an extreme form of a X-way associative cache in which X is equal to the number of cache block frames, making the complete cache one single set [2, Appendix C]. Therefore, it does not suffer from conflicts (in which addresses map onto the same set and cause frequent evictions in this set, whilst other sets are underutilised) but is infeasible in practice. As a cache line can now reside anywhere in the cache, all entries in the cache need to be checked during every lookup. The size of the cache also increases because of the tags that need to be kept; as there is only a single set, the tag is equal to the complete cache line address.

2.1.4 Introduction to cache coherence protocols

In shared memory CMPs, the caches contain data that is shared among all processors. The communication between a processor core and memory will occur through the data in the caches. If no precautions are taken, two different processor cores might see different values of the same data block. The job of a cache coherence protocol is to prevent such incoherencies.

There are two main types of coherence protocols: snooping protocols and directory-based protocols. In the case of snooping protocols, all actions in a cache that change the value of a cache line (i.e. writes), need to broadcast to all other caches. All caches listen in or snoop to a bus-like communication fabric and discard any local copies when they receive such a message. Whilst this is a very simple concept, it does not scale well with increasing processor core count. As the number of processor cores increases, the number of private and shared caches will increase as well, making it harder to broadcast efficiently.

The directory-based protocols have a higher implementation overhead but scale better than the snooping protocols. In a directory-based protocol, all information about the state of cache blocks present in the on-chip memory hierarchy is kept in a directory. It keeps track of which caches have a copy of the block, whether the block has been written since it was fetched from main memory (making the block 'dirty') etc. To prevent the directory from becoming a bottleneck, it is distributed along the processor cores in a similar manner as the memory. The implementation of the directory is very important to keep it as fast and small as possible. The part of the directory where the information about a certain cache block is kept is called the home node. A node that makes a request for a certain block is called the local node. If copies of the block are present in multiple caches, they are called remote nodes or sharers.

It has been assumed that coherence (regardless of the exact type of protocol) is hard to maintain when increasing the number of processor cores per chip. Coherence protocols are thought to scale badly because of multiple concerns over the amount of traffic generated, the storage needed to keep track of sharers, the latency of requests that need to be communicated to remote nodes and the incurred power cost. However, this has been refuted recently by Martin et al. [23]. The authors discuss the scalability of the directories, where they classify a directory architecture to be scalable if the associated overheads grow slower than the number of sharers in the system. By means of amortised analysis they show the traffic growth per will at most be 20%, compared to a system with caches but no coherence. The second problem with regards to directories is the associated storage, however, the use of hierarchical systems[4] can help. The increased latency of directory-based systems forms a third disadvantage, because of indirection via the directory: a cache miss first needs to be pass by the directory before further action can be taken. However, the latency of these 3 hop misses does not depend on the

[4]In a hierarchical system, the lower level caches (e.g. L1s) are grouped into clusters. The top level directory tracks which clusters hold a copy of the cache line but does not know exactly which caches in the cluster share the data. It falls to the per-cluster directory to track the individual sharers inside the cluster.

amount of nodes in the system.

In this work, a Modified Exclusive Shared Invalid (MESI) directory protocol is used [24]. In this protocol, there are 4 basic states.

- **I - Invalid** : the cache line is not present in this cache.
- **E - Exclusive** : the cache line is present in only one cache and has not been written to and as such, is consistent with the next level of the cache hierarchy (e.g. the L2 cache or main memory)
- **M - Modified** : the cache line is present in only one cache and has been written to and as such, is no longer consistent with the next level of the cache hierarchy.
- **S - Shared** : the cache line is present in multiple caches and as such, cannot be written to

There are also many more transient states. Every coherence protocol has a set of rules determining the transitions between all states, resulting in a Finite State Machine (FSM). The FSM can then get implemented in hardware, resulting in a coherence controller. The exact number of states and the exact transitions depend on the level in the cache hierarchy, resulting in different coherence controllers per level in the cache hierarchy. Every cache has an associated coherence controller, keeping track of the cache line this cache holds. What follows is a small example to illustrate how a cache coherent system works. Imagine processor A wishing to write to a location in cache block ① (Figure 2.4). This block is present in nodes B and C (① is in the Shared state). Processor A will first send out a request to its local cache. Because it is not present there (① is in the Invalid state in cache A), a request will be send out to the directory. The request states that processor A requests a copy cache block ① to write to it. Once this request arrives at the home directory, the information on block ① will be checked (Figure 2.4a). Considering there are remote nodes caching the block, the directory will need to ensure they invalidate the block. It does so by transmitting an invalidation request to all sharers whilst sending a message to the requester which contains the data and the number of sharers that need to invalidate their copy before cache A is allowed to write to the block (Figure 2.4b). After nodes B and C have acknowledged the invalidation of ① to A (Figure 2.4c), node A will then be the exclusive owner of this block and communicates this to the directory (Figure 2.4d). The reason the remote copies need to be invalidated first, is to keep the view of the memory coherent. If processor A were allowed to write ① without first invalidating all remote copies of ① , remote nodes might read the value of ① and see an incorrect value. After ① has been written to, the state of the cache line changes to the Modified state.

In this thesis, messages are described by their source, type and kind. The source is a L1 cache, the L2 cache, or memory. The type is either a request or response. Finally,

Source	Type	Size	Functionality
L1	REQ	Control	Start of transaction after L1 miss
L1	REQ	Data	Writeback after L1 eviction
L1	RES	Control	Acknowledgment
L1	RES	Data	Ack containing data
L1	UNB		Ack upon receipt of exclusive line
L2	REQ	Control	Invalidation request
L2	REQ	Data	Writeback after L2 eviction
L2	RES	Control	Writeback ack, upgrade ack
L2	RES	Data	L2 response
MEM	RES	Control	Writeback after ack
MEM	RES	Data	Memory response

Table 2.1: Message types

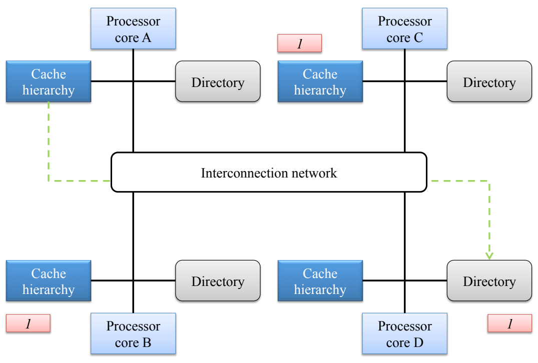
the kind is either control (a short 8 B message) or data (a 64 B cache line). For example, *L1 REQ C* is a short control request leaving a L1-cache controller, and this type of message typically signifies the start of a transaction. Table 2.1 contains all message types occurring in a MESI-based system.

The short example in Figure 2.4 gives some more information on the type of traffic that is to be expected on-chip. Most messages will not be messages carrying data, but short messages that keep the caches coherent such as the invalidation messages or just simple request messages. These type of messages are called control messages and are around 8 B long as they just contain the request, the memory address concerned and the requesting node. Messages carrying cache blocks are called data messages and are much less frequent. They are generally 72 B to 264 B long, depending on the cache line size. Increasing the cache line size will initially increase the hit rate because of spatial locality but will lead to diminishing returns as less cache lines will fit in the cache. The optimal cache line size depends on the cache organisation (size, associativity) and the workload used. The most commonly used cache line size is 64 B, which is also used in this thesis. Nevertheless, there are architectures which use different cache line sizes across the different cache levels: the SPARC T5 for example has a 32 B cache line size for the L1 and L2 whereas the Third Level Cache (L3) has a 64 B cache line size [6].

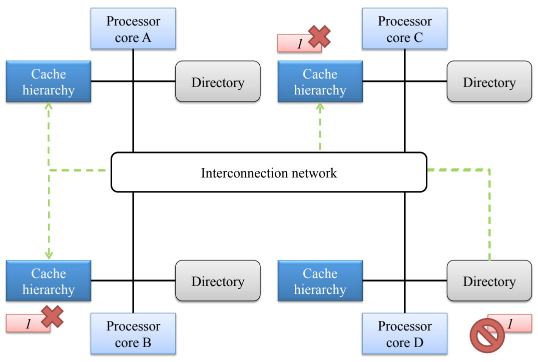
The directory tracks all cache lines present in the on-chip cache hierarchy. It does so by tracking all these cache line addresses and some meta-information associated with each address like the sharers etc. If the Last Level Cache (LLC) (the cache level closest to the main memory) is inclusive [5] a copy of all cache lines present in the on-chip cache

[5]In an inclusive LLC, all cache lines present in the caches closer to the processor cores need to be present. An exclusive LLC cannot hold a copy of a cache line that is also present in a lower level cache. A non-inclusive cache enforces neither inclusion nor exclusion [14, Chapter 4.3.4].

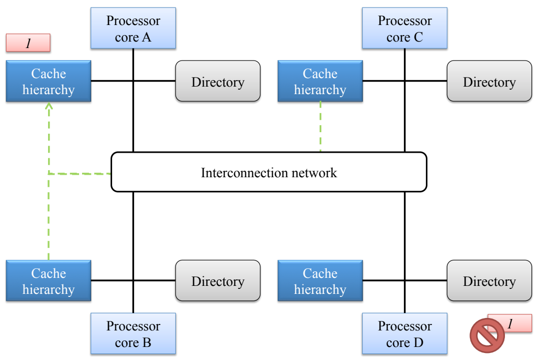
hierarchy need to be kept in the LLC. When a cache has a copy of a cache line, it also needs to hold meta-information like cache line address and state. The meta-information kept by an inclusive LLC and the directory overlaps quite a bit: they both store the cache line address and the cache line state. Data replication can be avoided by merging the directory and LLC [25, Chapter 8.6.2]. In this thesis, when referring to the L2 controller or directory controller, they refer to the same controller.



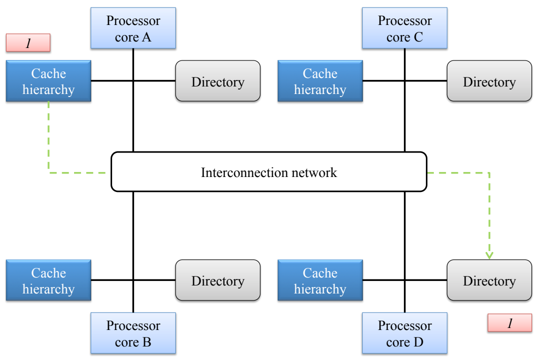
(a)



(b)



(c)



(d)

Figure 2.4: Example of a cache coherence transaction. Processor core A wishes to write to cache block a. D is the home node for a. B and C, which are caching a, are remote nodes. a is in the Shared state at the start of this example for B and C and is in the Modified state for A at the end of the example.

2.1.5 Industrial and academic examples of chip multiprocessors

Many CMP architectures have been proposed and in this section some notable cases will be discussed. Not all of the following examples are pure CMPs according to the definition (multiple CPUs on the same chip). This list is not meant to be exhaustive but to give an idea of how the individual characteristics of the system architecture, which will be used in this thesis, exist in real-life systems.

SGI Origin (publication in 1997)

The SGI Origin system [26] consisted of two processor cores per node. Up to 512 nodes could be connected in a complete system, resulting in a 1024 core system. As these cores were not on the same die, this is not a real CMP. However, the SGI Origin system did employ a distributed shared memory, leading to the use of a MESI-type directory coherence protocol. To make the directory scheme scalable to 1024 cores, the directory dynamically switches between a normal sharers vector and a coarse sharers vector[6]. To connect all nodes in the system in-house NUMALinks[7] were used: point-to-point links connected in a fat hierarchical hypercube topology.

Raw (publication in 2002)

The Raw architecture proposed by MIT [27] was one of the first proposals using a tiled organisation. The authors also designed a specific Instruction Set Architecture (ISA)), to take advantage of this layout. Every system contains 16 tiles in which each tile accommodates a simple MIPS-style processor core, 3 routers, a floating-point unit, a data cache and a software managed instruction cache. All tiles are connected using four mesh networks, two of which are static whilst the other two are dynamic. The interesting feature of this architecture is the fact that the networks are an integral part of the Raw-ISA. To start transmission of a word[8] on one of the networks, the word simply needs to be written to the corresponding register. In the case of the static networks, the routing is determined at compile time. When the program uses the static networks to transmit data between tiles, the connection between those tiles needs to be set as well. The dynamic networks function like conventional mesh networks as will be described in more detail in Section 2.2.1.

TILE64 (publication in 2007)

The Tilera system [28] is a commercial product, inspired by the RAW architecture and shares a lot of features with this system. Every processor core has a private L1, but can share its L2 with other tiles, thereby creating a shared L3. No details are given, however, on how these are kept coherent. The interconnect infrastructure [29] in particular is a clear follow-up from the Raw architecture. There are now five mesh networks, each with a single purpose. The memory dynamic network for example handles the traffic between the LLC and the main memory.

[6]A coarse sharers vector with N entries can track up to N sharers exactly. If there are more than N sharers, the region of sharers is tracked.

[7]These are also known as CrayLinks.

[8]A word being the most common size of a data access and indicates for most architectures the register size [1].

Nehalem (publication in 2009)

The Nehalem microarchitecture proposed by Intel in 2009 [30] is not a complete CMP but rather a concept that was reused in multiple subsequent CMPs. Nehalem was designed to provide scalability both at runtime (via Dynamic Voltage and Frequency Scaling (DVFS) [29] for example) and design time (via a modular design). Every processor core has a private L1 and L2 cache. Multiple processor cores share an inclusive L3 cache. The inclusivity prevents unneeded snoops. This architecture formed the base of multiple chip multiprocessors, with the number of processor cores ranging from 1 to 16, depending on the target market. The view of memory is kept coherent using a snooping Modified Exclusive Shared Invalid Forward (MESIF) coherence protocol [31].

Single-Chip Cloud Computer (publication in 2010)

The Single-Chip Cloud Computer (SCC) by Intel [32] is a research CMP containing 48 processor cores and is seen as the follow-up of the Tera-Scale Research Processor [33]. The Tera-Scale Research Processor contained 80 processing engines, unlike the SCC Computer which contains 24 tiles, each holding two Pentium processor cores. Every processor core has access to a private L1 and L2 on the tile. The processor cores in the SCC have access to three address spaces (1) private address space, only accessible from this core. These addresses can be cached, in both the L1 and L2. (2) shared address space, accessible by all cores and, hence, non-cacheable (3) shared address space, cacheable in special message passing buffers and kept coherent by the software using message passing. The NoC is a conventional mesh network.

SPARC series (publication in 2013)

The SPARC series by Oracle [6] is based upon the Niagara processor by Sun [34]. The latest version, the SPARC T5 features 16 cores, each able to run eight threads. Each processor core has access to a private L1 and L2. The L3, however, is shared among all processor cores and is divided into eight cache banks. These banks can be accessed via a high bandwidth crossbar. The architecture supports up to 8 SPARC T5 chips to be combined into one shared memory system, connected via direct links. One of the interesting features of the SPARC T5 is the hierarchical coherence protocol: the L2 are kept coherent using a MESI protocol. The L3 is kept coherent across all nodes using a Modified Owner Exclusive Shared Invalid (MOESI) directory protocol.

All the previous examples have identical processor cores, repeated multiple times per chip. One body of research is focusing on heterogeneous computing in which different processor cores, with the same ISA, are combined, allowing for a better match between software and hardware. However, these types of systems are not discussed in this thesis. Table 2.2 contains a summary of all previously discussed CMPs.

This thesis focuses on a distributed shared memory system where the view of memory is kept coherent using a directory coherence protocol. The CMP uses a tiled layout. While the individual characteristics of such a system exist in real-life systems (for example, the tile concept as presented in Tilera series and directory protocol as used in the SGI Origin series), a system with these characteristics does not yet exist as a commercial product.

In the remainder of this work, when talking about a CMP, it is implied this is a distributed shared memory CMP where the processor cores share the LLC and the view

Name	# Cores/ # Nodes	Intra-chip interconnect	Cache organisation	Coherence	Ref.
SGI Origin	2/512	Router	Distributed shared	MESI-like directory	[26]
RAW	16/ Not app.	Mesh	Private & shared	Unclear	[27]
TILE64	64/ Not app.	Mesh	Private & shared	Unclear	[28]
Nehalem	4/ Variable	Bus	Private & shared	MESIF-snooping	[30]
SCC	48/ Not app.	Mesh	Private	Message passing	[32]
SPARC T5	16/ 8	Crossbar	Private & shared	MESI & MOESI directory	[6]

Table 2.2: Example of CMP systems and proposals

of the memory is kept coherent and consistent using a directory protocol. The details of the CMP used in this thesis are summarised in Table 2.5.

As the number of processor cores increases, the importance of communication increases. For example, one of the latest CMPs by Oracle, the SPARC M7 contains 32 cores and features an actual NoC, rather than a crossbar [35]. The NoC in the M7 consists of 3 networks, each responsible for a message type. The request messages are transmitted on a ring NoC, the responses use a Point-to-Point (Pt2Pt) network and finally, there is a mesh NoC to route the data. The same trend can be seen in the Tilera CMPs where the NoC is an integral part of the CMP. The following sentiment, lifted verbatim from [36] when discussing the Tera-Scale Research by Intel, confirms this and brings us to the next section in this chapter;

> "Intel is not designing a processor core. It's designing a network on a chip; ... "

2.2 Introduction to Networks On-Chip

The increasing core count in CMPs puts more pressure on the communication fabric between the various processor cores, the cache hierarchy and main memory. In Section 2.2.1 the evolution from a simple bus-based communication system to a NoC will be sketched. Conventionally the NoC consist of electrical links but as will be detailed in Section 2.2.2, electrical links might no longer suffice and I will argue for the use of optical links in the NoC. Section 2.2.3 will review the various components in an optical NoC and discuss the most important optical NoC proposals. This will be followed by a discussion on various holistic methods that can be used to improve the performance of the NoC (be it an electrical NoC or optical NoC) in Section 2.2.4.

2.2.1 Transition to networks on-chip

As the previous section made clear, the different nodes on a CMP (processor cores, various parts of the memory hierarchy, I/O devices) need to be able to communicate. The pressure on this communication fabric will only increase with increasing core count.

End of bus-based systems

Initially communication was handled by bus-based systems [22, Chapter 1]. The architecture of a bus is quite simple: all nodes are connected to one shared medium, the bus. When a node wishes to communicate, it makes a request to the central arbiter. Once the request is granted, the message is broadcast to all nodes on the bus and only the destination node will further process the message. Unfortunately, buses do not scale. The first disadvantage is formed by the arbitration process. The delay of a centralised arbiter increases with increasing node count. As most buses were formed using long electrical links, they face the same issues commonly faced by electrical interconnect, which will be elaborated on in Section 2.2.2. In short, every new node on the bus will add a parasitic capacitance. Not only will this increase the power consumption of the bus, it will slow down the switching speed of the bus as well. A bus is also not power efficient as messages are broadcast to all nodes. Only a limited number of messages in a shared memory architecture are true broadcasts: most of the messages are either unicast or multicast messages (depending on the coherence protocol used). Increasing the number of nodes will not only increase the power needed to charge and discharge the capacitance of the bus but also the power wastage as even more nodes are addressed unnecessarily. The last disadvantage is formed by the fact the bus is a shared medium. No parallel transactions are possible on a bus meaning that if one node is communicating, all other requests need to be stalled. The chance that at least one node is communicating increases with the number of nodes, making contention and saturation rise.

To mitigate these effects, the concept of a NoC was proposed [37],[38],[39]. It needs to be noted that it is possible to improve the power, area, cost and performance of a bus-based system by choosing an adequate bus topology such as a split bus in which multiple shorter buses are used, which are in turn connected using tri-state buffers or a partial crossbar bus which is a hybrid between a full point-to-point scheme and a conventional bus [40, Chapter 2.4]. In a way these topologies can be seen as an intermediate step between bus-based interconnects and NoCs. NoCs are also known as On-Chip Network (OCN) or On-Chip Interconnection Network (OCIN). In this work, the term NoC is used.

Characteristics of networks on-chip

NoCs offer some advantages over traditional buses [41],[38]. First of all, they are scalable to higher node counts for several reasons. There is (in general) no dependency on centralised arbitration. The components that make up a network, routers and the connecting links, react better to technology scaling than long dedicated wiring (elaborated in Section 2.2.2). Secondly, wires are used more efficiently as multiple data flows use the same wiring. Not all communication flows are frequent so if dedicated wiring were to be used, those wires will rarely toggle. But they still need to be designed to cater for peak moments, leading to an over-provisioned channel. The third advantage is that NoCs offer modularity. The network architecture consists of links and routers. The design effort can be focused on the design of a link or router which can then be replicated multiple times across the NoC. This is similar to the argument made for CMPs. Modularity also offers advantages in terms of ease of testing of components. While NoCs and larger scale networks share characteristics, they do differ [41]. For example, the traffic patterns are different, both in terms of message distribution [41] and message size. Before discussing specific NoC architectures, some network terminology will be introduced.

A NoC design can be subdivided into different components [42, Chapter 1]. The topology of network specifies how the various nodes, routers and links are connected and can be seen as the floorplan of a network. How messages travel the network is determined by the routing algorithm. A routing algorithm should balance the traffic as evenly as possible, to prevent the formation of hotspots and avoid contention. It affects network performance as the message latency depends on the number of hops a message makes to cross the network and the number of hops is determined by the routing algorithm. Flow control determines how resources such as buffering and link bandwidth are allocated to the various messages that are travelling the network.

Most networks need allocation or arbitration of some form. An arbiter handles requests from n requesters, all vying for one single resources. An allocator on the other hand oversees requests from n requesters for m different resources. A $n \times m$ allocator can be created by using m $n \times 1$ arbiters[9].

Networks can be packet-switched or circuit-switched [22, Chap. 5.2] [42, Chap. 12.3]. In a circuit-switched network, messages travel the network using resources that have been assigned before message transmission started. Messages therefore get access to a contention free path but there is a certain latency associated as transmission can only start after the path or circuit has been setup. Circuit-switching tends to be favourable for long lived data flows. However, the use of the term circuit does not mean multiple messages use the path or circuit that has been setup. Ideally, to improve the utilisation and reduce the path setup overhead, multiple messages will use the same path but this does not have to be the case. Packet-switched networks on the other hand allow messages to be transmitted across the network and acquire resources on the go. Packet-switched networks can also work at a smaller granularity as messages can subdivided into packets which are then transmitted across the network. Packet-switching avoids the need for central arbitration, making this a more scalable network model.

Non-blocking networks can accommodate all path requests that are permutation of inputs and outputs i.e. an unused input can always be connected to a yet unused output. A strictly non-blocking networks goes even further and such a circuit can be setup, without having to rearrange existing connections.

Commonly used network topologies

The most commonly used NoC topologies are crossbars, meshes and rings.

Crossbars

A $m \times n$ crossbar connects n inputs to m outputs (Figure 2.6a). The main advantage of a crossbar lies in its strictly non-blocking property and as such can provide a higher

[9]In this thesis, the term arbiter and allocator are used intermingled. However, when taking about an arbiter in a circuit-switched NoC strictly speaking an allocator is meant.

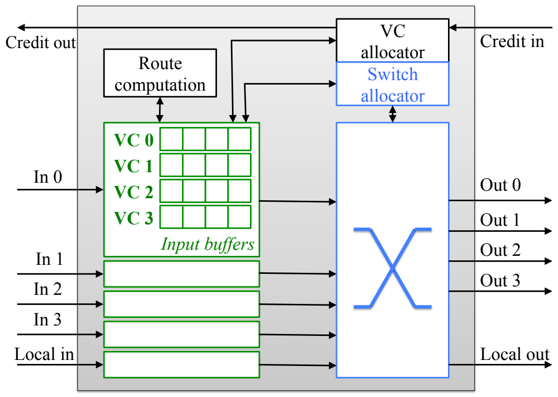


Figure 2.5: Microarchitecture of a credit-based virtual channel router based on [22, Chapter 6]

throughput than buses. However, crossbars do not scale well because there is still need for centralised allocation. An electronic crossbar also has a large area footprint and associated power consumption [42, Chapter 6.2] as the cost of a $n \times n$ crossbar increases with n^2 (the same holds for the allocator). Added to that come the connections from the network interfaces to the actual crossbar. Crossing long lines on a chip can take more than one clock cycle in modern technology nodes: it has been projected that at 70 nm, with a very conservative clock speed, only 20% of the chip can be reached in one clock cycle [17]. To counter this, intermediate buffers are placed on these long links. This is similar to the use of inverters as repeaters on long electrical links as will be discussed in Section 2.2.2. Nevertheless, crossbars should not be excluded from use in high-radix NoCs as careful design can improve their scaling characteristics, both in the actual crossbar switch [43] and the centralised allocator [44][45]. One of the examples of a commercial product using a crossbar is the SPARC-T series [6], as discussed in Section 2.1.5.

Meshes

However, the most commonly used topology for CMPs with high core counts is a mesh network. It consists of a grid of routers, each connected to its neighbours using bidirectional links (Figure 2.6b). The algorithms that take care of routing and flow control are implemented in the routers (Figure 2.5). A router generally consists of the following units: input buffers, routing logic, allocators and a crossbar that acts as a switch. They are often pipelined to improve throughput. The latency of a router is very important as all messages pass through multiple routers and as such have a major impact on network latency. The links are generally short interconnects. A lot of the design effort will be focused on the design of these routers and links as they will be replicated throughout the network. Banerjee et al. [46] compare multiple router designs and shows the main source of power dissipation lies in the routers, not the links. It is argued that the wide data paths in the routers, compared to the control paths, have a higher energy consumption, mainly because of the use of buffers. This is an argument for the use of more complex router control planes as has been proposed in by Mullins et al. [47], in which speculation is used to reduce the router latency to only 1 clock cycle.

Meshes can be described in two ways. First of all, a mesh is a type of torus network and can therefore be described using terminology borrow from torus networks. Torus networks are characterised using two parameters: n and k. A k-ary, n-cube network connects $N = k^n$ nodes where the nodes are arranged along a n-dimensional cube, with

k nodes in each dimension and links between neighbouring nodes[10]. The mesh depicted in Figure 2.6b can therefore, using this terminology, be described as a 3-ary, 2-mesh. For it to be a 3-ary, 2-torus, direct links would have to be provided between the left-most and right-most column of the grid for example. However, mesh topologies will more commonly be described to as a $m \times n$ mesh where m stands for the number of rows and n for the number of columns.

Meshes have various attractive properties, making them very suitable NoC topologies. First of all, links in this topology are identical and short, avoiding one of the main disadvantages of electrical interconnect as will be discussed in more detail in Section 2.2.2. Long lines that cross the chip need to be buffered or pipelined and a router (and by extension, mesh networks) can be seen as the natural successor of these buffers. Secondly, because a mesh is a very regular 2D topology, it is extremely suited for a Very Large Scale Integration (VLSI) environment. Routers and links can be designed once and repeated across the design multiple times. In general, every tile in the CMP will have its own router functioning as interface to the network. Finally, mesh networks provide path diversity. A message can take multiple, equally possible routes to travel from its source to its destination. This allows for the routing algorithm to perform load balancing and increases the reliability of the NoC. Unfortunately, the major disadvantage of mesh networks lies in the latency of a message travelling the network. This latency is determined by the number of hops it needs to make to reach its destination. Reducing the hops by reducing the number of routers in the network can reduce latency. To reduce the number of routers, concentration is used: multiple tiles will share one router. This has the additional advantage that it reduces the silicon footprint of the network.

The latency of a mesh network strongly depends on the dimensions of the network. The latency T of a message traversing a network is made up of two parts [42, Chapter 3.3.2]: the time it takes the head of the message T_H to travel the network and the time T_S it takes the rest of the message to catch up.

$$T = T_H + T_S \tag{2.1}$$

The serialisation latency T_S is determined by the bandwidth characteristics of the network. The header latency on the other hand is determined by the average hop count H_{min} which directly depends on the number of nodes in the network. T_r is the latency experienced when passing through a router, T_l is the link traversal latency.

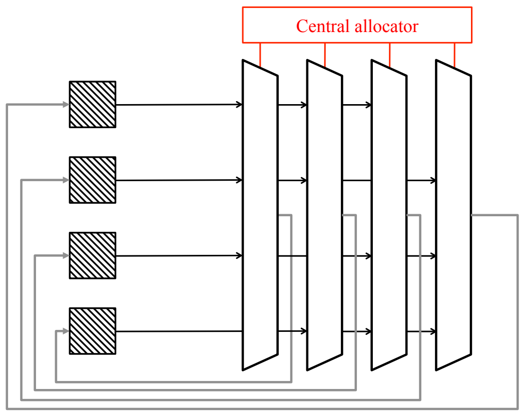
$$T_H = H_{min} T_R + (H_{min} - 1) T_l \tag{2.2}$$

H_{min} is determined by Equation (2.3)[42, Chapter 5.2.2].

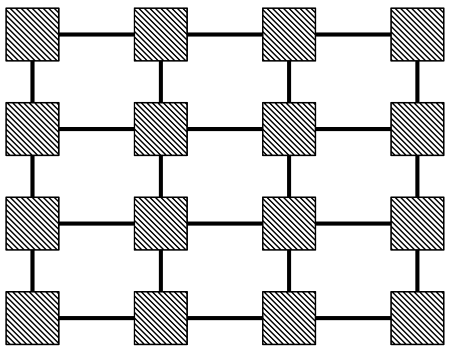
$$H_{min} = \begin{cases} \frac{nk}{3} & \text{if } k \text{ is even} \\ n\left(\frac{k}{3} - \frac{1}{3k}\right) & \text{if } k \text{ is odd} \end{cases} \tag{2.3}$$

In this equation n stands for the number of dimensions and k for the number of nodes per dimensions, as defined previously. This shows the number of hops will increase

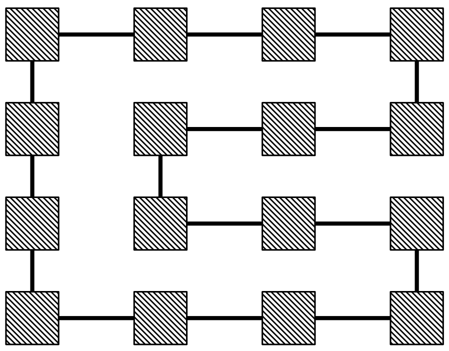
[10]Neighbouring nodes in this case are defined as nodes which only differ in one address digit. An address is based on the coordinates, in a $n \times k$ dimensional space



(a)



(b)



(c)

Figure 2.6: The most common NoC topologies (a) Crossbar connecting 4 nodes. The crossbar consists of four 3×1 multiplexers, controlled by a central allocator (b) Mesh topology connecting 16 nodes as a 4×4 mesh (c) Ring topology connecting 16 nodes

when the network increases in size. To limit the effect on the overall network latency, the router latency needs to be kept to a minimum. The link traversal latency will be fixed and equal to a clock cycle in most cases.

Mesh networks are mostly used for CMPs with a high core count, as the advantages outweigh the increased latency. If threads that frequently communicate are mapped onto neighbouring processor cores, the communication latency can be quite low as the number of hops will be minimal. However, for this to happen, the thread scheduler, which handles the mapping of threads onto processor cores also needs to have access to information on the communication latencies between the various processor cores. Thread migration, in which threads migrate between processor cores to, for example, minimise communication latency, however, is a research area by itself [48]. Examples of existing mesh architectures are the TILE64 and SCC systems. The Tilera TILE64 processor connects all tiles using five individual 8×8 meshes, where every mesh carries a different type of traffic. The Intel SCC also employs mesh networks.

Rings

Rings are another type of torus networks, where the number of dimensions n is equal to 1 and k, the number of nodes per dimension, is equal to the number of nodes in the NoC. They form a compromise between the traditional buses and complex packet-switched networks like meshes. Commercial examples of CMPs using ring networks are the Cell processor by IBM [49] and the Xeon Phi by Intel [4].

Low latency networks on-chip

The NoC is an integral part of the CMP. It connects the processor cores with various parts of the memory hierarchy and every message on the network is the consequence of a memory reference made by a processor core[11]. This means that a processor core is waiting for the outcome of these network transactions and that the design and performance of the on-chip network will have an indirect impact on the overall performance of the CMP. The latency of the NoC will, therefore, have a significant impact on overall performance (this will be discussed and measured in Section 4.1) and as such, there has been research focused on low latency NoCs.

There are multiple avenues to explore when trying to minimise network latency. Firstly, the message latency can be reduced by minimising the number of hops a message needs to take. This can be achieved by, for example, allowing messages to bypass routers. The work by Kumar et al. propose the use of express channels in a mesh network, in which messages are allowed to take express lanes until they got closer to the destination, after which they take the normal lanes [50]. This reduces the amount of hops a message needs to take. Using high radix routers can also reduce the number of hops a message needs to take, as the network diameter is reduced. For example, Kim et al. proposed a flattened butterfly [51]. A butterfly is an indirect network which consists of a number of terminal nodes and intermediate nodes, which while it provides a low hop count, also has no path diversity which can be disadvantageous[12] [22, Chapter 3.3]. By providing high radix routers to create a flattened butterfly, the authors provide better path diversity. Another possibility is, for example, remapping cache lines to caches closer to the CPU using those cache lines [52, Chapter 2.1]. A second body of work focuses on reducing the link latency to reduce the overall message latency. Chen et al. proposed the SMART NoC in which low-swing, repeated links allow messages to potentially travel multiple hops in a single clock cycle, in the absence of contention in the routers [53]. Muralimanohar et al. investigated the effect of the wire type on performance [54] as the type of wire (e.g. metal layer, wire width etc.) affects the link latency. Thirdly, the overall message latency can be reduced by reducing the time spent in the routers. The work discussed earlier on speculative routers [47], is one example which reduces overall network latency by reducing the router latency. Li et al. propose the runahead NoC which allows for lossy transmission in the runahead layer of the NoC. Messages are dropped when contention occurs in the runahead routers, thereby allowing for single cycle hops [55]. This work also uses the critical word scheme to their advantage as the critical word is sent via the lossy layer, while the complete cache line follows on the normal layer. This allows for a low latency delivery of the critical word, whilst the non-lossy layer guarantees the delivery of the complete cache line.

2.2.2 Motivation behind optical networks on-chip

In this section, I will argue that the electrical connections that make up the NoC have reached a performance plateau and motivate the choice for a transition towards an optics-based NoC. On-chip interconnect, which connects devices on the same chip, can

[11]One could argue a prefetch is also a consequence of an earlier memory reference made by a processor core.

[12]Path diversity is the possibility for a message to take multiple routes from source to destination. This allows for better load balancing.

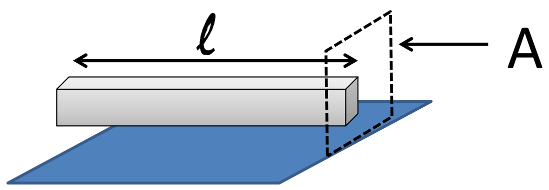


Figure 2.7: Schematic view of a RC line. The area A includes the spacing in between adjacent wires (after [9]).

be subdivided into two classes based upon their length. Interconnects with a length on the order of the size of the chip (like the NoC) are called global interconnects, in contrast to local interconnects. The use of electrical links to create these global interconnections might not suffice anymore. One of the alternatives being proposed is the use of optical communication [8]–[10], [56]. There are four reasons why electrical links can no longer meet the demands of NoCs. These four reasons all have to do with the continuous signal loss because of the capacitance and resistance of the wire, an effect that (at the computer system scale) does not exist in optics. In this section, these effects will be discussed and compared with the optical case.

Dependence on aspect ratio

The delay and attainable bandwidth of an electrical link depend on the length and area of the link. Conventionally electrical connections on-chip are formed by copper wires, surrounded by a low k dielectric. It is the rise time that determines the maximal bitrate on the line: if pulses are transmitted too close together in time, the finite fall and rise times will make them overlap, resulting in errors at the output. The rise time also determines the signal delay.

Most on-chip lines are resistive-capacitive lines (RC) because of their small cross-section[13]. An example of such a line is shown in Figure 2.7. A is the effective total cross-section and includes the spacing in between lines needed to reduce crosstalk[14]. The total length of the line is l. The rise time in a RC line is limited by its RC time constant which is equal to $R_l C_l l^2$ where R_l and C_l are respectively the resistance and capacitance per length unit. Some approximations can be made regarding the RC constant [9]. R_l can be approximated as $\sim 1/A$. C_l is constant and depends on the design of the line. This makes the RC constant approximate $\sim l^2/A$.

The bitrate B of a RC limited line is $\sim 1/\mathrm{RC}$ making $B \sim A/l^2$. Assuming copper as conducting material, well-designed lines and reasonable requirements for the eye diagrams the bitrate of an RC limited line will be limited to [9];

$$B \cong 10^{16} \frac{A}{l^2} bits/s \tag{2.4}$$

[13]Transmission lines, which have lower latency and higher signal integrity, cannot be considered for on chip links. The resistance of such a line needs to be negligible which implies a large cross-sectional area. This cannot be achieved while supporting the bandwidth density requirements of modern Complementary Metal Oxide Semiconductor logic (CMOS) chips.

[14]Switching a wire will affect neighbouring wires through capacitive coupling. Crosstalk can cause noise on the neighbouring lines or depending on the switching activity, change the rise times of the neighbouring lines[16, Chapter 6.3.3].

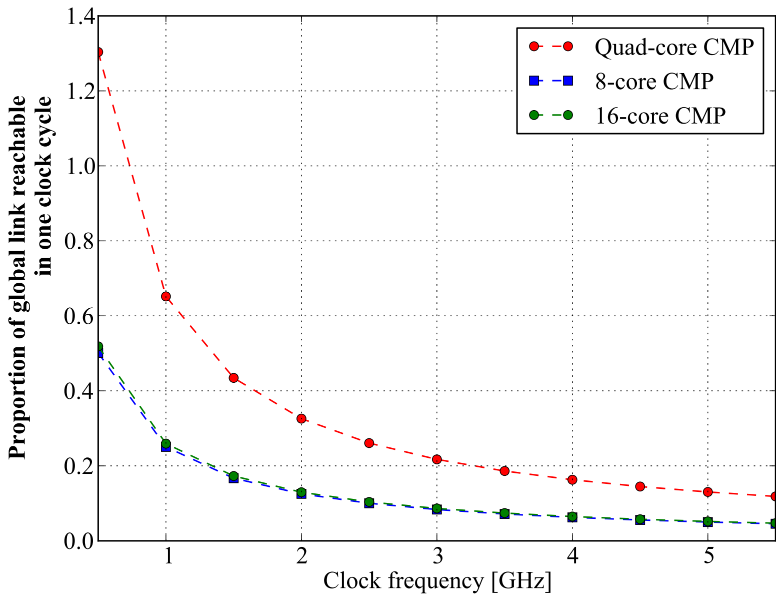


Figure 2.8: The proportion of a global interconnect line crossable in less than one clock cycle at a 45 nm technology node. The die size of the (Intel) 16-core (green line), 8-core (blue line) and quad-core (red line) is respectively 661 mm^2, 684 mm^2 and 263 mm^2.

The bitrate across an electrical link is therefore limited by the *aspect ratio* of the line, the ratio of the length and the cross-sectional area of the line. Global lines on-chip have a fixed l and so the upper limit on A is given by the number of metal layers available for global interconnects. Once all available metal layers are occupied by wiring, the width of the wire cannot be increased further and it is not possible to increase the bitrate. This puts an upper limit on the total bandwidth of the network. The propagation delay is also a function of the RC time constant and it follows from the above discussion that it increases quadratically with distance. Figure 2.8 shows the fraction of a global link that can be crossed in less than one clock cycle, assuming a 45 nm technology node. There are multiple metal layers present in a VLSI design: global interconnect will be using the M8 metal layer. The results are shown for three Intel CMPs: a 16-core, 8-core and a quad-core CMPs which differ in die size. Only when the quad-core would be clocked at 500 MHz would it be possible to cross the global interconnect lines in less than a clock cycle.

There are multiple possibilities to reduce the impact of the aspect ratio: equalisation of the lines[15], using multilevel signalling or the use of repeaters [58],[16]. Repeaters divide the link into shorter segments, each driven by a CMOS inverter. This reduces the RC time constant per line segment. The RC constant of the total line is therefore $\sim N \times R_l C_l (\frac{l}{N})^2$. Assuming the number of segments N is proportional to the total length l, the total RC time constant is now linearly dependent on the distance (compared to the quadratic dependence of an unrepeated line). The exact number of repeaters per line is a trade-off: the delay per segment decreases with increasing number of repeaters while the delay due to the repeaters increases with increasing repeater count. Whereas repeaters are advantageous in terms of signal delay and bitrate, they come with multiple disadvantages. The exact number of segments needs to be optimised for the desired bitrate and link length. Any change to the bitrate for example will imply a redesign of

[15]An equalised signal has been compensated for the effect of the channel transfer function, resulting in a higher data rate at the cost of higher signal to noise ratio [57].

the repeated lines (if the number of repeaters is chosen to supply up to the requested bitrate). Repeaters also increase the energy consumption per bit. When designing the repeated line for minimum delay, there is an 87% increase in energy per bit compared to the unrepeated line [16]. To prevent signal inversion, one has to either provide an even number of repeaters on the line or use buffered repeaters (flip-flops) which are easier to work with but come with slight increase in delay, area and energy consumption. The CMOS inverters also need to be connected to the metal layers providing the global interconnect, resulting in an increase in the number of vias.

Optical interconnect on the other hand does not suffer from a dependence on the aspect ratio of the link [9]. Signal degradation in electrical links and optical links differ radically, at the link lengths used in this work. Simply put, one could say a signal on an electrical link degrades continuously, all across the link. In an optical link, however, the highest losses are more localised. A bend in a waveguide for example will be a source of signal loss. In Section 2.2.3 the different components in an optical link and the losses they incur, will be discussed in more detail. The loss of a Silicon-on-Insulator (SOI) waveguide ranges from 0.1 dB per cm to 3.6 dB per cm (Table 2.3). The loss of a polymer waveguide is lower in general and ranges around 0.03 dB per cm -0.04 dB per cm [59]. Individual components on the optical path, however, can add significant losses. However, passing by a microring resonator used as modulator can incur between 0.1 dB to 1.5 dB (Table 2.3). To compare these values, when assuming the quad-core die size used in Figure 2.8, the total loss of an optical link crossing the complete die is equal to approximately 0.006 dB, assuming the lossiest type of waveguide is used.

Optical signals, in contrast to electrical signals, have no frequency dependent losses for up to tens of GHz, at the scale of computer systems. This is due to the fact that the optical carrier frequency is so high (in the order of THz) that any changes modulation frequency are negligible, relative to the carrier frequency [60]. This is also means an optical link (with the exception of the devices driving and receiving the signal) does not need to be redesigned when changing the modulation frequency in contrast to a repeated link, for which the optimal placement of the repeaters depends on the bitrate on the link.

The signal delay on an optical link consists of two major parts: the conversion from the electrical to the optical domain (also known as the serialisation latency) and the time of flight of the signal in the waveguide. The time of flight of the signal is determined by the speed of light in the waveguide material and as such, is very short. An optical signal in a silicon waveguide with refractive index $n = 3.4$ will travel at $\sim$11 ps per mm. In a polymer waveguide, with $n = 1.6$ this comes down to $\sim$5 ps per mm. Haurylau et al. note that the propagation delay of an electrical signal will remain fixed around 20 ps per mm, assuming the widest wire pitch [61]. Figure 2.9 plots the propagation delay of electrical and optical interconnect. Most of the latency of an optical signals originates from the serialisation process in the transmitter as the data needs to be modulated serially onto the light stream. The serialisation latency can be reduced by increasing the number of wavelengths available or increasing the modulation frequency. The O/E conversion latency at the receiving end is negligible as it lies in the ps range[62].

It needs to be noted though that the delay of a signal, be it electrical or optical, as discussed here only looks at the link itself and does not look at the delay of scheduling transmission across that link, which might be problematic for optical links, which will be discussed in more detail in Section 2.2.3.

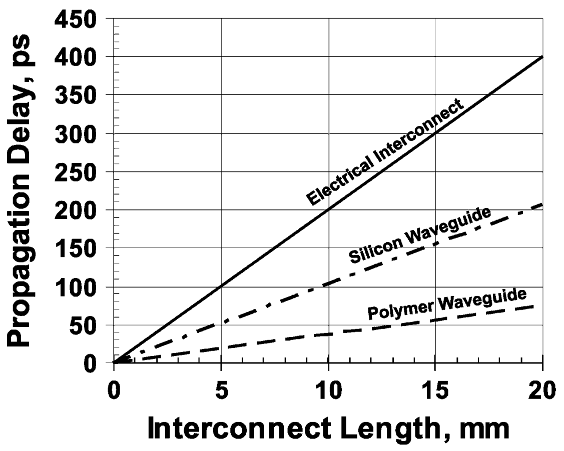


Figure 2.9: Propagation delay of electrical interconnect compared to the propagation delay of polymer and silicon waveguides (from [61]).

Scaling

The scalability limits of electrical links are the second reason they are less suited for use in NoCs. A major driving force behind the progress in microprocessors has been the continuous scaling of CMOS technology. Improved fabrication techniques have allowed Metal Oxide Semiconductor Field Effect Transistors (MOSFETs) to be steadily reduced in size since the 1960s. However, MOSFETs and electrical links react differently to technology scaling. Scaling the dimensions of a MOSFET with an arbitrary factor S (while keeping the electrical field constant), decreases the switching latency with a factor S. At every new technology node, transistors become faster. This is not the case for electrical interconnects. Scaling a wire in all dimensions will increase the resistance per unit length R_l because the cross-section decreases. But because the length will decrease as well, the total RC time constant of line does not change. Only local interconnect will scale in length, global interconnects are unaffected as the chip dimensions do not change significantly [17]. Therefore the latency of global links that make up the NoC will increase [9].

Another unwanted side-effect of scaling is the increase in the parameter variability. Any uncertainty on the delay of a component or link will lead to increased timing margins which lower the clock frequency. This is because even though signals are clocked, the following clock edge can only come when all signals have arrived at the next stage. There is an increased uncertainty in propagation delay across electrical wires when scaling: first of all, any process variations[16] can lead to changes in wire width and therefore a variation in the RC time constant. As the resistance of copper is temperature dependent, variations in temperature across the chip (e.g. proximity to a very active processor core) can lead to delay variations, again because of the change of the RC constant. The repeaters are CMOS inverters and as they are subject to increased variability due to scaling, their contribution to the link propagation delay will be increasingly variable. As the number of repeaters per link is expected to increase, this effect will be reinforced even more [63] . The capacitance of a line will also be affected by switching activity on nearby wires due to the Miller effect[17]. All this means that

[16]Process variations are caused during fabrication whereas environmental variations occur during the lifetime of a chip.

[17]Wires have a parasitic capacitance with their neighbours. This capacitance will also affect the delay

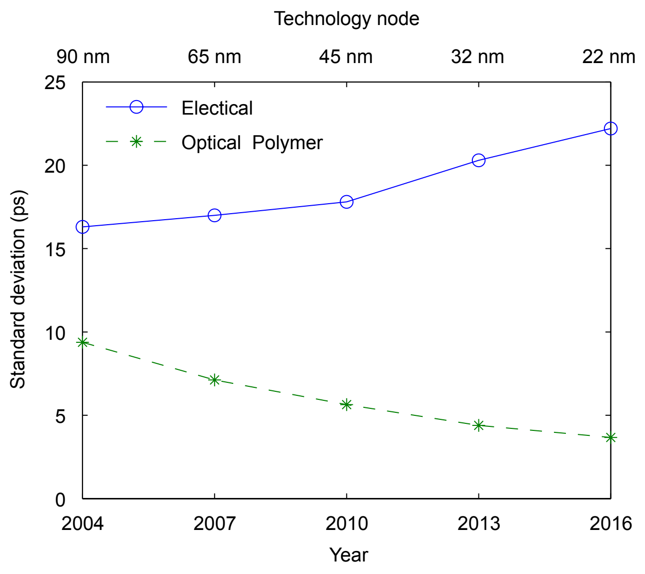


Figure 2.10: Expected standard deviation of the delays of optical and electrical interconnects (from [62]).

the delay uncertainty of electrical interconnections will increase in future technology nodes.

Scaling the width and height of interconnect also introduces reliability issues: scaling down electrical connections will also lead to an increase in electromigration[18] which results in a lower reliability [56]. It is possible to avoid electromigration (e.g. by applying a surface coating [64]) or reduce its effects on reliability (e.g. by inserting reservoirs of interconnect material at points where electromigration will lead to failure [64]) but electromigration is an important effect to consider in electrical interconnect. In contrast, optical interconnects do not suffer from scaling significantly. Any variability of the delay of an optical signal (be it on the same device or between devices) will be coming from the electronic transmitter and receiver circuits but the overall variability will still be lower than the electrical case as shown in Figure 2.10. Geometric variations of the waveguide are possible and the effect of sidewall roughness will increase with scaling but any delay uncertainty caused by the waveguide is small in comparison to other factors [62]. This would make optical signals very useful for the clock delivery for example. In the case of off-chip signals, this can remove the need for resynchronisation at the receiver. This frees up both area and power budget.

Bandwidth density

Bandwidth density can give an idea about how efficiently an interconnect technology uses the available chip area. It therefore should be as high as possible as the perimeter around every processor core is limited [65]. The literature looks at this issue in two manners: Miller quantifies the bandwidth density of an electrical link as Gbit/s per μm^2 where the cross-sectional area is seen as defined previously in Equation (2.4) [60].

of that line. The size of the parasitic capacitance will depend on the signal on the neighbouring lines, being maximal when they both switch in an opposite direction. This leads to the delay of a line becoming data dependent.

[18]Electromigration is gradual movement of conductor atoms, because of constant electron bombardments i.e. the slow displacement of Cu atoms because of the current the wire is conducting.

In this equation the achievable bitrate is determined by $\frac{A}{l^2}$. Both A and l are fixed quantities: the link l length remains constant across technology nodes in the case global interconnects. The available area A is determined by the available metal layers. Once these are exhausted, the bandwidth density can no longer be increased without employing repeaters for example.

Koo et al. [65] and Chen et al. [63] quantify bandwidth density as Gbit/s per µm to simplify the comparison between electrical and optical links. Chip area is, in these papers, interpreted as circuit area: link length multiplied by link width. Considering the link length of optical and electrical links is the same (both providing global interconnect), only the link width will be included when quantifying bandwidth density. The advantage of optical interconnect over electrical interconnect in terms of bandwidth density is less clear because of the presence of transmitter and receiver circuitry in the optical links and the small wire pitch of the electrical wires [9],[62]. For local interconnect, electronics will outperform optics for this reason. For longer distances, this is not the case as the bandwidth of an optical signal can quite easily be increased by using Wavelength Division Multiplexing (WDM). Using WDM it is possible to send multiple data streams down one waveguide. This allows to increase the bandwidth even further.

The overall advantage of optical interconnects over electrical interconnects in terms of bandwidth density will most likely continue in the future as the bandwidth of an optical link can easily be increased by adding more wavelengths by means of WDM. Increasing the number of wavelengths decreases the channel spacing, incurring higher signal losses, requiring more power to be injected per channel. This in turn increases the aggregate power in the link, which is limited by non-linear waveguide losses and instabilities in the microring resonators due to heating [66]. The upper limit imposed on aggregate power due to non-linear waveguide losses can be mitigated by changing the waveguide material to silicon nitride which suffers less from these effects. However, the integration of this material remains challenging [67]. Both electrical and optical interconnect could make use of more complicated modulation and coding formats to increase the bandwidth, which has not been considered in any of the works discussed previously ([63][60][65]).

By using wavelength striping [68], WDM also can be employed to reduce the latency. Figure 2.11 compares the conventional use of WDM with wavelength striping. Wavelength striping allows to decrease the serialisation latency by increasing the bandwidth. By modulating the first part of the message first, this part of the message will arrive first at the receiver. This is especially important for data messages in a memory hierarchy. As mentioned in Section 2.1.4, a cache line consists out of multiple data words, which are grouped together for efficiency reasons. A processor core, however, will only request one single word to work on. The other words in the cache line are not used immediately. Therefore, the critical word first scheme is employed [2, Chapter 5]. When a data message containing a cache line is travelling the memory hierarchy, the word that was requested will be placed in the first part of the message. When the message arrives at the L1, the word can then immediately be given to the processor core while the rest of the message, containing the other words in the cache line, is still arriving (or in the case of an optical network, still deserialising). Using wavelength striping, it is possible to do this.

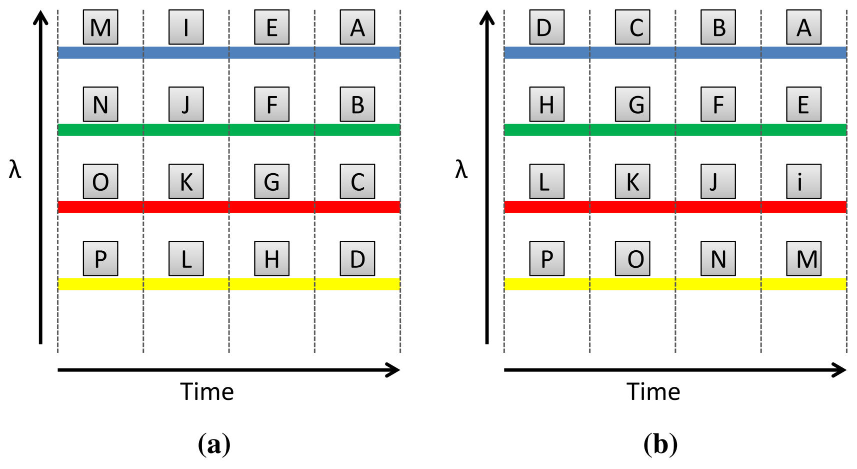


Figure 2.11: Comparison of the use of wavelength striped and conventional WDM in a NoC. The message to be transmitted is the alphabet. The letters are used to show the order of bits in a message (a) when using wavelength striping, the first part of the message will be modulated first (b) conventional use of WDM. In this case the 4 different wavelengths could also be used to transmit 4 different messages.

Power consumption

The last disadvantage of electrical interconnects is their power consumption. Power consumed on-chip has to be dissipated as heat. This puts an upper limit on the allowable on-chip power dissipation. The International Technology Roadmap for Silicon (ITRS) [69] has stated in 2007 that the amount of heat that can be removed from chip will most likely level around 200 W making power management a very important issue [60]. To place this into context: the quad-core shown in Figure 2.8 for example, was released in 2009 and had a Thermal Design Power (TDP)[19] of 130 W [70]. The 16-core shown in the same figure was released in 2015 and has a TDP of 165 W [71]. Power consumed in the interconnect consists of dynamic power and static power. Part of the static power in electrical connections comes from leakage power which increases when scaling down transistors. The most important part of the power consumption, however, is dynamic power consumption. Dynamic power consumption is determined by the capacitance of the line, C, the switching frequency f and the switching voltage V_s.

$$P_{dyn} = CV_s^2 f \tag{2.5}$$

By not using rail-to-rail signalling the voltage swing can be reduced thereby lowering the power consumption. However, the capacitance of a wire is determined by design and cannot be reduced. Added to this comes the power consumed in the repeaters.

The possibility of lower power consumption when using optical interconnects forms another advantage [60]. The static power consumption is determined by the power of the laser source. The best location of the laser source will be discussed in Section 2.2.3. The dynamic power, however, is not determined by distance, a significant advantage over electrical interconnect. The only switching activity (resulting in dynamic power

[19]The Thermal Design Power is defined by Intel as *"the average power, in watts, the processor dissipates when operating at Base Frequency with all cores active under an Intel-defined, high-complexity workload"*.

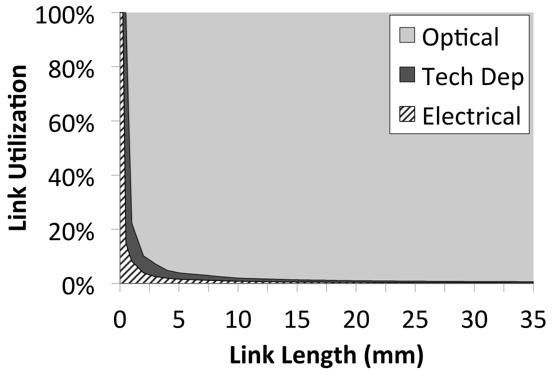


Figure 2.12: Transition point at which optical interconnect becomes more energy-efficient than electrical interconnect, depending on both link length and link utilisation, which captures how often the link is actually in use to transmit data (directly taken from [72]).

consumption) in optical links is coming from the receiver and the transmitter circuits.

Conclusion

All of the previous factors make it increasingly difficult to use electrical links to provide global on-chip connectivity. Various techniques to improve electrical signalling have been proposed but none of these solve the underlying physical problems limiting the performance of electrical interconnect [73]. For example, the length of connections can be reduced by using 3D integration [74] but the heat dissipation of such a system will be challenging. A second approach would be the use of additional cooling (e.g. liquid cooling) to reduce the intrinsic resistance of copper. The cost of such cooling systems, however, is only justified for chips aimed at the supercomputing market [9]. A third option would be to increase the amount of wiring available for global on-chip communication: the number of available metal layers could be increased or off-chip connection could be used for on-chip communication [9]. A last alternative would be a transition from copper-based interconnects to carbon nanotubes (CNT) [65] which have a lower resistance. These four alternatives would improve the current performance of copper-based electrical interconnect but the physical properties that limit performance are still present.

The use of transmission lines to reduce latency and energy consumption has also been proposed, for example by Carpenter et al. [75]. They argue the use of transmission line-based buses will remove the need for packet-switched NoCs which incur non-negligible energy overheads due to routing, packet transmission and switching. However, the scalability of this system is limited and the bandwidth density of transmission lines in general is smaller than that of optical interconnect [76]. The advantage of optical interconnect over transmission lines in terms of energy efficiency will, however, depend on both the link length and whether or not the laser is on-chip [76].

Because of all these reasons, optical interconnect might be a viable alternative for electrical interconnect. Apart from the advantages discussed previously, optical interconnect might result in design simplifications [9]. First of all, optical links are bitrate transparent and do not need to be redesigned when the clock frequency is changed (in contrast to electrical links). The second simplification is that fact that, at these scales, there is no distance dependence for optical signals so there is no difference in behaviour and design of on-chip and off-chip interconnects (in contrast to the high inductance off-chip pins). Thirdly, because of the low latency of optics they might allow for the use of larger synchronous zones on-chip. The last simplification is the voltage isolation between the communicating nodes provided by optical links.

Overall, the advantages of optical interconnects over electrical interconnects increase with increasing link length. This is due to the fact that optical interconnects have a relatively distance-independent latency and power consumption, while both these properties degrade with increasing distance in the case of electrical interconnects. However, because of the fixed costs (both in terms of latency and power) associated with optics, there is a critical length for interconnects, known as the partition length [77] beyond which optical interconnects outperform their electrical counterparts. When looking at the energy efficiency of optical interconnects, it is important to take link utilisation into account to assess the partition length as for optical interconnects the static power is quite high. Figure 2.12 shows the transition point between electrical and optical interconnects, depending on both the length of the link and the link utilisation [72]. The link utilisation captures how often the link is actually in use to transmit data For very short links, electrical interconnect will be more energy-efficient. However, the longer the link becomes, the more efficient optical links are. There is a dark zone on the plot where the exact transition point is unclear, because it will depend on the exact technology used. Overall though, this figure confirms optical interconnects can be a more energy-efficient option for NoCs.

However, the use of optical interconnect in the NoC is not without challenges: the technology is relatively immature, especially compared to electrical interconnect. Another technical challenge is the temperature dependence of various optical components. This will be discussed in more detail in the next section.

2.2.3 Optical networks on-chip

Optical interconnects can be seen as a natural extension of the use of optics for long-haul transmission [78] and were first proposed by Goodman et al. in 1984 [56]. More publications, by Miller [8], [9] and [10] followed in the early 2000s.

In this section the various components needed for an optical NoC will be discussed, followed by a review of the most important optical NoC proposals. This review will allow me to compare these optical NoCs with the system proposed in this thesis.

Components in an optical network on-chip

Optical on-chip links require optical components which satisfy two requirements. First of all, they need to be small enough to fit in the area of chip designated for the NoC[20]. Secondly, their total power consumption should not exceed the total power the NoC is allowed to consume. These two upper limits depend on the actual CMP but they indicate the need for energy and area efficient components. In the following section the components needed for an optical NoC will be discussed. This discussion will be

[20]In a 3D integrated system, it would be possible to place the photonics in a designated layer as proposed by Udipi et al. [79]. However, as previously mentioned, the power dissipation of a 3D integrated system is challenging. Considering the thermal stability needed by some optical components (as will be discussed later on), this might not be the optimal solution. It would also be possible to place the photonics on a silicon interposer, so called 2.5D integration [80]. However, the preference for small photonic components would remain.

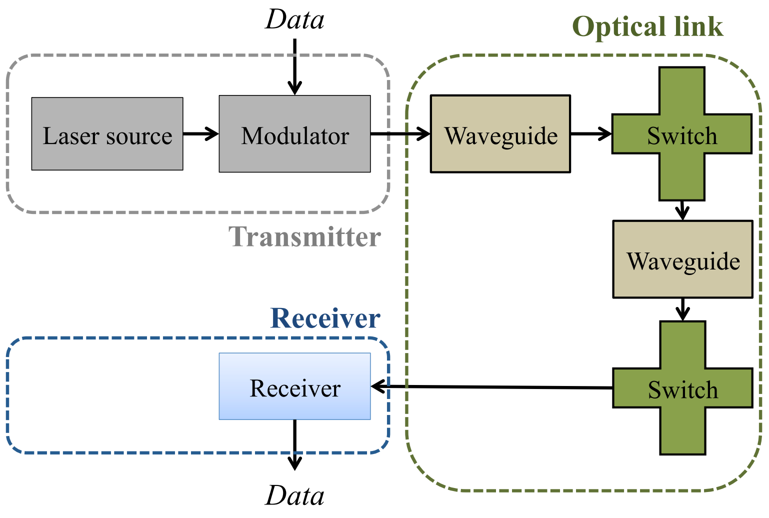


Figure 2.13: Schematic representation of a photonic link.

brief as the physical implementation of an optical NoC is not the aim of this thesis. Figure 2.13 shows the various parts of an optical link. The link consists of three parts: the transmitter, the transmission medium and the receiver.

General

Microring resonators can be used in multiple parts of an optical link. They are suitable for use in an NoC as they are area-efficient (in the µm range) and energy-efficient (in the fJ per bit range)[60]. They are formed by building a SOI waveguide in the shape of a ring, in the vicinity of a waveguide transporting the signal as can be seen in Figure 2.14 [81]. The exact circuitry surrounding them determines their final use but the basic principle is always the same. When the optical path inside the ring is equal to an integer number of wavelengths, the optical signal on this wavelength will be coupled into the ring [81]:

$$\begin{aligned}\lambda_{res} &= \frac{n_{eff}L}{m} \qquad\qquad m = 1,2,3.... \\ &= \frac{n_{eff}2\pi r}{m}\end{aligned} \tag{2.6}$$

The optical path inside the ring depends on the refractive index n_{eff} as can be seen in Equation (2.6). By changing the refractive index of the material, the resonant frequency λ_{res} can be shifted. Changing the refractive index can be done by adjusting the carrier density which can be achieved by changing the voltage across the ring. This will shift the resonant frequency towards the blue. As the refractive index of a material is temperature dependent, the temperature of these devices needs to be strictly controlled to guarantee reliability: the resonant frequency of a ring will shift by 0.09 nm/°C [82]. This is one of the major disadvantages of microring resonators as tuning can be quite power consuming: 130 µW/nm for a shift to the blue by means of a current injection and 240 µW/nm for a shift to the red by means of heating [83]. Microring resonators can be used as modulators, switches and detectors.

Transmitter

The start of an optical link is formed by the transmitter , where the conversion from the electrical to the optical domain takes place. This is done by means of a modulator which modulates the data onto a stream of light.

The unmodulated light will be generated by a laser. The light can be generated on-chip or off-chip. Whilst off-chip lasers are highly efficient in terms of light-emitting and do not add to the temperature instability on-chip (thereby affecting the microring resonators that make up the modulators and switches less), the coupling to the silicon chip is very lossy and packaging will more strenuous [84]. On-chip lasers could exhibit better energy efficiency and higher integration intensity, but the limited heat dissipation on-chip might be problematic. The research into on-chip lasers is still very active [84], [85]. Generating light in silicon is difficult as silicon is an indirect band gap material: the recombination of a hole and electron will generate both a photon and a phonon. This makes silicon a less efficient material for stimulated emission.Nevertheless, there is a body of work focusing on the demonstration of a silicon laser [85]. A recent review by Zhou et al. [84] lists three types of light sources that are based on other materials besides silicon: erbium-related, germanium-on-silicon and III-V materials based. It is important that these materials are compatible with existing CMOS fabrication techniques. Quantum dot lasers are one example of on-chip lasers using III-V materials and were first demonstrated by Liu et al. in 2011 [86]. External lasers could provide the additional advantage [82] that one central laser could provide optical power to the complete chip or even multiple chips. The light beam of an off-chip laser does need to be coupled onto the chip, at the cost of additional coupling losses which can range between 0.9 dB [59] and 3 dB [87], depending on the type of waveguide used. There are multiple possibilities for the exact implementation of the modulating device itself. The most common type proposed for future integrated electronic/photonic systems is the microring resonator due its small size, high modulation speeds (10 GHz to 25 GHz [81]) and the lowest modulation energy per bit [81][21]. It can act as a modulator by placing it next to a passing waveguide (Figure 2.15a). By changing the refractive index of the ring, On-Off Keying (OOK) is possible. When using WDM, multiple rings will be used, each tuned to one wavelength in the stripe. There are two main challenges associated with the use of microring resonators as modulators. Firstly, there is the thermal tuning power discussed previously. The second challenge is the loss an optical signal experiences when passing by a microring resonator. When the microring is off-resonance (i.e. not tuned to the wavelength passing by), this loss can range from 0.0001 dB to 0.1 dB in the literature [82]. The on-resonance loss is reported to vary from 1 dB to 3 dB [82]. The on-resonance loss cannot be avoided but the off-resonance loss can incur a high penalty in systems with a large number of wavelengths per waveguide (a high WDM factor).

Waveguides, switches and splitters

After the signal has been modulated onto the stream of light, the stream will then travel the optical links. Optical links consist of waveguides and switches. The waveguide confines and guides the light. There are two main candidates [62]: SOI can be used for short and dense connections as it allows for a smaller pitch whereas low-loss polymer could be used for longer paths. Waveguides introduce propagation loss, intersection loss and bending loss. Light can be coupled from a waveguide in one plane to a waveguide in another plane which induces a loss as well. The coupling can be done by means of grating coupler or mirrored surfaces [89] .The parameters for the loss factors again

[21]Mach-Zehnder interferometers could also be used as modulators. Whilst they offer a larger bandwidth than microring resonators, their extinction ratio is lower, meaning the output power in the OFF state is non-zero [88]

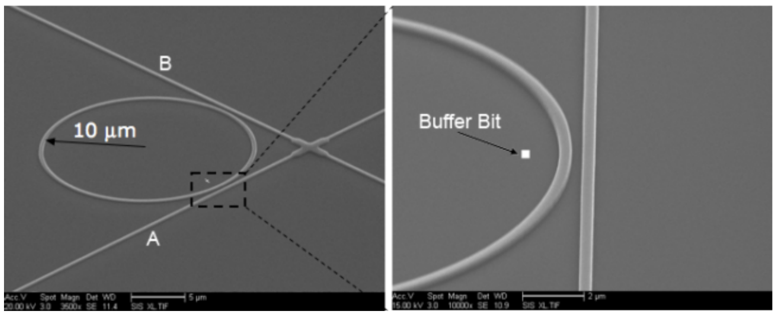


Figure 2.14: SEM picture of switch microring resonator. This picture also shows the difference between the electrical components (in the form of a buffer) and photonic components (from [82]).

vary quite significantly across the literature as shown in Table 2.3. Optical switches are used to change the direction of the light. These can be implemented using microring resonators [90], with the same challenges with regard to tuning power and losses as discussed previously. A schematic view is shown in Figure 2.15b. There are two types of networks when using WDM. If the wavelength of a signal determines its path in the network, passive microrings are used. These rings are always in resonance with that signal. Active rings on the other hand switch between the off-resonance and on-resonance state, thereby at runtime determining where the signal goes exactly. Active rings can be used to create optical crossbars [90], which logically function exactly like their electrical counterparts. Broadband splitters are used to split the incoming optical signal across multiple waveguides. Splitters can be implemented in multiple different ways [88]: Y-branch splitters, which split the power in half [91], or Multi Mode Interference (MMI)-based structures which can split the power across multiple branches (≥ 2) [92]). Splitters introduce another loss to the optical signal: Y-branch splitters for example are associated with 0.1 dB to 0.2 dB loss[91].

Receiver

The last device in an optical link is the receiver. The receiver stage consists of three parts: a microring resonator functioning as a filter guiding the selected wavelength to the photodetector. The most commonly proposed photodetector is the P-I-N diode [87]. The photodetector responsivity, which relates the incident optical power with the output current, determines the maximum allowable path attenuation. The optical power in the signal at the beginning of the path (i.e. when it is generated off-chip) will be attenuated by all the lossy elements encountered (e.g. coupling loss, microring resonators etc.) but enough optical power needs to reach the photodetector in order for an output current to be generated. In the receiver unit, the integration of the photodetector is vital as a low input capacitance is needed to keep the receiver as small and power effective as possible. If the physical capacitance of the receiver and the connection to the amplifier are small, they will give larger input signals to the amplifier, resulting in less stages and better noise immunity. A small RC time constant will also ensure the receiver has an acceptable bandwidth (similar to the RC time constant of an electrical link). Another option (still speculative though) would be for the receiver to be receiverless [60]. In this case the photodetector would be able to directly drive a logic level voltage swing that can be routed to the logic circuits. Overall, the receiver is not seen as the main challenge [60].

The losses associated with each component in an optical NoC will determine the

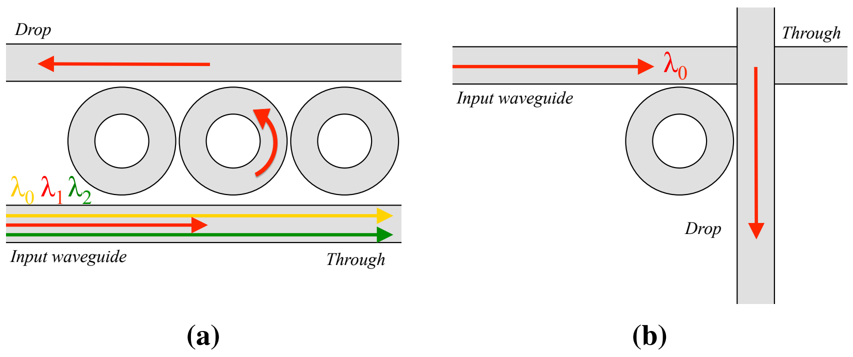


Figure 2.15: Schematic depiction of the various functions a microring resonator can fulfil (a) Microring resonator as modulator (b) Microring resonator as switching element.

Component	***Description***	***Minimum***	***Mode***	***Maximum***
Waveguide	Propagation	0.1 dB/cm [93]	1 dB/cm [85], [94], [95]	3.6 dB/cm [96]
	Intersection	0.05 dB [97], [98]	0.05 dB [97], [98]	0.2 dB [93]
	90° bend	0.002 15 dB[22]		0.5 dB [87]
Microring resonators	On-resonance	0.1 dB [94]	1.0 dB [87], [93]	1.5 dB [93], [98]
	Off-resonance	0.0001 dB [98]	0.001 dB [94], [97]	0.1 dB [93]
Coupler	Layer-layer (via)	1 dB [87]		3 dB [93]
	Fibre-to-chip	1 dB [85], [94], [95], [97]	1 dB [85], [94], [95], [97]	3 dB [87]
Photodetector	Loss	0.1 dB [97]	3 dB [94], [95]	3 dB [94], [95]
Splitter	Loss	0.1 dB [94], [95]	0.2 dB [87], [97], [98]	0.2 dB [87], [97], [98]

Table 2.3: Optical loss parameters in the literature, taken from [82]. Only the values for silicon waveguides are included because of their smaller pitch.

overall power consumption. However, the values reported in the literature can vary quite a lot. This variation can be due to differences in technology used or methodology used (analytical versus experimental). This makes it hard to quantitatively assess power consumption figures across proposed topologies. Table 2.3 integrally taken over from [82], shows the variation of various parameters.

[22]This value is said to come from a reference to Vlasov et al. [99] in Koch et al. [100]

Proposed optical networks on-chip

Optical and electrical interconnection networks differ greatly. The major difference is the fact that there is no practical optical memory available which can easily be integrated on-chip to be used as intermediate buffering[23]. Electrical interconnects depend heavily on buffering, either in buffered repeaters or in routers. This implies messages can travel the network in multiple hops and while the message is safely stored in intermediate buffers, the resources for the next part of the route can be obtained (packet-switched approach). This is impossible in optical networks, as there is no optical memory . For a message to be buffered in an optical network, it has to be converted from the optical domain into the electrical domain, buffered and converted again into the optical domain, which is costly as discussed previously. Most optical networks will therefore be circuit-switching networks (i.e. the end-to-end path is acquired before message transmission) to avoid this intermediate buffering. To make optimal use of a circuit-switched network, the number of messages per circuit should be maximised to reduce the circuit setup overhead. However, this is not a given as circuits can be setup for a single message and be torn down immediately if the circuit resources are needed elsewhere. Packet-switching is possible in optical networks on-chip, but as the messages acquire (such as optical links or wavelengths for example) resources whilst they travel the network, they might need intermediate buffering during this arbitration process.

In either network type, the arbitration process is important and will be a very important difference between the various optical network on-chip proposals. Arbitration can be global (circuit-switched) or distributed (packet-switched). However, centralised arbitration can form a bottleneck at high loads[24]. High loads are not very likely in an on-chip environment however[25] More importantly, any form of arbitration adds latency. Before the transmission of a message, a request of some sort needs to be made for the resources needed, followed by arbitration over all incoming requests and a grant of some sort needs to return. A grant is a short message from the allocator to tell the original requester that the resource requested has now been granted. This arbitration latency will have an impact on the overall network performance.

Many optical network on-chip incorporate some form of message transmission in the electrical domain. For example, there could be a second, electrical network in parallel that routes certain message types. Many proposals also use electrical links to connect the processor cores etc. to the actual optical network in order to reduce the number of optical nodes. Part of the message journey will then be in the electrical domain. However, in the following overview, this part of the network will not be discussed as this is just a form of network concentration [22, Chapter 3.6].

Whilst optics have been in use for a long time in long-haul communication [82], the requirements are very different, leading to very different network proposals. In long-haul optics for example, it is possible to use optical burst switching [105]. In this scheme, packets are collected at the network edge until a large enough burst is collected.

[23]Optical memory has been proposed in the form of delay lines [101][102] and bistable microring lasers [103]. Neither of these are particularly suited for on-chip operation as they do not function like electrical buffers in which the data can be held for an indefinite amount of time.

[24]For one of the most used allocation algorithms, the iSLIP algorithm [104] the curve of the load-latency relationship will change and become steeper around the 60% mark. The exact saturation point however, will depend on the traffic offered,the exact algorithm used and to a lesser degree the number of ports.

[25]This assumption will be confirmed in Section 3.2.1.

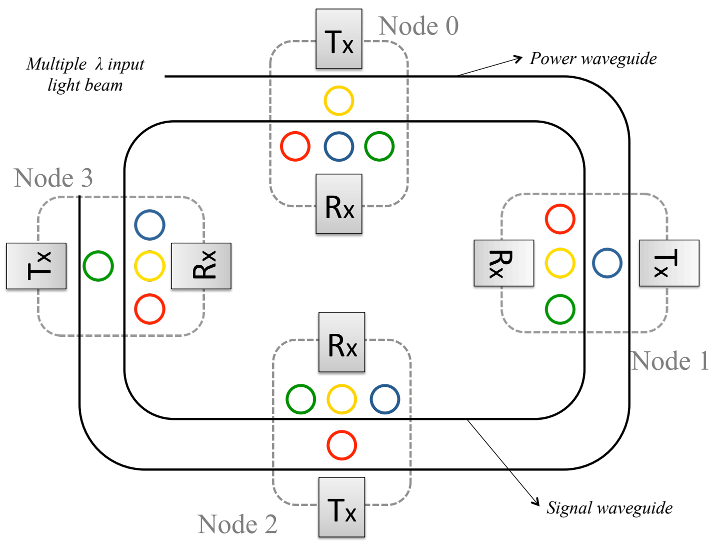


Figure 2.16: Schematic depiction of a SWMR scheme as employed in Firefly [96]. Every node has a dedicated wavelength to write to and from to which all other nodes read. Arbitration is not needed.

This burst is then transmitted across the network, thereby reducing the arbitration overhead. However, this would be impractical in a CMP environment, where every message is an (indirect) consequence of a pending memory request from a processor core. Buffering messages at the edges of the NoC would lead to an unacceptable delay of the memory requests. Another difference lies in the use of modulation formats in long-haul optical network. These increase the information transmitted by increasing the number of bits/symbol. Using OOK, only 1 bit is sent per symbol. However, there are no proposals yet incorporating complicated modulation formats in an optical NoC.

The proposals discussed below are grouped according to how resources needed by a message are allocated. In packet-switched networks, these resources are allocated on the go, whilst the message travels the network. In circuit-switched networks, the resources are allocated before transmission. I distinguish between two types of circuit-switched networks. First, there is the classic circuit-switching, resources are allocated at runtime, before message transmission. The second type are networks in which the resources are allocated during design time, by setting up un-arbitrated direct links between all nodes for example. As in this type no circuits are setup at runtime, I will refer to these as non-switching architectures.

Non-switching: Firefly

In the Firefly architecture proposed by Pan et al. [96] the Single Writer, Multiple Reader (SWMR) scheme is used. Each optical network node has a dedicated wavelength to modulate its data on, which is received by all other nodes. All receivers need to be continuously on, making this type of broadcasting quite power consuming. Therefore, some changes are made to the scheme. All receivers are turned off by default. When a node wishes to start transmission, it will make a reservation using a dedicated reservation channel. The reservation message contains which receivers need to be activated and the duration of the message to come. This reduces power consumption. The advantage of an SWMR scheme is that it eliminates arbitration and the resulting latency as all nodes have a dedicated channel to write to, assuming no reservation scheme. In the case of Firefly, the reservation scheme will introduce a small amount of latency. Some

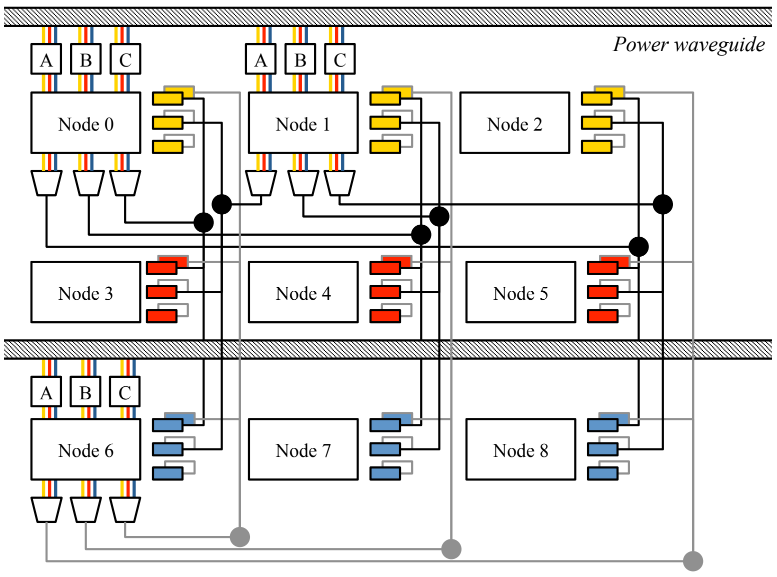


Figure 2.17: The macrochip scheme as proposed by Krishnamoorthy et al. [89] for a 3×3 network. This scheme is a combination of space-division multiplexing across waveguides and optical planes and WDM inside waveguides. Only the Tx structures of node 0, node 1 and 6 are depicted for clarity. WDM demuxing happens in layer-to-layer couplers in the column waveguides.

sort of flow control is still needed to prevent overflowing buffers at the destination. Every node needs to be able to simultaneously receive a message from all other nodes ($N-1$ receivers per node). This scheme therefore results in $N \times (N-1)$ receivers with the associated buffers in total. These receivers not only consume area but increase the optical power needed as well. Figure 2.16 shows a schematic view of a SWMR network.

A similar scheme is employed in the ATAC architecture, proposed by Kurian et al. [106]. The main optical network is in essence the same as Firefly, as it functions as a SWMR network. However, to reduce the number of optical nodes, concentration is used: multiple processor cores share an optical node. Messages coming from the optical network, will be broadcast to all processor cores in such a cluster. The most interesting part about this proposal is the interplay between the design of the coherence protocol and the network design. The ATAC architecture makes use of a limited pointer directory scheme [25, Chapter 8.5.2]. To reduce the number of bits required in each directory entry, only K sharers can accurately be recorded. If a request from the $K+1$th sharer arrives in the directory, the global bit is set for the entry, indicating that from now on, rather than recording which cores share this entry, the total number of sharers is recorded. This results in an increased number of broadcasts, which is an easy operation in this scheme, making this type of directory organisation well suited to the ATAC architecture.

Non-switching: Macrochip

Krishnamoorthy et al. proposed the macrochip [89] which is interconnected using a Pt2Pt network. Routing is done by static WDM. By combining WDM inside one waveguide and space division multiplexing across multiple waveguides, a unique link between any two nodes in the network can be established (Figure 2.17). For example, node 0 can communicate with node 8 by modulating on the blue wavelength, in wave-

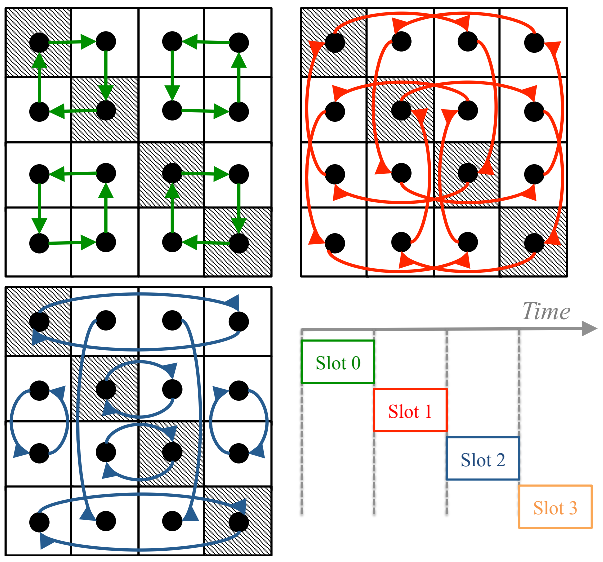


Figure 2.18: Schematic view of TDM arbitrated 4×4 network as proposed by Hendry et al. [107], showing the circuits setup in the first three slots. In total 6 slots will be needed to provide all-to-all connectivity. The hatched squares denote unnecessary connections.

guide A. The waveguide determines the column, the wavelength determines the row. The result is a network that is non-blocking and needs no arbitration. Because of the low number of optical components in a link, the optical losses in a Pt2Pt network are small. The disadvantage lies in the serialisation delay which depends on the number of wavelengths per link and as such might be quite long. Assuming one wavelength per channel, the serialisation latency will be high. When using multiple wavelengths per channel, there will be a rise in the number of components needed. One of the major disadvantages of this scheme lies in the fact that the network does not adapt to the traffic. Regardless of the amount of traffic, only a fixed number of channels is available per source-destination pair, as determined at design time. Other channels might be idle but cannot be used. Another disadvantage is the high component count: the scheme requires $N \times (N-1)$ receivers and $N \times (N-1)$ transmitters.

Non-switching: Time Division Multiplexing

Hendry et al. proposed an optical network [108] and further refined it in follow-up work which completely avoids arbitration all together [107]. All processor cores have their own unique router and the routers are connected using a 2D mesh network . To avoid arbitration, TDM is used. During a time slot, the routers in the network are configured to allow communication between specific source-destination pairs. The duration of such a time slot is determined by the setup time, the transmission time and the worst case propagation latency. After a certain amount of slots (called a time frame), all nodes have been able to communicate. The advantage of this is simplicity. The connections during every slot are determined during the NoC design phase, by a genetic algorithm which optimises the connection pattern as to reduce the length of the time frame. At runtime, all routers just need to follow the clock: knowing which time slot they are in suffices to know which connections to setup. No arbitration is needed. In [107] the number of time slots that make up a time frame is reduced by avoiding to connect all nodes directly during a time frame. Messages will be transmitted along the X dimension first, converted back to the electrical domain and buffered. In the second stage, they

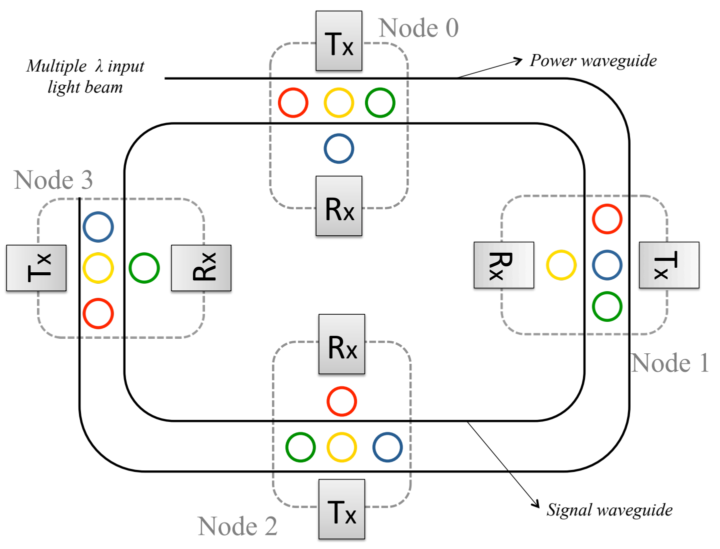


Figure 2.19: Schematic depiction of a MWSR scheme as employed in Corona [109]. Every node has a dedicated wavelength to read from to which all other nodes can write. Arbitration is needed to determine which node can write to a certain wavelength.

will be transmitted optically in the Y direction to the final destination. In this case, the architecture becomes packet-switched. The control matrix of this scheme is shown in Figure 2.18. In this TDM scheme, arbitration is avoided by determining the connections per time slot statically. This results in a very simple scheme which is advantageous in terms of area and power but the network does not adapt itself to the needs of the traffic. If two nodes are communicating heavily, for example, this will be spread out over different time frames. To make the time frames as short as possible, intermediate buffering in the electrical domain is used which is not very power nor latency efficient.

Circuit-switching: Corona

The Corona architecture which was proposed by Vantrease et al. is an example of an all-optical network[109]. The logical network topology is a 64×64 crossbar but it is physically implemented as a serpentine, i.e. the network nodes are laid out in a grid and the waveguide forms a winding ring, passing by all nodes. The network is organised as a MWSR scheme (Figure 2.19). Every node can write onto a given channel[26] but only one node can read from this channel. There will be contention when two nodes wish to write to the same channel so global arbitration is needed. To reduce the arbitration latency, this is done using an optical token message which is constantly circulating on a dedicated wavelength. Whenever a node wishes to write to a channel, it will grab the associated token and release it again when transmission is over. The authors later optimised the arbitration schemes [110]. The advantage of a MWSR scheme as employed in Corona lies in its ability to efficiently handle multicasts and broadcasts as every node can transmit on multiple channels simultaneously. Every node needs to be able to transmit on all channels so $N \times (N-1)$ transmitters are needed where N is the number of nodes. The number of receivers needed is equal to N as every node only reads its own channel.

[26]In this thesis a channel is defined as one or more wavelengths, across one or more waveguides which connect a source and destination at a given moment in time. Channels can be fixed or changed at runtime.

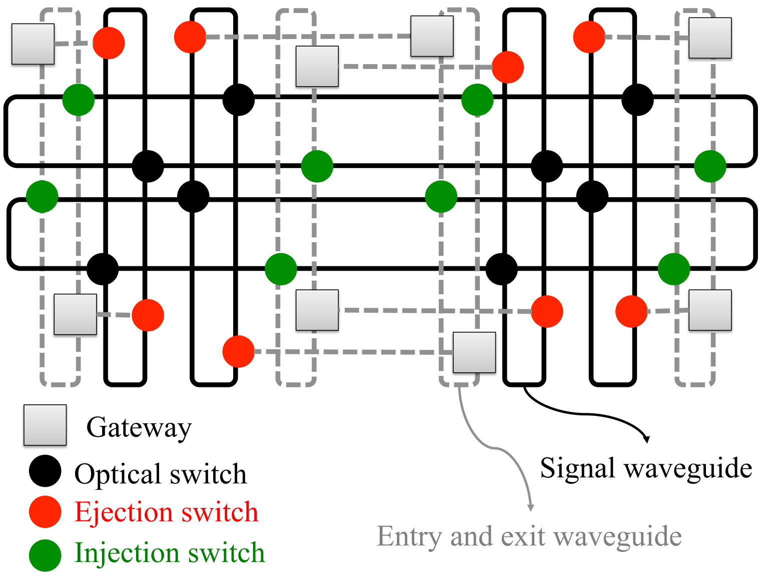


Figure 2.20: Schematic view of the optical torus topology as proposed by Hendry et al. [111]. Path setup is done via an electronic network, superimposed on the network depicted here. The electrical setup message reserves resources on the way and configures all necessary switches (injection, path switches and the ejection switch).

Circuit-switching: FlexiShare

Both the SWMR (Firefly) and MWSR (Corona) scheme suffer from the fact that not all bandwidth available at a node (either to read from or to write to) will be utilised if the network load is unbalanced. In the FlexiShare architecture, proposed by Pan et al. (the same authors who proposed Firefly) a solution is proposed by decoupling the number of channels from the network radix [97]. On the sender side, to gain access to a channel, a MWSR scheme is used. The arbitration is done using an optical token. On the receiver side, a SWMR scheme is used: a channel can be received by all receivers. To minimise power consumption, a reservation scheme similar to that of Firefly is used. The FlexiShare architecture is more efficient in terms of component count than ’pure’ SWMR/MWSR schemes as can be seen in Table 2.7.

Circuit-switching: Optical torus

Hendry et al. proposed a circuit-switched, hybrid network [111]. The overall network consists of an optical data layer and an electrical control layer (Figure 2.20). Both layers have a folded torus topology. The transmission of data on the optical network takes place in three phases. During the first phase, an electronic setup message travels on the electrical layer to obtain the resources needed. When the electrical message reaches the destination, an electrical acknowledgement message will be transmitted back to the source after which the source can start data transmission. The break down phase starts when all data is transmitted and a message travels the network to release the optical resources again. The path setup latency will only be amortised for long messages. Only selected messages will therefore be transmitted on the optical network. Short messages will be transmitted on the electrical control network which has a dual function now. This proposal is not viable for shared memory architectures because the long setup time will not be amortised for the short messages transmitted. In this proposal the arbitration process is decentralised: the electrical setup message still needs to acquire various optical resources along the path. This removes the central bottleneck

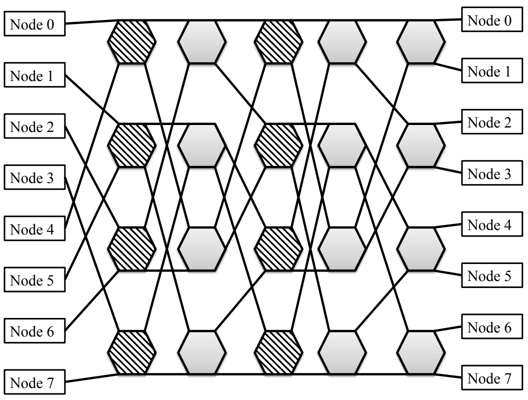


Figure 2.21: The SPINet scheme for an 8×8 network. The grey hexagons denote the switches present in a conventional Omega topology, the hatched hexagons denote the scattering nodes.To reduce the number of messages that are dropped, scattering nodes are added which will be inserted before some switching nodes. When the scattering node detects that two message will contend for the same output port in the switching node, it will deflect one of the message to a neighbouring node [113].

but the problems regarding arbitration latency still persist.

Packet-switching: SPINet

Shacham et al. proposed a time slotted network called SPINet [112]. The network has an Omega topology [27] and consists out of intermediate switches and network nodes (Figure 2.21). To avoid centralised arbitration, speculative transmission is used in combination with distributed arbitration. Messages are transmitted without obtaining an end-to-end path beforehand. All source nodes start transmitting simultaneously. Messages travel the network hop-by-hop, from switch to switch. A dedicated wavelength header contains the message destination and is used at the switches to make routing decisions (distributed arbitration). To prevent buffering of the messages, the message headers must be seen ahead of the actual messages so the switch arbiter reaches a decision before the message arrives. If two messages in a switch contend for the same output port, one of them will be dropped. At low loads this scheme is very effective as there will be very little contention in the network. The authors improve on the traditional Omega network and add deflecting nodes as shown in Figure 2.21. If two messages contend for the same output port in these nodes, one of the messages will be deflected to a neighbouring node. This is also known as deflection routing or hot potato routing. Once a message reaches its destination node, an acknowledgement will be sent back, using the same path. Source nodes that did not receive an acknowledgement at the end of the time slot will retransmit the message in the next slot. At high loads, these retransmissions will have an adverse effect on overall throughput. Silicon area is expensive on-chip so an indirect topology like an Omega network (especially with the deflection nodes to reduce the number of dropped messages) is probably not optimal. Watts et al. compared speculative and scheduled transmission schemes, for chip-to-chip communications and showed the network latency of uniform traffic can be reduced by the use of speculative transmission [45].

[27]The Omega topology is indirect and for N network nodes consists of $\log_2 N$ stages. Every stage holds $\frac{N}{2}$ intermediate 2×2 switches. The intermediate stages are connected in a shuffle pattern [112].

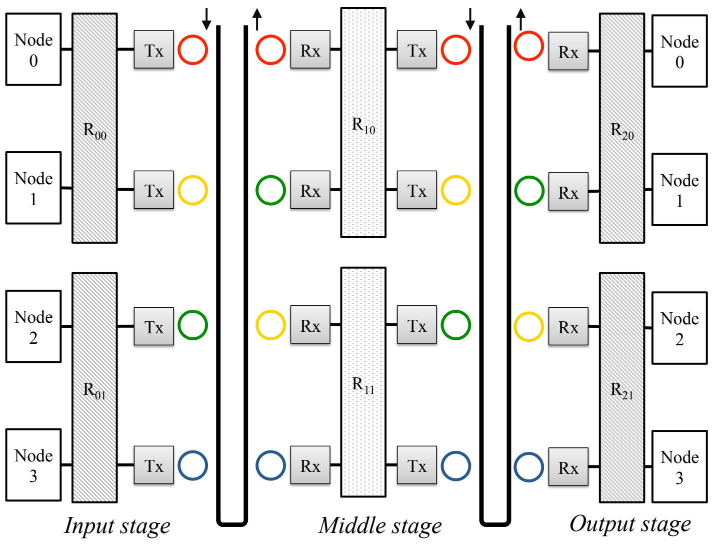


Figure 2.22: An optical Clos network [98]. There is an intermediate O/E/O conversion in the middle stage routers. If node 0 wishes to communicate with node 3, it will use the yellow wavelength to reach the R_{11} (the lower middle stage router). The message will be buffered and then transmitted to R_{21} on the blue wavelength.

Packet-switching: Optical Clos network

A Clos network is a three stage network in which the first and last stage connect the network ingresses and egresses to the middle stage [22, Chapter 3.3]. Joshi et al. proposed a Clos network to decouple the number of transmitters and receivers from the number of processor cores [98]. When the middle stage routers are electrical as depicted in Figure 2.22, this is a packet-switched network. The intermediate O/E/O conversion in the middle stage will be disadvantageous in terms of latency and power consumption.

Packet-switching: Phastlane

The Phastlane network as proposed by Cianchetti et al. is a speculative packet-switched network [114] . The routers are laid out in a mesh and messages are send to the next router without knowing whether or not the necessary resources will be available. The architecture uses dimension-ordering routing [28] making the complete route known in advance. The setup of each router on the path can therefore be encoded and optically transmitted, in parallel to the message, using two separate waveguides. Every router reads out its designated setup bits and assesses the request. There are three possible outcomes. If the requested output port is free, the switch in the router is setup accordingly and the message continues on (without buffering). The path is kept open in case the message is dropped further down the path. If the requested output port is contended and there is a free buffer, the message is converted to the electrical domain and buffered, making this router becomes responsible for the final message delivery. If the requested output port is contended and there is no free buffer, the message is dropped and a Negative Acknowledgement (NACK) is sent back, using the return path. Once the NACK reaches the router where the message was last buffered (either source router or intermediate router), this router will start retransmission. This scheme would

[28]Dimension-ordered routing is a deterministic routing algorithm in which messages first travel the network in one dimension, before they travel in another dimension [22, Chapter 4]. For example. in the case of XY routing, messages travel in the X-dimension until they have reached the X-coordinate of the destination, followed by the same process in the Y-direction.

Name	***Type of switching***	***Arbitration***	***Disadvantages***	***Ref.***
Firefly (SWMR)	No switching	Not needed	Flow control and serialisation	[96]
Macrochip	No switching	Not needed	Flow control and serialisation	[89]
TDM	No switching (packet)	Not needed	(Intermediate O/E/O)	[107]
Corona (MWSR)	Circuit	Optical token	Serialisation	[109]
FlexiShare	Circuit	Optical token		[97]
Optical torus	Circuit	Electrical setup packet		[111]
SPINet	Packet	Electrical by router	Area	[113]
Optical Clos	Packet	Electrical by router	Intermediate O/E/O	[98]
Phastlane	Packet	Electrical by router	Speculative	[114]

Table 2.4: General comparison of the most important academic proposals for optical NoCs

be very cumbersome if the latency of a hop (link and router traversal) would be high as the message would need to buffered until a NACK could return. However, as the architecture name alludes to, multiple hops can be traversed in less than one clock cycle and so a NACK will either not be needed or return in less than a clock cycle.

Conclusion

All the arbitration schemes come with their advantages and disadvantages. Each arbitration approach favours certain types of traffic. Circuit-switched networks might work for large messages (as incurred by message passing CMPs) because the setup latency is amortised by the message size. SWMR schemes allow nodes to address multiple destinations simultaneously. Speculative transmission schemes are more suited for low load traffic.

A short summary of all these optical NoC proposals has been given in Table 2.4. In the case of hybrid or hierarchical networks only details on the optical network are included in this table.

Table 2.4 shows most optical NoC proposals focus on avoiding arbitration all together, at the cost of high serialisation latency. Almost all proposals focus on circuit-switched networks if one sees a non-switched network like Firefly for example as a circuit-switched network in which the circuits are setup at design time. When arbitration does take place for the circuit-switched networks, it is either in the optical domain using tokens (e.g. Corona) or distributed (e.g. electrical setup packet in the optical torus).

2.2.4 Holistic proposals for network on-chip optimisation

There are various ways of improving the performance of the NoC, be it electrical or optical. It is possible to improve the latency or power-efficiency of individual components in the network. For example, the router latency can be targeted by reducing the number of stages [115] or improve the energy-efficiency of a link by applying Dynamic Voltage and Frequency Scaling (DVFS)[29] In this section, the use of holistic methods will be discussed. These are methods that draw on knowledge of different aspects of the NoC to improve the overall NoC performance. Such holistic methods could be used to reduce the impact of arbitration on the circuit-switched optical NoC being proposed in this thesis. One example of a holistic method is the co-design of the NoC and coherence protocol proposed by Jerger et al. [116]. Das et al. proposed the concept of slack (a metric indicating how the delay of a message would affect the performance of the application) to prioritise messages during the arbitration stage [117]. The methods discussed below are subdivided into predictive methods and methods that employ knowledge of the coherence protocol. This boundary, however, is not very strict as there are predictive methods that use coherence knowledge.

Use of prediction in the network on-chip

Prediction can be used to improve the performance of the NoC.

Adi et al. use prediction to reduce the path setup latency of a hybrid NoC [118] . The data plane is an optical, circuit-switched torus network. The optical routers are configured by an electrical mesh network. To set up an optical path, a packet travels the electrical control plane, acquiring resources along the way. The path setup latency is reduced by using channel prediction in the electrical routers (based upon the work by Matsutani et al. [115]) in combination with lookahead routing. A predictive structure in each input channel will predict the output channel and send a speculative request to crossbar arbiter in the router.

Ogras et al. achieve flow control by predicting congestion in the network and using these predictions to control the injection rate at the network ingresses [119].

DVFS[29] is a method used to reduce the power consumption of a NoC . Hesse et al. use predictions of future communication to steer DVFS [120]. These predictions are made by collecting communication sets in every node (i.e. nodes with which this node will communicate), based upon information in the coherence controller. At the end of every epoch, this information is sent out to all the receiving nodes. The nodes aggregate all the incoming information and make a DVFS decision for the upcoming epoch.

Huang et al. apply a similar principle: traffic is predicted in the TILE64 CMP and then used to calculate future link utilisation which is used to control the DVFS unit [121].

Wen et al. propose to use reuse distance to setup circuits in optical networks [122]. Reuse distance is a concept borrowed from cache optimisation and the authors adapt to capture the number of different circuit requests made in between two uses of the same circuit. This information can then be used to setup circuits before they are requested.

[29]DVFS adapts the frequency and voltage of a voltage/frequency island to the requirements of the overall system e.g., reduce voltage/frequency when the island is idle, increase voltage/frequency to increase performance. An island can be one or more routers, one or more processor cores, a complete system etc. [16, Chapter 5.2.3.2].

Use of knowledge about the coherence protocol in the network on-chip

There are various ways of avoiding or reducing the latency of memory requests by predicting certain memory transactions. Whilst these do not change the NoC as such, they are mentioned in this section as they do improve performance by streamlining communication.

Acacio et al. alleviated the cost of cache-to-cache transfers by using prediction [123]. The need for cache-to-cache transfers is predicted based upon the program counter, while caches holding copies of the requested data are predicted using both the program counter and the requested memory address. Kaxiras et al. used prediction to forward memory addresses to future readers, thus avoiding L1 misses and sending their associated messages to the directory [124]. Martin et al. proposed a cache coherence protocol in which other caches sharing a cache line are predicted [125]. Coherence control messages can then be forwarded to this predicted list of sharers, to avoid indirection via the directory which makes this protocol a hybrid between a snooping and a directory protocol. While these proposals decrease the latency of memory requests by avoiding unnecessary network transactions, they do not speed up the messages that still need to travel the NoC.

Abousamra et al. proposed a path setup scheme proposed which makes use of very general knowledge of the coherence protocol [126]. As soon as a cache hit is detected in the LLC, a circuit reservation is sent to the NoC. This ensures the circuit is ready as soon as the message arrives at the network ingress. This scheme relies on the latency of reading out a cache line and preparing the message.

Demir et al. proposed prediction to reduce the laser power consumption in an optical NoC [127]. The communication fabric is subdivided into a part carrying the control information in a message (e.g. the memory address) and the part carrying the actual data. The data slice is predictively activated, only for messages which will carry data. Control messages which do not carry a cache line, will not need the data slice. Multicast messages and the subsequent many-to-one messages are quite frequent in most directory coherence protocols (for example, an invalidation request to all sharers, which will then acknowledge the invalidation to the original requester). However, a mesh network is very ineffective at handling these. Ma et al. proposed a logical multicast tree in which multicast messages fan out, like in a tree [128]. When the many-to-one messages return, they recombine, again in tree-like fashion so ideally only one message arrives at the original requester, carrying the acknowledgements of all sharers.

There is a lot to be said for a black-box approach, in which the NoC would be treated as an independent unit by itself which does not influence the design of the coherence protocol, caches etc. and in its turn, will not be be influenced by these things. However, in this thesis, I wish to argue for a more holistic approach in which the NoC design is guided by the overall memory hierarchy design. The proposals discussed in this section are an example of this.

2.3 System Under Study

This section will discuss the system architecture and parameters, as used in this thesis. If any parameters of the system would be different than listed in this section, it will be explicitly mentioned. The first subsection will discuss the system architecture, which remains constant throughout the thesis. The second subsection will discuss the

NoCs used in this thesis. The workloads and simulator used to compare the different architectures will be discussed in Section 3.2.1 and Section 3.2.2 respectively.

2.3.1 System architecture

Core architecture

In this thesis, I assume a CMP with 16 in-order Intel X86 cores. The choice for X86 cores is motivated by their dominance in the desktop and server market [129], which is also the segment where a change to optical NoCs would make sense [78]. From a purely practical point of view, it is one the ISAs used in gem5 that can be used in conjunction with Ruby, which is needed to accurately simulate the cache hierarchy and coherence protocol (see Section 3.2.2). The number of cores in the architecture was a trade-off between the number of cores of high performance CMPs and a realistic simulation time, where 16 cores was chosen as a good balance. The CPUs execute instructions in-order. Out-of-order cores, in which instructions can be executed in a different order than the program order [1, Chapter 4.10], partially hide the latency of memory requests as another instruction can be executed whilst waiting for the memory request to be completed. In this work in-order cores are used as they ease reasoning about changes in the NoC. However, in a realistic system, out-of-order core would be used which presumably would gain less (in terms of computational performance) from a change in the network architecture as a cache miss affects performance to a lesser degree than in-order cores.

Cache and memory architecture

Figure 2.23 gives a schematic depiction of the cache and memory hierarchy. Each CPU has a private L1 cache. The L2 and directory are logically shared, but physically distributed into banks to prevent the formation of a bottleneck. The mapping of the cache lines onto the 16 L2 and directory banks is static and based upon the cache line address. Cache line interleaving is used to prevent an uneven distribution of requests to the slices. In this thesis set-interleaving is used, where cache lines belonging to consecutive sets are mapped onto different banks [52, Chapter 2]. This is achieved by using the address bits right after the block offset to calculate the L2 and directory slice the address is supposed to map onto. This technique will be discussed in more detail in Section 5.2. The choice for a two level cache hierarchy was motivated in part by the restrictions in terms of the coherence protocols available in gem5 which were compatible with the setup used in this thesis. However, some existing CMPs, like the Intel Single-Chip Cloud Computer (Section 2.1.5), also have a two level hierarchy so this is not an unrealistic choice. Multiple works in the literature assume a memory interface per tile (e.g. Corona [109]) as recent advances in technology such as 3D stacking make this possible. Using a memory interface per tile also avoids any effects of suboptimal memory interface placement as investigated by Abts et al. [130]. This cache and memory architecture will always be the same, regardless of the NoC.

[30]The DRAM latencies assumed in this work are the default values as assumed in gem5. The DRAM controllers are based upon work by Hansson et al. [150]. The frontend latency represents the latency incurred by the pipeline stages of the DRAM controller, the backend latency is based on the DRAM

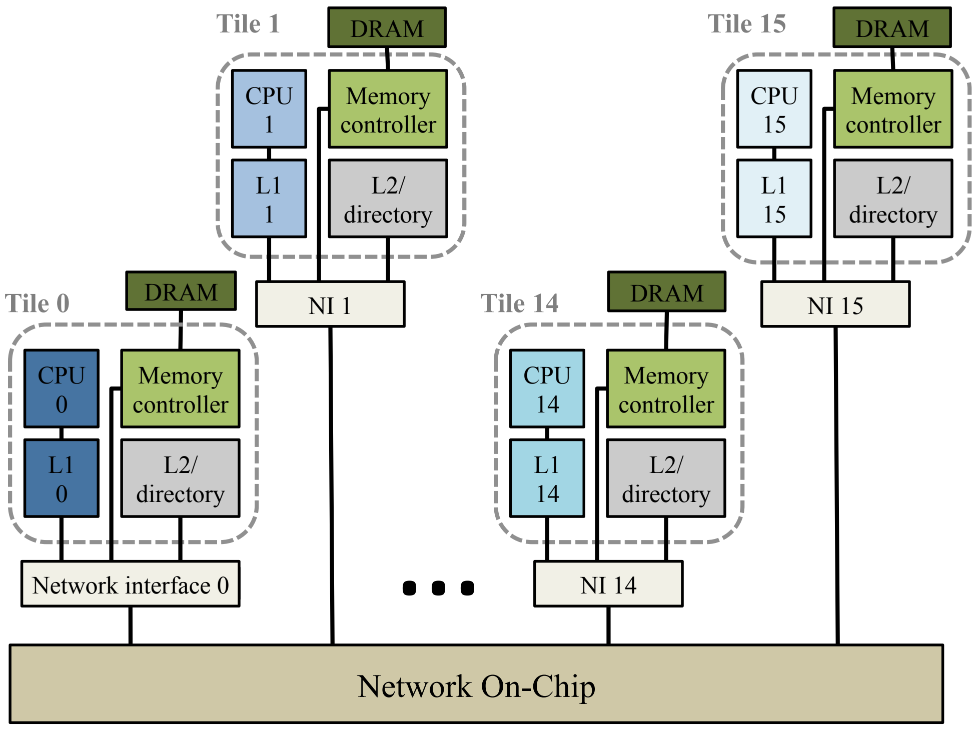


Figure 2.23: Cache and memory architecture under study. The L1 caches are private, all other parts of the cache hierarchy are physically distributed but, logically shared.

2.3.2 Network on-chip architecture

The simulator gem5 used in this thesis, uses the concept of virtual networks. These networks are used to separate the different message types to prevent protocol-level deadlock from occurring[31]. Each virtual network has its own set of physical resources, to prevent the formation of a blocking dependency between the various message classes. The exact classification of message class to virtual network, depends on the coherence protocol. In the coherence protocol used in this work, there are 3 virtual networks. The first network carries request messages, the second network transmits responses and the third networks is responsible for unblock messages. The networks that already existed in gem5 use these virtual networks. The optical networks I implemented, only have virtual networks and their associated separate resources (such as buffers) when entering and leaving the network. The optical NoCs itself only has a single virtual network layer, routing all message types.

Electrical mesh networks

The most commonly used topology for large NoCs is the mesh network. Table 2.6 contains the parameters assumed in this thesis. These will be discussed in more detail in Section 4.2.1.

interconnect latency.

[31]Protocol-level deadlock takes place when there is cyclic dependence between messages from different message classes that are vying for the same physical resources in the network.[22, Chapter 2.3.1]

CPU	16, in-order X86 cores
Clock frequency	2 GHz
L1-I cache organisation	Private, 4-way associative, 32 kB
L1-I controller latency	2 clock cycles to CPU / 1 clock cycle to L2
L1-I cache latency	2 clock cycles
L1-D cache organisation	Private, 4-way associative, 32 kB
L1-D controller latency	2 clock cycles to CPU / 1 clock cycle to L2
L1-D cache latency	2 clock cycles
L2 cache organisation	Distributed shared, 32-way associative, 16 $\times$ 256 kB
L2 controller latency	2 clock cycles/ 1 clock cycle to L1
L2 cache latency	15 clock cycles
Coherence protocol	Directory MESI Two Level
Number of memory controllers	16
Memory controller latency	6 clock cycles
DRAM latency [30]	10 ns (frontend/backend latency)

Table 2.5: Parameters of the system architecture i.e. the core architecture and memory hierarchy. These parameters remain constant, while the NoC parameters vary.

Network frequency	2 GHz
Number of nodes	16
Topology	Mesh
Number of rows in mesh	4
Terminal nodes per network interface	3 (tiled architecture)
Routing algoritm	Deterministic, dimension ordering
Link latency	1 clock cycle
Link bandwidth	8 B
Router pipeline stages	5
Buffer size	1 (control) / 4 (data)
Virtual channels	4 per virtual network

Table 2.6: Parameters used for the electrical network with which the optical proposals will be compared.

Optical networks

The existing optical NoC proposals focus mostly on reducing arbitration latency by avoiding switching and, hence, removing the need for arbitration, resulting in rigid schemes. The NoCs that do need arbitration such as MWSR aim at reducing the associated latency by doing arbitration in the optical domain (for example with optical tokens in the case of Corona [110]). In this thesis, I propose the use of a circuit-switched optical NoC with centralised arbitration (Figure 2.24) in which the arbitration

Scheme	***# Tx***	***# Rx***	***Total Tx/Rx***	***# Switching elements***
MWSR	$N \times [(N-1)m]$	$N \times m$	$N^2 \times m$	0
SWMR	$N \times m$	$N \times [(N-1)m]$	$N^2 \times m$	0
Pt2Pt	$N \times [(N-1)m]$	$N \times [(N-1)m]$	$2 \times m \times N \times (N-1)$	0
Crossbar	$N \times m$	$N \times m$	$2N \times m$	$N \times [(N-1)m]$

Table 2.7: Number of transmitters and receivers per network scheme, where N is the number of network nodes and m the number of channels

latency will be hidden by the use of holistic message prediction methods (these will be discussed in more detail in Section 2.2.4).

The central circuit-switch would be an optical crossbar. In the remainder of this thesis, both the term circuit-switch and crossbar will be used. The term optical crossbar can be problematic as in a lot of publications the NoC is referred to as an optical crossbar while in reality, it is logically a crossbar but not physically (e.g. FlexiShare [97]). Optical crossbars can be constructed using Mach-Zehnder interferometer switches [131] or microring resonators [132]. In this work, it is assumed the crossbar is constructed using microring resonators, organised as a matrix switch, as can been seen in Figure 2.24. The allocator used is electrical. In the system as is, communication to and from the allocator is assumed to be in the electrical domain. However, the communication medium to and from the allocator does not affect the concept as proposed here.

The main advantage of such a system is the simplicity. Almost all proposals discussed previously use multiple logic network layers, increasing the complexity of the flow control system for example. Flow control in a system with a centralised allocator is easier: as the allocator organises all communication it knows when all buffers of a receiving node will be taken. The simplicity of this system can also been found in the low number of transmitter and receiver units, compared to the two main schemes. As Table 2.7 shows both the MWSR and SWMR show a quadratic dependency on the number of nodes in the network, whereas for the crossbar scheme, this is linear. However, Table 2.7 also shows the crossbar scheme needs switching elements compared to the other schemes. However, the exact cost of these switching elements (both in area and power consumption) will depend on the exact physical implementation of the crossbar and as such, outside of the scope of this work. Another advantage is the efficient use of the channels available. When node A wishes to communicate with node B, it can use all the channels at its disposal. This is in the contrast to a MWSR scheme, where node A can only use the channel read by node B, even if all other channels are unused.

The disadvantage of the crossbar scheme as shown in Figure 2.24 is arbitration latency. Especially for short messages 8 B, the arbitration overhead will be significant. The aim of this thesis is therefore, to investigate whether it is possible to implement a system that is able to reduce the arbitration latency by means of prediction in a circuit-switch scheme. Is it possible for a simple NoC like a crossbar with arbitration to outperform an non-arbitrated scheme like SWMR?

	Switched NoC	*Non-switched NoC*
Network frequency	2 GHz	2 GHz
Number of nodes	16	16
Topology	Crossbar	SWMR
Terminal nodes per network interface	3 (tiled architecture)	3 (tiled architecture)
Modulation frequency	25 Gbit/s	25 Gbit/s
Allocation latency	2 clock cycles	Not applicable
Electrical signalling latency	1 clock cycle	Not applicable
Time of flight	1 clock cycle	1 clock cycle
WDM wavelengths available	1-32	1-32

Table 2.8: Parameters used for the optical networks in this thesis.

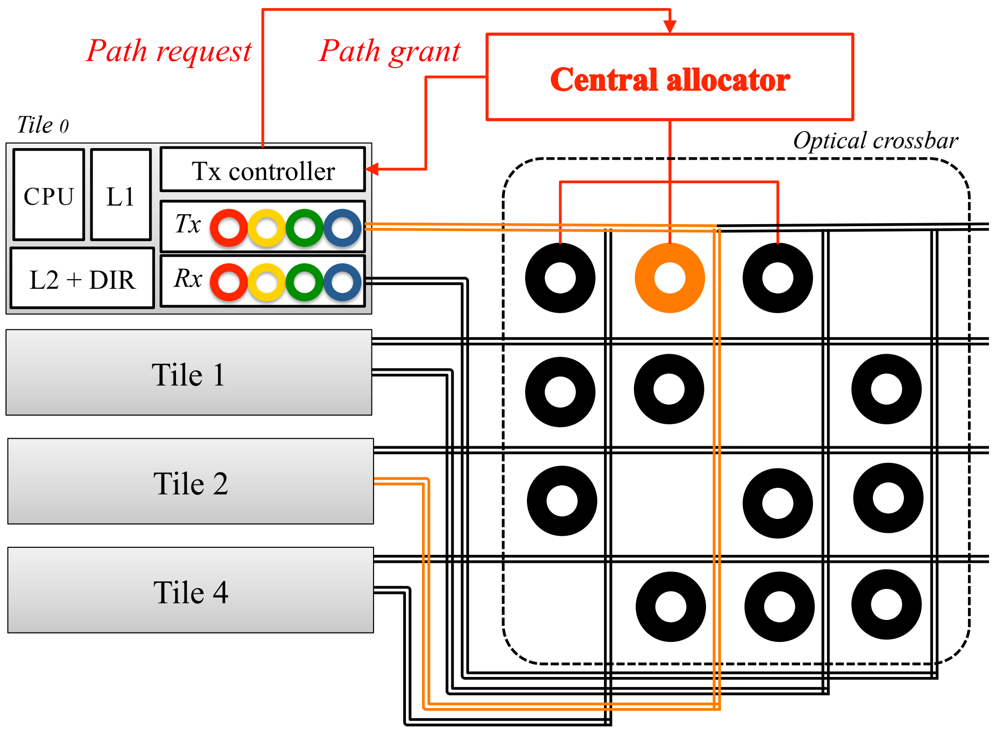


Figure 2.24: Diagram of the type of circuit-switched optical NoC used in this thesis, for a CMP with four tiles. Every tile has a network interface, consisting of a transmitter, receiver and transmitter controller. The Tx controller requests an optical path by means of an (electrically transmitted) path request to the central allocator. Once the requested path has been setup, a path grant is sent back to the Tx controller which then starts transmission. In this figure, tile 0 is transmitting to tile 2. The microring resonators are controlled by the central allocator. For clarity, only the connection to the first row of microring resonators is shown.

2.4 Conclusion

This chapter set the scene for the remainder of this thesis. The system used is a shared distributed memory CMP, where the view of the memory is kept coherent by means of directory protocol. As a result, the messages on-chip will be short (8 B to 72 B) and latency sensitive. Therefore, the NoC needs to be able to provide low-latency interconnect with very little overhead. Optical interconnect has been proposed, because of the limitations of electrical interconnect. The question is two-fold: first, this thesis wishes to investigate whether the use of an optical NoC (be it a a crossbar as pictured in Figure 2.24 or a SWMR as pictured in Figure 2.16) and defined in Table 2.8 will provide better performance than an electrical NoC (Table 2.6). Secondly, the optical crossbar we propose has an inherent arbitration latency and, hence, overhead. I wish to investigate whether this optical crossbar can outperform a non-switched optical NoC, when applying holistic methods. Both questions involve quantifying an architecture. In the next chapter possible figures of merit and measurement methods will be discussed.

3 Figure of Merit

THE two main questions in this thesis revolve around the concept of *performance*: can an optical Network On-Chip (NoC) outperform an electrical NoC? Can a circuit-switched optical NoC with centralised arbitration outperform a non-switched optical NoC? Evaluating and comparing different computer architectures is not a trivial task, however. Setting up a good experiment consists of three steps [133]. In the first step, one should decide upon the metric to be used in the comparison, the figure of merit. The second step decides on how to obtain this figure of merit, by means of an analytical model or a simulation. The third step in the setup process is the decision for a workload.

In Section 3.1 the choice for a figure of merit will be discussed. Subsequently, Section 3.2 will discuss the measurement of this figure of merit in more detail: Section 3.2.1 discusses the workloads to be used, followed by a discussion on the simulator to be used in Section 3.2.2. Afterwards Section 3.2.3 will justify the use of full-system simulators, compared to analytical modelling.

3.1 Performance Evaluation

3.1.1 Performance evaluation of single-threaded workloads

The performance of an architecture running using a single-threaded workload can be characterised using a single metric: the total runtime T [133].

$$T = I \times CPI \times \frac{1}{f} \tag{3.1}$$

Equation (3.1) [134] shows how the total runtime T can be related to I, the total number of useful instructions, Cycles per Instruction (CPI) and the clock frequency f. I captures the total amount of useful work and therefore excludes any instructions caused by branch mispredictions etc. The amount of work I is determined by the interplay between the Instruction Set Architecture (ISA) and the compiler. The second factor, CPI, depends on the memory system, the micro-architecture and the implementation of that architecture. The last factor, f, depends on the system-implementation and the technology node. This equation is also known as the Iron Law of Performance as it relates ISA, micro-architecture and technology node [135, Chapter 1.3.1]. Assuming the amount of work done I and the frequency f are kept constant, CPI is a good metric for the comparison between different architectures. The inverse metric, Instructions per Cycle (IPC) is often used as well as it is quite intuitive. The higher the IPC, the better the design performs.

Speedup is a metric to compare two architectures, using IPC. The speedup s is defined as $\frac{IPC_A}{IPC_B}$ and indicates the performance of architecture A is $s\times$ that of architecture B.

3.1.2 Performance evaluation of multithreaded workloads

Whilst IPC is a good metric for single-threaded application, IPC by itself is not suitable for multithreaded workloads [133] and can lead to wrong conclusions as discussed by Alameldeen et al. [136], [137]. The interplay between the various threads in multithreaded workloads can lead to different execution paths being taken in different simulations (of the same system!). This effect is also known as space variability [136]: small differences in timing might lead to different scheduling decisions by the thread scheduler and, hence, lead to different execution paths. This effect also exists in real applications [136] but the different scheduling decisions level out over the global runtime of long programs. However, because of the short runtime of benchmarks, as noted in Table 3.1, a small variation in execution path will have a non-negligible effect on IPC.

I have visualised this effect as well in the system setup used in this thesis (see Table 2.5 for more details on the memory hierarchy). In these simulations, the NoC is a Point-to-Point (Pt2Pt) network, connecting all parts of the memory hierarchy directly (i.e. instead of providing one network node per tile, every First Level Cache (L1), Second Level Cache (L2) slice and memory controller had an individual network node) to make the NoC as ideal as possible. Figure 3.1a shows the thread scheduling across the processor cores for ***canneal***. The y-axis denotes time, measured in epochs, the x-axis denotes the processor core number. A blue square indicates the thread on that processor core was suspended during an epoch, whilst a yellow square indicates a running thread. The figure shows CPU-5 is idle for a significant amount of time. CPU-14 on the other hand is, compared to the other processor cores, very active. When a small perturbation was added to the simulation i.e. the latency of a link was varied randomly at runtime[1], the thread scheduling differs significantly as shown in Figure 3.1b. The behaviour across processor cores is much more uniform in Figure 3.1b compared to Figure 3.1a. Both simulations were run to completion. However, the overall runtime differed as the number of epochs per run differs, as can be seen in Figure 3.1. It seems in the

[1]See Section 3.2.2 for more details on the exact implementation

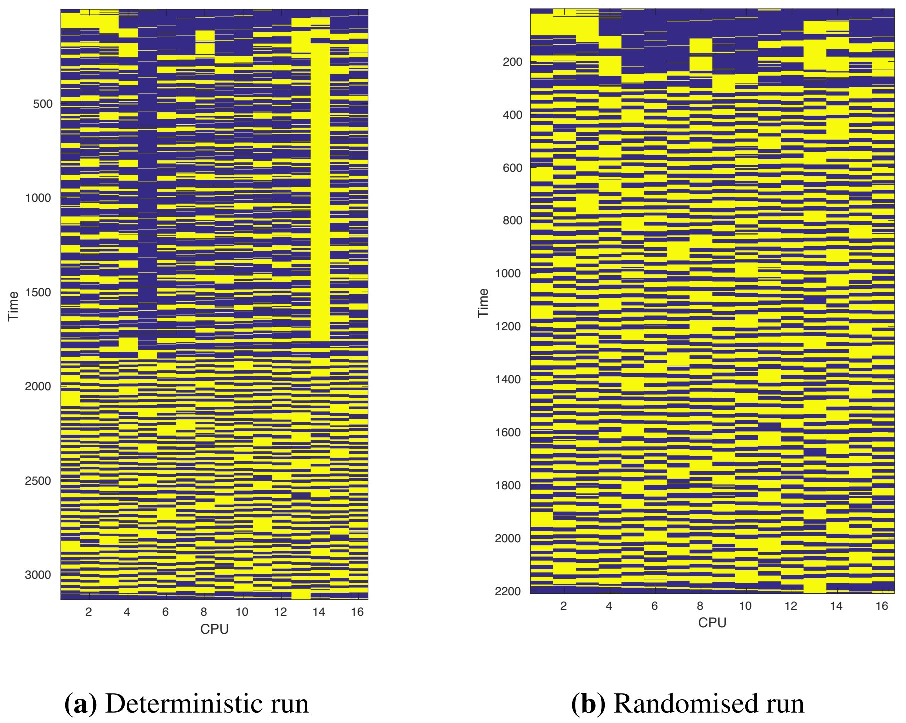


Figure 3.1: Variability in thread scheduling for ***canneal*** when running on a CMP with a Pt2Pt network. Blue squares denote a suspended thread, yellow squares denote a running thread (a) Deterministic run (b) Randomised run – the link latency is increased/decreased at random

deterministic case CPU-14 is not releasing a lock for a significant amount of time, thereby blocking CPU-5.

This example shows that only using a single data point per architecture evaluation can lead to wrong conclusions as that one data point is the combination of both the architecture and the execution path taken in that particular simulation. It is also important to note that the amount of space variability depends on the workload as well [136]. Therefore it is necessary to take multiple data points per architecture evaluation. Alameldeen at al. propose the use of confidence intervals to cope with space variability [136]. A confidence interval indicates the range of values that are expected to hold the true population parameter (Equation (3.2)). Confidence intervals are useful for two reasons: first of all, they indicate the spread between the data points obtained. Secondly, if the experiment is aimed at comparing two architectures and their confidence intervals overlap, it is very likely a wrong conclusion will be drawn. The values can lie anywhere within the interval so also within the overlapping region, making it impossible to conclusively compare both values. The confidence interval of the mean (assuming a normal distribution) is given by Equation (3.2) [136]

$$\text{confidence interval} = \overline{\mu} \pm \frac{t\overline{\sigma}}{\sqrt{N}} \tag{3.2}$$

In this equation $\overline{\mu}$ and $\overline{\sigma}$ stand for the sampled mean and standard deviation respectively, t stands for the normal deviate determined by the desired confidence interval and N stands for the sample size[2]. The value of t can be found in statistical tables. Increasing

[2]N has previously been used in this thesis to indicate the number of core per CMP but in this chapter

N will decrease the confidence interval. Equation (3.3) [136] gives the sample size needed to obtain the desired confidence interval.

$$N_{ideal} = \left(\frac{t\sigma}{r\mu}\right)^2 \tag{3.3}$$

In this equation, μ and σ stand for the population mean and standard deviation respectively and r for the desired relative error of the sampled mean to the population mean. As the population mean and standard deviation are unknown, their sampled counterparts can be used.

3.1.3 Conclusion

This thesis, where possible, will aim at obtaining the mean with a 95% confidence interval, with relative error between the true mean and experimental mean of 5%. However, this can be computationally very expensive as every data point equates to several hours of simulation time so it might be unmanageable to obtain N_{ideal}. In any case, error bars will be added to indicate the space variability of the experiment.

In this thesis computational performance, measured as IPC, is used to compare various architectures. However, it should not be forgotten there are other metrics needed to provide a fully rounded comparison. Architectures should be tested for their reliability, power consumption, energy-efficiency, design complexity etc. However, in the limited scope of this thesis when comparing network organisations computational performance is the metric of choice. When optimising an architecture, certain aspects affecting design complexity will be taken into account as well.

3.2 Performance Measurement

The previous section discussed the figure of merit to be used in the evaluation of multithreaded architectures and decided upon the use of computational performance in this thesis. This section will discuss the next steps in an experiment: which workloads should be used to measure performance? How can this the performance be measured i.e. via the use of a simulator or by means of analytical evaluation? The final section will argue for the use of full-system simulations rather than analytical modelling.

3.2.1 Choice of workloads

As mentioned previously, one of the steps in the setup of an experiment is the choice of the workloads to be used. The set of workloads to be used should satisfy two requirements. First of all, they should give a good representation of the applications which would be running on the architecture under investigation. Secondly, the various benchmarks in the suite should show a variety of computation oriented workloads and more communication oriented workloads, traffic patterns, data sharing etc. This section will first discuss all benchmarks in the PARSEC benchmark suite. Subsequently the

the sample size will also be indicated by N as the letter N is often used in a statistical context to indicate the number of samples whole suite will be reviewed and behaviour such as the ratio of communication to computation and data sharing will be quantified. These results are achieved using the system architecture described in Section 2.3, using the simulator gem5 which will be discussed in more detail in Section 3.2.2.

PARSEC benchmark suite

In this thesis the PARSEC benchmark suite is used [138]. PARSEC aims to include multithreaded workloads, based on emerging workloads without a focus on high-performance computing.

Table 3.1 gives an overview of the benchmarks present in the PARSEC suite. As can be seen in the table, 8 out of the 13 benchmarks function correctly in the simulator. The exact simulator will be discussed in Section 3.2.2. The remaining 5 benchmarks show problematic behaviour such as segfaults in the simulated workload etc. There are 6 different types of input sets for the benchmarks of which only ***simsmall, simmedium*** and ***simlarge*** are of interest as they can be used for micro-architectural exploration. In this thesis the ***simmedium*** input set is used.

blackscholes

This is a financial application used to analytically calculate the price of a portfolio of stock options. It does so by evaluating the Black-Scholes partial differential equation. The threading model is data-parallel meaning the work is divided into chunks whereby each chunk is assigned to a thread. While ***blackscholes*** shows quite a lot of sharing, the data is only shared between two threads [138]. At the start of the parallel phase of the benchmark, one thread divides the work among other threads. Therefore the data representing the stock portfolio is shared between the worker threads and the original thread but there is no communication between the worker threads. As a result, there are very few communicating reads and writes [139][3].

canneal

The ***canneal*** benchmark is based on the simulated annealing method, which is used for optimisation problems in which there is a very large discrete configuration space and the desired global extremum can be hidden among local extrema [140, Chapter 10.9] [4]. In the case of ***canneal***, the annealing method is used to minimise the routing distances of a digital design during the place-and-route phase of a chip design. It has a very large working set and will therefore use the caches very ineffectively. This means that even though the threads communicate heavily via shared data, this is not visible when tracking the number of sharers per cache line in the Last Level Cache (LLC) as the chance that more than one thread will access the data before it is evicted to make space for another cache line is negligible [138]. It has the second highest amount of communication with main memory, following ***streamcluster***. ***canneal*** uses a data-parallel thread organisation. Compared with the other benchmarks in the suite ***canneal*** has the most sharers per communicating write. This might have an impact on the coherence traffic.

[3]A communicating write is the writing of a value which will be read by another processor core than the writer. A communicating read occurs when the value read was produced by another processor core and the value has never been read before. [139]

[4]In essence, the simulated annealing method calculates the cost of various configurations and moves towards configurations with a lower cost. However, at random times, the cost is allowed to increase to escape a local minimum.

Name	***Application domain***	***Correct function in gem5***	***Runtime in gem5***	***Operational intensity in $\frac{instr}{byte}$***	***LD to ST ratio in %***
blackscholes	Financial analysis	✓	0.243 s (σ = 0.01 s)	$\frac{27}{26}$	88.57 ($\sigma \approx 0$)
bodytrack	Computer vision				
canneal	Engineering	✓	0.271 s (σ = 0.023 s)	$\frac{1}{59}$	73.20 ($\sigma \approx 0$)
dedup	Enterprise storage	✓	1.813 s (σ = 0.137 s)	$\frac{5}{78}$	71.08 (σ = 0.06)
facesim	Animation				
ferret	Similarity Search				
fluidanimate	Animation	✓	1.047 s (σ = 0.027 s)	$\frac{23}{80}$	79.30 (σ = 0.03)
freqmine	Data mining	✓	33.347 s (σ = 0.506 s)	$\frac{1}{5}$	80.33 (σ = 0.18)
raytrace	Rendering	✓	47.687 s (σ = 0.886 s)	$\frac{77}{94}$	95.10 (σ = 0.01)
streamcluster	Data mining	✓	1.981 s (σ = 0.127 s)	$\frac{7}{96}$	81.79 (σ = 0.01)
swaptions	Financial analysis				
vips	Media processing				
x264	Media processing	✓	0.717 s (σ = 0.038 s)	$\frac{1}{32}$	95.93 (σ = 0.02)

Table 3.1: Workloads in the PARSEC benchmark suite

dedup

The ***dedup*** benchmark compresses data by a process called deduplication: it detects redundancy in a data stream and removes these blocks. The processed stream is then compressed. Deduplication is a data compression technique used by cloud storage applications for example. The threads in ***dedup*** are organised in a pipeline, there are 5 stages in the application and each stage has a pool of threads associated. The stages (and therefore the associated threads) pass on information. The three middle stages of the application run in parallel which means that at a given moment in time threads with completely different characteristics and behaviour can be running. The pipeline-threading model makes that this benchmark exhibits a lot of sharing.

fluidanimate
fluidanimate is used to simulate non-compressible fluids using the Navier-Stokes equation for use in interactive animations such as computer games. ***fluidanimate*** has the highest parallelisation overhead of all PARSEC benchmarks. Its working set is relatively large and exhibits streaming behaviour. The threads are organised in a data-parallel thread manner. There is very little data sharing among the threads but quite a lot of communication, possibly due to the high number of locks present [5].

freqmine
The ***freqmine*** benchmark identifies the most frequent patterns in a database and represents data mining applications. It has a very large input set and the threads work according to the data-parallel model. It has an average amount of sharing.

raytrace
In the second version of the PARSEC benchmark suite ***raytrace*** was added to include a rendering benchmark in the suite [141]. ***raytrace*** creates a realistic image by reversing the path of the light in the scene. The working sets are large as they hold the complete scene. The threads are working in parallel resulting a lot of data sharing (the scene is shared) but relatively little actual data exchange.

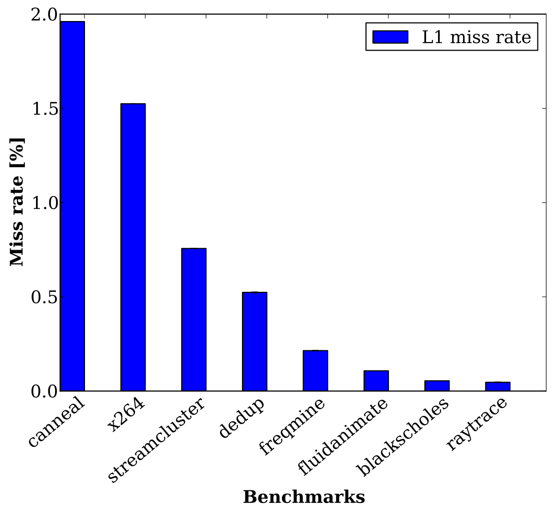
streamcluster
streamcluster is another data mining benchmark which calculates the optimal clustering of a given set of multidimensional data points. The data points are streamed as many clustering applications need real-time responses. The data points are streamed, which explains the high amount of communication with main memory in ***streamcluster***. ***streamcluster*** has a high amount of read-only sharing: one processor core writes the data and various other processor cores read it. The ***streamcluster*** benchmark also exhibits a lot of false sharing: processor cores share a cache line, but not the exact memory addresses [142].

x264
The final benchmark used is ***x264*** which is a video encoder based upon the frequently used ITU-T H.264 standard. The threads are organised in a pipeline-like manner: every input frame gets assigned to a thread. Each frame is categorised as an I, P or B frame. I-frames are independent and encoded on themselves, without information from other frames. P-frames contain the parts of the frame which have changed since the previous I or P-frame. B-frames use the same concept but are encoded using the next and the previous I or P-frame. This organisation ensures there is a lot of inter-thread data exchange, resulting in a large number of reads and writes to shared data locations. The number of threads involved is quite low though as each thread only needs to exchange information with a couple of reference frames. This makes ***x264*** quite communication intensive and therefore interesting from a NoC point of view.

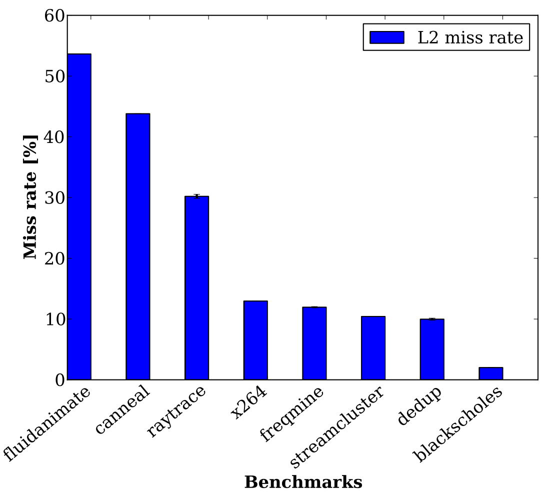
Assessment of the PARSEC benchmark suite

A set of benchmarks used in architectural exploration, as done in this thesis, should show varied behaviour. For example, if all the benchmarks exhibit the same traffic

[5] To ensure the correct execution of the workload, it is important that threads execute their instructions in the correct order. This can be enforced by protecting parts of the code with locks for example. Before a thread can access a commonly shared variable (e.g. a shared counter), it needs to gain access to the lock. While the thread holds the lock, no other thread can grab the lock and therefore access the protected variable. Once the thread has finished working on the variable, it releases the lock, so other threads can try grabbing it [20].



(a)



(b)

Figure 3.2: Cache miss rates of the various PARSEC benchmarks (a) L1 miss rate (combined L1-D and L1-D rates) (b) L2 miss rate

pattern, a network architecture that plays into this exact pattern might be very positively judged, even though the architecture might not work for other traffic patterns. In this section, multiple characteristics which were mentioned in the discussion of the PARSEC benchmark are quantified. The reason for this is twofold: first of all, it allows to check for variation across the benchmarks. Secondly, the effect of every characteristic on the NoC will be discussed. The measurement is done for an architecture as described in **??**, when using an ideal NoC[6]. These characteristics are summarised in Table 3.1.

Cache miss rates

The cache miss rate is defined as the ratio between all the requests a cache receives and those requests it cannot immediately satisfy (misses). It is important to note that a miss does not necessarily mean the cache line is not present in the cache, as it could be present but not in the correct coherence state. Figure 3.2a shows the L1 miss rates of the various PARSEC benchmarks. Benchmarks with a higher L1 miss rate (like ***canneal*** and ***x264***) can, therefore, stress the network more as the L1 needs to forward the misses to their respective directories, via the NoC. Of course miss rate is a relative number and as such, the exact amount of traffic a L1 cache incurs on the NoC depends on the absolute number of requests it actually receives. The consequences of a high L2 miss rate (as depicted in Figure 3.2b) are slightly more complicated. If the cache line is simply not present in the L2 cache, the request will be forwarded to main memory. This type of traffic will be routed via the memory controller and will not pass by the NoC. However, if the cache line is present in the L2 cache but in a wrong coherence state (e.g. a write request to a cache line shared by multiple L1 caches), the request will be forwarded to the sharers (L1 caches holding a copy of the cache line) via the NoC. A high L2 miss rate can therefore be due to either a large working set (e.g. ***fluidanimate***), a lot of sharing (e.g. ***raytrace***) or a combination of both. It is also interesting to note that benchmarks with a high L1 miss rate do not necessarily have a high L2 miss rate and vice versa. Overall though, the miss rates of the PARSEC benchmarks used in the thesis show a lot of variation.

[6]A Pt2Pt NoC where every coherence controller has a network interface and the link latency is equal to 1 clock cycle

Operational intensity

In this thesis, a concept from the roofline model is used to assess the ratio of communication to computation. The roofline model is an analytical model used to provide insights in the design of CMPs by relating the performance of the processor cores with the off-chip memory traffic [143] and is similar in concept to the program/machine balance model [144]. The roofline model uses a metric called operational intensity (I) which relates the work done by a workload (W) with the amount of memory traffic generated by the workload (Q) as shown in Equation (3.4).

$$I = \frac{W}{Q} \tag{3.4}$$

W is usually measured in Floating Point Operations (FLOPs) but can be measured in any type of operations, depending on the workload. Q is most commonly measured as the amount of traffic to the Dynamic Random Access Memory (DRAM) but the metric can be changed to only include the traffic to the LLC for example [143]. The unit used for operational intensity is FLOP per byte. Therefore, a workload with a high operational intensity is more oriented towards computation, whereas a workload with a low operational intensity is more communication-heavy. The work done by a benchmark will be measured both in FLOP and total number of instructions. The memory traffic Q is adapted to account for the fact that this thesis focuses on NoC optimisation and will, therefore, measure all traffic on the NoC (in byte).

Figure 3.3 shows the operational intensities of all PARSEC benchmarks. The figure shows operational intensity, both in instructions per byte and FLOP per byte. In the previous section ***x264*** was described as communication intensive. When looking at the operational intensity of ***x264*** and expressing the value as a fraction, this assumption is confirmed. The operational intensity of ***x264*** is equal to $\frac{1}{32}$ meaning every instruction will incur 32 B on the NoC. In the case of ***blackscholes*** however, the operational intensity has increased to $\frac{27}{26}$. One instruction will, approximately, only generate one byte of traffic on the NoC. Ideally, the set of workloads used should represent a spectrum of workloads, ranging from computationally heavy towards more communication oriented and should therefore have a large variation in terms of operational intensities, which is confirmed in Figure 3.3.

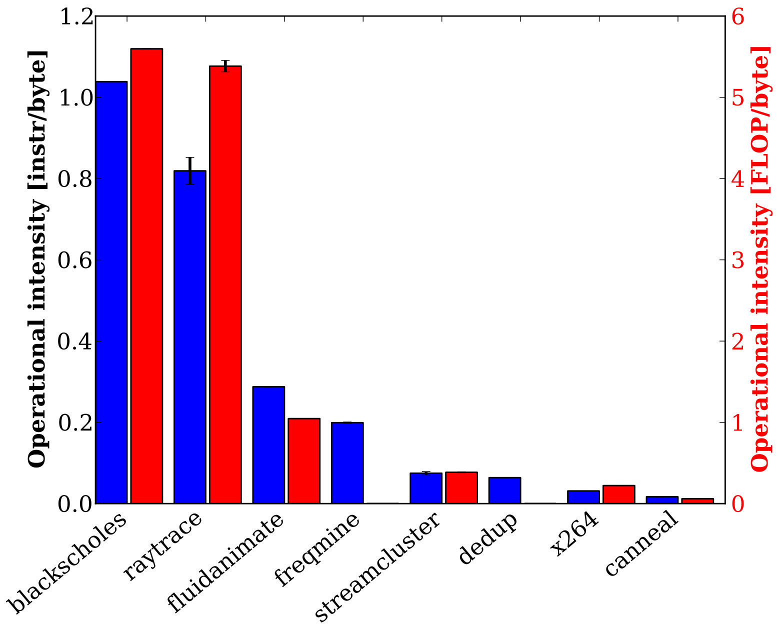


Figure 3.3: Operational intensities of the benchmarks in the PARSEC suite. The operational intensity is give in both instructions per B (blue) and FLOP per B (red) as the benchmarks differ greatly in the number of floating point operations.

Network load

Whilst operational intensity gives information about the type of workload (communication or computation), it does not give any information with regard to the amount of actual traffic on the NoC. Figure 3.4 shows the injection rates of the PARSEC benchmarks. The injection rate as shown is averaged over the complete runtime: the total number of bytes transmitted across the NoC divided by the runtime (in clock cycles). The red bars show the traffic to and from main memory. This is significantly lower than the injection rate on-chip. The reason for this is two-fold: first of all, the LLC acts as a filter, not all memory references that miss the L1 will miss the L2 and need to go to main memory. Secondly, the coherence transactions on the NoC are more complicated and consist of more messages, incurring more traffic. When a memory reference misses in the LLC, the resulting transaction via the memory controllers is quite simple: the cache line will be directly fetched from main memory. However, when a memory reference misses in the L1, the requested cache line can be in any of the L1 caches which can result in complicated coherence transactions (Section 5.3.1). The lower memory injection rate might seem at odds with the higher L2 miss rate as depicted in Figure 3.2b but it is important to remember that a miss in the L2 does not inevitably leads to a memory request and again, L2 miss rate is a relative number.

Whilst Figure 3.4 already gives an indication of the low network loads that can be expected in the NoC, Figure 3.5 confirms this. The figure plots injection rate, not in bytes per clock cycle but in messages per clock cycle. Again, ***canneal*** has the highest injection rate but even so, less than one message is injected every three clock cycles. Both in Figure 3.4 and Figure 3.5 the benchmarks are sorted in descending order. However, whilst ***x264*** and ***streamcluster*** have similar injection rates when measured in bytes per clock cycle, ***streamcluster*** incurs more messages. This indicates the proportion of small control messages being sent in ***streamcluster*** is higher, resulting in more traffic but less bytes. Overall though, the load on the NoC is low.

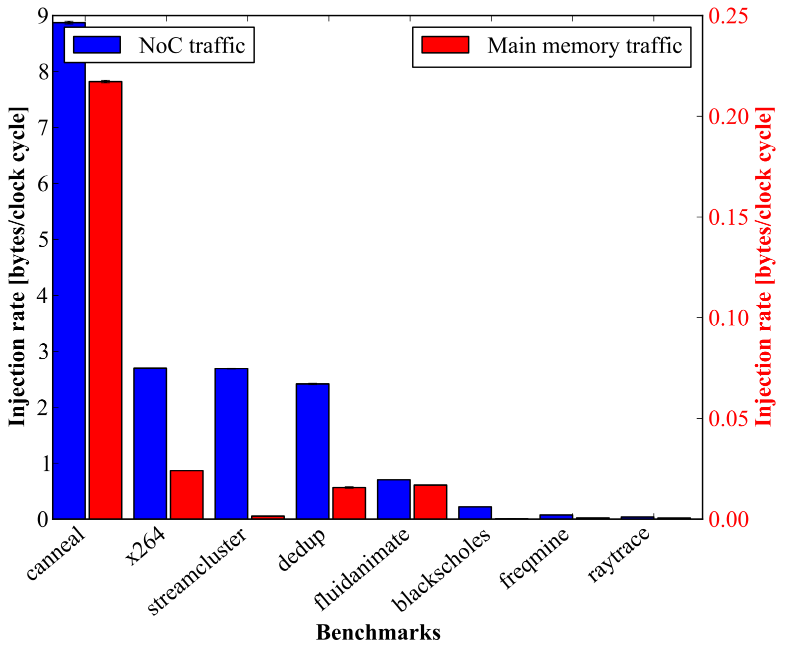


Figure 3.4: Injection rates in the PARSEC suite.

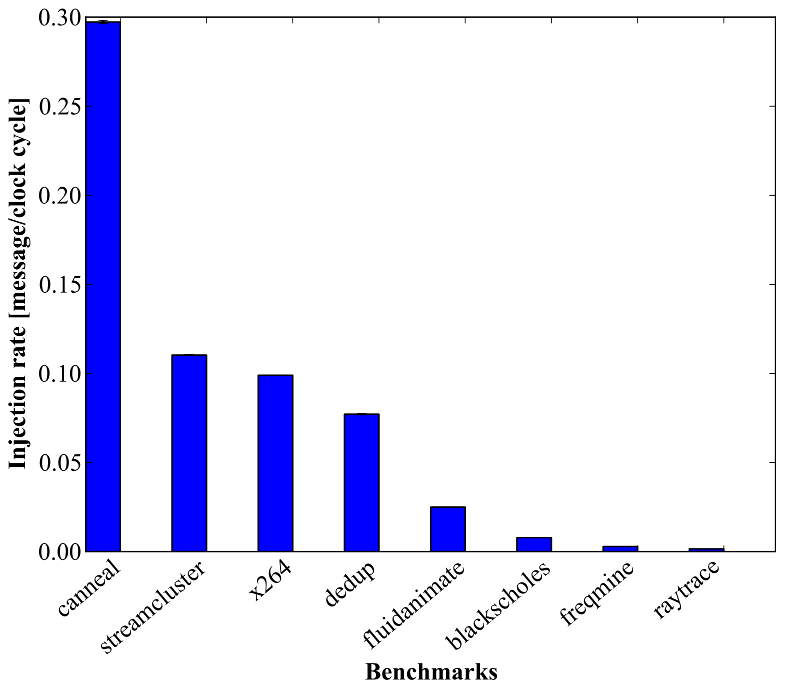


Figure 3.5: Message injection rates in the PARSEC suite.

The injection rates shown in Figure 3.5 show the average injection rate. In Figure 3.6 the injection rates are shown over time (per epoch). These results were obtained in a slightly different manner, using traces that contain every message transmitted on the NoC, with its timestamp[7]. An epoch is defined as 1 million clock cycles. These plots also only show one single run, whereas the other results are averaged out over multiple runs. These plots are only used to show that even benchmarks with a quite similar average injection rate like ***streamcluster*** and ***x264*** can exhibit different behaviour over

[7]More details about this can be found in chapter Section 5.1.

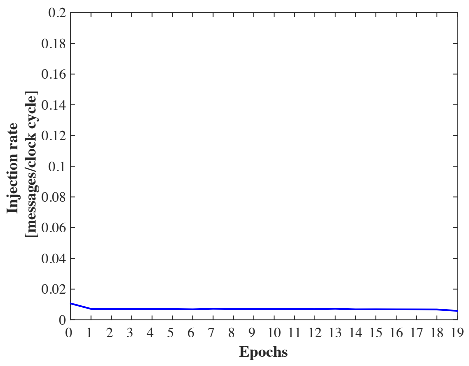
time.

Sharing

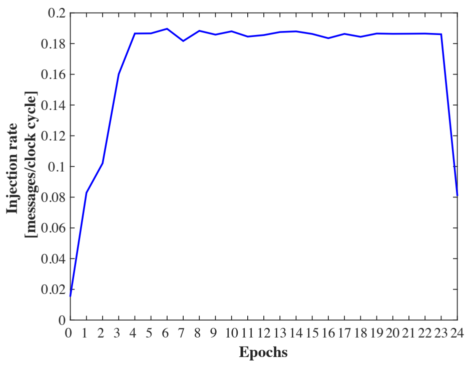
Sharing is a difficult concept to quantify, partially because the term is so vague. A concise definition of sharing could be two or more threads using the same data. Using data can mean multiple things: the threads could only be reading the same data (e.g. shared information), one thread could write to the data, which other threads then read (e.g. information exchange) or all threads could be attempting to write the data (e.g. lock). It is not straightforward to extrapolate whether operations on a cache line are actually aimed at the same memory address (i.e. the same data structure), because multiple memory addresses are grouped in the same cache line. The aim of this section is to quantify the type of sharing behaviour that will affect the NoC traffic. To do this, requests to the L2 cache are recorded and categorised according to the state of the requested cache line. Figure 3.7 shows the proportion of read requests (also referred to as load (LD) requests) that arrive at the L2 cache for cache lines currently held by multiple L1 caches (red bars) or cache lines that are currently held exclusively by a L1 cache (blue bars). These states correspond to the 'Shared' and 'Exclusive' state respectively (Section 2.1.4). In ***blackscholes*** a lot of information is shared but not exchanged which is confirmed here by the large number of read requests to cache lines also present in other L1 caches. The same holds for ***raytrace***. ***streamcluster*** shows the most reads to cache lines in the Exclusive state (7.1%). There is no significant difference with ***fluidanimate*** though (6.7%). The amount of cache line sharing for write requests (store (ST) requests) is generally lower as can be seen in Figure 3.8, with the exception of ***raytrace***. ***streamcluster*** exhibits the second highest amount of sharing (when combining both states) which might be attributed to the large amount of false sharing present. However, it needs to be noted that these again are relative numbers as they are calculated by dividing the number of read/write requests to a certain state by the total number of read/write requests. For example, ***raytrace*** has a large amount of sharing relatively speaking but it has a very low L1 miss rate meaning that less requests need to be sent on to the directory. This is confirmed by the low injection rate of ***raytrace*** Figure 3.5.

Another way of quantifying sharing is looking at the number of sharers per cache line upon invalidation. An invalidation occurs when the directory orders one or more L1 caches to invalidate their copy of a cache line. This can be because the L2 cache is replacing the cache line and, hence, the cache line can no longer be present in any of the L1 caches (inclusive L2 cache) or because the directory received a write request to a shared cache line. The number of sharers per cache line is approximated here by dividing the number of invalidation messages received by all L1 caches with the number of invalidations executed by the directories. This gives approximately the number of L1 caches that need to be addressed per invalidation round and, hence, the number of destinations in the resulting multicast message placed on the NoC. This can be important as certain network architectures struggle with multicasts. Figure 3.9 shows ***raytrace*** having the highest number of sharers, on average (blue bar). However, ***raytrace*** has the lowest probability of an invalidation actually occurring (red bars).

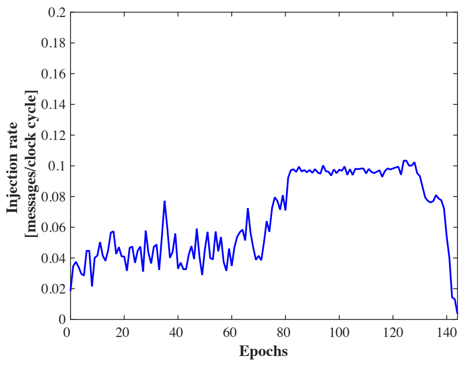
Again, the benchmarks used in this thesis show a lot of variation in terms of sharing behaviour. The characteristics discussed in this section are summarised in Table 3.1, in case the metric is not clear on the corresponding figure.



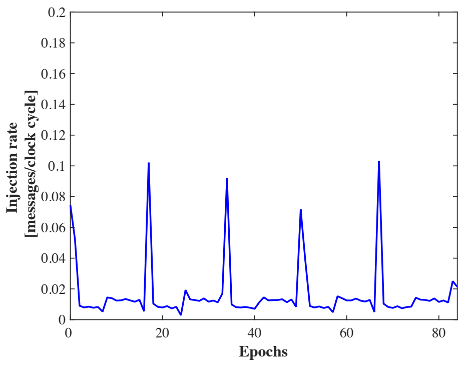
(a) ***blackscholes***



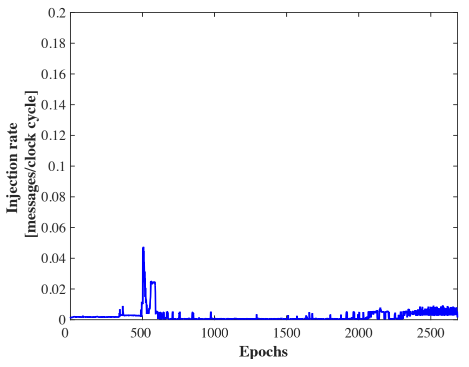
(b) ***canneal***



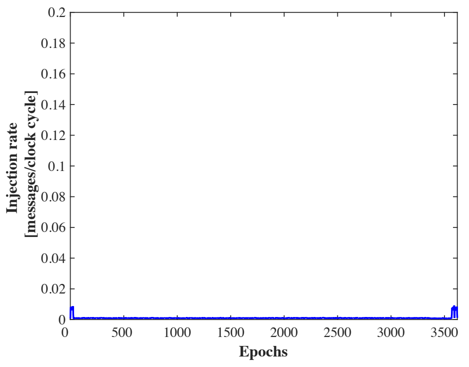
(c) ***dedup***



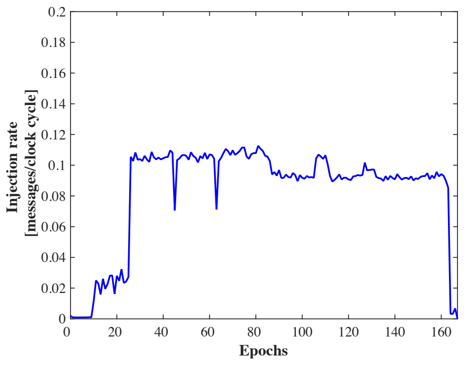
(d) ***fluidanimate***



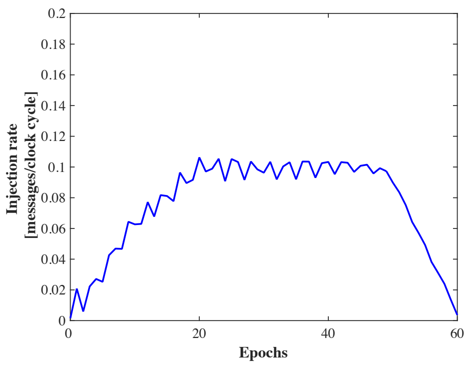
(e) ***freqmine***



(f) ***raytrace***



(g) ***streamcluster***



(h) ***x264***

Figure 3.6: Injection rates of the PARSEC benchmarks, plotted over time

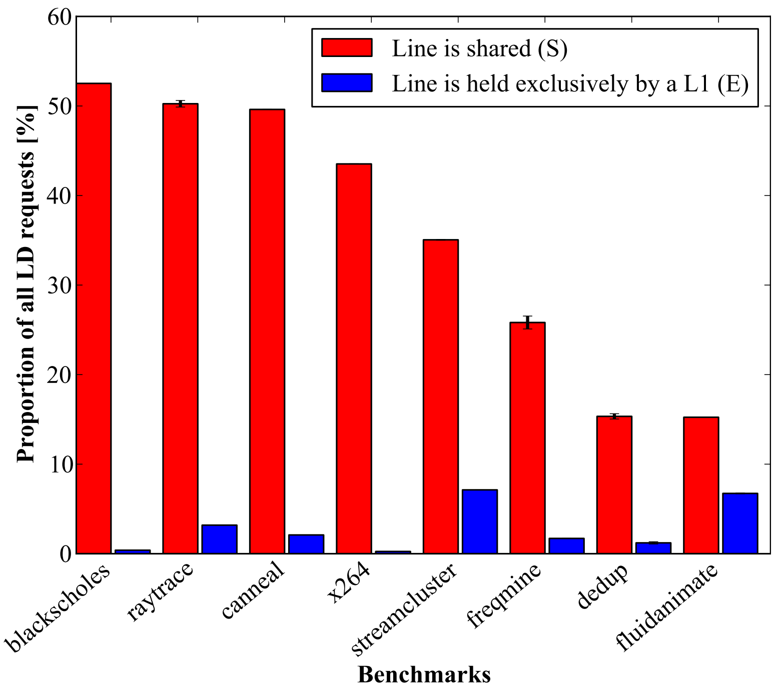


Figure 3.7: Sharing behaviour of read requests that reach the L2 for the different PARSEC benchmarks.

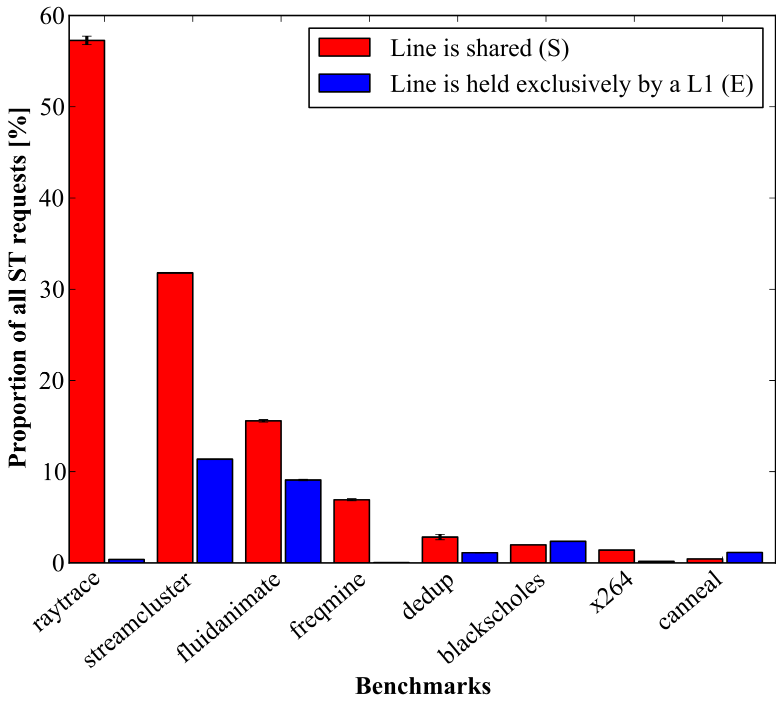


Figure 3.8: Sharing behaviour of write requests that reach the L2 for the different PARSEC benchmarks.

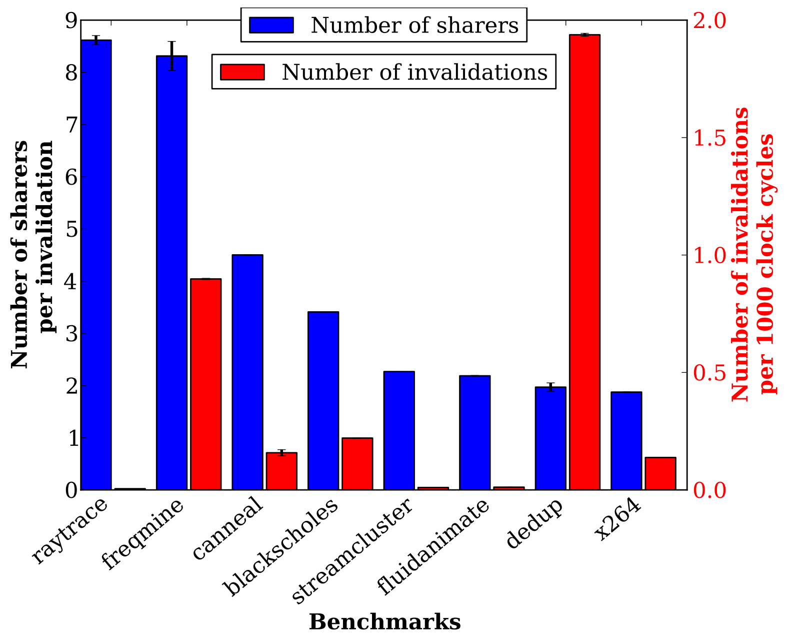


Figure 3.9: Average number of sharers involved in an invalidation (blue bar) and the probability of an invalidation actually occurring (red bar) for the different PARSEC benchmarks. Invalidations happen after a replacement in the L2 cache or a write request to a shared cache line.

3.2.2 Simulator choice

The next step in the setup of an experiment is the choice of type of modelling used to test the target architecture. Analytical modelling is fast and can give high-level and fundamental insights but lacks the accuracy of simulators because of the complexity of the complete system and its interactions [133]. As the interaction between the NoC, processor cores and memory hierarchy is extremely complex, full-system simulation is used in this thesis. This decision will be justified in Section 3.2.3. After discussing the simulator of choice in this thesis, the evaluation methods used in other optical NoC related work will be reviewed.

Choice of the gem5 simulator

In this work a full-system, cycle-accurate simulator called gem5 is used [145]. Full-system simulators are micro-architectural simulators which allow the complete software stack to be run on the simulator [133]. This gives more reliable results when running multithreaded workloads as the operating system handles the thread scheduling. There are challenges associated with full-system simulators: their development is far from easy, simulations take a long time and non-determinism is introduced as discussed in Section 3.1.2. Cycle-accurate simulators emulate the behaviour of the target architecture on a cycle-by-cycle base. The advantage is correct timing behaviour at the cost of longer development and simulation time.

The simulator used in this work combines the best aspect of two earlier simulators, M5 [146] and GEMS [147] which were both focused on different parts of the computer architecture. gem5 is an open-source simulator, written in C++ and supported by both academia (University of Wisconsin-Madison, University of Michigan etc.) and industry (e.g. ARM, AMD, etc.). The simulator is very flexible as it contains various

ISAs (ARM, x86, Alpha etc.), various CPU models (simple, aggressive out-of-order), different system modes[8] and various memory systems.

There are two memory system models in gem5. The classic memory system, taken from M5, is the simplest and fastest model. It models a bus-based memory system with an abstract Modified Owner Exclusive Shared Invalid (MOESI) snooping protocol [9]. The second model, the Ruby model (from GEMS), is more flexible and provides more accurate simulations at the cost of additional developing complexity and simulation time. The Ruby model allows simulation of a memory hierarchy using an arbitrary cache coherence protocol and an arbitrary NoC. When researching a novel NoC concept it will most likely not be present in the gem5 repository, if not it has been investigated by other researchers. Therefore, the developing cost of the Ruby model, has to be paid every time a new concept is investigated. In this work the Ruby model is used, which I extended with a new optical network model to simulate the behaviour of various optical NoCs.

Various papers have assessed the accuracy of the gem5 simulator. Butko et al. compared the real-life dual core ARM system with its gem5-simulated counterpart, using various workloads [149]. The authors found the accuracy varied, based upon the amount of memory traffic generated by the workload. This was found to stem from a simplified DRAM model. Since then the DRAM model has been updated [150]. Overall, the authors found the accuracy (which ranges between 1.4% and 17.9% depending on memory traffic) satisfactory. Gutierrez et al. also investigated the accuracy of the gem5 simulator by comparing the runtime statistics and micro-architectural statistics with an existing hardware platform [151]. They found the runtime error of gem5 in combination with the PARSEC benchmarks to be less than 20%. The micro-architectural statistics were found to be within an error margin of 20% as well.

Introduction of variability in gem5

As mentioned in Section 3.1.2 it is necessary to account for different execution paths in multithreaded simulations by running multiple simulations. To ensure these different simulations take different execution paths it is necessary to introduce non-determinism.

To achieve non-determinism in the simulations, random variation was added to the latency of the NoC links (irrespective of the network type). It is important to note that the non-determinism bears no resemblance to fabrication tolerances etc. The random variation in latency is needed to nudge the different runs to take different thread scheduling decisions and thereby to follow different execution paths. The unperturbed link latency is calculated first. The exact calculation depends on the NoC type. After this, it is decided at random whether or not the link should be perturbed. Perturbations occur with a probability ρ_{rand}. If the link will be perturbed, the only thing left to do is choose the direction of the perturbation as the size of the perturbation is fixed and equal to ϑ_{rand}. The perturbation is then added to the original link latency. The pseudo-code used for this can be found in Figure 3.10. There are three system wide variables which need to be set at the start of the simulation.

[8]gem5 also has syscall-emulation mode in which the system level effects like for example interrupts are emulated by the simulator. This mode is not used in this thesis.

[9]The MOESI protocol was proposed by Sweazey and Smith [148]. It resembles the Modified Exclusive Shared Invalid (MESI) protocol (Section 2.1.4) but has an additional coherence state for shared cache lines: the Owned state.

```
link latency ← Calculation of unperturbed link latency ( ∼ NoC type) ;
int pertubation = 0 ;
unsigned pert ← generate random number between 0 and $\rho_{precision}$ ;
if $pert < \rho_{rand}$ then
    perturbation = $\vartheta_{rand}$ ;
    bool up ← generate randomly with 50/50 distribution ;
    if not up then
        if link latency - $\vartheta_{rand} \geq 0$ then
            perturbation = perturbation * (-1)
        end
    end
end
link latency += perturbation
```

Figure 3.10: Algorithm used to add variability to the link latency

- ρ_{rand}**:** The probability of a perturbation i.e. the probability the link latency will be changed (default = 0.05)
- $\rho_{precision}$**:** The precision of ρ_{rand}
- ϑ_{rand}**:** The perturbation introduced i.e. the number of clock cycles with which the link latency will be increased or decreased (default = 1 clock cycle)

The average link latency will not be affected by these perturbations as line 6 in Figure 3.10 ensures the latency gets increased and decreased an equal amount of time. These small perturbations can cause the simulations to follow different execution paths, as shown in Figure 3.1.

Performance evaluation of other optical NoC proposals

Most of the proposals for optical NoCs discussed in Section 2.2.3 were evaluated in terms of performance by using synthetic traffic and/or real application traces in a network simulator. This was the case for the macrochip [89], the hybrid optical torus [118], SPINet [112], the Time Division Multiplexing (TDM) network [107], Firefly [96] and Corona [109]. The network simulators were either in-house simulators, PhoenixSim [152] or booksim [42].

The only exception is the ATAC architecture [106] which is simulated using the Graphite simulator [153]. The Graphite simulator is an open-source multiprocessor simulator. A simulation consists of a target multicore architecture, defined by models in the simulator, on which an application is running. To decrease the simulation time, the different tiles in the multicore architecture are decoupled and simulated in parallel. At predefined times, the events on different tiles will be synchronized. The strictness of the synchronization determines the simulation time. This makes Graphite, by default, not cycle-accurate. This makes it hard to accurately simulate contention. The contention for common resources is determined using statistics.

Network simulators, using application traces, will report correct numbers in terms of characteristics and performance metrics of the NoC by itself but give no definite answers on how the CMP as a whole will react to a change in the interconnection network. This is due to the fact that these type of simulations do not capture the intricate interplay between the NoC, memory hierarchy and processor cores as there is no or limited feedback between the NoC simulation and the application level. The Graphite simulator does simulate the application level but is not cycle-accurate and is not able to accurately model the contention in the network, a very important parameter of networks.

To the best of my knowledge, the work discussed in Chapter 4, which draws upon a publication by Van Laer et al. [11] was the first work to measure the performance benefits gained from an optical NoC using full-system, cycle-accurate simulations[10].

3.2.3 Need for full-system simulations

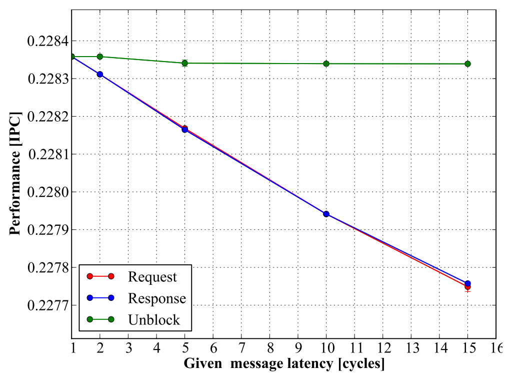
Full-system simulations are time consuming in both simulation time and development time. In this section I want to show why they are invaluable, nevertheless, by means of an experiment.

There are three message classes in the MESI protocol used in gem5. *Request messages* carry requests from one coherence controller to another, *response messages* carry responses whilst *unblock messages* indicate the end of a coherence transaction. A coherence transaction consists of an arbitrary number of messages from these classes, depending on the exact type of transaction. These types will be discussed in more detail in Section 5.3.1. This experiment aims at investigating the relationship between the latency of a certain type of message and the overall CMP performance. The NoC is a Pt2Pt network, in which every coherence controller has an individual network interface. It is possible to show the effect of slowing down or speeding up one message class on the overall performance, by varying the latency of a certain class, whilst keeping the latency of all the other messages classes constant.

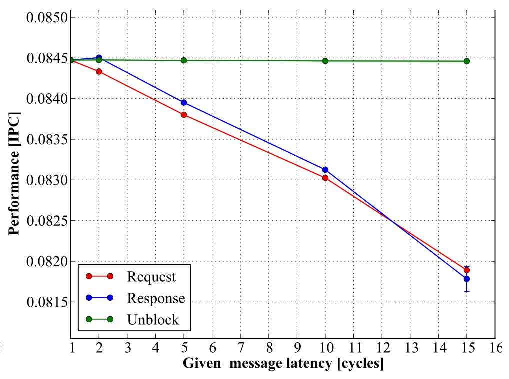
Figures 3.11a and 3.11b show the effect of changing the latency of various message classes for ***blackscholes*** and ***x264***. The x-axis shows the latency of the message class under investigation whilst the y-axis shows performance in IPC. The red lines denote the cases in which the latency of the request messages is varied whilst the latency of the other messages is kept constant and equal to one clock cycle. The blue lines denote the same but for the response messages. Both Figures 3.11a and 3.11b shows performance going down when increasing message latency, which is not an unexpected result. It also shows increasing the latency of unblock messages does not affect performance which again, is not unexpected. Unblock messages signal the end of a transaction when a L1 cache controller signals to the directory it has received the requested cache line. Holding up unblock messages by increasing their latency will, therefore, not slowdown outstanding requests in the L1 cache and, hence, have little effect on overall performance.

However, Figures 3.11c and 3.11d show the same results but now the x-axis denotes the average message latency which is calculated by multiplying the number of messages per message class with the associated latency. The given latency (as shown in Figures 3.11a and 3.11b) only captures the latency of a single message class, whilst the average latency (as shown in Figures 3.11c and 3.11d) shows the effect of a change in given latency on the latency across all message classes. The average message latency

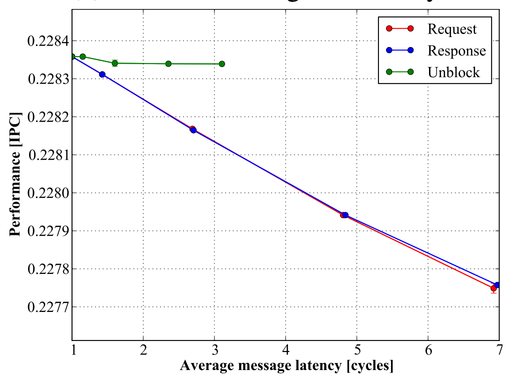
[10]The work on an optical hierarchical ring NoC by Bartolini et al. [154] was published in the autumn of 2013, whilst the work by Van Laer et al. [11] was published in the spring of 2013.



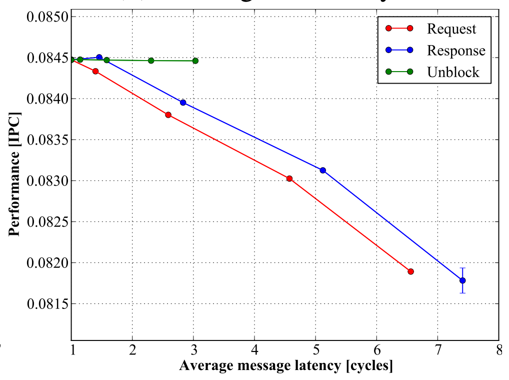
(a) ***blackscholes*** - given latency



(b) ***x264*** - given latency



(c) ***blackscholes*** - average latency



(d) ***x264*** - average latency

Figure 3.11: Effect of changing latency per message class on performance, shown by plotting IPC versus message latency. (a) and (b) show performance versus given message latency for ***blackscholes*** and ***x264*** respectively. (c) and (d) show the IPC versus resulting average message latency for these benchmarks.

will also vary across runs as the exact number of messages will vary. Error bars denoting the variation have been applied in Figures 3.11c and 3.11d but the variation is so small they are not visible. Increasing the latency of requests and response messages has the largest effect on the average message latency as they are more frequent.

These figures make the need for full-system simulations clear: the average latency of a system in which requests are slowed down (red line) versus a system in which unblock messages are slowed down (green line) might be very similar but the effect on the overall performance will be quite different. Average message latency is a useful metric when comparing NoCs. However, to measure the effect of a NoC on the overall CMP performance full-system simulation is needed as individual messages will affect performance in different ways.

4 Effect of Latency on Performance

THE two previous chapters defined the concepts that were needed to answer the first research question posed in this thesis: can an optical Network On-Chip (NoC) outperform an electrical NoC? Firstly, the different types of optical NoCs were discussed in Section 2.2.3, followed by a discussion on how performance should be measured exactly in Chapter 3. As the concepts needed to define the question are defined now, this chapter will aim at actually answering this question.

Firstly, the effect of link latency and link bandwidth in an ideal NoC will be investigated in Section 4.1. Isolating the effects of latency and bandwidth makes it easier to understand the effect of an optical NoC in which latency and bandwidth are intertwined. This section will also confirm some general assumptions about operational intensity and network loads. Secondly, Section 4.2 will then look at three NoC types: an electrical mesh, a crossbar-based circuit-switched optical NoC and a Single Writer, Multiple Reader (SWMR)-based non-switched optical NoC. Their implementation in the gem5 simulator will be discussed (Section 4.2.1 and Section 4.2.2). Characteristics of an optical NoC such as the number of wavelengths per channel and the number of channels per tile determine its performance. An increase in wavelength or channel count increases both the complexity and power consumption of an optical NoC. In order to find the most optimal optical NoC setup, a sweep will be performed for both these characteristics in Section 4.2.3. This will be be followed by a discussion and comparison between the two optical NoC types in Section 4.3.

4.1 Effect of Latency and Bandwidth on Performance

The first question this thesis aims at answering is the effect an optical NoC has on the overall performance. An optical crossbar in particular has very different characteristics than an electrical mesh. To allow for a better understanding of the effects of an optical NoC on performance, this section will look at the isolated effects of both latency and bandwidth on performance. To the best of my knowledge, there is no such work in the literature. Similar, yet quite different work was done by Sanchez et al. [155], in which three different network topologies (mesh, flattened butterfly and fat tree) are tested on 128-core Chip Multiprocessor (CMP), using a combination of an in-house simulator for the cores and a predecessor of gem5 for the cache and memory hierarchy. The system setup clearly differs from the one used in this thesis[1] but the authors reach similar conclusions about latency and bandwidth.

To explore the effects of latency and bandwidth, an ideal NoC is used. All coherence controllers are directly connected to a Point-to-Point (Pt2Pt) network, this in contrast to the tiled organisation depicted in Figure 2.23, where three coherence controllers (First Level Cache (L1), Second Level Cache (L2) slice and memory controller) share a single network interface. Giving each coherence controller direct access to the network by means of a private network interface, allows for a clearer analysis of the effects of NoC latency and bandwidth as it removes the effects of contention in the shared network interface. In these simulations, latency and bandwidth are completely decoupled. The latency of a link (given in clock cycles) is the time taken by a complete message to traverse the link. A message travelling a link with a latency of 5 clock cycles will appear in the destination buffers exactly 5 clock cycles later. The bandwidth of a link (given in B per clock cycle) determines the time a message will occupy the link whilst the message is being transmitted i.e. the time it takes before another message can use the same link. A control message (8 B) passing a link with a bandwidth of 8 B will therefore only occupy the link for 1 clock cycle. However, if a data message (72 B) passes by the same link, the link will remain occupied for 9 clock cycles. In a real system, however, latency and bandwidth are intertwined characteristics. A data message passing by a link with a bandwidth of 8 B and a latency of 1 clock cycle, will arrive completely at the destination after 10 clock cycles, as the message will be divided into 9 separate 8 B flits and it will take 1 clock cycle for the head flit to cross the link[2]. This artificial dissociation between latency and bandwidth, however, allows to completely separate the effects of latency and bandwidth.

4.1.1 Effect of latency on performance

Figure 4.1 shows the effect of increasing latency on performance. The x-axis shows the latency which varies from 1 clock cycle to 30 clock cycles. The y-axis on the other hand shows performance. Performance is shown in speedup, to ease comparisons across the benchmarks. To calculate speedup, the performance (in Instructions per Cycle (IPC)) is normalised to the maximal performance (in IPC) reached when the latency is equal to 1 clock cycle. The simulations per latency value have been repeated until the sample size (i.e. the number of simulations for this latency value) is larger than the ideal sample size

[1]The authors investigate a CMP with 128 in-order, 2-way multithreaded X86 cores and use the PARSEC [138], SPLASH-2 [156] and BioParallel [157] workloads. This differs from our setup (Section 2.3) in the numbers of cores (16 versus 128), the type of cores (single-threaded versus 2-way multithreaded), the workloads (only PARSEC versus 3 suites), the protocol (MESI versus MOESI) and the simulator setup (gem5 versus in-house + GEMS. However, broadly speaking, the type of system is similar to ours (a tiled, directory-based coherent system with distributed caches).

[2]Messages can be subdivided into flits where a flit is at the granularity of the flow control mechanism, which in most case is for example the channel width [22, Chapter 5.1].

(as per Equation (3.3)). The figures also show the maximal and minimal performance obtained for that specific latency value (red and green dots respectively). This again indicates the need for multiple runs, as completely different conclusions could be drawn when only using one run per latency value. For example in the case of ***canneal***, if only one run was performed it could seem the benchmark reacts especially bad to a link latency of 4 clock cycles.

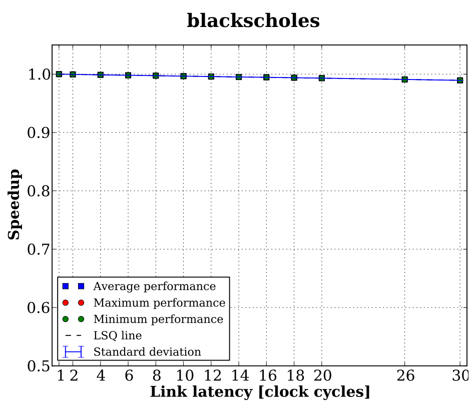
Overall, it is clear all benchmarks react negatively to an increase in the link latency. The exact impact, however, differs per benchmark. When increasing the latency to 30 clock cycles, the performance when running ***canneal*** has dropped by approximately 20%. ***blackscholes*** on the other hand barely loses performance when the latency is increased from 1 clock cycle to 30 clock cycles.

However, Figure 4.1 also shows there are some simulation artefacts present. In the case of ***canneal*** (Figure 4.1b), it seems a link latency of four clock cycles is detrimental to performance. The red data points represent the maximal performance obtained in this set of samples. Therefore, one could say these represent the run with the most optimal scheduling decisions. When only looking at the red data points, there is no drop when the link latency is equal to four clock cycles. In the case of ***streamcluster***, it seems the performance loss tails off at increasing latency, or even increases. Simulations of higher latencies (depicted in Figure 4.2) show though this is a local increase and, hence, a simulation artefact. The results of the latency sweep for ***raytrace*** on the other hand are quite random and I was not able to conclusively find the cause of this inconsistent behaviour. It was impossible to plot ***raytrace*** on the same scale as the other benchmarks as that range would not capture all data points. The ***raytrace*** benchmark will be excluded from the remainder of this chapter, because of the long simulation time of ***raytrace*** (2 days per data point) and what most likely are simulation artefacts.

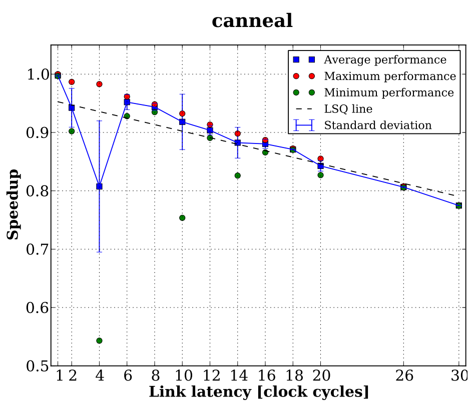
To compare the effect of latency on the various benchmarks in a more general way, the concept of operational intensity is used again (Section 3.2.1). Benchmarks with a high operational intensity (such as ***blackscholes***) are more computation-oriented, rather than communication-oriented. This would mean they are less affected by changes in the NoC. Figure 4.3 shows speedup versus operational intensity. The lowest and highest latencies are marked by the $\star$ marker and the $\diamond$ marker respectively. The latency sweep of a benchmark depicted in Figure 4.1 can be traced on this overview figure by going from the $\diamond$ marker to the $\star$ marker. The more spread out these markers are, the more effect an increase in latency has on performance. This figure confirms that, generally speaking, benchmarks with a low operational intensity (e.g. ***x264*** and ***canneal***) are more affected by a change in the NoC than workloads with a high operational intensity (e.g. ***fluidanimate*** and ***blackscholes***). The figure also shows these 4 benchmarks, at either end of the operational intensity spectrum, capture the behaviour of the PARSEC benchmark suite as a whole quite well. Therefore, in the remainder of this chapter only results for these 4 benchmarks will be shown.

4.1.2 Effect of bandwidth on performance

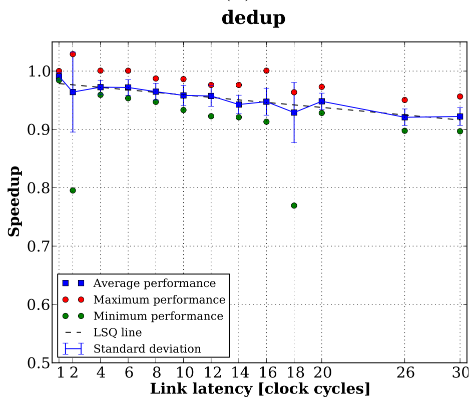
Figure 4.4 shows the effect of bandwidth on speedup. The bandwidth decreases from left to right and the link latency is kept constant at 6 clock cycles. The smallest bandwidth is 4 B per clock cycle: a control message (8 B) will therefore occupy the link for 2 clock cycles and a data message (72 B) for 18 clock cycles. This means that for the subsequent 2 to 18 clock cycles no other messages can use this link. However, because



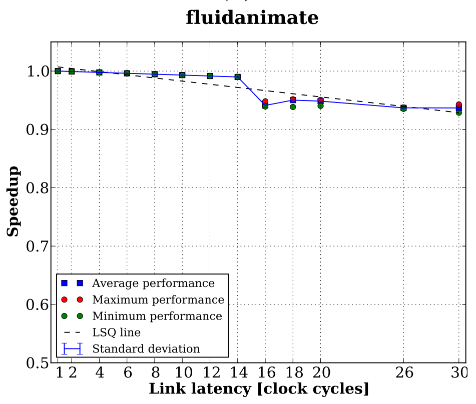
(a)



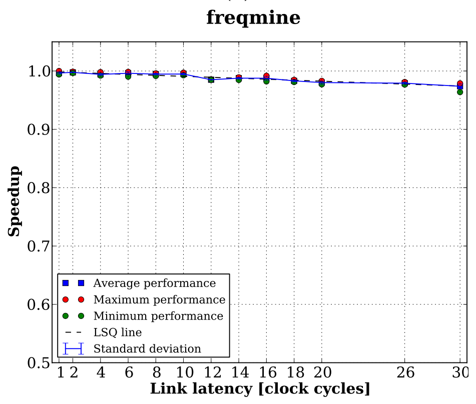
(b)



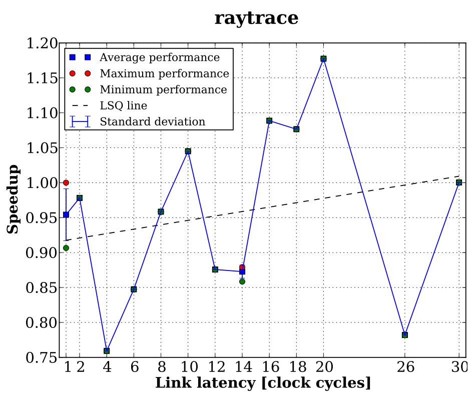
(c)



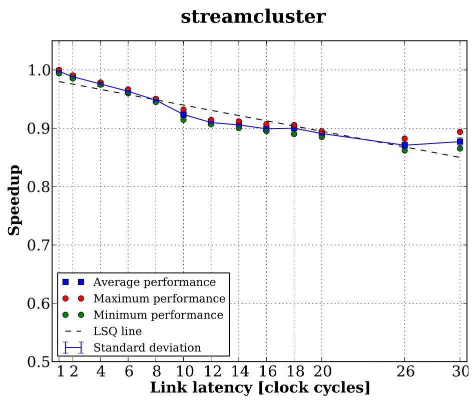
(d)



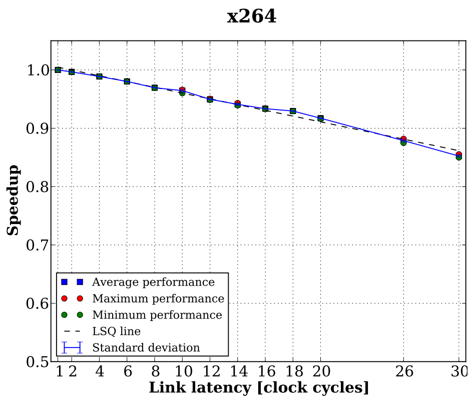
(e)



(f)



(g)



(h)

Figure 4.1: Effect of changing latency on speedup.

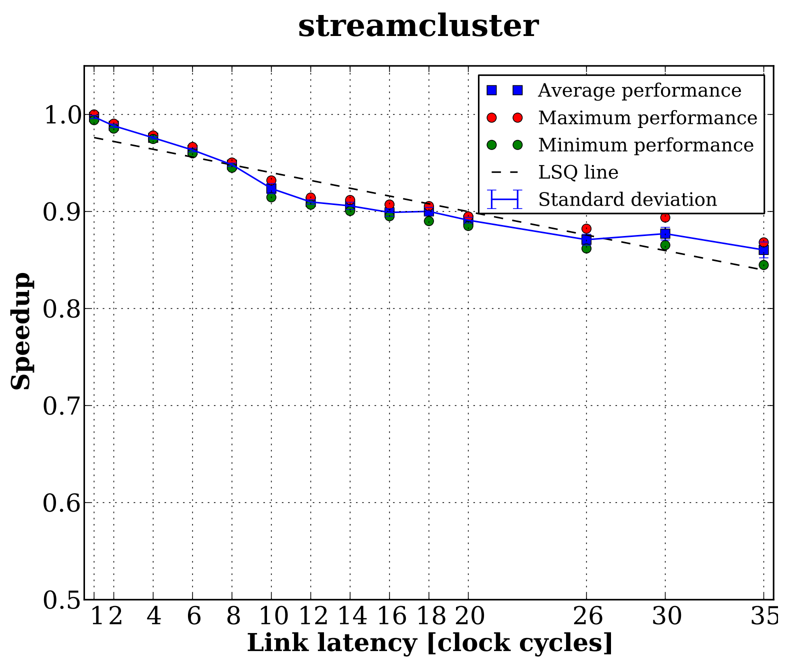


Figure 4.2: Effect of changing latency on speedup for ***streamcluster***. By increasing the latency above 30 clock cycles, it becomes clear the increase in Figure 4.2 around 30 clock cycles is only a local increase.

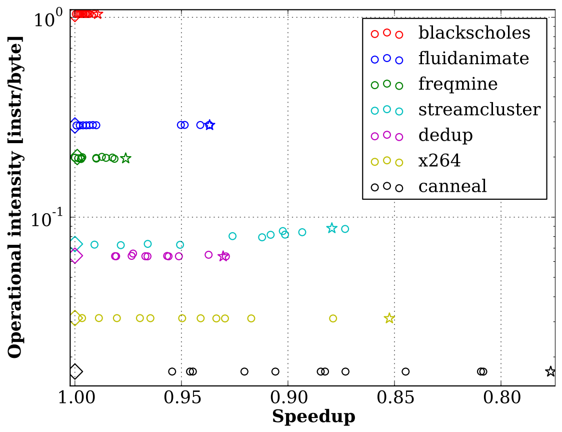
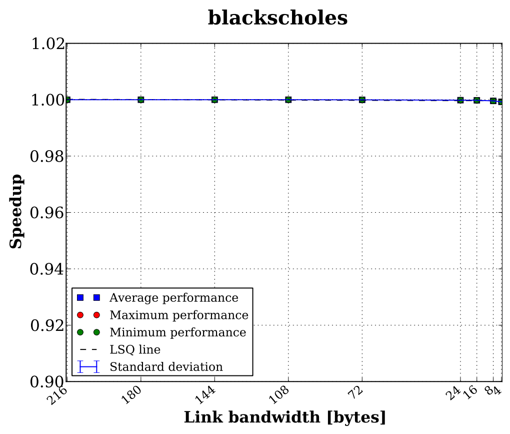


Figure 4.3: Decrease in speedup due to increasing latency, plotted against operational intensity. The ◇ marker represents speedup at the lowest latency, the ⋆ marker represents the speedup at the maximal latency.

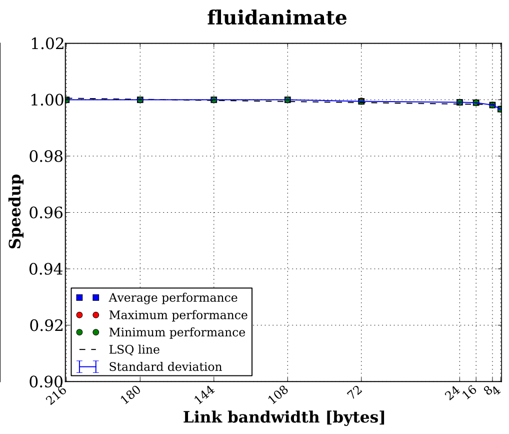
of the low injection rates in the NoC, this is not a problem. The associated decrease in speedup is minimal. Even the least operationally intensive benchmarks (i.e. ***canneal*** and ***x264***) which are most affected by changes to the NoC, only lose between 2% and 5% performance.

4.1.3 Conclusion

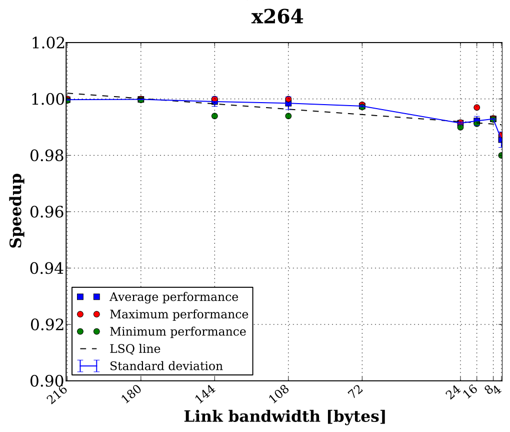
Figure 4.4 and Figure 4.1 seem to suggest an increased link latency has more effect than a decreased link bandwidth, which is in line with the results reported by Sanchez et al. [155]. In Figure 4.5, both these latency and bandwidth sweeps are plotted. The baseline is a NoC with a link latency of 6 clock cycles and a link bandwidth of 72 B. To



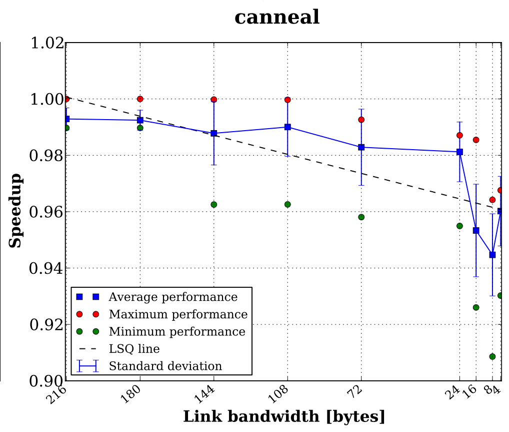
(a)



(b)



(c)



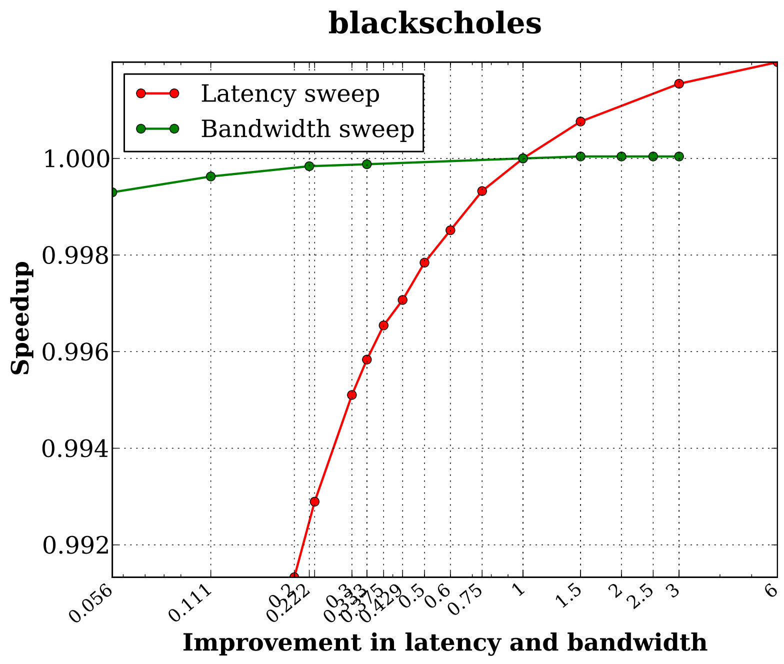
(d)

Figure 4.4: Effect of changing bandwidth on speedup.

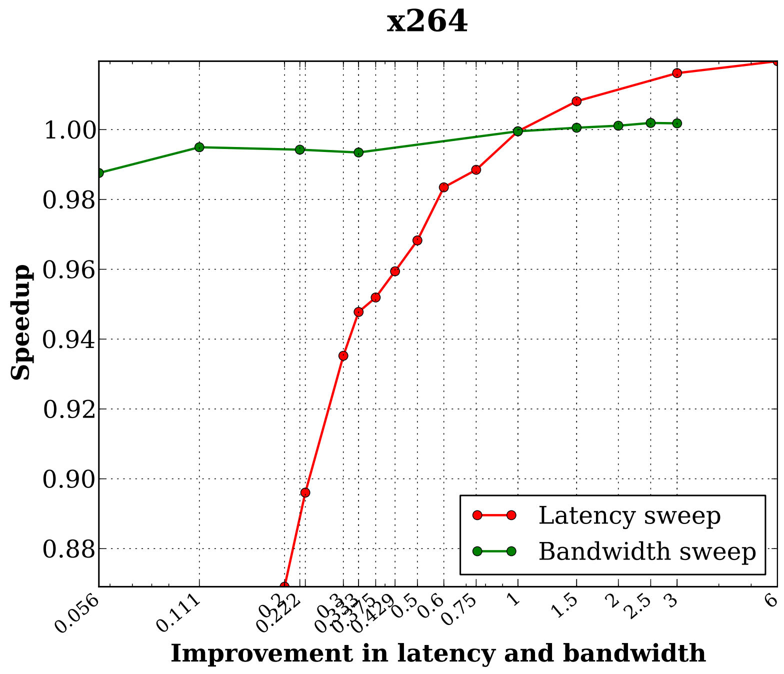
normalise latency, this baseline latency is divided by the latency of a data point. For example, a normalised value of 6 on this axis indicates a link latency of 1 clock cycle. To normalise the bandwidth, the bandwidth of a data point is divided by the baseline bandwidth. For example, a normalised value of 3 on this axis indicates a link bandwidth of 216 B. This normalisation is applied to make the figure more enlightening. Going from the left to right on the x-axis indicates, both for the latency and bandwidth sweep, an improvement in NoC performance: the link latency decreases and the link bandwidth increases. Only the results for ***blackscholes*** and ***x264*** are shown as they suffered the least from simulation artefacts both the latency and bandwidth sweep, making the general comparison more comprehensible. Both for ***blackscholes*** (Figure 4.5a) and ***x264*** (Figure 4.5b) the performance of the CMP as a whole is more affected by the latency of the NoC, than the offered bandwidth.

4.2 Effect of Latency in an Optical Network On-Chip on Performance

This section looks at the effect of an optical NoC on the performance of a CMP. Two types of optical NoCs are compared: a non-switched NoC and a circuit-switched NoC. In the case of the circuit-switched NoC, optical circuits need to be setup before message



(a)



(b)

Figure 4.5: Effect of both latency and bandwidth normalised to an architecture with a link latency of 6 clock cycles and a link bandwidth of 72 B.

transmission can start, incurring a path setup latency. In this work, it is assumed the central switch is an optical crossbar but the exact optical layout of the switch is not important in this and subsequent chapters. The non-switched NoC on the other hand use a Single Writer, Multiple Reader to avoid the latency associated with the setup of the optical circuits. These networks were discussed in more detail in Section 2.3.

The NoCs (the optical NoCs and the electrical mesh) used in this section differ from the ideal Pt2Pt network used previously. First of all, a tiled structured (as depicted in Figure 2.23 is used: a L1 cache controller, a L2 cache/directory controller and memory controller share a router. This means there will be less traffic on the NoC as intra-tile traffic does not enter the NoC. Coherence controllers that share a router can directly communicate with each other e.g. a memory request from the L2 cache can travel straight to its memory controller, provided they are on the same tile. The second difference lies in the effect bandwidth has on latency. In the previous section, latency and bandwidth were two isolated characteristics. However, in a real system (as simulated here) the bandwidth of a link affects the latency a message experiences. The time taken by a message to completely cross a link (i.e. the link latency) is the sum of the actual transmission time (also known as the time of flight) and the serialisation latency. The serialisation latency which is the time needed to place the complete message on the link depends on the link bandwidth.

The first two sections elaborate on the simulation of the NoCs in gem5. First, the existing implementation of an electrical mesh is reviewed as it will function as the baseline NoC to compare the optical NoCs with (Section 4.2.1). This is followed by a discussion on the implementation of the optical NoCs in gem5 (Section 4.2.2). These models are then used to simulate the various ways in which the latency of an optical NoC can be changed, allowing us to compare the electrical mesh network with the non-switched and circuit-switched optical NoCs in Section 4.2.3.

4.2.1 Electrical mesh as comparison network on-chip

Electrical mesh networks are seen as the most viable NoC for CMPs with high core count (Section 2.2.1). Therefore, this will be used as a baseline NoC, to compare the optical NoCs with.

The gem5 simulator already contains an implementation of mesh network, called garnet [158]. There are two flavours of garnet which differ in the way the router is modelled. In the first model, the fixed-pipeline model, the router being simulated is a 5-stage virtual channel router with credit-based flow control. The second model, the flexible-pipeline, is more flexible and allows to change the number of pipeline stages in a router. In this work, the fixed-pipeline model is used. The links in the mesh network are set to have a latency of 1 clock cycle and to be 8 B wide. The mesh parameters are summarised in Table 2.6. This is a very basic mesh network, which is not very competitive, especially in terms of latency, as the router is not optimised at all. The bandwidth of the mesh is quite similar to the optical networks: the bisection bandwidth of this mesh is equal to 512 Gbit/s where the optical crossbar, with 16 wavelengths, has a bisection bandwidth of 512 Gbit/s. However, I will discuss how the change to a more optimised mesh network will affect the comparison in Section 4.3.

The randomisation of these networks is done in a slightly different manner as described in Section 3.2.2. As the garnet network is an extensive piece of code, it was prohibitive to implement the variability inside the code representing the links. To

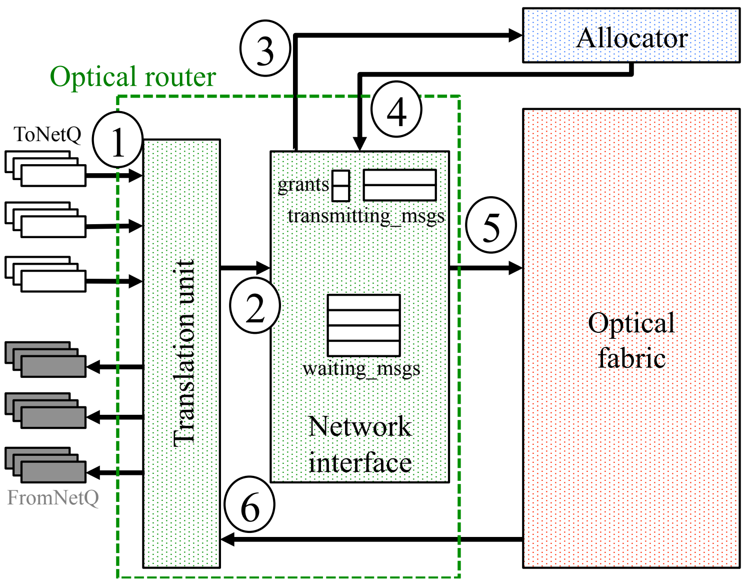


Figure 4.6: Schematic overview of the optical NoC as implemented in gem5.

introduce variability across simulations, the fixed latency of a link is randomised at the start of a simulation. Five links are chosen at random to have a higher or lower latency than their original latency. As the links chosen for randomisation differ per simulation, each simulation will follow a different scheduler path.

4.2.2 Simulation model of the optical networks on-chip

The main gem5 repository does not contain an optical NoC model. For this thesis, I implemented two models: a switched, non-blocking optical NoC and SWMR NoC. This section will mostly elaborate on how the switched optical NoC is implemented. The non-switched NoC is quite similar to the switched case, albeit less complex.

Figure 4.6 gives a general overview of an optical NoC as implemented in gem5. The `Allocator` for example will not be present in a non-switched network. As gem5 is C++ based, all the modules represent C++ objects. All the objects together functionally emulate an optical NoC. Whilst the individual objects in the network bear resemblance to the structures of an optical NoC, they only emulate the behaviour of those structures. They should not be seen as hardware descriptions of those structures.

In both the non-switched (SWMR) and switched network (crossbar) a message will cross the network as follows. The `FromNetQ` contains the buffers that connect the coherence controllers with the NoC ①. There are three sets of three buffers as each tile connects three coherence controllers to the NoC (L1 cache controller, L2 cache controller and memory controller). gem5 uses three virtual networks to prevent coherence deadlock which is why there are three buffers coming from each coherence controller. The `Translation` unit moves the incoming messages to the `NetworkInterface` ②. The `NetworkInterface` will request an optical path for the message, in the case of a switched NoC ③. The `Allocator` will reply with a grant ④. Once the message is ready for transmission i.e. after a grant has been received or immediately in the case of a non-switched NoC, the message will be transmitted optically using the `OpticalFabric` ⑤. After the message has travelled the optical NoC, the `Translation` unit again will move the message from the optical NoC to the `FromNetQ` ⑥, which connect the NoC with the coherence controllers.

An overview of the message flow in the `NetworkInterface` is given in Figure 4.8.

Optical fabric

The optical fabric represents message transmission in the optical domain. The optical fabric is responsible for emulating the timing of message transmission in an optical NoC. It buffers a message until the complete transmission time has passed and will then deliver the message straight to the destination optical router. The transmission time represents the time taken to serialise the message and transmit it across an optical link. The exact delays added by the optical fabric depend on the NoC type. The optical fabric is also responsible for the randomisation as described in Figure 3.10. Again it needs to be stressed that the optical fabric is an abstract concept used for optical NoC emulation and is not a faithful representation of the physical implementation of an optical NoC.

Translation unit

As the name implies, the translation unit is responsible for the translation between the ingresses and egresses of the NoC and the optical parts of the NoC. First of all, it filters incoming messages. It moves intra-tile messages straight from the incoming network queues to the outgoing network queues, without passing the network interface. Inter-tile messages on the other hand are moved to the network interface. It is also important to note that the translation unit converts multicast messages into unicast messages as the optical fabric, as implemented here, cannot handle multicast messages. Lastly, it moves messages from the optical fabric to the desired outgoing network queue.

Network interface

The network interface represents the conversion from the optical to the electrical domain. The exact implementation of the network interface depends on the network type but both types share two important structures as can be seen in Figure 4.6. The first one is a message buffer, called `m_waiting_msgs` in Figure 4.6 which holds messages that are not yet transmitted. This could be because no transmitter is available, or in the case of a switched NoC, the requested circuit has not yet been established. The second structure, `m_transmitting_msgs` is responsible for tracking the occupation of the transmitter. In a real NoC, messages will be buffered until the end of the serialisation. However, in the simulation, messages are moved to the optical fabric as soon as soon as transmission starts. These structures represent the buffers in a real optical NoC.

Allocator

The allocator is only present in the switched NoC. It receives requests and circuit updates from the network interfaces. These updates inform the allocator about messages transmitted on speculative circuits. The allocator uses round-robin allocation: the requester that was serviced last, will have the lowest priority in the next round of allocation [22, Chapter 6.5.1]. Following an allocation round, the allocator informs the network interfaces of the newly established circuits using grants. There are two types of grants: positive grants carry information about a newly established circuit, negative grants inform the network interfaces of a circuit being torn down. Whilst the grants are being transmitted, the crossbar is reconfigured thereby completely hiding the reconfiguration time. The allocator is pipelined, meaning new requests can come in and their allocation process can begin, even when older requests are still being allocated.

Figure 4.7 shows how the latency encountered by a message in an optical NoC is emulated by the different C++ objects discussed previously. Figure 4.7a and Figure 4.7b show the timings of a control and data message respectively in a switched NoC. Figure 4.7c shows the timings of a control message in the case no path arbitration is needed i.e. in a non-switched NoC or when a message in the switched NoC can make use of an existing circuit. It needs to be noted that the timings shown are the minimum possible, they can increase in case of congestion. For example, the allocation latency experienced by a message will be doubled if the associated circuit request can only be allocated in the second allocation round. The non-switched NoC can also suffer from increased latencies if a previous message is still being serialised.

4.2.3 Effect of reduced latency in an optical network on-chip

The message latencies as depicted in Figure 4.7 all consist of two parts: a fixed latency, determined by the message size (serialisation latency) and the time of flight and a variable latency. The variable latency is not determined by the message itself but by the state of the NoC. This becomes clearer when looking at Figure 4.8, which depicts the decision process once a message has been injected in the network interface. This decision process is similar for both the switched and non-switched NoC, with the exception that in the non-switched case, there will always be a circuit to the destination (SWMR). The time between the injection of a message in the network interface and the start of serialisation can be seen as the *waiting latency* as the message is being buffered in anticipation of message transmission. This waiting latency is variable: as long as the transmitter is busy for example, the message is waiting. This variability is due to other messages in the system: the transmitter could be busy or the requested circuit cannot be immediately granted due to destination congestion.

Section 4.1 showed the positive effect of reducing the message latency. To reduce the message latency in optical NoC, there are two avenues: either the fixed latency is reduced (by reducing the serialisation latency) or the waiting latency can be reduced. In this section, these two possibilities are explored for both network types.

Effect of reduced serialisation latency

The work in this section is an extension of the work presented by Van Laer et al. [11]. The serialisation latency in an optical NoC is the time needed by the transmitter to convert a message from the electrical domain to the optical domain. The serialisation latency is given by the following equation (keeping in mind that channel is defined as a connection between source and destination):

$$\text{serialisation latency} = \frac{\text{message size}}{\text{modulation speed} \times \text{\# wavelengths per channel}} \tag{4.1}$$

Equation (4.1) shows the serialisation latency can be increased by increasing the modulation speed or by increasing the number of wavelengths per channel. In the next section the effect of an increasing number of wavelengths will be investigated, both for the switched NoC and the non-switched SWMR NoC. The effect of increased modulation speed will be, in essence, the same as an increase in the number of wavelengths and not be investigated. Currently, the modulation speed is limited by the electrical circuitry driving the transmitter but this could change in the future. The number of wavelengths per channel on the other hand is a characteristic of the optical NoC as a whole.

The number of wavelengths per channel will affect multiple characteristics of the

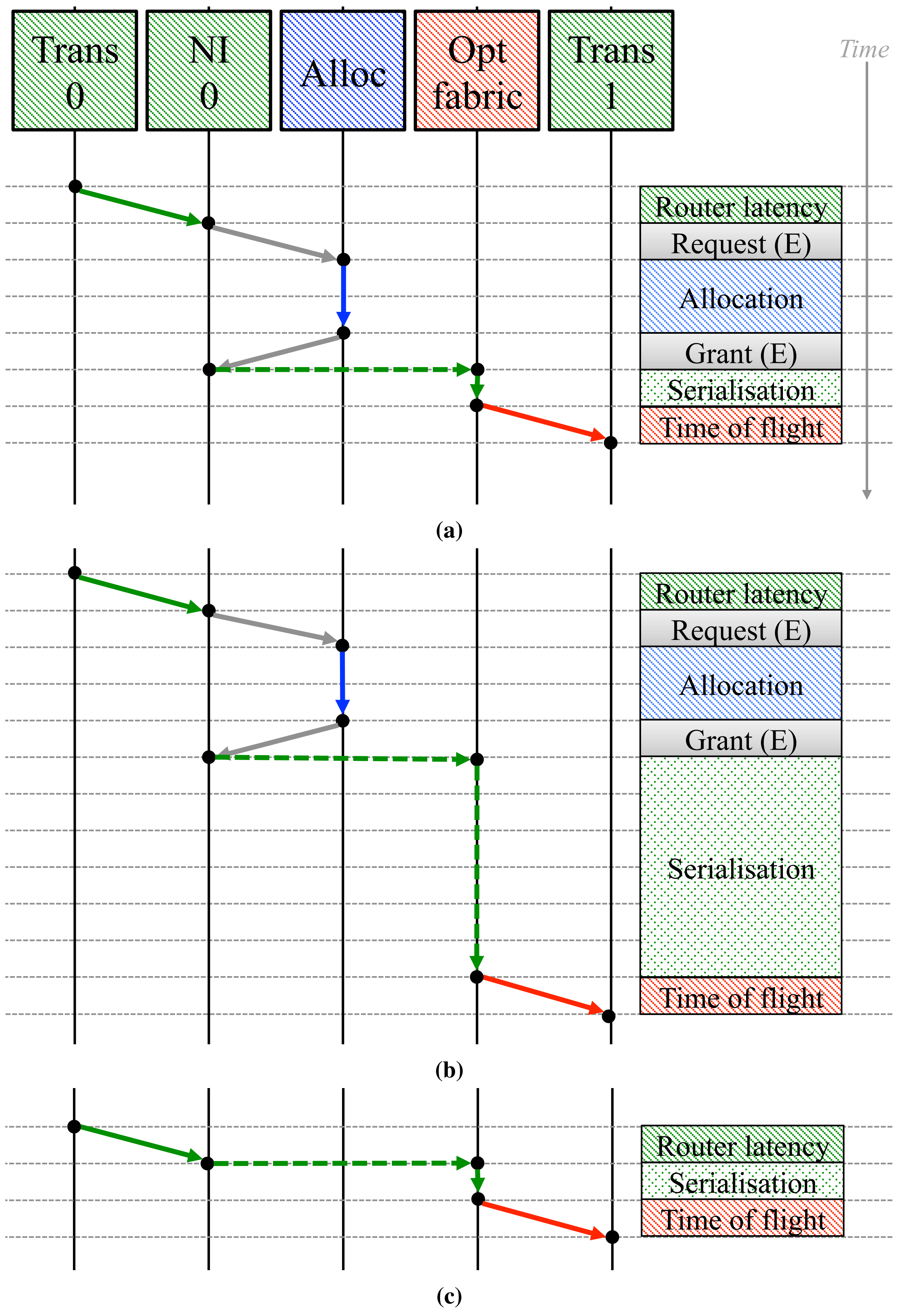


Figure 4.7: Timing diagrams depicting the latency a message encounters in an optical NoC and how they are emulated in gem5. Time is depicted on the y-axis, whereas the x-axis depicts the various elements in the emulation of the optical NoC. Diagram assumes 8 wavelengths in the stripe (a) transmission of a control message in a switched NoC (b) transmission of a data message in a switched NoC (c) transmission of a control message in a SWMR NoC.

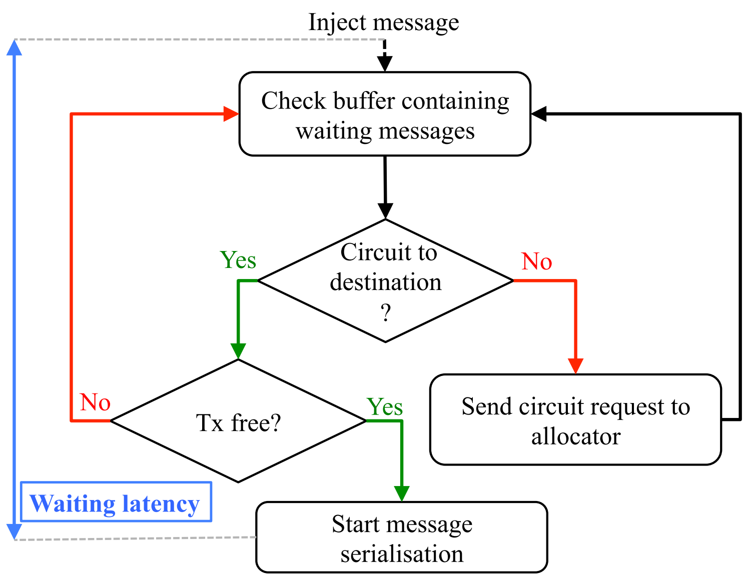

Inject message
Check buffer containing waiting messages
Circuit to destination ?
Yes
No
Tx free?
No
Yes
Send circuit request to allocator
Waiting latency
Start message serialisation

Figure 4.8: Flow diagram depicting the decision process in the network interface

transmitter and the optical crossbar (in the case of the switched NoC). First of all, an increase in the number of wavelengths per channel will increase the required optical signal power. In the transmitter for example, the optical signal will pass by $w-1$ passive microrings where w is the number of wavelengths in the channel. Each of these rings adds between 0.0001 dB and 0.1 dB loss to the optical path, depending on the state of the microring (off-resonance or on-resonance, respectively) (Table 2.3). The optical path loss in the crossbar (in case of a switched NoC) will also increase with increasing number of wavelengths, the extent of which is determined by the exact crossbar layout. Secondly, the electrical power consumed by the transmitter will also increase when increasing the number of wavelengths. The total electrical power of the transmitter is the sum of the trimming power, the signal modulation power, the Serialisation/Deserialisation (SERDES) power and local transport power [72]. Trimming is needed to correct for the temperature dependence of silicon as to keep the resonant frequency of a microring resonator stable (Section 2.2.3). The power associated with this is shown to be non-linear [83]. Nitta et al. investigated the effect of the modulation speed and the number of wavelengths per channel, when aiming to minimise power consumption and to keep the target bandwidth constant [72][159]. Increasing the number of wavelengths per channel is a more efficient option, in terms of modulation power and SERDES power, than an increase in modulation speed as the latter options leads to an increase in switching activity which, in turn, will lead to an increase in dynamic power. However, only increasing the number of wavelengths whilst keeping the modulation speed constant (thereby increasing the link bandwidth) will lead to an increase in modulation and SERDES power. The transport power is the power required to carry the electrical signals that drive the microring resonators from the router control unit to the actual location of the microrings. This should be included in the total power consumption as these links can be relatively large (due the difference in scale between the microring resonators and the electronics, see Figure 2.14). An increase in number of wavelengths per channel increase the length of this link and, hence, the power consumption, even more so when repeaters are needed. Because of the impact on power consumption, the number of wavelengths per channel should be kept as low as possible and not be over-designed.

Figure 4.9 shows the effect of increased wavelengths for both network types. Figure 4.9a to Figure 4.9d show the performance (in IPC), averaged over multiple runs. These figures also show the confidence intervals for every wavelength value. In order to correctly assess the effect of the switched and the non-switched NoC, their respective confidence intervals should not overlap. First of all, it is clear ***canneal*** behaves oddly. This might be due to its aggressive synchronisation strategy which depends on data race recovery[3] [138]. The average values as shown in Figure 4.9b are the results of more than 25 runs per wavelength value for each network type (to satisfy the sample size requests given by Equation (3.3)). The variation across this set of runs is due to different scheduler decisions. One can, therefore, assume the run with the highest performance (for a certain wavelength value) had the most optimal scheduler decisions. By only comparing the runs with optimal thread scheduling, one can argue the effect of the scheduling is removed and only the NoC differences are compared[4]. By replotting performance using the highest obtained IPC (rather than average IPC), the results for ***canneal*** are completely in line with the other benchmarks as shown in Figure 4.9f. There are some conclusions that can be drawn from these figures. First of all, the optical NoCs always outperform the electrical mesh network, as long as the optical NoCs provide 2 or more wavelengths per channel. Secondly, the effect of an increasing number of wavelengths per channel levels off after 16 or more wavelengths. This can be explained by the fact that the majority of all messages are control messages. The serialisation latency of a control message reaches the lowest possible value (a single clock cycle) once 8 wavelengths are provided. Any additional wavelengths above 8 wavelengths therefore only add bandwidth to the NoC, which does not affect performance significantly which confirms the findings in Section 4.1.3. The non-switched NoC also outperforms the switched NoC when both NoCs provide the same number of wavelengths per channel.

To get a more general view on how the optical NoCs compare, Figure 4.10 shows the performance in terms of speedup, against the bisection bandwidth. The speedup is calculated using the average performance, with the exception of ***canneal*** in which case the highest performance is used[5]. The bisection bandwidth is defined as the minimum bandwidth between two partitions in the NoC, assuming there is an equal number of network nodes in each partition [42, Chapter 3.1]. As the optical NoCs are symmetrical, the bisection bandwidth is given by the following equation:

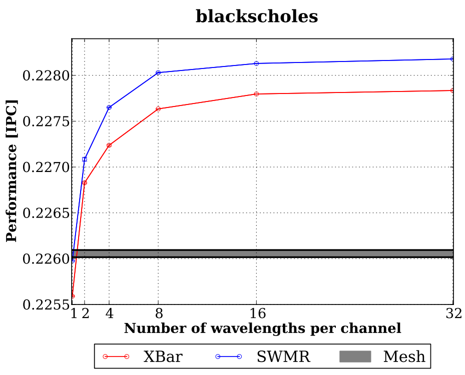
$$\begin{aligned} \text{bisection bandwidth} &= n_C \times \text{bandwidth per channel} \\ &= n_C \times \text{modulation speed} \times \text{\# wavelengths per channel} \end{aligned} \tag{4.2}$$

where n_C is the number of channels across the partition. In a SWMR network n_C is equal to $2 \times (N-1)$ where N is the number of network nodes. In the switched, crossbar-based network n_C is equal to $\frac{N}{2}$. The SWMR network will therefore have a higher bisection bandwidth than the crossbar, when both have the same bandwidth per channel. Figure 4.10 shows that at lower bisection bandwidths, the switched NoC outperforms

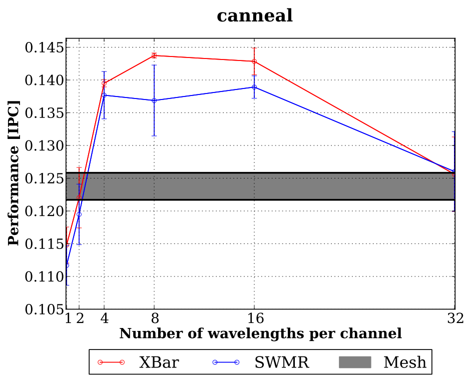
[3]A data race occurs when two threads try to access the same memory location concurrently and at least one of the accesses is a write. There is no synchronisation mechanism in place to prevent the race from occurring and a data race can change the program flow [1, Chapter 2.11].

[4]This does not necessarily mean the optimal scheduler decisions are the same for each wavelength value. The optimal decisions most likely depend on the state of the NoC.

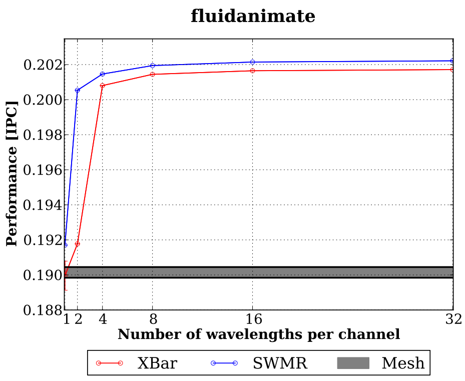
[5]This is done to exclude the effect of thread scheduling, see Section 4.1.1



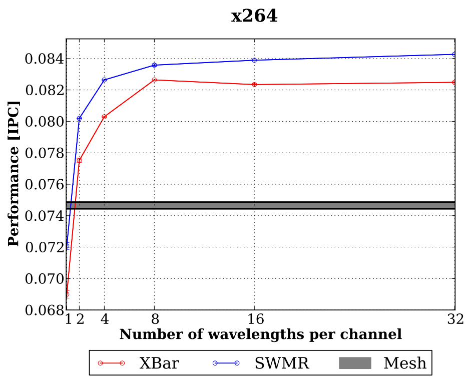
(a)



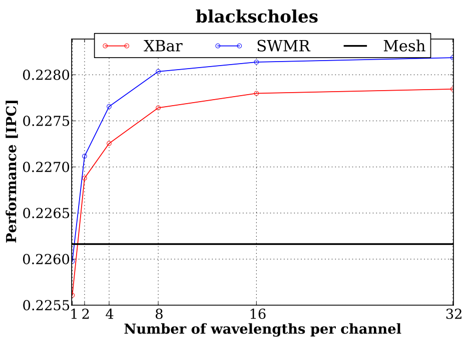
(b)



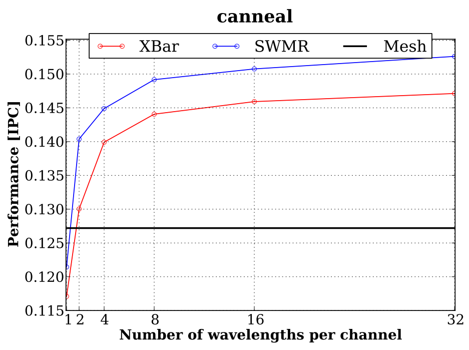
(c)



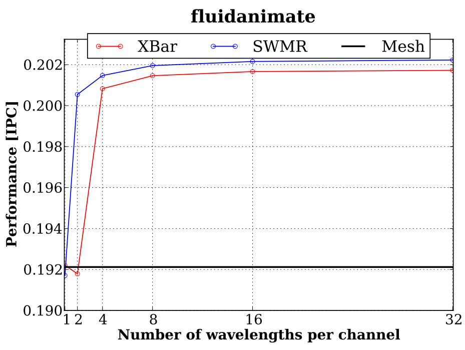
(d)



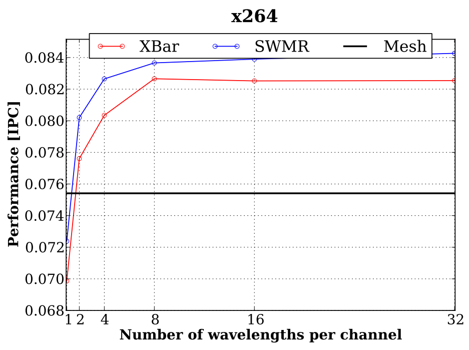
(e)



(f)



(g)



(h)

Figure 4.9: Effect of increased number of wavelengths for a crossbar-based NoC and a SWMR NoC with one transmitter per tile, compared to an electrical mesh. (a) - (d) show the average performance (in IPC), where the brackets indicate the confidence intervals. (e) -(h) show the best possible performance (in IPC).

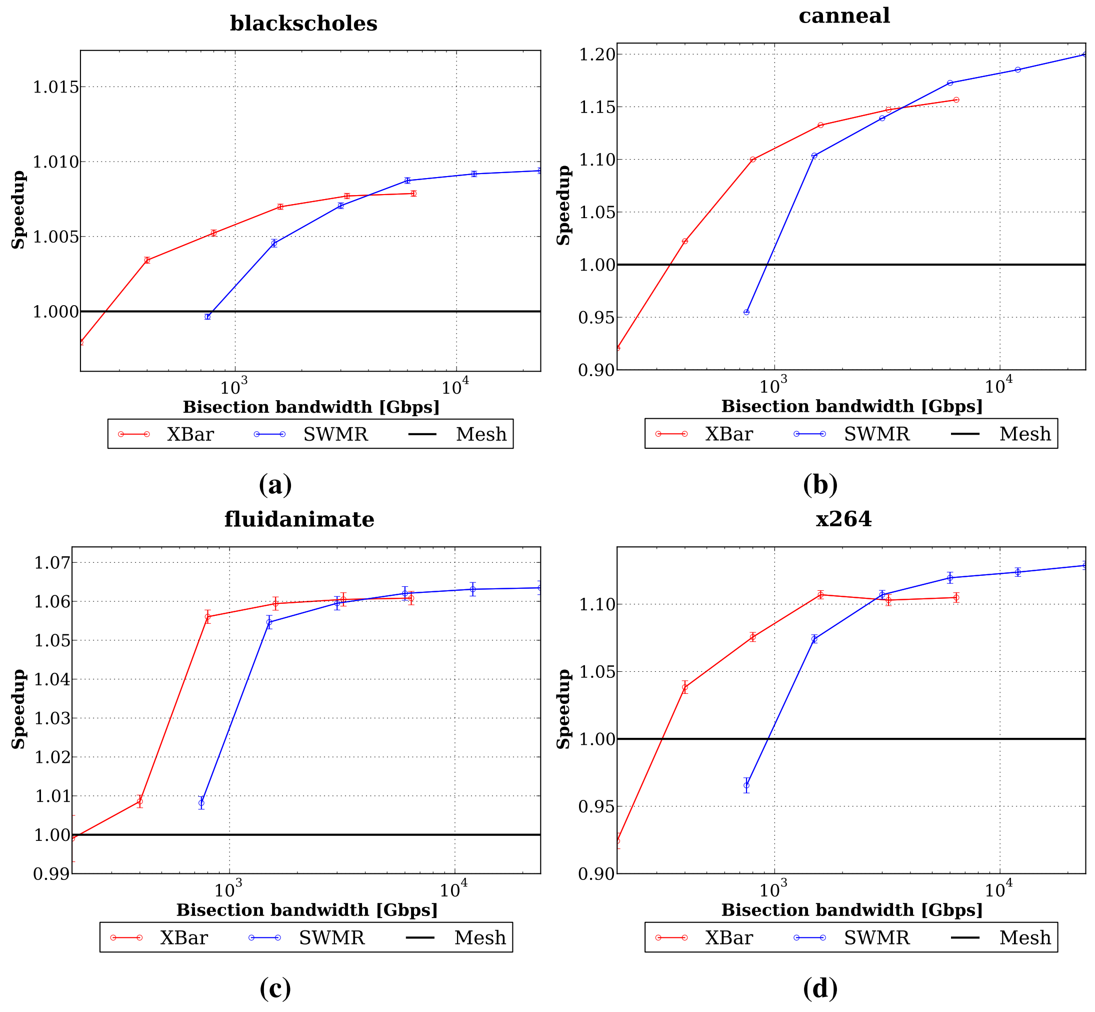
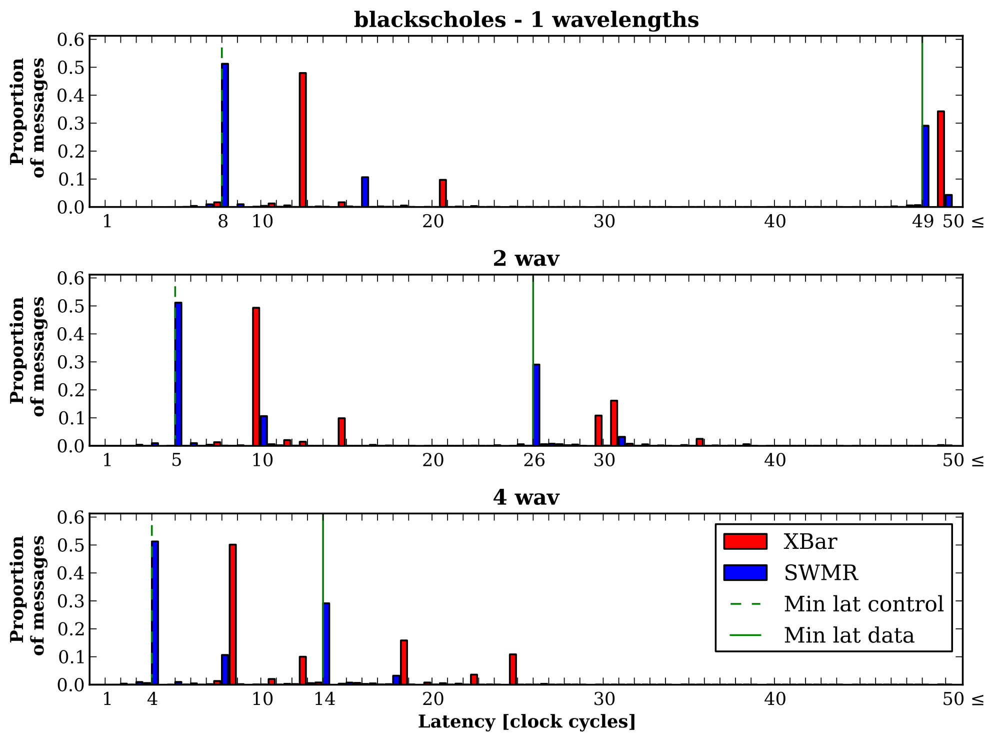


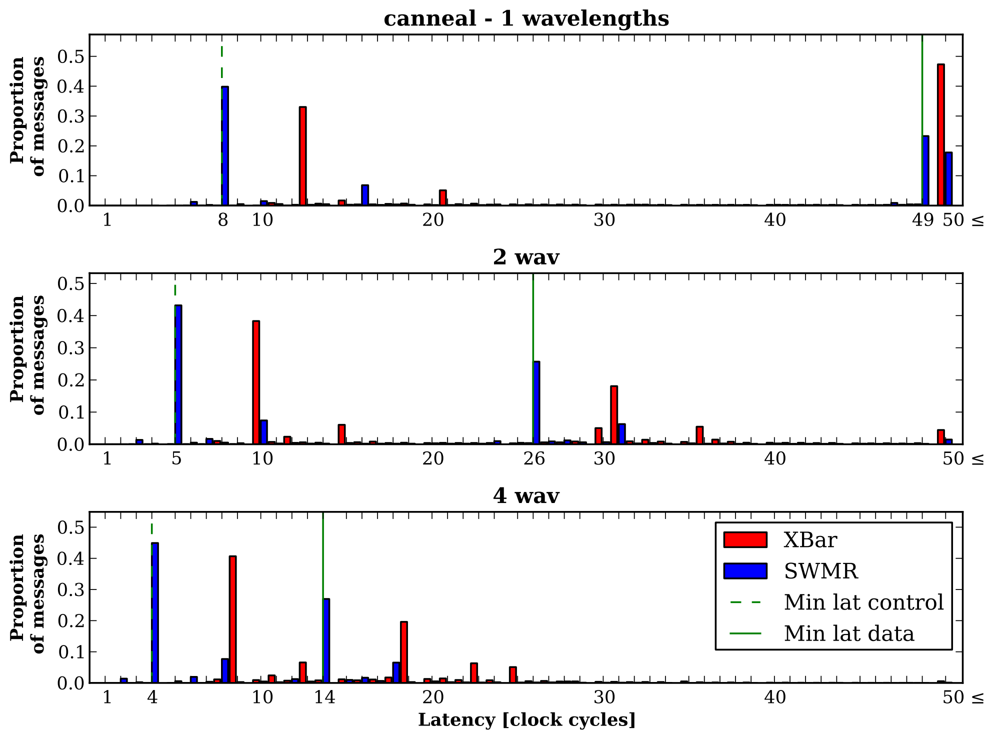
Figure 4.10: Effect of increased number of wavelengths for a crossbar-based NoC and a SWMR NoC with one transmitter per tile, normalised to an electrical mesh, plotted against bisection bandwidth. Brackets indicate the confidence intervals. The y-axes of the various plots differ as the effect on speedup differ significantly across benchmarks.

the non-switched NoC because it provides more bandwidth per channel. Again, the better performance is not due to the higher channel bandwidth in itself but because a higher channel bandwidth provides a lower serialisation latency. However, at increased bisection bandwidths, the non-switched NoC outperforms the switched NoC: both have high channel bandwidths and, hence, low serialisation latencies so the non-switched NoC provides the advantage that there is no path setup required. Figure 4.10 also confirms again benchmarks with a low operational intensity (***canneal*** and ***x264***) are more affected by a change in the NoC. This becomes clear when looking at the scale of y-axes across the various benchmarks: the speedup in the case of ***canneal*** varies between 0.9 and 1.2, whereas the range in ***blackscholes*** is significantly smaller (0.995 to 1.010). All benchmarks have a slowdown rather than speedup compared to the electrical mesh at the lowest bisection bandwidth. However, ***canneal*** and ***x264*** suffer more from this (more than 5% slowdown). The gain to be had from an optical NoC is also much higher for these benchmarks: around 20% (speedup of 1.2) for ***canneal*** compared to less than 1% (speedup of 1.01) for ***blackscholes***.

As explained at the beginning of this section, there are two components in the



(a)



(b)

Figure 4.11: Message latency distribution in optical NoC with one transmitter per tile. Increasing number of wavelengths removes the tail composed of messages with a very high waiting latency due to contention. Only ***blackscholes*** and ***canneal*** are shown for brevity.

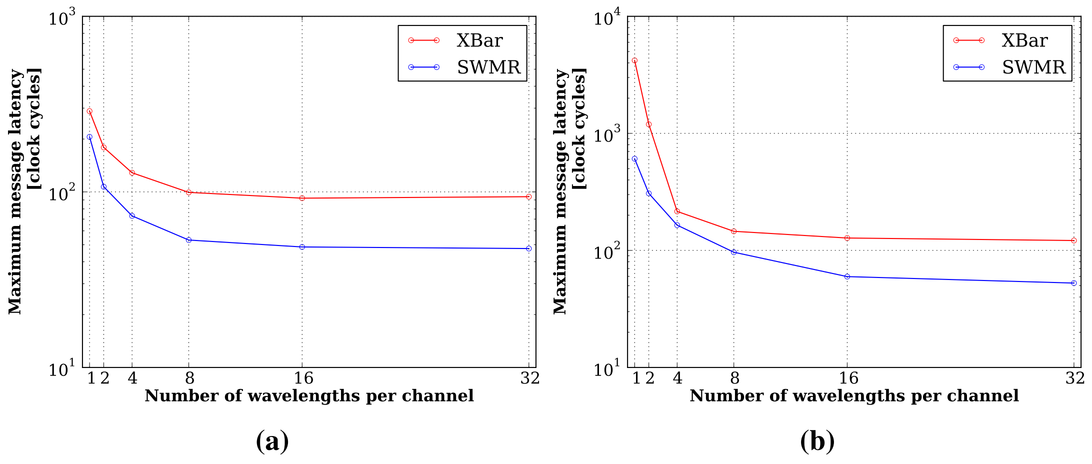


Figure 4.12: Maximum message latency for a crossbar-based NoC and a SWMR NoC with one transmitter per tile for (a) ***blackscholes*** and (b) ***canneal***.

message latency: the fixed component (serialisation latency) and the variable component (the waiting latency). Figure 4.11 shows the distribution of the message latency for optical NoCs with 1, 2 and 4 wavelengths. The green lines denote the minimum message latency of control and data messages i.e. when the message did not have to wait before transmission. Let's first look at the SWMR case. The highest peak for the SWMR NoCs (blue bars in Figure 4.11) corresponds to control messages that could be transmitted immediately after they were injected in the NoC (dotted green line). The height of this peak differs slightly between ***blackscholes*** (Figure 4.11a) and ***canneal*** (Figure 4.11b) because of differences in the traffic patterns and the overall ratio of control to data messages varies slightly across benchmarks[6] . The second highest peak on the other hand corresponds to the data messages with the minimum latency. There are smaller peaks which correspond to messages that were queued. For example, in the single wavelength case there is a peak around 16 clock cycles: these are control messages that had to wait for the serialisation of an earlier control message to finish. The switched NoC (red bars) show similar behaviour, with the difference that there are additional peaks corresponding to messages that had to go through allocation, for example the peak at 12 clock cycles in Figure 4.11a.

However, it is interesting to look at the last peak of the distribution for both cases. This peak corresponds to all messages with a higher latency than 50 clock cycles. Especially in the single wavelength case these peaks (for both NoC types) are quite pronounced. In the case of the switched NoC, this is partially due to data messages that had to undergo allocation before message transmission could start. However, the length of the tail of the distribution should not be underestimated. As shown in Figure 4.12 the maximum message latency encountered can be quite high at a low number of wavelengths per channel. The high message latency is due to a high waiting latency: messages had to wait until the transmitter became available and/or until the circuit had been setup. At a low number of wavelengths per channel, the transmitters are occupied for a long period of time due to the low serialisation speed. The maximum latencies encountered are consistently higher in the case of the the switched NoC where messages both have to ensure both the availability of a transmitter and a circuit to the destination (as shown in Figure 4.8). Considering the effect of the waiting latency on the maximum

[6]The data to control message ratio lies around ∼30% for all PARSEC benchmarks.

latency, the next section will look into one possible way of reducing the waiting latency, namely by providing multiple transmitters per tile.

Effect of reduced waiting latency

As shown in Figure 4.8, the waiting latency will be (partially) determined by the availability of a channel. In this section, the effect of 2 channels per tile will be investigated. The second channel should reduce the amount of source contention where the source has multiple messages to transmit, each to a different destination. This occurs for example in the case of multicast messages. In a system with two channels per tile, each tile can now setup two independent lines of communication: two connections to two different destinations. The amount of outgoing channels per tile and number of incoming channels per tile is chosen to be identical in this work. This is done to prevent destination contention and to keep the network symmetric. Adding a second outgoing channel to each tile will therefore affect the total number of transmitters and receivers twice (Table 2.7).

Adding such a second network layer will increase the complexity of the NoC significantly. A network layer in this case is defined as the set of channels that allows each transmitter to setup one connection. For example, the schematic depictions of the SWMR scheme and the crossbar scheme as shown in Figure 2.16 and Figure 2.24 respectively, depict one network layer. In the case of the switched NoC, the allocation process will be more complex as the allocator now oversees multiple network layers. These network layers are logically identical but when assessing a request, the allocator now needs to loop over all layers to find a suitable circuit.

The exact physical implementation of a second network layer is not investigated in this work. It could be done, for example, by placing a second optical layer. It could also be done by assigning a subset of the wavelengths to each network layer. So, for example, in the single channel system, each channel would consist of 8 wavelengths. In the dual-channel system, each channel would consist of only 4 wavelengths but each tile has two independent channels at its disposal.

Figure 4.13 shows the addition of a second channel to each tile does increase performance but only for the non-switched NoC. This makes sense as a second transmitter per tile will reduce the waiting latency by increasing the chance a transmitter is available. However, in the case of switched NoC the waiting latency is affected by the allocation process as well, not only the availability of a transmitter. When looking at Figure 4.13, it is indeed hard to judge the effect of a second channel on the switched NoC as the confidence intervals overlap in almost all cases. The effect is also more pronounced for low numbers of wavelengths per channel, where the channels are occupied for longer stretches of time due to higher serialisation latency.

Figure 4.14 shows the speedup plotted against the bisection bandwidth. Both the single and dual-channel system have the same number of wavelengths per channel. In the case of the two-channel system, this doubles the bisection bandwidth according to Equation (4.2) as n_C, the number of channels crossing the partition doubles. Figure 4.14 shows the increase in bisection bandwidth (and associated network complexity) voids the performance benefits provided by the second channel.

The addition of a second channel does reduce the waiting latency as evidenced by Figure 4.15 (only the results of ***canneal*** are shown for brevity). First of all, the last peaks corresponding to messages with a high message latency decrease. The effect

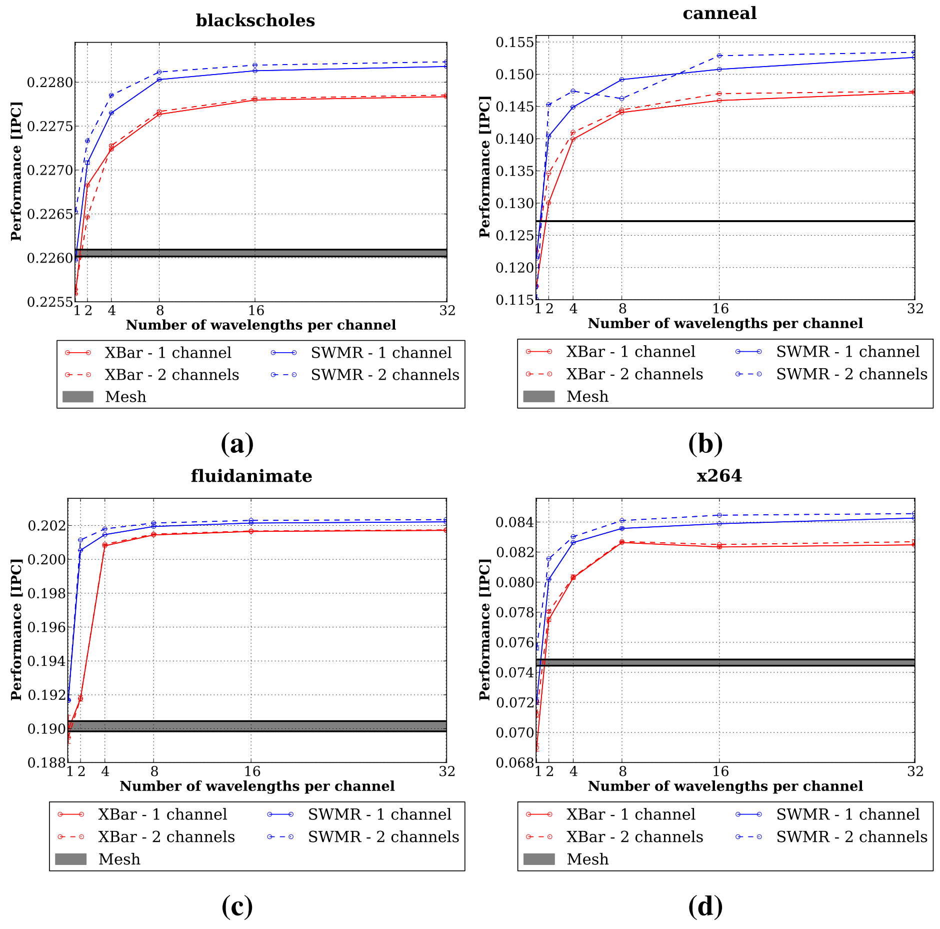


Figure 4.13: Effect of multiple channels on performance (in IPC) for a crossbar-based NoC and a SWMR NoC. Only the best performance is given for ***canneal*** in (b)

on the SWMR NoC is also more pronounced (blue bars). A peak at 16 clock cycles in a system with 1 wavelength for example corresponds to a control message waiting until the previous control message has been serialised. This blue peak is present in the single channel system (Figure 4.11b) but has almost vanished in the dual-channel system (Figure 4.11b) The maximum message latency encountered in the system is also reduced by the addition of a second channel as shown in Figure 4.15b.

However, overall, it can be concluded that multiple channels per tile do not provide a gain in performance that would outweigh the increased complexity of the NoC as a whole.

4.3 Conclusion

In this chapter, I set out the answer the first research question of this thesis: can an optical NoC outperform an electrical NoC? The answer is *yes*, provided the serialisation latency of the optical NoC is kept at bay by providing enough wavelengths per channel. In the systems compared in this thesis, a mesh NoC (consisting of 5-stage routers and links which are 8 B wide and have a link latency of a single clock cycle) is outperformed by optical NoCs with one channel per tile and at least 2 wavelengths per channel. The exact crossover point will be determined by the interplay between the CMP, the

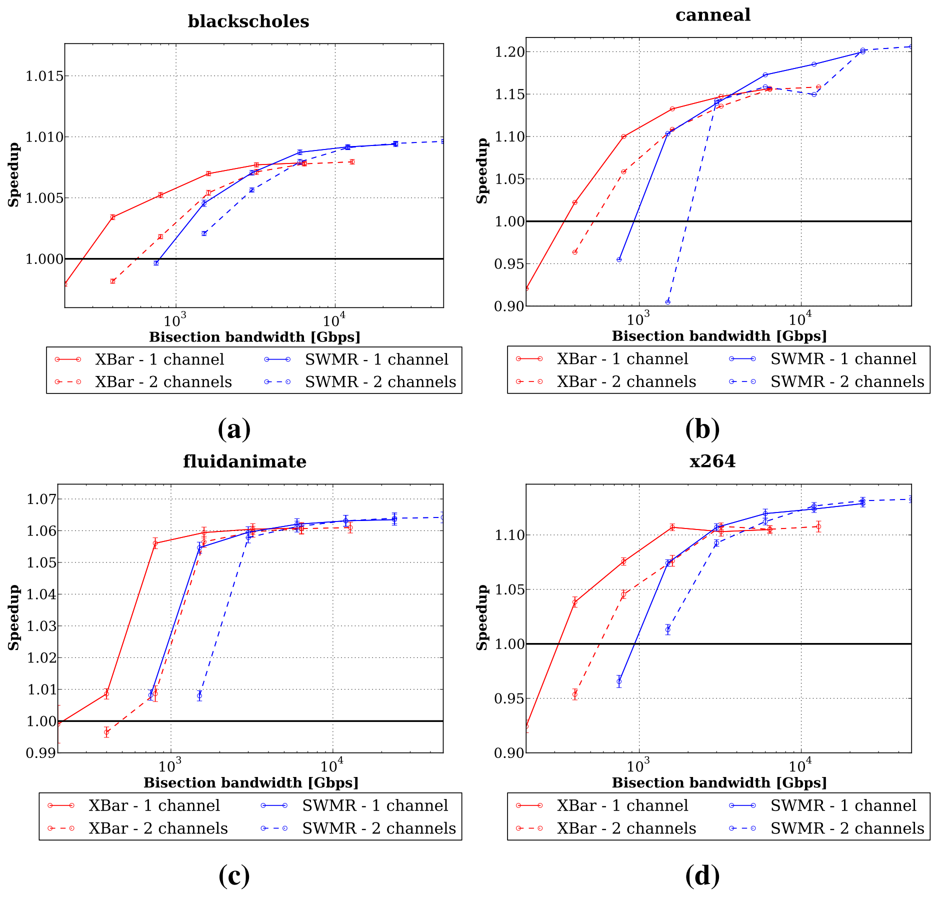
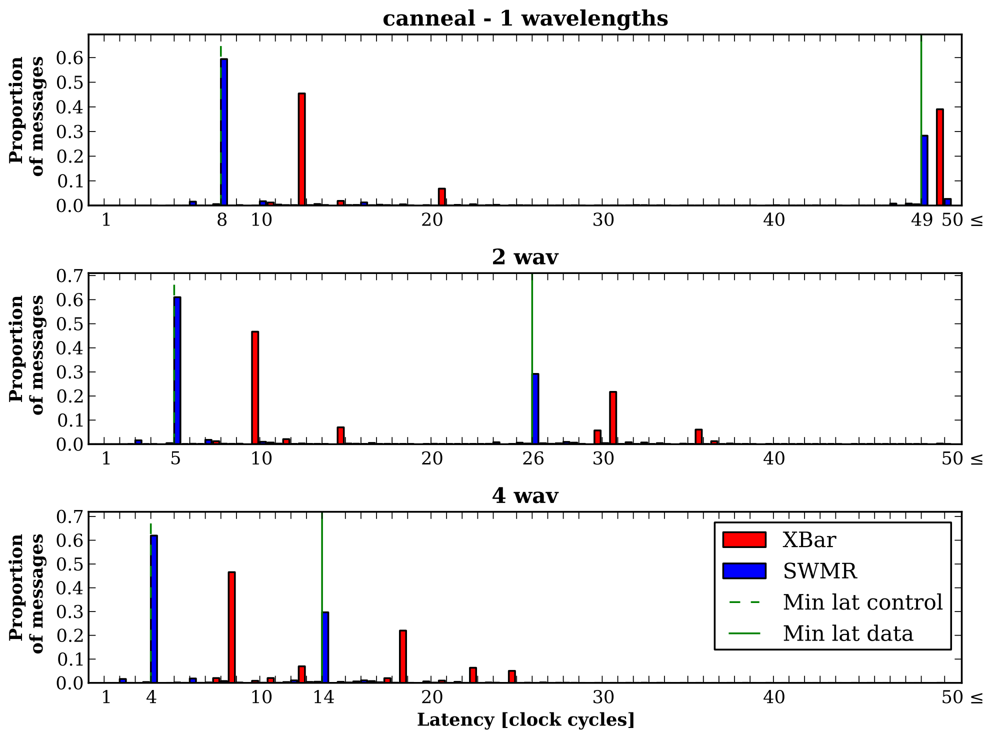
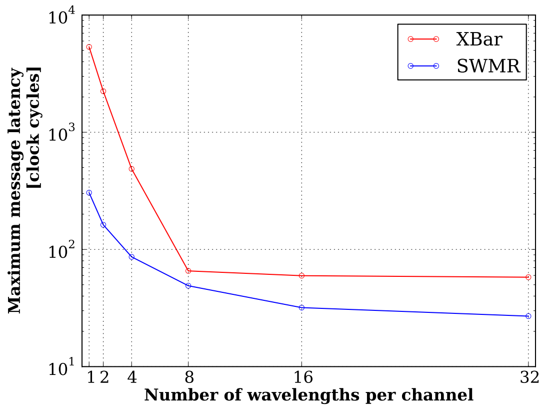


Figure 4.14: Effect of multiple channels per tile for a crossbar-based NoC and a SWMR NoC on speedup, normalised to an electrical mesh, plotted against bisection bandwidth. Brackets indicate the confidence intervals. Only the best performance is given for ***canneal***.

electrical mesh and optical network characteristics. An earlier version of this work [11] for example, compared an 8-core CMP connected via a mesh network in which the links were 16 B wide with the same CMP interconnected using an optical crossbar-based NoC. In this case, 4 or more wavelengths per channel were needed. The crossover point in terms of wavelengths per channel was higher in this case as the disadvantageous effect of hop count on message latency is less pronounced for a 8-node mesh than for a 16-node mesh. The mesh network (Table 2.6) which was used as the comparison network in this chapter was a very standard mesh network: the links were fast (1 clock cycle) but have a relatively small bandwidth (8 B). The routers were also not optimised and had 5 pipeline stages, whereas there is a significant body of work focused on reducing the number of pipeline stages [47], [115], [116]. To investigate the effect of a more competitive network, the mesh network was optimised by first reducing the number of pipeline stages (by changing to the flexible garnet network in gem5), increasing the link bandwidth and increasing the buffer size. Table 4.1 holds the parameters of the optimised meshes, where the red font is used to indicate differences with the original mesh network. The optimised mesh uses the flexible garnet work. In Figure 4.16 the effect of these optimisations on performance is shown. Optimising the mesh networks speeds up the benchmarks, where the improvement for ***canneal*** is most significant. An optimised mesh network will also push the crossover point to the right: the optical NoCs need to provide more wavelengths to outperform the mesh network. This is shown



(a)



(b)

Figure 4.15: Effect of multiple transmitters per tile on message latency (a) Message latency distribution (b) Maximum message latency. Only ***canneal*** is shown for brevity.

in Figure 4.17, in which it becomes clear at least 8 or more wavelengths need to be provided for the optical NoCs to outperform the optimised mesh network, whereas, in the case of an unoptimised mesh network only 2 wavelengths had to be provided.

Secondly, the effect the number of wavelengths per channel in an optical NoC has on performance confirms the findings of the first section in this chapter: the latency of an optical NoC will affect performance to a higher degree than its bandwidth. Whilst increasing the number of wavelengths per channel does increase the channel bandwidth, it also decreases the serialisation latency. Providing more than 16 wavelengths per channel does not affect performance significantly showing that an increase in channel bandwidth is only useful when it comes with a decrease in serialisation latency. The serialisation latency of control messages, which form the majority of the network traffic, reaches its minimum value of a single clock cycle once each channel consists of 8

Network frequency	2 GHz
Number of nodes	16
Topology	Mesh
Number of rows in mesh	4
Terminal nodes per network interface	3 (tiled architecture)
Routing algoritm	Deterministic, dimension ordering
Link latency	1 clock cycle
Link bandwidth	8 B
Router pipeline stages	1
Buffer size	Infinite
Virtual channels	4 per virtual network

Table 4.1: Parameters used for the optimised electrical network (Flex+) with which the optical proposals will be compared.

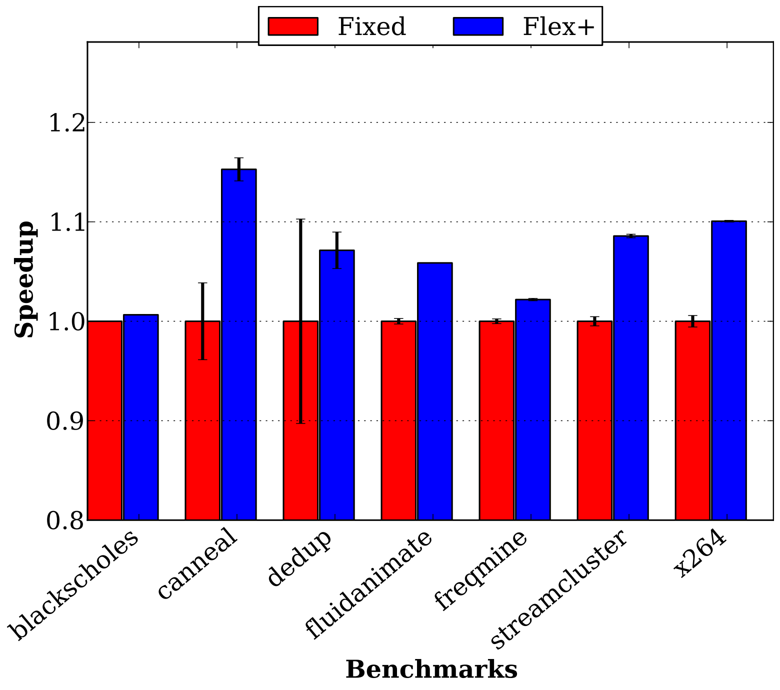
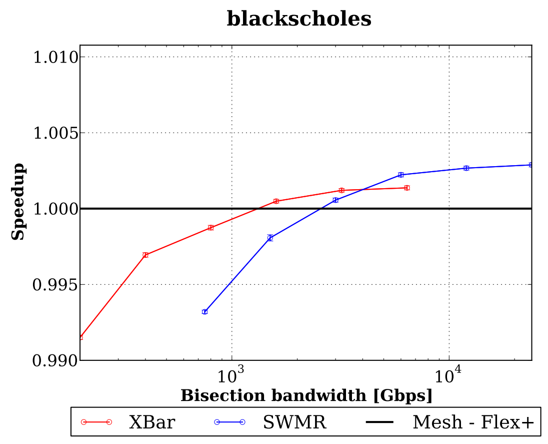
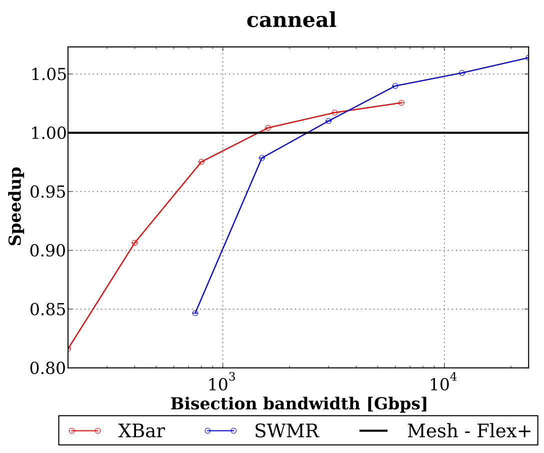


Figure 4.16: Effect of optimising a mesh network by increasing the bandwidth and reducing the router latency. The parameters of the mesh networks can be found in Table 4.1.

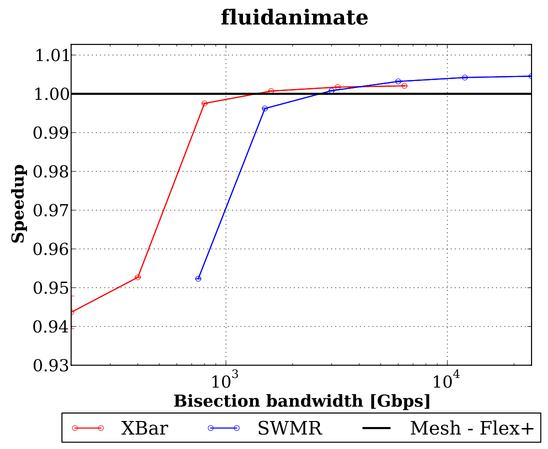
wavelengths. This also ties in with the finding that increasing the number of channels per tile does not affect performance significantly. Increasing the number of channels, whilst keeping the number wavelengths per channel constant, increases the bandwidth of the network. Providing two channels per tile was done to reduce the waiting latency of a message. However, the trade-off between channel count versus waiting latency yields far less improvement in terms of performance than the trade-off between wavelength count versus serialisation latency. The latter is completely deterministic: increasing the wavelength count will lead to a lower serialisation latency (assuming the minimum is not yet reached). However, the effect of the first trade-off is far less clear cut as



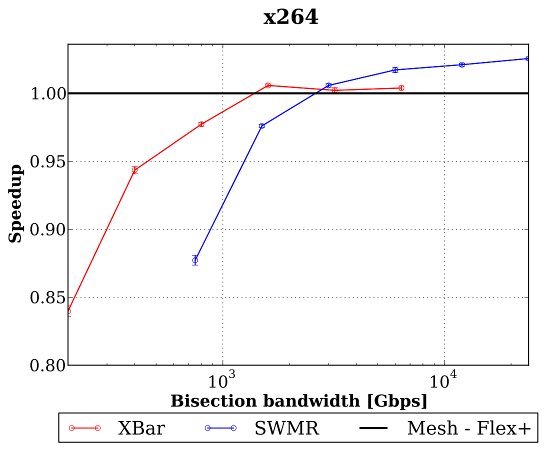
(a)



(b)



(c)



(d)

Figure 4.17: Effect of increased number of wavelengths for a crossbar-based NoC and a SWMR NoC with one transmitter per tile, normalised to an optimised electrical mesh, plotted against number of wavelengths per channel. Brackets indicate the confidence intervals. The y-axes of the various plots differ as the effect on speedup differ significantly across benchmarks.

it depends on the traffic: the second channel will only be used if the first is already occupied by another message.

This chapter also compared a non-switched optical NoC (SWMR) with a switched optical NoC (crossbar). When both networks have the same number of wavelengths per channel, the non-switched NoC leads to a better performance. This is because whilst the serialisation latency in both networks is the same, the waiting latency is not. Messages in the switched NoC need to go through a path setup phase, thereby, spending more time in the buffers. However, when both network types have the same bisection bandwidth, the switched NoC performs better as for the same bisection bandwidth it can provide more wavelengths per channel, thereby, lowering it serialisation latency. The bisection bandwidth can be seen as a substitution for the complexity of the optical routers. A higher bisection bandwidth will lead to more wavelengths per channel. An increased number of wavelengths leads to a higher number of transmitters and receivers in the routers which will affect the power consumption. However, it needs to be noted that in the future, when higher modulation speeds might be more attainable, this might no longer hold as at high bisection bandwidths the SWMR outperforms the crossbar. Therefore, in general, the crossbar-based NoC outperforms the SWMR

NoC if both have the same number of transmitters and receivers in the optical routers. The switched NoC in this thesis is based on an optical crossbar. The exact number of microrings in the crossbar can differ per implementation but depends on the number of wavelengths per channel and should not be neglected. The microrings in crossbar will need trimming, just like the transmitting and receiving rings, which is a power hungry process. However, the resonant frequency of the crossbar rings needs to be shifted far less frequently , in comparison to the transmitter rings which constantly shift in and out resonance to modulate the signal. The power consumption of the crossbar, therefore, will also depends on the number of wavelengths per channel, but to a lesser degree than the transmitter structures. The exact dependency though will be determined by the implementation, which is out of the scope of this thesis.

The non-switched NoC performs better (at higher bisection bandwidths) because it has a lower waiting latency than the switched NoC. The messages in a switched NoC are buffered whilst the optical path is being setup. The next chapter will look at techniques that draw from knowledge of the coherence protocol, to setup paths before messages arrive, therefore, lowering the waiting latency of messages in a switched NoC.

5 Use of Prediction in Switched Optical Networks On-Chip

IN the previous chapter it was shown that the latency a message experiences in an optical Network On-Chip (NoC) consists of two parts. The serialisation latency and time of flight form the first part. This part of the message latency is fixed and determined at design time. The second part of the message latency is the waiting latency which is variable. This is the time a message spends in the buffers, awaiting transmission. The waiting latency in a non-switched NoC is generally lower as there is no lengthy path setup process to go through as is the case in switched NoCs. This chapter investigates whether it is possible to reduce the waiting latency of a switched NoC to that of non-switched NoC by using message prediction methods that draw on knowledge of the coherence protocol.

Even though this thesis is focused on optical NoCs, the methods proposed in this chapter can be used in any type of switched NoC in which there is a centralised allocation process. Most works assume electrical crossbars are not scalable (Section 2.2.1). However, by carefully designing the crossbar and allocator, some works (for example by Passas et al. [160]) have pushed the node count of electrical crossbars beyond 100 nodes. The techniques presented in this chapter are equally applicable to an electrical crossbar.

Section 5.1 will first discuss the methodology used in this chapter which differs from the one used in the previous chapter. Once the methodology has been established, Section 5.2 will look at a simple path setup strategy in which paths are not immediately broken down after usage. Section 5.3 introduces a coherence-based predictor, which is optimised in Section 5.4. In Section 5.5 the waiting latencies that can be obtained by using these strategies in an optical crossbar-based NoC are then compared with those of a Single Writer, Multiple Reader (SWMR)-based NoC.

Network frequency	2 GHz
Number of nodes	16×3
Topology	Pt2Pt
Terminal nodes per network interface	1
Link latency	15 clock cycle
Link bandwidth	72 B

Table 5.1: Parameters used for creating the trace

The work in this chapter is an extension of the work presented by Van Laer et al. [12].

5.1 Trace-Based Methodology

The metric used to study the efficiency of various path setup strategies, is the waiting latency (as introduced in Section 4.2.3). The optimal path setup strategy will reduce the waiting latency as much as possible. The waiting latency is the only part of the total message latency that will be affected by the path setup strategy which is why this the preferred figure of merit in this chapter. I will investigate the various path setup strategies using a trace-based NoC simulator. The traces are obtained by running the PARSEC benchmarks on a Chip Multiprocessor (CMP) with an ideal NoC: a Point-to-Point (Pt2Pt) network, which directly connects all coherence controllers to avoid interference of the topology on the traces. The links have a link latency of 15 clock cycles and a bandwidth of 72 B. The system architecture remains exactly the same as summarised in Table 2.5 but the NoC parameters are summarised in Table 5.1.

This latency is in line with the average message latency in a non-switched optical NoC with 8 wavelengths per channel when simulated in gem5, as can be seen in Table 5.2, albeit slightly higher to account for the fact that the traces will also be used for a switched NoC simulation, which has a higher message latency. The link latency also aims at making the NoC as neutral as possible so the traffic captured in the trace is as realistic as possible. If the NoC is too efficient (low latency - high bandwidth), the traces might, in essence, contain the traffic of a CMP without a NoC. A simulation using such a trace might report stress on the network as there is no delay (bar the controller latencies) between coherence messages in a coherence transaction, which is unrealistic. If the NoC were to have a high latency and low bandwidth, the traffic captured by the traces might be that of a CMP in which the NoC actively hinders the coherence controllers. The trace will then not stress the simulator at all, as the delays between the messages are too long. To avoid these corner cases, an ideal NoC with middling latency and bandwidth is used. The traces are taken over the complete parallel phase of computation (region of interest) and capture all messages travelling in the NoC during that time. These traces were then fed to a C++ based network simulator which simulates the life of a message in an optical NoC, from its arrival at the source node until the actual start of transmission. To do this, the same model as presented in Section 4.2.2 was used. As gem5 is based on C++, the code is mostly identical. When investigating the predictive path setup methods, described in Section 5.3, the predictor setup containing the predictor itself and the interception structure as depicted in green

Benchmark	***Average message latency (clock cycles)***	***Standard deviation of message latency***
blackscholes	10.06	2.60
canneal	11.22	2.66
dedup	10.95	2.66
fluidanimate	10.91	2.44
freqmine	11.05	3.06
streamcluster	11.39	2.68
x264	10.24	2.29

Table 5.2: Total message latency in non-switched optical NoC with 8 wavelengths per channel when simulated in gem5

in Figure 5.6 is added to the model.

Every benchmark is represented by a single trace, which holds all inter-tile traffic generated during a single simulation. This seems to contradict Section 3.1.2 where it was argued multiple simulations per architecture are needed to account for the effect of space variability. However, even though multiple runs can result in completely different performance results (as shown in Figure 4.1), the traffic offered to the NoC does not seem to change significantly across runs as will be shown by discussing the injection rate and the source & destination traffic across multiple runs (Figure 5.1 and Figure 5.2 respectively). The cases with the highest variation in performance as seen in Figure 4.1 are those representing a CMP with an ideal NoC with a link latency equal to 4 and 18 clock cycles, running ***canneal*** and ***dedup*** respectively. If the traffic varies significantly across simulations, it should be most visible in these cases. The traffic offered to the NoC is compared in terms of injection rate, the proportion of messages transmitted by each tile and the proportion of messages received by each tile.

Figure 5.1 shows the injection rate normalised to the average injection rate across all runs. The brackets indicate the standard deviation across the different runs. Even in the cases where the performance varies with approximately 45% and 15% (***canneal*** & 4 clock cycles and ***dedup*** & 18 clock cycles respectively), the standard deviation on the injection rate is only around 1%.

Figure 5.2a and Figure 5.2b show the proportion of the total number of messages transmitted per tile for ***canneal*** and ***dedup***. The red bars indicate the traffic in a NoC with link latency equal to 4 clock cycles, whereas the blue bars represent traffic in a NoC with link latency equal to 18 clock cycles. The number of messages transmitted by a tile can differ across architectures (red bars versus blue bars) but the standard deviation across runs is again around 1% at most. The same holds for the amount of traffic received per tile (Figure 5.2c and Figure 5.2d).

A single trace will sufficiently represent the traffic behaviour of a benchmark for the measurement of waiting latency, because of the low amount of variation in injection rate and traffic patterns across multiple runs for the same architecture.

The choice for a trace-based model is purely practical as it cannot replace full-system simulations. gem5 is a very complex simulator which makes adding a predictor to the optical NoC a daunting task. Therefore I first wanted to ascertain whether the use of speculative and predictive path setup strategies would give positive results using a

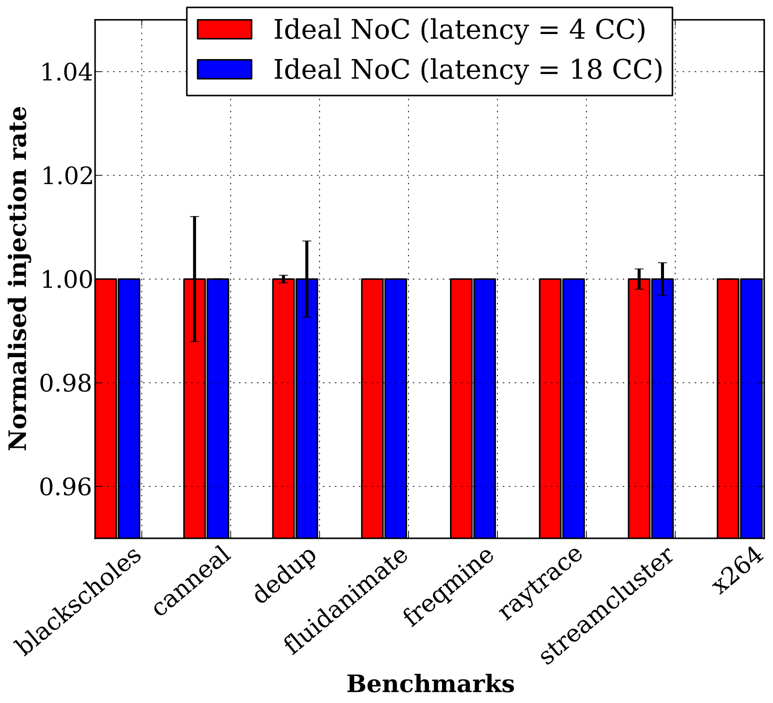


Figure 5.1: Variation in injection rate across multiple simulations of a CMP with an ideal NoC with link latency of 4 clock cycles and 18 cycles respectively. The number of simulations varies between 5-17 runs, depending on the benchmark.

simple trace-based simulator. The actual effect of prediction on the overall performance of the CMP can only be measured using full-system simulations. The results in this chapter therefore should not be seen as absolute improvements in performance but rather as a proof of concept, showing various path setup strategies can (positively) affect the NoC.

5.2 Use of Speculative Circuits in Switched Networks On-Chip

In the baseline path setup scheme, there is a one-on-one relationship between a circuit and a message. When a message arrives at the network interface, a path will be requested and set up which the allocator signals to the requesting network interface using a positive grant. After message transmission, the circuit gets torn down immediately. The advantage of the baseline scheme lies in its simplicity: when requesting a circuit, the size of the message and as such, the time at which the circuit can be torn down, is known. After the grant signalling the setup of a circuit, there needs to be no further communication between the allocator and the network interface. The latencies associated with the path setup process differ slightly from the previous chapters and are summarised in Table 5.3. The latency associated with path setup is assumed to be 8 clock cycles in this chapter. The electrical transmission of the requests and grants takes 2 clock cycles. Assuming the die size is around 263 mm^2, the allocator is in the middle of the chip and the M8 metal layer is used (see Figure 2.8), this is a valid assumption. However, for larger die sizes such as the Haswell series by Intel which can hold 16

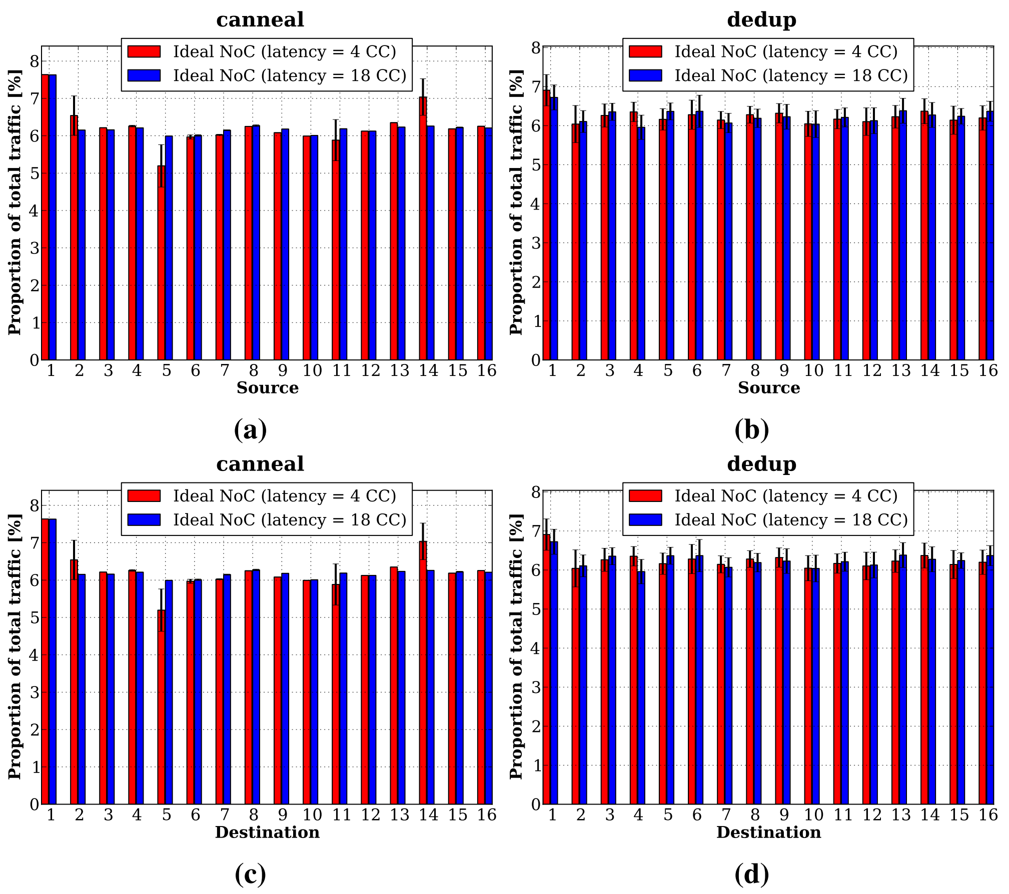


Figure 5.2: Variation in traffic per source and destination tile across multiple simulations of the same system. Only ***canneal*** and ***dedup*** are shown as they show the largest variation in Figure 4.1. The number of simulations varies between 5-17 runs, depending on the benchmark.

Xeon cores and has a die size of 661 mm^2, this electrical latency will take up to 4 clock cycles. The electrical transmission latency assumed might therefore be on the low side. The allocation decision process is assumed to take 4 clock cycles. The exact delay of the round-robin allocator used here can only be determined by synthesising the allocator. However, looking at the literature the delay of similar allocators (round-robin, 16×16 system) ranges from 3.66 ns to 1.79 ns (in 180 nm technology) [161] to approximately 1 ns (in 90 nm technology)[160]. An allocation latency of 4 clock cycles lies within the range of values mentioned in the literature, but is definitely not that of an optimised allocator.

Keeping a circuit open after transmission has the advantage that any subsequent messages from the same source to the same destination will not have to go through the grant-request cycle, which will have a positive effect on the waiting latency. The circuit will only be broken when any of resources it is using (i.e. transmitter or receiver) is needed by another circuit. This strategy will only work if there is a relatively high probability of circuit re-use. However, because of the interplay between cache indexing and Second Level Cache (L2) mapping, circuit reuse is common in this type of CMP architecture. This is because address interleaving is used to determine which slice of the L2/directory a cache line belongs to [162]. By using the address bits to the left of the block offset, neighbouring cache lines will be assigned to different L2/directory slices. This prevents the formation of bottlenecks. In our specific case, bits 6 to 9 will

	Switched NoC	*Non-switched NoC*
Network frequency	2 GHz	2 GHz
Number of nodes	16	16
Topology	Crossbar	SWMR
Terminal nodes per network interface	3 (tiled architecture)	3 (tiled architecture)
Modulation frequency	25 Gbit/s	25 Gbit/s
Allocation latency	4 clock cycles	Not applicable
Electrical signalling latency	1 clock cycle	Not applicable
Time of flight	1 clock cycle	1 clock cycle
WDM wavelengths available	1-32	1-32

Table 5.3: Parameters used for the optical networks in this thesis. Values in red differ from Table 2.8.

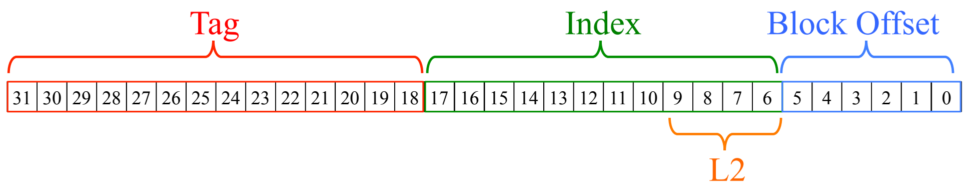


Figure 5.3: 32-bit address structure used in our system architecture. Bits 6-9 determine the L2 slice which each block is mapped to.

be used for L2 mapping as shown in figure 5.3. These bits coincide with the bits used for set indexing in the First Level Cache (L1) cache, as again the lower-order bits are used as index bits (in our L1 setup, bits 6 to 17). Basically this means that all cache lines mapped onto the same set will be mapped onto the same L2/directory slice. If a L1 request results in an eviction, both the newly requested address and the victim cache line belong to the same L2/directory slice.

Eviction of an exclusive or dirty line from the L1 will result in a writeback message to the L2 (Section 2.1.4). In the Modified Exclusive Shared Invalid (MESI) protocol (as implemented in gem5), the message carrying the new request and the writeback message will leave the L1 controller at the same time. These two messages can use the same circuit: either the request or writeback message opens the circuit after which the subsequent message reuses the circuit. Assuming the ratio of L1 writebacks to L1 misses is relatively high, this simple path setup strategy should result in savings in the waiting latency.

5.2.1 Implementation

Keeping circuits open after usage introduces speculative circuits: circuits which are kept open without an explicit request. These speculative circuits also increase the complexity of the allocator and the network interfaces.

First of all, the complexity of the allocator increases as there now are two types of circuits: firm circuits and speculative circuits. A firm circuit has either been requested by a network interface or was a speculative circuit which is currently in use and has been upgraded to a firm circuit. A firm circuit will be downgraded to a speculative circuit after message transmission.

The second increase in complexity is due to the allocator needing to be able to determine which speculative circuits are in use. This could be achieved by making the network interface signal all transmissions to the allocator. Another option is the use of an interceptor structure: before the optical signals enter the crossbar, a small part of the power is tapped off and used to inform the allocator of a circuit in use. This is similar to the structure depicted in Figure 5.6, albeit it simpler in this case, as only the fact that the circuit is being used needs to be detected.

Thirdly, a new type of grant which is used to signal the removal of a circuit is needed. When the allocator receives a firm request which needs resources (i.e., a transmitter or receiver) which are currently used by a speculative circuit, it sends out a *negative* grant to the transmitting network interface to signal end of the speculative circuit. In practice, this adds one extra bit to the grant connections to signal the type of grant.

Finally, the use of speculative circuits also increases the buffering complexity on the transmitter side. When a network interface receives a positive grant, it will assume the connection is intact until further notice and will keep on transmitting messages across the circuit. However, as there is a communication delay between the network interfaces and the allocator, it might be that the circuit is being torn down while the network interface is transmitting. When the network interface receives a negative grant, it immediately cancels the transmission. However, to prevent message loss, messages need to be to kept in the network interface buffers until this uncertainty period has passed, which consists of the time of flight and the electrical communication delay.

5.2.2 Results

Figure 5.4 shows the savings in waiting latency, compared to the baseline case, that can be obtained by keeping circuits open after usage. The benchmarks are ordered by latency savings (red bars). ***blackscholes*** has the highest latency savings, ***freqmine*** the lowest. The number of evictions per L1 miss is plotted as well, showing that in general a higher ratio of evictions per L1 miss favours this scheme. There are three exceptions to this rule: ***blackscholes***, ***canneal*** and ***freqmine***. This is due to the waiting latency of a system being determined by various factors, such as the injection rate and traffic patterns. In the case of ***blackscholes***, the discrepancy can be explained by its higher proportion of messages carrying a L1 request compared to the overall number of messages. It generates fewer writeback messages per L1 request message but the proportion of L1 request messages to the total number of messages is the highest (28% of all messages in ***blackscholes*** are carrying a L1 request, compared to the PARSEC average of 23%). On the other hand, ***canneal*** has the highest injection rate of all PARSEC benchmarks, which negatively affects waiting latency, making it harder to make any latency savings. The injection rate of ***canneal*** is 150$\times$ that of the lowest injection rate in the PARSEC benchmark suite. Finally, in the case of ***freqmine***, the discrepancy is due to the traffic pattern it creates. We find that there are a couple of tiles transmitting to all tiles and, in return, all tiles transmit to these sender-tiles. This hinders circuit re-use as the sender-tiles break up circuits so they can transmit to multiple destinations, and the many receiver-tiles respond to many sender-tiles. Overall, keeping circuits open after transmission is a relatively effective strategy and reduces the waiting latency by 15% on average. When the path setup process is more costly in

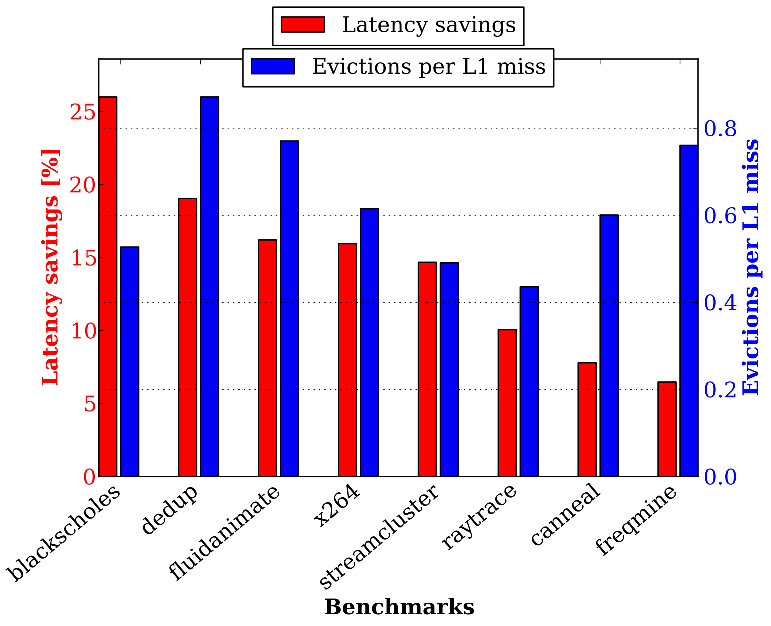


Figure 5.4: Savings in waiting latency by using speculative circuits in a switched NoC.

terms of latency, using speculative circuits is more beneficial. If the path setup process takes 16 clock cycles, the latency savings increase to 16%. The effect on performance is less certain as these results seem to indicate the savings are due to to a more efficient path setup for eviction messages. However, as evictions do not lie on the critical path of a memory transaction so the effect on the average latency of a memory transaction as a whole will be less pronounced.

The use of speculative circuits increases the complexity of the system as collisions might occur. However, collisions very rarely occur: the proportion of messages that will collide lies below 1%, for all benchmarks. The added buffering in the network interfaces will therefore rarely be needed. However, as messages cannot be dropped in this setup, these buffers are necessary. Speculation as used here is less complex than the work done by Wen et al. [122] in which circuit reuse distance is used to predict upcoming circuits by tracking past circuits and their probability over time (Section 2.2.4) The speculative work presented here is similar to the work done by Abousamra et al. [126] (Section 2.2.4) in which a similar principle is used: when a message comes by, the return path is automatically set up, as they argue a request message will most likely cause a corresponding response message will need to be transmitted back to the original source. They also discuss the use of cache events to start path setup early, for example, as soon as a cache hit is detected, a signal could be transmitted to network to start path setup as there is a delay between hit detection and the read-out of the completely cacheline. In comparison, the speculative scheme is less complex than both these works, as it only keeps existing circuits open.

5.3 Use of Predictive Circuits in Switched Networks On-Chip

The use of speculative circuits reduces the waiting latency. Larger latency savings can be obtained by recognising that messages in a shared memory CMP are not randomly injected into the network: they are part of a coherence transaction following a memory request by one of the processor cores. In this section, the various types of coherence transactions are discussed to see how this knowledge can be exploited to predict the optical paths that will be required later on. This discussion centres around MESI-based coherence controllers connected using an optical crossbar-based NoC but the concepts presented here are applicable to any type of switched NoC running any type of directory-based coherence protocol.

5.3.1 Coherence transactions

Figure 5.5 show coherence transactions for a system running the MESI cache coherence protocol following a write request to memory address ① by the processor core.

Figure 5.5a show a read access to memory address ① which is already present in the L2. The coherence transaction is straight forward as the L2/directory can directly respond to the request by returning a copy of the requested cache line. Figure 5.5b depicts a subsequent write access to memory address ① which results in far lengthier coherence transaction. The address is already present in the L1 cache, but with read-only permission. The cacheline needs to be upgraded to the exclusive state, which requires the invalidation of all copies of ① in other L1 caches. All other sharers need to acknowledge their invalidation of the address to the originating L1 before the upgrade can complete. The length of the upgrade transaction therefore depends on the number of sharers. In this example, the L1 sends the upgrade request to the L2 directory, which sends out a response to the original L1 and an invalidation request to the other sharer. This sharer responds directly to the original L1 once it has invalidated its copy of the memory address. Finally, the original L1 sends an unblock message to the directory to inform it that the upgrade is complete. In Figure 5.5c the L1 performs a write to memory address ① which is not yet present in the on-chip cache hierarchy, so needs to be retrieved from main memory. It makes a request to the L2, which has no record of it, so in turn makes a request to the memory controller which retrieves the value from Dynamic Random Access Memory (DRAM). This sends a response back to the L2, which can then send another response back to the L1. Finally, once the L1 has received the data, it can unblock the L2.

5.3.2 Coherence-based prediction

The previous examples can be generalised: when a processor core requests a memory address, the resulting coherence transaction and its associated messages seen by the network are a combination of the type of access requested and the current state of the requested address. For example, in the case of the upgrade transaction depicted in Figure 5.5b, by combining knowledge about the request (the *L1 REQ C* tells us this is an upgrade for address ①) and the current state of ① (cached in L1'), the exact

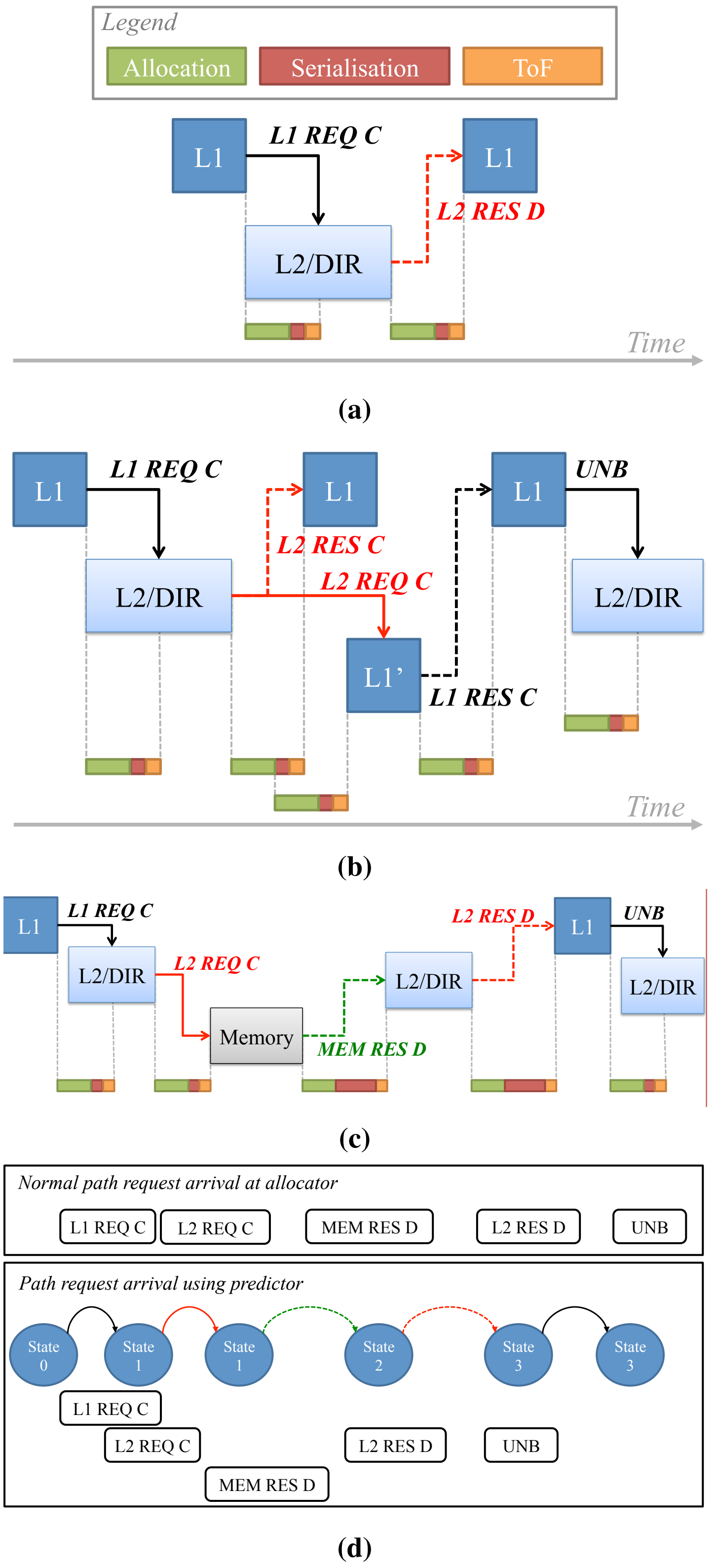


Figure 5.5: Examples of memory transactions and the messages generated by the coherence protocol. The timescales in this figure only show the relative order of events and not the duration of these events. (a) read access to an address present in L2 cache (b) a L1 upgrade transaction (c) a write access to an address not present in the on-chip cache hierarchy. Below the messages, their corresponding stages in the network are depicted: path allocation, serialisation and time of flight (ToF). (d) Resulting transitions in the predictor FSM. For explanation about the different message types, see Section 2.1.4)

coherence transaction can be determined. Once the coherence transaction is known, the next messages can be predicted with high accuracy based upon the previous message.

This type of prediction naturally lends itself to an FSM implementation. In order to evaluate the upper bound on performance of this kind of coherence-based prediction, a predictor structure is implemented using a 5-state FSM for each memory address, with the state transitions based upon all possible coherence transactions. The FSM is designed for the MESI protocol but could be easily modified for other coherence schemes. At runtime, every message travelling the network is intercepted and used to make the state transitions within the predictor table. This allows the prediction of future messages by looking at the new state and the last observed message. There are 6 possible states:

State 0 Block is not present anywhere in the on-chip cache hierarchy (*Start state*)

State 1 Block is not present in the on-chip cache hierarchy, but has been requested by the directory/L2

State 2 Block is only present in the L2

State 3 Block is only present in one L1 cache

State 4 Block is present in multiple L1 caches

State 5 Transient state, block is in the midst of being evicted

Figure 5.5d shows the state machine transitions for the write request depicted in Figure 5.5c. In the conventional case, all 5 messages in the transaction are delayed by the path setup latency before transmission as path setup only starts after the controllers have handled the previous message. Using the predictor, as soon as a message is intercepted, a prediction can be made to setup the path for the next message while the message is still being handled by the cache/directory controller. Only the first message in the transaction is delayed by the path setup latency. Section 7.1 and Section 7.2 contain the code representing prediction FSM.

The predictor consists of combinational logic and a Look-up Table (LUT) in which every entry is associated with one unique address. Every entry has 5 fields:

1: Valid bit;

2: Current FSM state (3 bits, but the valid bit could be combined into this field as another state);

3: Sharers list in which a processor is seen as a sharer if it holds a copy of the cache line in its L1 cache (N bits for a N-core CMP) ;

4: The type of transaction (3 bits for 5 possibilities: load, store, upgrade, L1 eviction and L2 eviction);

5: The L1 cache currently upgrading or evicting a cacheline ($log_2(N)$ bits).

The LUT is located next to the central allocator. The address of the intercepted message is used as an index into the LUT which (if needed) issues a predictive path request to the allocator. Firm path requests will be prioritised over predictive requests by the allocation algorithm. If the predictive request wins a path, the corresponding source node will be notified of the path that has been set up. If the predictor did not predict the correct path, the future message that should have used this path will go through the normal request-grant cycle, without any additional penalties. Considering the low injection rates in the NoC, as depicted in Figure 3.5, unused predictive requests are unlikely to overload the allocator. To prevent predictive requests from immediately tearing down recently established (and as yet unused) predictive circuits, every predictive circuit gets assigned a countdown timer. Once the countdown timer has expired, the priority of the circuit is lowered so the circuit can be broken down to accommodate another predictive request[1].

The holistic proposals discussed in Section 2.2.4 also use knowledge of the coherence protocol to decrease the latency of a memory request as a whole, by using sharer prediction, for example, to avoid unnecessary network transactions. These proposals, however, do not speed up the messages that still need to travel the NoC. The coherence-based predictor in this work aims at speeding up message transmission, rather than message avoidance.

5.3.3 Implementation details

Implementing such a coherence-based predictor requires changes to the optical NoC and allocator. Firstly, the changes that need to be made to the NoC and allocator will be discussed followed by an overview of the timings of the prediction process.

Changes to the NoC

To allow coherence-based prediction, additional components need to be added to the NoC. Figure 5.6 shows the new system setup. A small optical splitter gets added to every waveguide entering the crossbar, so the messages can be intercepted. Reading these intercepted messages requires additional receivers leading to a modest increase in the number of active components and power consumption. However, not all wavelengths need to be read. The trade-off involved in choosing the number of interceptor receivers is discussed in more detail in Section 5.4.2. The predictor itself is centrally located and has a $2 \times log_2(N)$-bit wide connection to the allocator to transmit predictive requests containing the source and destination of the predicted circuit where N is the number of tiles on the CMP. The power consumption and complexity of this ideal predictor are discussed in more detail in Section 5.4.

Changes to the allocator

Using a predictive path setup policy, in conjunction with the previously discussed open circuit policy, will introduce some changes to the allocator. A circuit can exist in four states:

Firm circuit A circuit either currently in use or a circuit established because of firm request coming from a network interface

Predictive circuit A circuit established because of a predictive request coming from the predictor

[1]The countdown timer was set to 1 clock cycle as experiments (not shown in this thesis) showed a higher value had little effect on the overall prediction hit rate

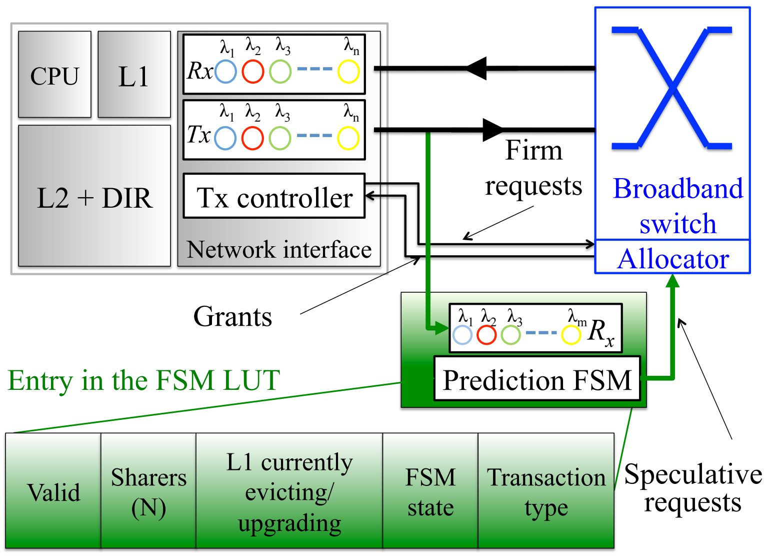


Figure 5.6: Schematic overview of the system setup when using coherence-bath path prediction. Note the number of wavelengths intercepted by the predictor structure does not have to be equal to the number of wavelengths in the stripe. Only the connections between a single tile and the central switch are shown. All other tiles are connected in an identical way.

Expired predictive circuit A circuit established because of a predictive request coming from the predictor of which the countdown timer has expired

Speculative circuit A previously existing circuit is kept open after usage

These 3 circuit types each have a different priority, with a firm circuit having the highest priority. Firm circuits are downgraded to speculative circuits after usage. Both speculative and predictive circuits are upgraded to firm circuits once they are in use. If a request arrives for an existing circuit and the request priority is higher than the circuit priority, the circuit gets upgraded e.g. a speculative circuit can become a predictive circuit if a predictive request arrives. When a request arrives at the allocator and the required circuit is not set up, its priority is compared with the current circuit priority to reach a decision on how to proceed. If the request priority is higher than the circuit priority, the circuit will be broken, to free the resources needed by the incoming request.

Timing

Figure 5.7 shows the 4 stages in the overall prediction process. The first stage ① is message interception in which the information needed to make a prediction is tapped off from a message currently travelling the NoC. Interception can only start after a period Δ, discussed in more detail in Section 5.4.2. During the second stage ②, the prediction information is extracted from the LUT using the intercepted address. This results in a prediction entry as previously discussed, which will be combined with the current message information in the third process step ③ when the actual prediction is made and a predictive request is sent to the allocator. The allocator takes care of the final step ④: setting up the actual path. All of these stages need to have finished before the coherence controller has handled the message so the path is setup before the next message in the coherence transaction (the predicted message) arrives at one of the network ingresses.

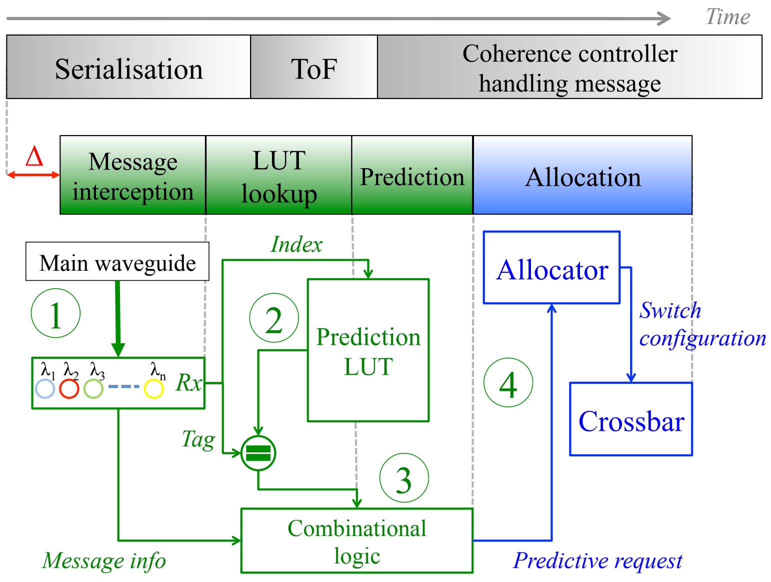


Figure 5.7: The phases of path prediction. (1) Message being intercepted on the main waveguide. Message interception can only start after Δ as the prediction info needs to be serialised (2) Prediction info is accessed in the LUT using the intercepted address (3) Prediction is determined using an FSM (4) Predicted path is being setup. Path needs to be ready before the coherence controller finishes handling the message.

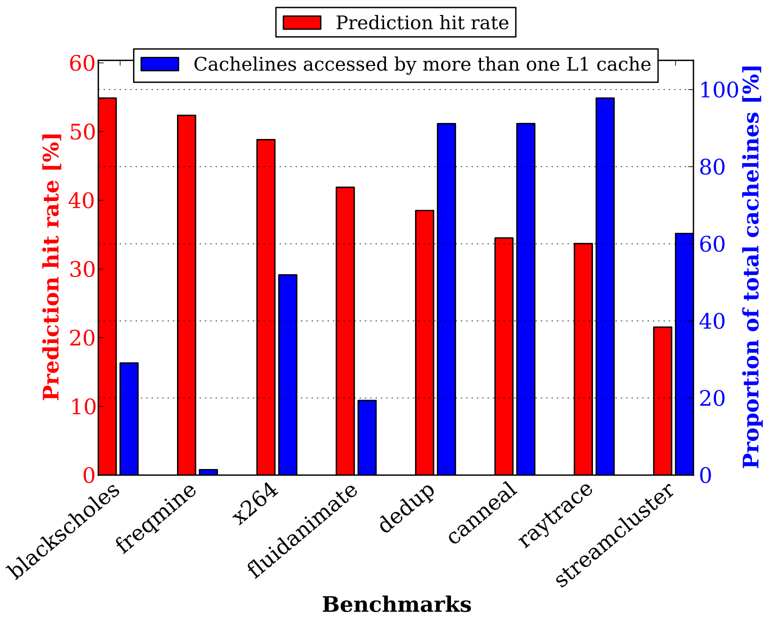


Figure 5.8: Prediction hit rates of an ideal prediction schemes compared to the number of sharers per cache line.

5.3.4 Results

The reduction in waiting latency that can be obtained by using the ideal predictor is affected by various factors. First of all, the predictions need to be correct. In principle, perfect prediction of all messages, excluding those that signal the start of a transaction, is possible if the predictor is accurately designed whereupon it basically becomes a directory controller without the actual caching. However, in the scheme proposed here the predictor can only track a single transaction so multiple transactions for the same address starting around the same time will not be predicted correctly. The directory controller handles concurrent transactions by means of blocking and transient states. The directory controller as defined in the MESI protocol in gem5 has 17 states in total, of which 13 are either transient or blocking. Tracking concurrent transactions in the predictor would reduce the prediction miss rate but would increase the complexity of the predictor. First of all, the size of an entry in the LUT would increase due to the increase in number of states. Secondly, the combinational logic used to determine the transitions between the states would become slightly more complex as the possible number of transitions will increase. As the predictor will not discern between different concurrent transactions involving the same cache line, the information in the prediction LUT will be a muddled combination of all these transactions. Therefore the prediction algorithm might not be able to combine the intercepted message and the associated LUT entry to create a predictive request. If the observed message would not lead to a valid state transaction from the current state held in the LUT, the predictor will simple make no prediction.

To measure concurrency a histogram containing the number of L1 caches that have accessed every individual cache line is kept over the complete ROI. This is an approximation for concurrency [2] as cache lines requested by multiple caches during their lifetime will most likely also be involved in concurrent transactions e.g. being requested by multiple L1s at the same time. In Figure 5.8 the benchmarks are ordered by prediction hit rate (red bar). The second bar (blue bar) shows the proportion of cache lines that have more than 1 sharer. The benchmarks with a higher prediction hit rate have a lower number of sharers than the benchmarks with a higher number of sharers. In the four most predictable benchmarks (***blackscholes, freqmine, x264*** and ***fluidanimate***), on average, 25.4% of the cache lines have more than 1 sharer. For the least predictable benchmarks (***dedup, canneal,raytrace*** and ***streamcluster***), 85.7% of the cache lines are shared. The chance of concurrent transactions is much higher for these benchmarks, resulting in less predictable behaviour. Figure 5.8 seems to indeed suggest that the degree concurrency affects the prediction hit rate. However, tracking more concurrent transactions to increase the prediction hit rate is not a good trade-off because of the increased complexity and development time. Especially so as the prediction hit rate is already quite high when taking the proportion of unpredictable messages into account. The coherence-based predictor cannot predict messages that signal the start of a transaction (*L1 REQ C*) as they are the direct result of a L1 miss. The same holds for L1 eviction messages. Combined these message classes are responsible for between 42.2% and 33.6% of all messages, depending on the benchmark. Taking this into account, the prediction hit rate as shown in Figure 5.8 is acceptable [3].

The second factor to consider is the fact that while a prediction might be correct, the effect it has on waiting latency is less straight forward. Figure 5.9 shows the benchmarks, again ordered by prediction hit rate. The second curve, denoting the latency savings to the baseline case, follows the same trend but the prediction hit rate will not automatically

[2]In this context concurrency stands for coherence transactions that take place at the same time and involve the same cache line but originate from different processors.

[3]This is not to say predicting L1 misses (and, hence, the start of a coherence transaction) is impossible. [163] presented a predictor based upon a 2-bit saturating counter which uses the memory address of a memory reference, in parallel to the associated L1 access to predict whether or not the reference would miss in the L2 (if it were to miss in the L1). The same concept could be used to predict L1 misses (and probably also L1 evictions). The idea was not pursued further as such a predictor would have to be able to work faster than the L1 cache.

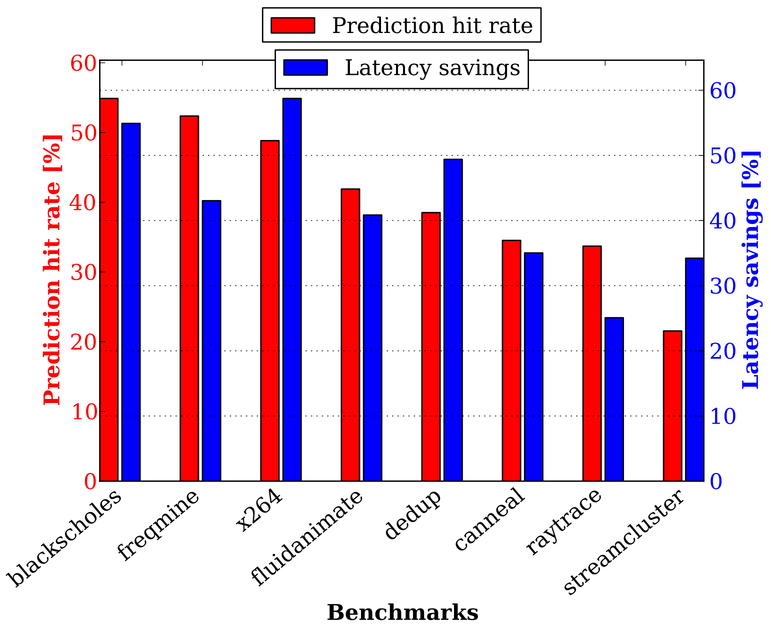


Figure 5.9: Waiting latency savings of an ideal prediction scheme compared with the prediction hit rate.

lead to exactly the same amount of latency savings. These discrepancies are hard to explain as they depend on various factors such as traffic patterns, injection rates, abrupt peaks in the injection rate etc.

Overall, though, the use of an ideal coherence-based predictor in a system with a path setup latency of 8 clock cycles will, averaged over all benchmarks, reduce the waiting latency by 42.6% (σ=11.3%), in contrast to the speculative technique discussed in Section 5.2 which reduced the waiting latency latency by 14.5% (σ=6%).

5.4 Realistic Use of Predictive circuits in Switched Networks On-Chip

The previous section evaluated an idealised LUT in which every address had a dedicated entry, which would be infeasible to implement in practice. However, the working set of an application is small at any point in time, so the predictor can be organised like the branch target buffer in a processor core where multiple addresses map to the same entry in the LUT, to then be used for prediction. The same principle is also used in cache organisation (Section 2.1.3). This section develops a realistic version of the predictor with acceptable overheads in size and complexity.

5.4.1 Realistic implementation of the predictor

The most obvious method for creating an implementable predictor is to re-organise the LUT. Multiple addresses can now map onto the same entry in the LUT. This reduces the size of the LUT compared to the ideal predictor where every address has its own dedicated entry in the LUT. Organising the LUT as a set-associative structure (borrowing concepts from Section 2.1.3) helps avoid collisions when multiple addresses

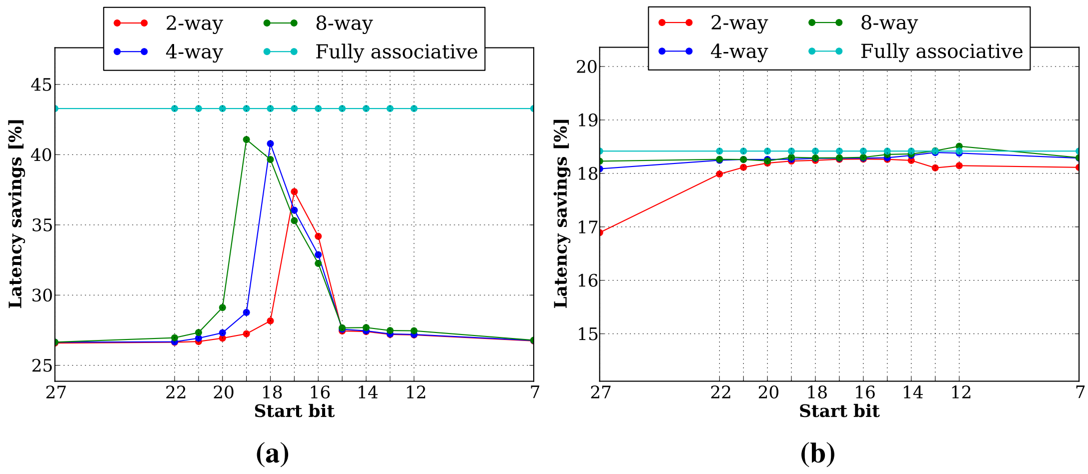


Figure 5.10: Effect of indexing function on latency savings for (a) ***blackscholes*** and (b) ***streamcluster***. Only these benchmarks are shown as they react the best and worst to prediction respectively.

map to the same entry. A Least Recently Used (LRU) replacement policy is used to evict entries out of the predictor LUT when required, but with slight modifications, as to reduce the number of evictions of addresses with a pending transaction. The replacement policy implemented to achieve this consists of two stages. First, only entries in the set which are in state 2 or lower are considered for replacement. These are entries for addresses only present in the L2, or currently being requested by the L2. Of this subset, the LRU entry is chosen. In case none of the entries in the set are in state 2 or lower, a pure LRU replacement policy is used.

To investigate the effect of sharing entries, a fully-associative LUT (Section 2.1.3) is used which provides an upper bound on the LUT hit rate, since it does not suffer from mapping conflicts. Although impractical, it is useful for comparison purposes.

Effect of LUT associativity

In the ideal prediction scheme, the complete memory address (with the exception of the block offset) is used as an index to lookup the associated prediction information in the LUT. The same principle holds for the fully associative LUT. In the realistic predictor that is not the case and a subset of the address bits will be used as the index to map onto a set in the LUT. The remaining address bits will be used as tag. The length of the index is determined by the number of sets in the LUT and decreases therefore with increasing associativity. The start of the index is important as it determines how efficient the mapping will be as shown in Figure 5.10. Only ***blackscholes*** and ***streamcluster*** are shown as they react the best and worst to prediction respectively. The effect is more pronounced for ***blackscholes*** (Figure 5.10a) than for ***streamcluster***(Figure 5.10b) but it is clear the most optimal bits for the index mask were found to be the most significant bits near the block offset, as they show the most variation and will therefore lead to a better mapping. This figure also shows that by using optimal indexing bits the performance of the realistic indexing functions is comparable to the upper limit case of the fully-associative LUT in which there are no mapping conflicts.

Figure 5.11 gives an overview of the latency savings for all PARSEC benchmarks, when using the optimal indexing function. Overall, increasing the associativity increases

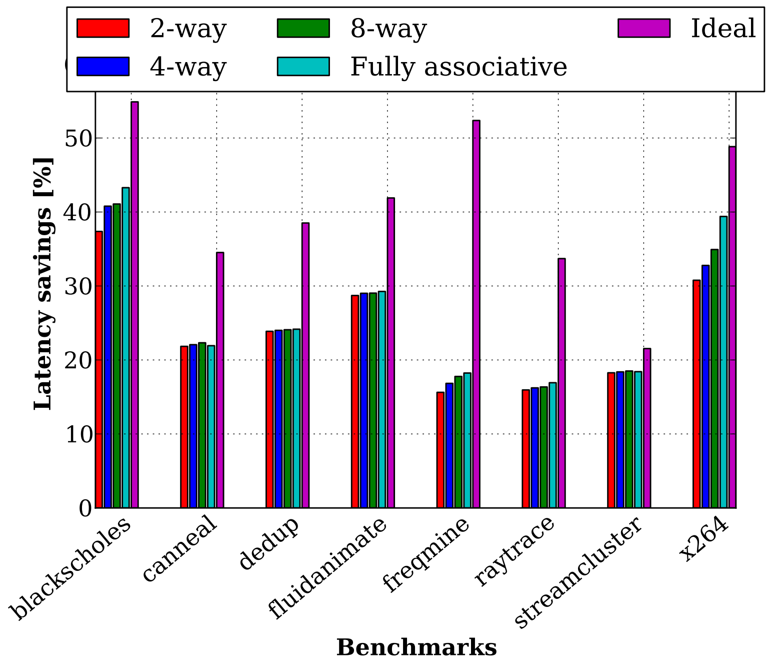


Figure 5.11: Latency savings for various indexing functions for a predictor with a LUT with 8192 entries.

2-way associative	40 kB
4-way associative	41 kB
8-way associative	42 kB
L1 (32 kB, 4-way)	32.88 kB

Table 5.4: Size of a LUT with 8192 entries

the latency savings as the number of conflicts decreases. An increase in the associativity will have an effect on the predictor complexity. Prediction can only start after the prediction information has been retrieved from the LUT making serially accessing the ways in a set not desirable. A 4-way associative LUT would provide a compromise between speed, added complexity and prediction accuracy. When comparing the latency savings of the ideal predictor with the set-associative predictor, the loss in performance becomes clear. However, the ideal predictor would be larger than the actual byte-addressable physical memory and is not implementable in practice.

An entry in a N-way associative LUT will be larger than its counterpart in the directly mapped LUT as the entry now also needs to contain the tag and the bits used by the LRU replacement algorithm, but this is offset by the decrease in number of entries. As the size of the tag depends on the associativity of the LUT, the total LUT size increases slightly when increasing the associativity. The LUT sizes can be found in Table 5.4 and are in the same order of magnitude as a L1 cache demonstrating that the LUT will be fast and will not significantly increase the area of the overall CMP.

Effect of LUT size

The effect of the number of entries on prediction accuracy was investigated by sweeping the number of entries in the LUT from 1K to 16K, for a 4-way associative LUT.

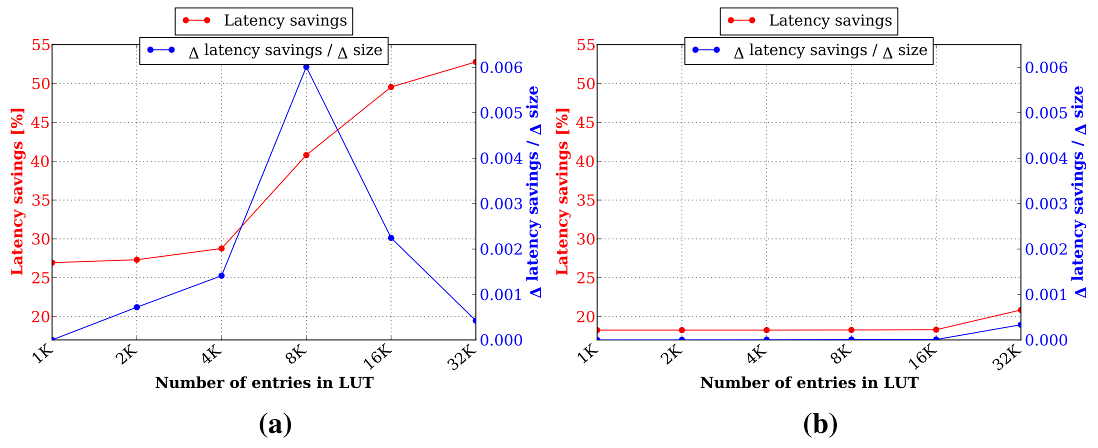


Figure 5.12: Latency savings when increasing the number of entries in a 4-way associative LUT for (a) ***blackscholes*** and (b) ***streamcluster***.

Optimal number of LUT entries	***LUT size***	***Benchmarks***
2K	10.75 kB	freqmine, raytrace
4K	21 kB	fluidanimate
8K	41 kB	blackscholes, canneal, x264
32K	156 kB	dedup, streamcluster

Table 5.5: Optimal number of LUT entries for a 4-way associative LUT.

Figure 5.12 shows the effect of increasing the LUT size on the latency savings for ***blackscholes*** and ***streamcluster***. Increasing the number of entries increases the latency savings (red line) as can be expected. The size, however, increases faster than the latency saved. Looking at the gain in latency savings over the increased LUT size for ***blackscholes*** in Figure 5.12a, shows that the return on investment (blue line) decreases when increasing the number of entries in the LUT above 8K. In the case of ***streamcluster***, the optimal number of entries in the LUT is 32K as shown in Figure 5.12b (assuming the allowed LUT size ranges from 2K entries to 32K entries) . However, the actual latency savings that can be obtained for ***streamcluster*** with a 32K LUT are smaller than those for ***blackscholes*** with a 4K LUT. Table 5.5 shows the optimal number of entries in the LUT per benchmark. The predictor setup needs to be able to adequately handle all benchmarks so a LUT with 8K entries seems most optimal.

Effect of timing

As soon as the predictor generates the predictive requests, they are transmitted to the allocator, which handles them as soon as possible. However, this might not be optimal as the message for which the predictive circuit is intended, might not arrive immediately. Depending on contention in the coherence controller, there might be a significant amount of time between the setup of the predictive circuit and the message arrival. During this gap, the predictive circuit might be torn down to use the resources for other circuits.

Figure 5.13 shows the effect of this timing gap. The red bars depict the proportion

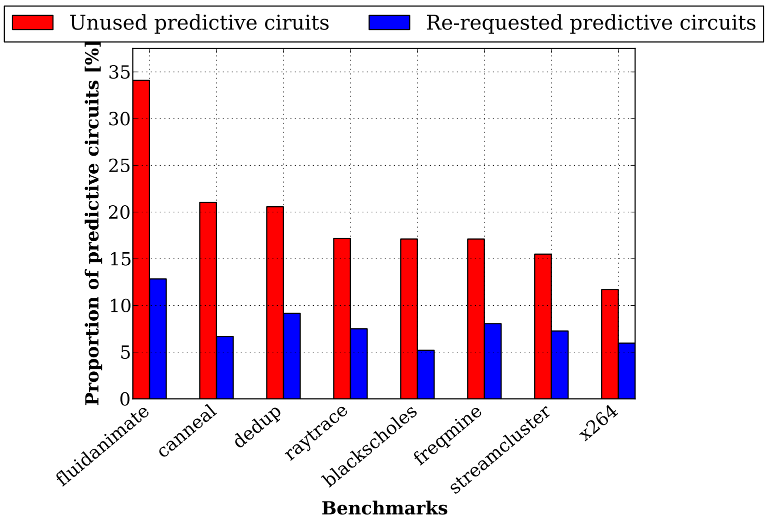


Figure 5.13: Effect of the timing gap between the predictive circuit and a possible corresponding message.

of predictive circuits which are torn down before they are used, over the total number of predictive circuits. The blue bars show the proportion of these unused circuits which will later on be requested again, by their corresponding message, again over the total number of predictive circuits. Due to constraints on simulation time, the traces were only run for 1,000,000 messages. However, these results show that while a non-negligible proportion of the predictive circuits are never used, the proportion of these circuits that then are requested again is quite small, which indicates the timing gap is not necessarily a problem.

5.4.2 Use of partial tags

The speed at which the predictor works will determine its efficiency. As shown in Figure 5.7, the actual prediction can only start after the message interception phase and the LUT lookup. A shorter tag length used in the LUT will decrease both the delay after which message interception can start (Δ in Figure 5.7) and the duration of the actual interception. Another positive side-effect of a shorter tag is a smaller LUT as the size of every entry decreases. The use of a *partial* tag has been proposed in the past to reduce the complexity of cache lookup [164].

Message interception delay

Figure 5.14a shows how a message is modulated on the 8 wavelengths in the stripe. With a modulation speed of 25 Gbit/s and a clock frequency of 2 GHz, 12.5 bits can be modulated on every wavelength per clock cycle. Every block in Figure 5.14 therefore represents 12 bits. The control info in a coherence message (C) is 8 B long and will need 6 blocks. If the messages carries data, the critical byte ($D*$)[4] will also be modulated

[4]The processor core requests data from memory on a word granularity. By placing this *critical* word first in the response message and transmitting the word to processor core immediately upon arrival in the cache, the miss latency as perceived by the processor core can be minimised [2, Chapter 2.2]. Some architectures can make memory references on a byte granularity, making the requested byte the critical byte.

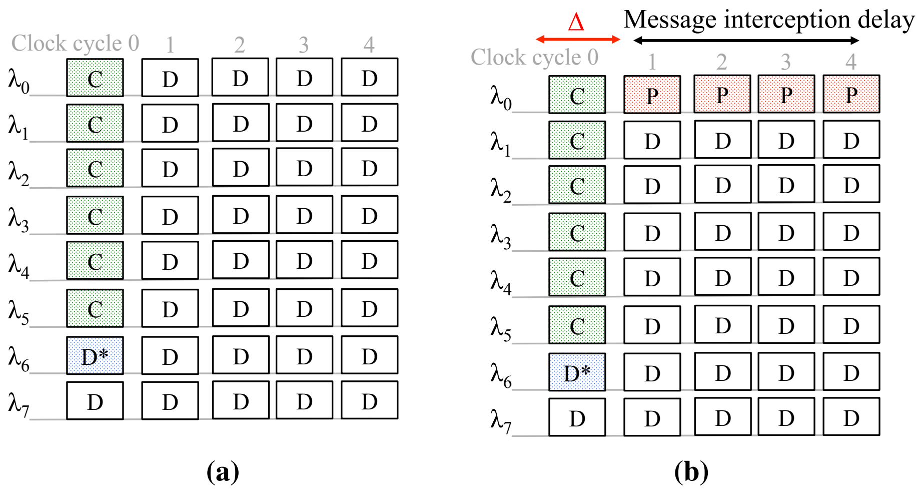


Figure 5.14: Encoding of the a message on 8 wavelengths. *C* denotes the control info, *D** stands for the critical byte, *D* for data and *P* represents the prediction info. Only the first 5 clock cycles are shown for clarity.

in the first clock cycle, immediately after the control information, and only takes one block. The remainder of the data will take 41 additional blocks, resulting in a total serialisation time of 6 clock cycles for a data message. The addition of the predictor should not interrupt the normal organisation of message transmission, so specific prediction information (*P*) is added to the message, which will be transmitted after the first clock cycle. These blocks contain the destination of the message ($log_2(N)$ bits), the message type (11 possibilities so 4 bits), the request type (5 possibilities resulting 3 bits), the destination type (1 bit) and the complete address (26 bits as this is the cache line address minus the block offset). This amounts to 38 bits, for which 4 blocks are needed.

In Figure 5.14b these 4 blocks are encoded on 1 wavelength. In this case, only 1 wavelength needs to be intercepted thus decreasing the number of receivers required at the crossbar but the LUT lookup can only start after 5 clock cycles. To reduce this latency we could increase the number of wavelengths used to modulate *P* or reduce the size of *P* where the latter is the most optimal solution. *P* can be reduced by lowering the overhead of the address information. The LUT needs the index and the tag of a memory address. By only using a part of the tag in the LUT, the size of *P* can be reduced as this lowers the overhead of the address information. A reduced tag length will decrease the accuracy of the LUT: two distinct address with the same index and partial tag will be seen by the LUT as identical and therefore be mapped onto the same entry. The prediction information held in an entry in a LUT that uses partial tags is therefore based on the coherence transactions of multiple distinct addresses.

The number of intercepted wavelengths will affect the power consumption of the predictor receiver. This effect can be quantified, albeit in a simplified manner by only including ring heating [165] and power per individual receiver [166]. Figure 5.15 shows the effect of decreasing the tag length on both the latency and the power consumption of the message interception stage. If the latency of message interception needs to be below 1 clock cycle, 4 wavelengths in total need to be intercepted. However, when using a tag length of only 2 bits, this can be achieved using only 2 wavelengths. This results in a $2\times$ decrease in power consumption.

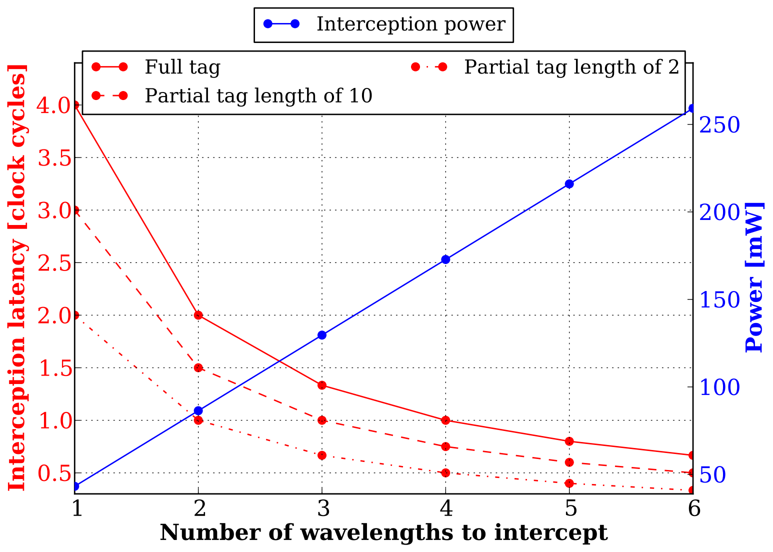


Figure 5.15: Effect of reduction in tag length on the latency and power consumption of the message interception stage, assuming a 4-way indexing scheme for a LUT with 8192 entries.

Results

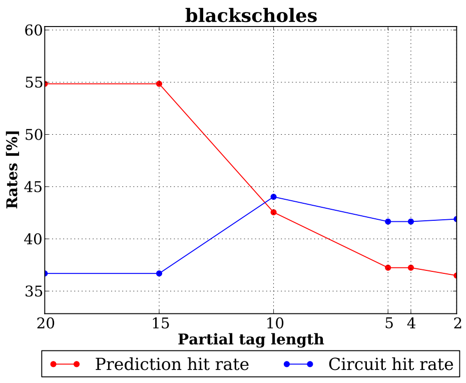
It would seem that varying the size of the tag length would lead to a trade-off between power and latency savings where a short tag would lead to reduced power and interception latency at the cost of a less efficient predictor. However, this is not always the case.

In Figure 5.16 - Figure 5.19, the use of speculative circuits is disabled to only show the effects of partial tag prediction. This means every circuit (based upon a predictive or a firm request) will be torn down immediately after it has been used or, in the case of a predictive circuit, when the resources are needed elsewhere.

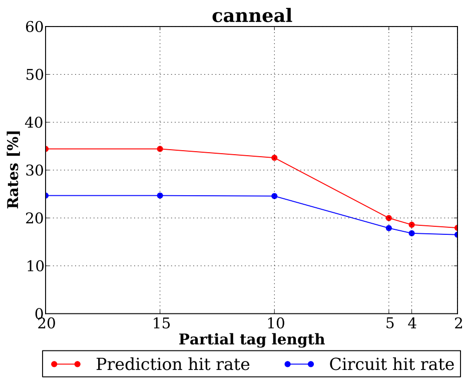
Figure 5.16 shows the effect of the tag length on the prediction hit rate (red line) and the circuit hit rate (blue line). The prediction hit rate is measured by checking the prediction LUT for the last prediction for this address when the message comes by. The circuit hit rate, however, is measured at the network ingress: a circuit hit occurs when a message arrives at the network ingress with a circuit set up. It is important to note in this case that an existing circuit can be the result of a prediction for a completely different address (and, hence, message) which is why the circuit hit rate can be higher than the prediction hit rate. This is because messages can piggyback on predictive circuits set up for other addresses. For example, imagine two concurrent coherence transactions, one for address Ⓐ and the other for address Ⓑ. Both addresses are completely unrelated and map onto different entries in the LUT. The first message in the coherence transaction for Ⓐ results in a predictive circuit from node 0 to 1. The first message in the transaction for Ⓑ then comes by, for which no prediction could been made as it is the first message in the transaction. However, because this message incidentally travels from 0 to 1, it can use the predictive circuit set up for Ⓐ. This will not be counted as a prediction hit (because the circuit was predicted for a message for Ⓐ, not Ⓑ) but it will be counted as a circuit hit (the circuit was established when the message arrived). However, if the second message in the coherence transaction for Ⓐ comes by and uses the circuit from 0 to 1, this will counted as both a prediction hit and a circuit hit. Therefore, the circuit hit rate can be higher than the prediction hit rate. This mechanism can occur more frequently in the case of a partial tag as multiple distinct addresses are mapped onto a single entry in the LUT, thereby making it more difficult to make a prediction. As the prediction algorithm will simply make no prediction if it cannot combine the prediction info in the LUT and the intercepted message into a prediction, there is no chance of prediction hit occurring but a circuit hit can still occur. The prediction hit rate decreases with decreasing tag length for all benchmarks. This can be explained by the fact that the predictions become less precise because multiple distinct addresses are mapped onto a single entry in the LUT. However, more surprisingly, some benchmarks (e.g. ***blackscholes*** (Figure 5.16a) or ***x264*** (Figure 5.16h)) have an increasing circuit hit rate when the tag length decreases. New terminology is now introduced to reason about this behaviour: a bucket is the unique combination of a tag and index. A bucket is the same as an entry in the LUT, but by using the concept of a bucket, it is clearer that this behaviour is independent of the LUT organisation. When decreasing the tag length, there are less unique buckets but more distinct addresses map onto each bucket.Figure 5.17a shows the number of buckets does indeed decrease with decreasing tag length, thereby increasing the number of addresses mapping onto each bucket as shown in Figure 5.17b. The exact numbers differ between the benchmarks, as they have different memory access patterns.

The prediction hit rate is affected by the similarity between the addresses mapping onto a bucket: if the distinct addresses that map onto this bucket behave very similar, the resulting prediction will be better than when the addresses behave completely differently, muddling the prediction information and algorithm. The latter effect is at play here, in Figure 5.16: the addresses mapping onto each bucket behave dissimilar, leading to a lower prediction hit rate with decreasing tag length. The circuit hit rate on the other hand is influenced by the similarity across the different buckets: bucket *A* might make a prediction, resulting in a circuit used by a message mapping onto bucket *B*. This would not count as prediction hit but will be seen as a circuit hit. This might explain why some benchmarks have an increasing circuit hit rate with decreasing tag length. In the case of benchmarks with stable average traffic patterns, because of the increased number of addresses mapping onto each bucket , the prediction information in each bucket will start approaching the average prediction information, across all buckets. This will lead to predictive circuits that might not be used by the message it was intended for, but by another message from another bucket. This averaging behaviour will not occur for all benchmarks, as benchmarks with quickly changing traffic patterns for example will not have an 'average' behaviour. This could explain the differences in circuit hit rates across benchmarks. An experiment was carried out to check if this averaging effect actual occurs. The circuits used by all messages mapping onto a bucket i.e. the average traffic pattern per bucket was used as an approximate for prediction information per bucket. This was done by creating a vector per bucket in which each entry in a vector stands for a circuit e.g the first entry in the vector stands for a circuit from node 0 to node 1. Every time this circuit is used by a message mapping onto this bucket, this increases a counter. The circuit vectors per bucket were built up over the complete runtime. At the end of the run, all vectors were normalised. The normalised circuits vectors belonging to different buckets were then compared. Ideally, all buckets would be compared. As this would be infeasible in practice because of the large number of comparisons, 10% of all buckets were randomly picked and compared by taking the absolute difference between their corresponding normalised traffic vectors. The results of this experiment are shown in Figure 5.18. This plot shows that on average, the difference in traffic patterns across all buckets (as a sort of representation of prediction information) decreases with decreasing tag length. The random jumps for some benchmarks could be due to the fact that only 10% of all buckets are compared. While Figure 5.18 could indeed be seen as a suggestion that the averaging behaviour across buckets plays a role in the increasing circuit hit rate of some benchmarks, is it by no means conclusive, if only because traffic patterns do not equal prediction information. The relationship between buckets, prediction information in a bucket, the prediction and an eventual circuit is not linear at all. There are so many effects at play (for example, the prediction itself is a complicated FSM based upon the prediction information where the prediction information is combination of sharers and current transaction), making it not straight forward to reason about this. The benchmarks differ as well in terms of distinct memory addresses, number of resulting buckets etcetera. As shown in Figure 5.19 the changes in latency savings closely follow the circuit hit rate behaviour.

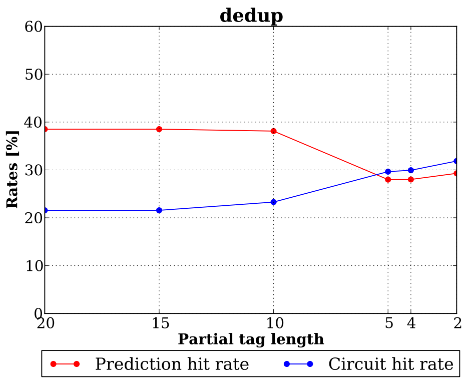
Overall, reducing the tag length, thereby decreasing the power and latency of the predictor structure, has a mixed effect on the latency savings. Figure 5.16 -Figure 5.19 showed the effects of prediction without speculative circuits, to remove the interplay between speculation and prediction. However, the actual predictor would use speculative circuits. Speculation does not alter the effect of decreasing tag length, only increases the circuit hit rate equally for all tag lengths. Figure 5.20 shows the effect of decreasing tag length for all PARSEC benchmarks by subtracting the latency savings for the full tag from those of a particular tag length. When the line dips below zero, it is no longer beneficial (from a performance point of view) to decrease the tag length. The negative effect of a decreased tag length (e.g ***canneal***) is smaller than the positive effect on most benchmarks (e.g ***dedup***). The most optimal tag length across all benchmarks in terms of latency savings, would be 10 bits as there there are no negative consequences yet at that point.



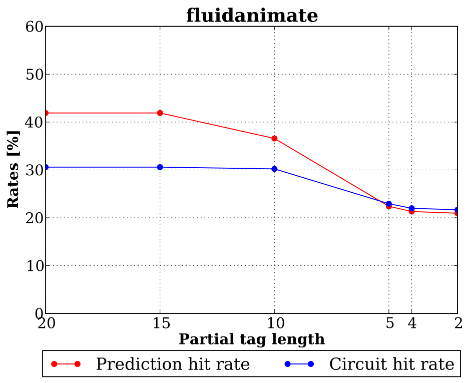
(a)



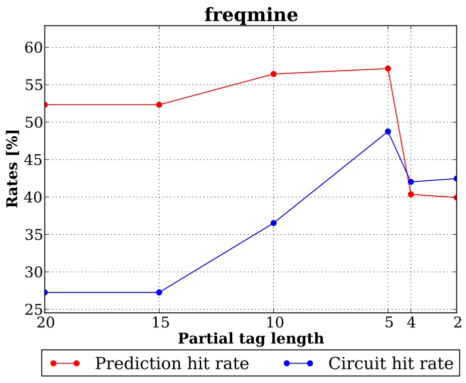
(b)



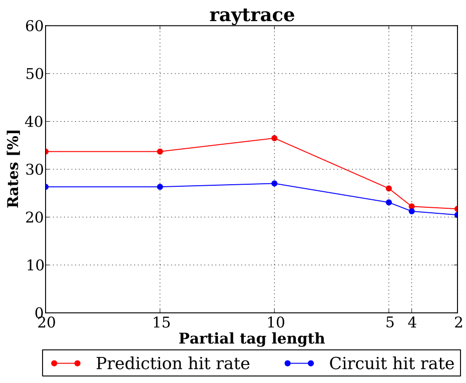
(c)



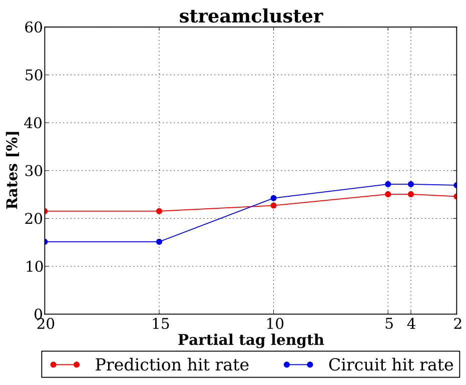
(d)



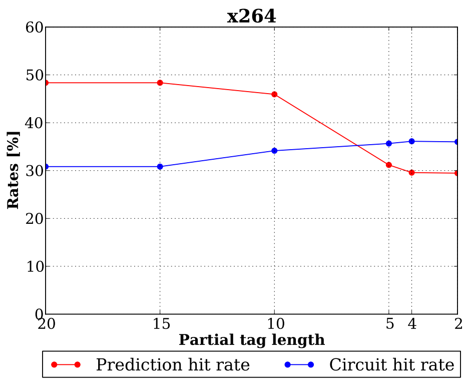
(e)



(f)

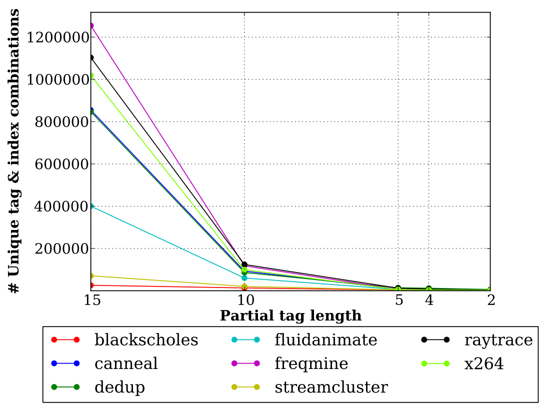


(g)



(h)

Figure 5.16: Effect of tag length on prediction & circuit hit rate



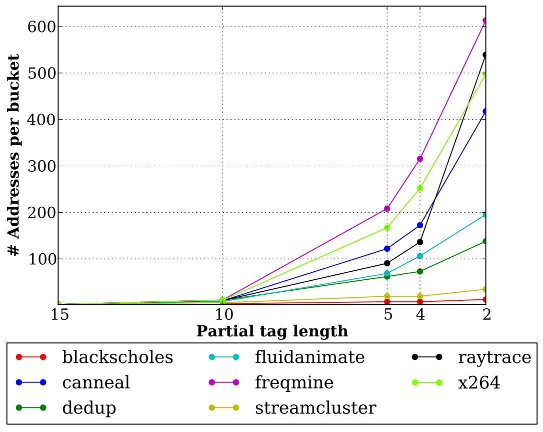


Figure 5.17: Effect of tag length on (a) the number of buckets where a bucket is the unique combination of tag and index and (b) the number of distinct addresses mapped onto each bucket,

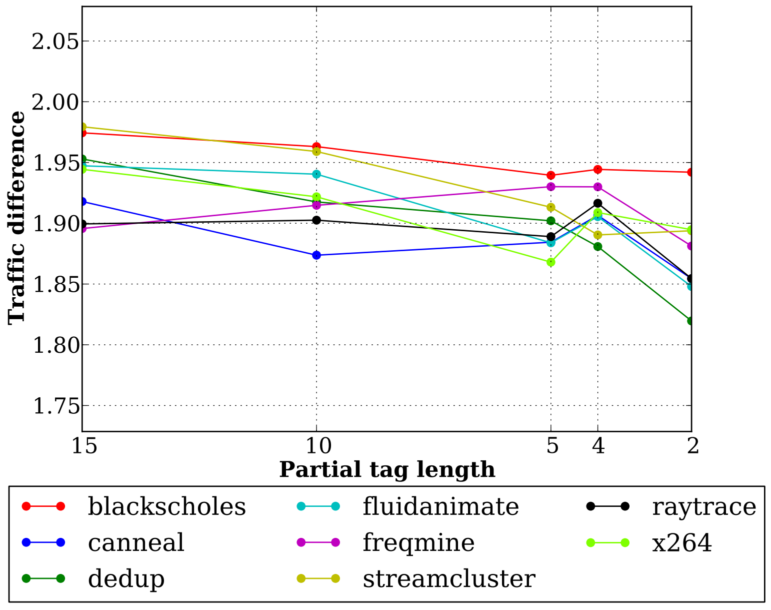


Figure 5.18: Effect of tag length on the difference in prediction information per bucket, where difference in prediction information is approximated by difference in traffic patterns

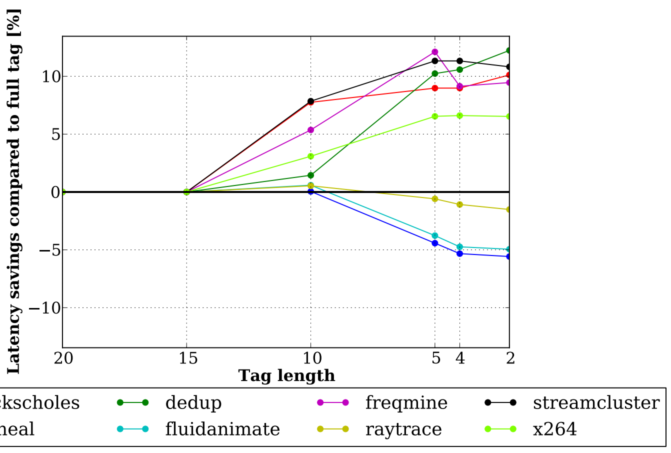
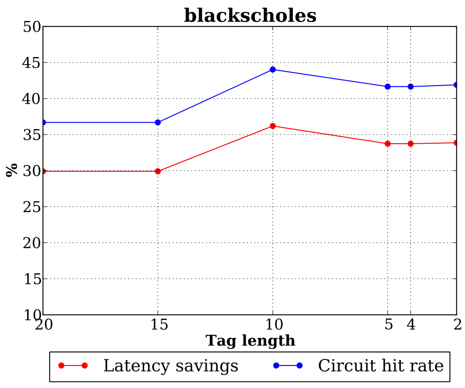
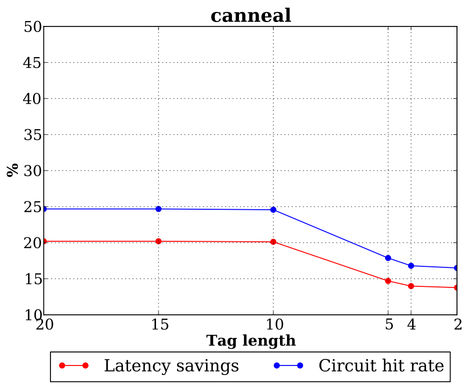


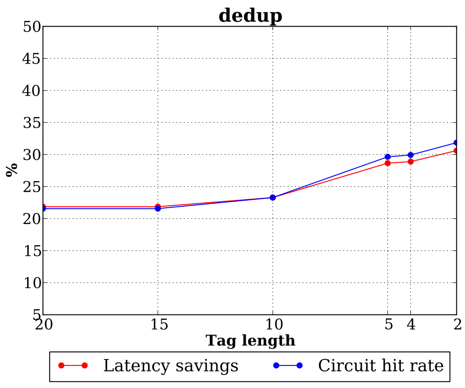
Figure 5.19: Effect of tag length of latency savings for the various PARSEC benchmarks.



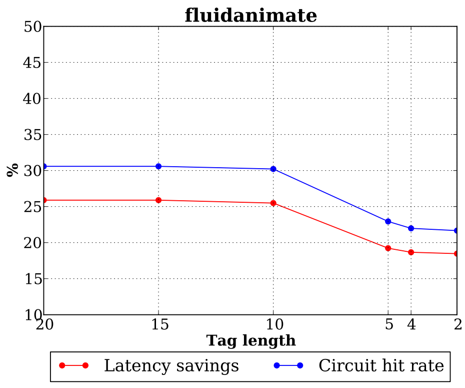
(a)



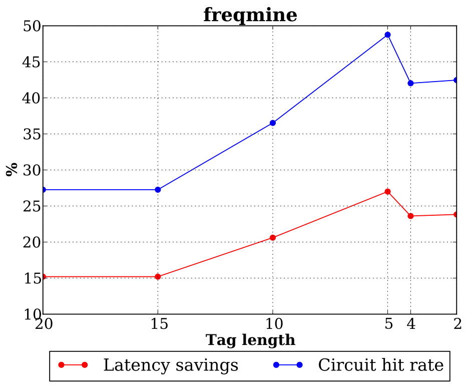
(b)



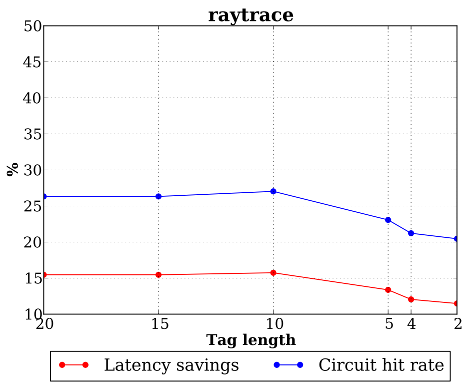
(c)



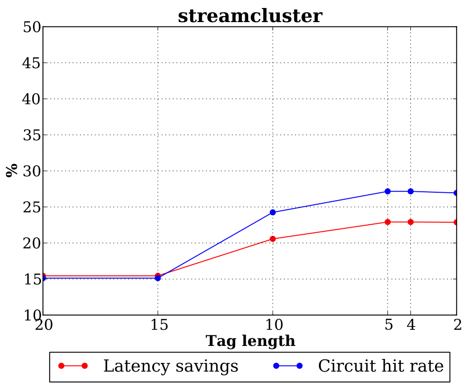
(d)



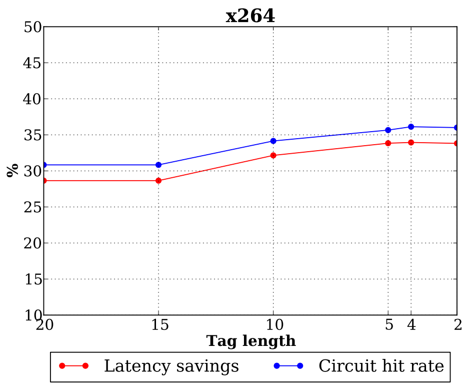
(e)



(f)



(g)



(h)

Figure 5.20: Effect of tag length on circuit hit rate & latency savings

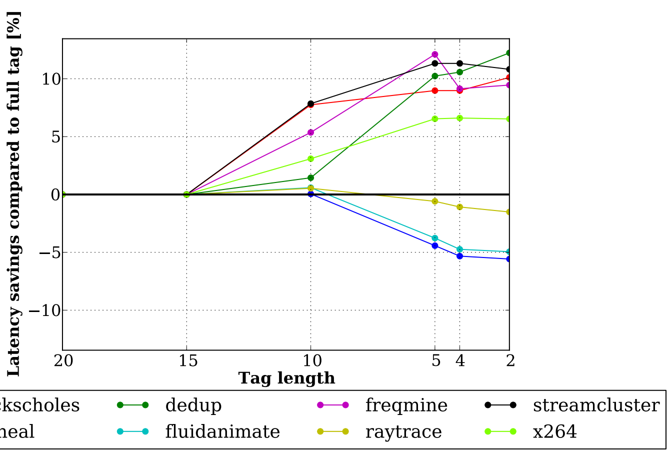


Figure 5.21: Effect of tag length of latency savings for the various PARSEC benchmarks.

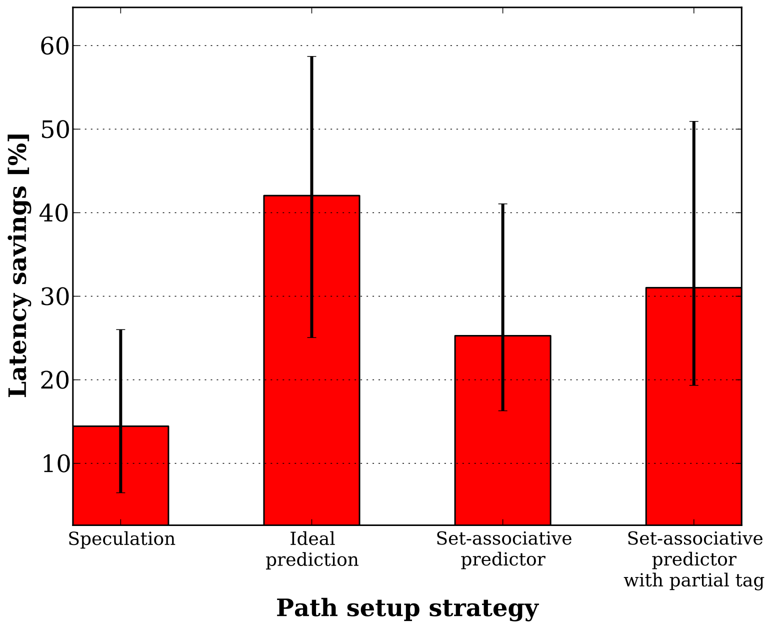


Figure 5.22: Waiting latency savings of the various path setup strategies, averaged over the PARSEC benchmarks. The error bars denote the variation across the benchmarks.

5.5 Conclusion

This chapter quantified how different path setup strategies can affect the waiting latency a message experiences in a switched optical NoC. Although a switched optical NoC with 8 wavelengths per channel was used in this chapter, the concepts presented here can be used in any type of NoC (be it electrical or optical) which needs centralised path setup. Figure 5.21 gives an overview of the savings in waiting latency per strategy, when compared to the baseline switched NoC. The error bars denote the variation across the PARSEC benchmarks. The first strategy consisted of keeping circuits open after transmission. This reduced the waiting latency by 15% on average. This strategy is most efficient for benchmarks with a high proportion of evictions per L1 miss as both the requested cache line and the victim cache line are mapped onto the same L2 slice and as such, can use the same circuit. However, by seeing messages in a NoC as part of a coherence transaction, the waiting latency can be reduced even further by using a coherence-based predictor. The maximal obtainable waiting latency saving was found by investigating an ideal predictor scheme, in which every address gets a dedicated entry in the LUT holding the prediction information. As the proposed predictor cannot handle concurrent transactions for the same address, benchmarks with a lower number of sharers per cache line react better to prediction. On average, the saving in waiting latency is reduced by 42%. To investigate the effect a realistically sized predictor could have, the size of the predictor LUT is reduced, in combination with set-associative indexing methods. Such a coherence-based predictor with a LUT of 40 kB can reduce the waiting latency by 15% on average. The predictor needs to intercept all messages transmitted on the NoC to predict upcoming circuits. The amount of information to be intercepted will affect both the latency of the interception operation (and thus the prediction delay) and the size of the predictor LUT. This leads to the third strategy which makes use of a partial tag scheme. Reducing the tag length will reduce the number of messages to be intercepted and reduce the interception delay to two clock cycles, whilst reducing the LUT size to 27 kB, smaller than a L1 cache. Reducing the tag length diminishes the prediction hit rate but as the open circuit policy takes over when less predictions arrive, the overall result is a increase in latency savings. Using a set-associative coherence-based prediction scheme with a partial tag decreases the waiting latency by 31% on average. The error bars in Figure 5.21 denote the variation across the PARSEC benchmarks, as the various benchmarks react differently to these methods. The speculative methods exhibits the least amount of variation as it is the simplest method. The effect of prediction on waiting latency will be affected by factors such as prediction hit rate, concurrency, traffic pattern and injection rate. These factors differ between all benchmarks resulting in a larger spread for predictive methods.

6 Conclusion

THE Network On-Chip (NoC) is a vital part of any Chip Multiprocessor (CMP) as it routes all the traffic between the different coherence controllers. Every message on the network is a direct or indirect consequence of a memory reference and, therefore, has the possibility of stalling the processor core. This work focussed on how the use of optics in the NoC can affect the performance of the CMP as a whole.

6.1 Thesis Summary

Optical NoCs have been proposed as an alternative to the currently accepted electrical NoCs as they offer a lower power consumption and higher bit rates. As optical buffers are not a viable option on-chip, the optical NoCs being proposed differ significantly from the electrical NoCs currently in use. In an optical NoC end-to-end paths need to be set up in advance (in the absence of buffering). In electrical NoCs on the other hand, messages are mostly transmitted on a hop-by-hop basis, using intermediate buffering. The total message latency in an optical NoC therefore, consists of two major parts: the serialisation latency and the waiting latency as was shown in Figure 4.7. The waiting latency is the time a message spends in the ingress buffers, awaiting transmission. The waiting latency is basically equal to the path setup latency as there is very little contention in NoCs as was shown in Figure 3.5. Existing optical NoC proposals therefore focus on minimising this path setup latency. One of the most viable of these proposals is the Single Writer, Multiple Reader (SWMR) scheme as discussed in Section 2.2.3. No path setup is needed as every node has a dedicated channel to which it writes whilst all other nodes read from this channel (broadcast). The disadvantage of this scheme is the high number of transmitters and receivers, which increases quadratically with node count (Table 2.7). In this thesis I proposed to use a circuit-switched NoC to keep the number of transmitters and receivers down and avoid paying the path setup latency by means of coherence-based message prediction. The crossbar-based optical NoC assumed in this thesis (see Figure 2.24) only depends linearly on the node count as was shown in Table 2.7.

This thesis can therefore be summarised by the following two questions; can an optical NoC lead to better overall CMP performance than the same CMP using an electrical mesh? And; can one get away with an optical NoC which is inherently slower by using holistic techniques i.e. can an optical circuit-switched crossbar in combination with a coherence-based predictor perform better than a SWMR-based optical NoC?

Can an optical NoC outperform an electrical mesh?

The first question was answered in Section 4.2 using full-system, cycle-accurate simulations. The electrical mesh as assumed in this thesis has links with a latency of 1 clock cycle, a router with 5 pipeline stages and a bandwidth of 8 B. The optical NoCs as defined in this thesis will modulate data on the wavelengths at a modulation speed of 25 Gbit/s . In this setup, a CMP with an optical NoC (be it circuit-switched or SWMR) will perform better than the same CMP with an electrical mesh once every channel has 2 or more wavelengths. The exact crossover point will depend on the system setup: a lower optical modulation speed or more efficient electrical mesh will move this crossover point to a higher number of wavelengths, as was discussed in Section 4.3. Figure 4.10 also showed how the different benchmarks reacted differently to the addition of an optical NoC, with the less operationally intensive benchmarks reacting more strongly. The maximal achievable speedup of ***blackscholes*** for example lies around 1% whilst the maximal achievable speedup of ***canneal*** lies around 20%.

The effect of an increasing number of wavelengths per channel also levels off for both NoC types. Most benchmarks reach this point around 8–16 wavelengths. This confirms the findings of Figure 4.5: the bandwidth of a NoC does not have the same impact as latency. Once 8 wavelengths are provided, the majority of messages (i.e. control messages) will have a serialisation latency of a single clock cycle. Any additional wavelengths will provide additional bandwidth to the NoC and decrease the serialisation latency of data messages (only 30% of all on-chip messages). Neither of these will affect the overall performance significantly.

Can a circuit-switched optical NoC outperform a SWMR based NoC?

Figure 4.9 also allows to compare the performance of the circuit-switched NoC without prediction with that of the SWMR-based NoC. When both NoC types have the same serialisation latency (i.e. the same number of wavelengths per channel), the SWMR scheme leads to better performance than the crossbar-based NoC: the overall latency will be lower as there is no path setup latency.

However, for a fairer comparison both NoC types should have the same bisection bandwidth (defined in Equation (4.2)) as this can be seen as a proxy for NoC complexity. The bisection bandwidth of the non-switched NoC is significantly larger when both NoCs have the same serialisation latency. In this case it can be calculated that the bisection bandwidth of the SWMR NoC is 3.75× that of the crossbar-based NoC in a system with 16 tiles. At low bisection bandwidths Figure 4.10 shows the crossbar-based NoC performs better than the SWMR NoC as it can provide more wavelengths per channel and, hence, a lower serialisation latency. However, there is a crossover point: once the SWMR NoC provides around 8 wavelengths per channel, it outperforms the circuit-switched NoC. This is due to the fact that for both NoC types the serialisation latency is quite low at that point (1 clock cycle for control messages) and the SWMR NoC can offer the additional benefit of the absence of any path setup.

To gauge the impact of prediction, the effect of the path setup process was exacerbated by increasing the path setup latency from 4 clock cycles (in Chapter 4) to 8 clock cycles (in Chapter 5). As the serialisation latency of both NoCs is identical, only waiting latency needs to be considered. Figure 6.1 shows the difference in waiting latency between the circuit-switched NoC and non-switched NoC without predictive methods when both optical NoCs have 8 wavelengths per channel. Averaged over all benchmarks, the waiting latency of the non-switched NoC is almost 10 clock cycles lower than that of the switched NoC. The highest reduction in waiting latency these holistic path setup methods can offer is the path setup latency itself, which is equal to 8 clock cycles and denoted by the black line. If the difference between the NoC types (determined by subtracting the latency of the non-switched NoC of the latency of the switched NoC) is larger than this value, the holistic methods simply cannot improve the performance of the switched NoC. For most benchmarks, this is the case: the gap in waiting latency between the SWMR and the crossbar scheme is too large for the predictive path setup methods to bridge it. Especially when considering that avoiding path setup for all messages is semi-impossible as the predictor is not geared at predicting all messages (e.g. L1 REQ). Figure 6.1 also shows the side-effects of a change of non-switched to switched NoC are not straightforward. The waiting latency a message experiences in a switched NoC is not the sum of the waiting latency of a message in a non-switched NoC and the path setup latency. The waiting latency is the result of the interplay between the traffic and the NoC type. This observation will make it hard to make generalised conclusions about the effect of holistic path setup methods on the waiting latency e.g. which path setup latency can be tolerated in a circuit-switched NoC, using prediction, so it could outperform a non-switched NoC.

However, if both NoC types have the same bisection bandwidth (and, hence, complexity), a circuit-switched NoC can actually outperform the non-switched NoC by using predictive path setup methods. Figure 6.2 shows the savings in average message latency when comparing a circuit-switched NoC with 8 wavelengths per channel to a non-switched NoC with 2 wavelengths per channel. These NoCs have approximately the same bisection bandwidth. To compare these two NoCs, total message latency is used rather than waiting latency. When both NoCs have the same number of wavelengths per channel, waiting latency can be used as a metric as the serialisation latency of both NoCs will be identical. However, a crossbar-based NoC with 8 wavelengths will have a significantly lower serialisation latency than the SWMR NoC with only 2 wavelengths per channel, making it necessary to use total message latency rather than waiting latency[1]. The baseline switched NoC will, in this configuration, perform slightly worse than the SWMR as can be seen in Figure 6.2. On average, the message latency in the circuit-switched NoC will be 11% larger than in the non-switched NoC, leading

[1]Both NoC types will differ in terms of complexity and layout. However, by setting the number of wavelengths per channel to 2 and 8 for the SWMR and crossbar respectively, the total number of transmitters will be relatively similar: the SWMR scheme will have 2×16^2 transmitters and receivers, the crossbar scheme on the other hand will have 16^2 transmitters and receivers, slightly less (Table 2.7. It would be difficult to exactly compare both network types in terms on complexity and component count without doing the physical implementation of the networks, which is outside of the scope of this thesis.

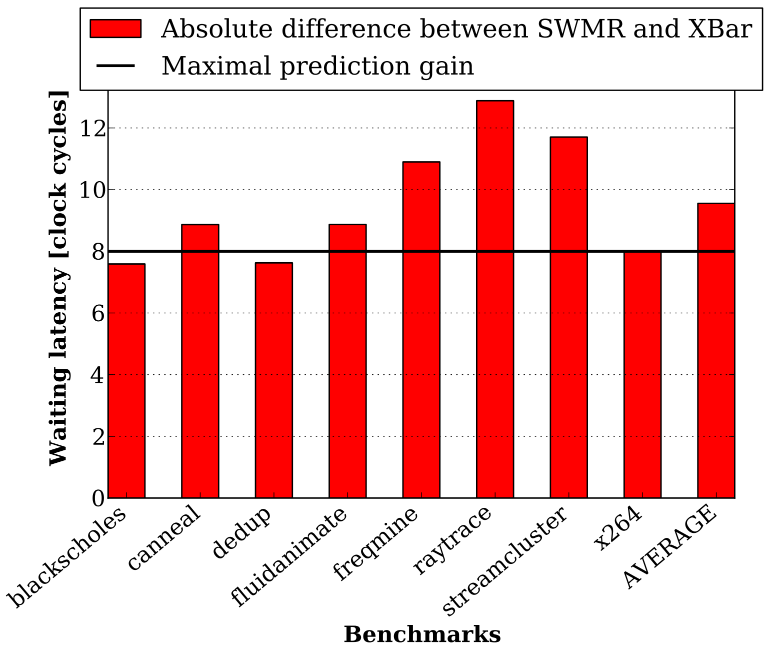


Figure 6.1: Difference in waiting latency between the the non-switched and switched optical NoCs. This difference is calculate by subtracting the non-switched latency of the switched latency and is marked as the *absolute* difference to indicate there is no normalisation done. The black line denotes the maximal achievable saving in waiting latency prediction can provide.

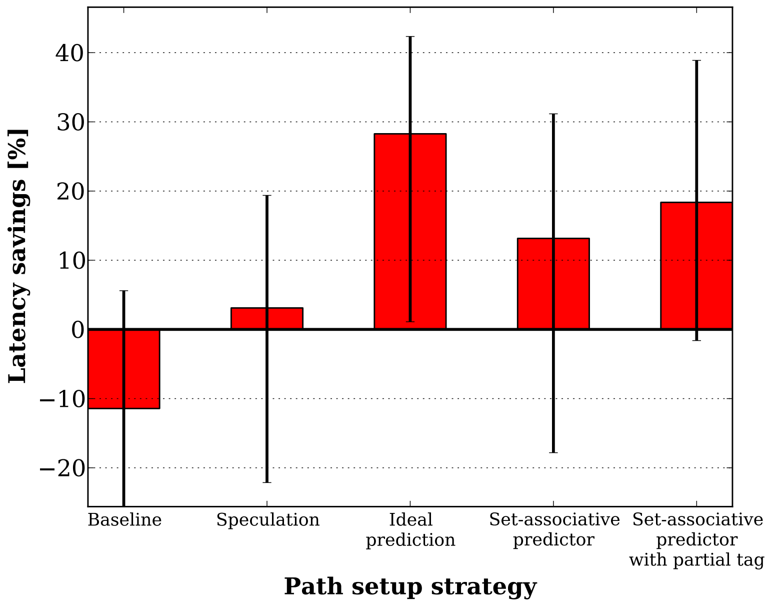


Figure 6.2: The effect of path setup strategies in switched NoC (crossbar) with 8 wavelengths per channel, normalised to non-switched (SWMR with the same bisection bandwidth. The error bars denote the variation across the benchmarks.

to negative latency savings. This result, in which the SWMR scheme outperforms the crossbar-based NoC might seem at odds with Figure 4.10 in which a crossbar with 8 wavelengths outperforms the SWMR scheme with 2 wavelengths. However, in the system depicted in Figure 4.10 the path setup latency is only 4 clock cycles whereas in this chapter path setup is assumed to take 8 clock cycles. The second bar in Figure 6.2 shows using speculative circuits in a circuit-switched NoC will, on average, lead to 3%

latency savings compared to the non-switched NoC. The use of an ideal predictor will reduce the latency even further to 28% on average. A more realistically sized predictor will have more modest latency savings of around 13%. By reducing the tag length, the predictor will not only become smaller and faster, the latency savings increases to 18% on average. The error bars in Figure 6.2 are larger than those in Figure 5.21 indicating more variation between the benchmarks. This is due to the fact that each bar in Figure 6.2 holds the comparison between two different NoC topologies and organisations. In Figure 5.21 on the other hand compares path setup methods, in the same NoC. A change in NoC topology will affect the latency to a higher degree than the addition of a speculative or predictive path setup method, resulting in a larger variation between the benchmarks. Again, it needs to be noted that even though Chapter 5 focussed on using these predictive path setup methods in optical crossbar-based NoC, they are applicable in any type of NoC with centralised arbitration.

6.2 Future Research

Whilst the two research questions at the centre of this thesis were answered, there are still avenues to be explored. First of all, the effect of the speculative and predictive techniques proposed in Chapter 5 has only been explored using traces. Whilst this is a good tool for initial exploration, the true extent of the effect of prediction can only be gauged by means of full-system simulation. Fundamentally, this is not a challenge. The predictor is already implemented in C++ and can collaborate with the allocator without a problem. The main issue lies in extracting the prediction information from the messages at runtime: I was not able to extract coherence information directly from the messages in gem5. Chapter 5 used traces containing prediction information but these traces were created in two steps: traces containing all messages and all activity in the caches were captured in gem5, followed by post-processing offline which combined the information gleaned from the caches with the messages to create messages containing the prediction information. However, this is no a fundamental limit and I am certain it can be solved.

Whilst the results presented in this thesis confirm the positive effects an optical NoC could have on the performance of a 16-core CMP, the move towards the actual use of optics in the NoC would be a very big shift. Complementary Metal Oxide Semiconductor logic (CMOS) technology used for conventional electrical interconnects is a mature technology whilst the integration of the various optical components on-chip is still in its infancy (Section 2.2.3). The performance improvements in Chapter 4 can already be seen as an argument for the use of optics in the NoC, but, nevertheless, I believe this argument can be strengthened in two ways. First of all, the core count could be increased. Optical NoCs will most likely be used for server chips, running scientific, financial and engineering workloads. These chips will contain more than 16 processor cores, as was assumed in this work. The simulation structure as is can handle such an increase in processor cores, the only hindrance is the wall clock time of the simulations: the larger the number of processor cores simulated in gem5, the lengthier the simulations become. A second way of strengthening the case for the use of optics could be made if the inclusion of optics could be expanded beyond the NoC. The macrochip as shown Figure 2.17 also proposed the topology could be used to connect multiple chips. The work by Beamer et al. for example propose the use of optics in the

Dynamic Random Access Memory (DRAM) chips [167].

A network that incorporates both the DRAM chips and all on-chip coherence controllers by using optics could level out the distinction between on-chip and off-chip communication. Such a comprehensive solution could possibly ease the step towards the use of optics on-chip. The integration of the DRAM chips in the optical NoC would lead to interesting questions. First of all, should a DRAM controller be a full node in the network? For example in the Modified Exclusive Shared Invalid (MESI) protocol as used in this thesis, the memory controllers only communicate with the Second Level Cache (L2)/directory. From this point of view, such a memory controller should not be a full and equal node in the network, as this allows for communication with every other node in the network and a memory controller would never use this functionality. On the other hand, if the memory controller were to be a full node in the network, some interesting setups could be investigated. Maybe it would be possible for the memory controllers to send a response directly to the First Level Cache (L1) that originally requested the cache line, in parallel to placing the cache line in the Last Level Cache (LLC). This would require changes to the coherence protocol but it would allow for a co-design of coherence protocol and network as done in the ATAC work (Section 2.2.3). The NoC as discussed in this thesis is completely symmetric: every node has the same number of channels and same number of wavelengths per channel. Maybe the case could be made for some asymmetry in the network as the nodes connecting the memory controllers might require a higher bandwidth and, hence, more channels and/or more wavelengths per channel. All this could be investigated in gem5 as it also contains detailed models of the memory controllers.

Another possibility would be to explore the use of more complicated modulation formats than On-Off Keying (OOK). Adding higher order modulation formats would allow for a trade-off to be made. As shown in Figure 4.10 the effect of a decreasing serialisation latency levels off at a certain wavelength count. In this thesis, the serialisation latency was decreased by adding more wavelengths to each channel. However, the serialisation latency could also be reduced by increasing the modulation speed which can be achieved by increasing the symbol rate. Using a more complicated modulation format would increase the complexity and power budget of the transmitter and receivers, but in turn the number of wavelengths per channel could be reduced. The number of wavelengths determines the number of optical components such as microrings to transmit, receive and guide (in the case of the crossbar switch). This would be an interesting compromise to investigate: optical complexity versus electrical complexity. The crossover point of this trade-off would most likely also differ between on-chip links and off-chip links to the memory controllers as the type of communication differs. The effect of a higher modulation format could be simulated in gem5 as it only emulates the higher bit rate and any additional encoding and decoding latencies. To determine the effect on electrical and optical complexity, a lower level investigation of the physical implementation of both the transmitters/receiver in the nodes and the optical NoC would be needed.

Overall this thesis has shown the benefits of a holistic approach: by knowing the functionality of the messages being transmitted, the NoC can be optimised. Some of the previously discussed ideas for future work are also system-wide approaches: co-design of the coherence protocol and the optical network incorporating the coherence and memory controllers, selective use of higher order modulation formats etc. It would also be interesting to investigate whether messages could be prioritised to obtain a form of Quality of Service (QoS) control. For example, eviction messages are transmitted at the same time as the corresponding L1 request but the eviction most likely does not lie on the critical path. An eviction by a L1 just needs to be acknowledged by the L2/directory whereas the L1 request might need to go to memory or needs invalidations from other L1 caches. Therefore, the priority of evictions could be lowered. Maybe QoS could even be applied at a higher level by slowing down or speeding down traffic on a thread-level. This could be done by assigning priorities or turning wavelengths on or off. Again, these proposals could be investigated in gem5.

References

[1] D. A. Patterson and J. L. Hennessy 2008, *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*, 4th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[2] J. L. Hennessy and D. A. Patterson 2007, *Computer Architecture, A Quantitative Approach*, 4th. San Francisco, CA, USA: Morgan Kaufmann, 2007.

[3] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo and B. Baas Jun. 2016, "A 5.8pJ/Op 115 Billion Ops/sec, to 1.78 Trillion Ops/sec 32nm 1000-Processor Array", in *Symposium on VLSI Circuits*, Honulu, USA: IEEE, Jun. 2016.

[4] Intel Corporation 2016, *Intel Xeon Phi Processor 7290*, 2016. [Online]. Available: `http://ark.intel.com/products/95830/Intel-Xeon-Phi-Processor-7290-16GB-1%7B%5C_%7D50-GHz-72-core` (visited on 22/08/2016).

[5] Mellanox Technologies 2016, *TILE-Gx72 Processor*, 2016. [Online]. Available: `http://www.mellanox.com/related-docs/prod%7B%5C_%7Dmulti%7B%5C_%7Dcore/PB%7B%5C_%7DTILE-Gx72.pdf` (visited on 22/08/2016).

[6] J. Feehrer, S. Jairath, P. Loewenstein, R. Sivaramakrishnan, D. Smentek, S. Turullols and A. Vahidsafa Mar. 2013, "The Oracle Sparc T5 16-Core Processor Scales to Eight Sockets", English, *IEEE Micro*, vol. 33, no. 2, pp. 48–57, Mar. 2013.

[7] G. Chrysos 2012, "Intel Xeon Phi Coprocessor - the Architecture", in *Proceedings of the 24th Hot Chips Symposium*, Cupertino, USA, 2012.

[8] D. A. B. Miller 1997, "Physical Reasons for Optical Interconnection", *International Journal for Optoelectronics*, vol. 11, pp. 155–168, 1997.

[9] D. A. B. Miller Jun. 2000, "Rationale and Challenges for Optical Interconnects to Electronic Chips", *Proceedings of the IEEE*, vol. 88, no. 6, pp. 728–749, Jun. 2000.

[10] I. O'Connor 2004, "Optical Solutions for System-level Interconnect", in *Proceedings of the 2004 International Workshop on System Level Interconnect Prediction*, ser. SLIP '04, New York, NY, USA: ACM, 2004, pp. 79–88.

[11] A. Van Laer, T. Jones and P. M. Watts 2013, "Full System Simulation of Optically Interconnected Chip Multiprocessors using gem5", in *Optical Fiber Communication Conference/National Fiber Optic Engineers Conference 2013*, Optical Society of America, 2013, OTh1A.2.

[12] A. Van Laer, C. Ellawala, M. R. Madarbux, P. M. Watts and T. M. Jones Mar. 2015, “Coherence Based Message Prediction for Optically Interconnected Chip Multiprocessors”, in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, Grenoble, France: EDA Consortium, Mar. 2015, pp. 613–616.

[13] K. Olukotun, L. Hammond and J. Laudon 2007, “Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency”, *Synthesis Lectures on Computer Architecture*, vol. 2, no. 1, pp. 1–145, 2007.

[14] B. Jacob, S. Ng and D. Wang 2007, *Memory Systems: Cache, DRAM, Disk*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[15] S. A. McKee 2004, “Reflections on the Memory Wall”, in *Proceedings of the 1st Conference on Computing Frontiers*, ser. CF ’04, New York, NY, USA: ACM, 2004, pp. 162–167.

[16] N. Weste and D. Harris 2010, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th. Boston, USA: Addison-Wesley Publishing Company, 2010.

[17] V. Agarwal, M. S. Hrishikesh, S. W. Keckler and D. Burger 2000, “Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures”, in *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, 2000, pp. 248–259.

[18] D. W. Wall 1991, “Limits of Instruction-Level Parallelism”, in *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS IV, New York, NY, USA: ACM, 1991, pp. 176–188.

[19] D. E. Culler, J. P. Sing and A. Gupta 1999, *Parallel Computer Architecture: A hardware/software approach*. San Francisco, CA, USA: Morgan Kaufmann, 1999.

[20] M. L. Scott Jun. 2013, “Shared-Memory Synchronization”, en, *Synthesis Lectures on Computer Architecture*, vol. 8, no. 2, pp. 1–221, Jun. 2013.

[21] Y. Hoskote, S. Vangal, A. Singh, N. Borkar and S. Borkar Sep. 2007, “A 5-GHz Mesh Interconnect for a Teraflops Processor”, *IEEE Micro*, vol. 27, no. 5, pp. 51–61, Sep. 2007.

[22] N. E. Jerger and L.-S. Peh 2009, “On-Chip Networks”, *Synthesis Lectures on Computer Architecture*, vol. 4, no. 1, pp. 1–141, 2009.

[23] M. M. K. Martin, M. D. Hill and D. J. Sorin Jul. 2012, “Why On-Chip Cache Coherence is Here to Stay”, *Communications of the ACM*, vol. 55, pp. 78–89, Jul. 2012.

[24] M. S. Papamarcos and J. H. Patel 1984, “A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories”, in *Proceedings of the 11th annual international symposium on Computer architecture*, ser. ISCA ’84, New York, NY, USA: ACM, 1984, pp. 348–354.

[25] D. J. Sorin, M. D. Hill and D. A. Wood 2011, “A Primer on Memory Consistency and Cache Coherence”, *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–212, 2011.

[26] J. Laudon and D. Lenoski 1997, "The SGI Origin: A ccNUMA Highly Scalable Server", in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ser. ISCA '97, New York, NY, USA: ACM, 1997, pp. 241–251.

[27] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe and A. Agarwal Mar. 2002, "The Raw Microprocessor: a Computational Fabric for Software Circuits and General-Purpose Programs", *IEEE Micro*, vol. 22, no. 2, pp. 25–35, Mar. 2002.

[28] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney and J. Zook Feb. 2008, "TILE64 Processor: A 64-Core SoC with Mesh Interconnect", in *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, IEEE, Feb. 2008, pp. 88–598.

[29] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III and A. Agarwal Sep. 2007, "On-Chip Interconnection Architecture of the Tile Processor", *IEEE Micro*, vol. 27, no. 5, pp. 15–31, Sep. 2007.

[30] J. Casazza 2009, "First the Tick , Now the Tock : Next Generation Intel Microarchitecture (Nehalem)", *Intel Corporation*, pp. 1–9, 2009.

[31] D. Hackenberg, D. Molka and W. E. Nagel 2009, "Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems", in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, 2009, pp. 413–422.

[32] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart and T. Mattson Feb. 2010, "A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS", in *2010 IEEE International Solid-State Circuits Conference - (ISSCC)*, IEEE, Feb. 2010, pp. 108–109.

[33] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob *et al.* 2007, "An 80-tile 1.28 TFLOPS Network-on-Chip in 65nm CMOS", in *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, 2007, pp. 98–589.

[34] P. Kongetira, K. Aingaran and K. Olukotun Mar. 2005, "Niagara: A 32-Way Multithreaded SPARC Processor", *IEEE Micro*, vol. 25, no. 2, pp. 21–29, Mar. 2005.

[35] K. Aingaran, S. Jairath, G. Konstadinidis, S. Leung, P. Loewenstein, C. McAllister, S. Phillips, Z. Radovic, R. Sivaramakrishnan, D. Smentek and T. Wicki Mar. 2015, "M7: Oracle's Next-Generation Sparc Processor", English, *IEEE Micro*, vol. 35, no. 2, pp. 36–45, Mar. 2015.

[36] M. Baron 2010, "The Single-Chip Cloud Computer", *Microprocessor Report, April*, 2010.

[37] W. J. Dally and B. Towles 2001, “Route Packets, Not Wires: On-Chip Interconnection Networks”, in *Design Automation Conference, 2001. Proceedings*, 2001, pp. 684–689.

[38] P. Guerrier and A. Greiner 2000, “A Generic Architecture for On-Chip Packet-Switched Interconnections”, in *Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings*, 2000, pp. 250–256.

[39] L. Benini and G. De Micheli 2001, “Powering Networks On Chips”, in *System Synthesis, 2001. Proceedings. The 14th International Symposium on*, 2001, pp. 33–38.

[40] S. Pasricha and N. Dutt 2010, *On-Chip Communication Architectures: System on Chip Interconnect*, ser. Systems on Silicon. Boston, USA: Morgan Kaufmann Publishers, 2010, p. 544.

[41] J. Henkel, W. Wolf and S. Chakradhar 2004, “On-Chip Networks: a Scalable, Communication-Centric Embedded System Design Paradigm”, in *VLSI Design, 2004. Proceedings. 17th International Conference on*, 2004, pp. 845–851.

[42] W. Dally and B. Towles 2003, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[43] G. Passas, M. Katevenis and D. Pnevmatikatos 2010, “A 128 x 128 x 24Gb/s Crossbar Interconnecting 128 Tiles in a Single Hop and Occupying 6 % of Their Area”, in *Proceedings of the 2010 Fourth ACM/IEEE International Symposium on Networks-on-Chip*, 2010, pp. 87–95.

[44] G. Passas, M. Katevenis and D. Pnevmatikatos 2011, “VLSI Micro-Architectures for High-Radix Crossbar Schedulers”, in *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip - NOCS ’11*, New York City, NY: ACM Press, 2011, pp. 217–224.

[45] P. M. Watts, S. W. Moore and A. W. Moore 2012, “Energy Implications of Photonic Networks with Speculative Transmission”, *Optical Communications and Networking, IEEE/OSA Journal of*, vol. 4, no. 6, pp. 503–513, 2012.

[46] A. Banerjee, P. T. Wolkotte, R. D. Mullins, S. W. Moore and G. J. M. Smit Mar. 2009, “An Energy and Performance Exploration of Network-on-Chip Architectures”, *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 17, no. 3, pp. 319–329, Mar. 2009.

[47] R. Mullins, A. West and S. Moore Jan. 2006, “The Design and Implementation of a Low-Latency On-Chip Network”, in *Design Automation, 2006. Asia and South Pacific Conference on*, Jan. 2006, pp. 164–169.

[48] M. Kandemir, O. Ozturk and S. P. Muralidhara 2009, “Dynamic Thread and Data Mapping for NoC Based CMPs”, in *Proceedings of the 46th Annual Design Automation Conference*, ser. DAC ’09, New York, NY, USA: ACM, 2009, pp. 852–857.

[49] C. R. Johns and D. A. Brokenshire Sep. 2007, “Introduction to the Cell Broadband Engine Architecture”, *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 503–519, Sep. 2007.

[50] A. Kumar, L.-S. Peh, P. Kundu and N. K. Jha 2007, "Express Virtual Channels: Towards the Ideal Interconnection Fabric", in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07, New York, NY, USA: ACM, 2007, pp. 150–161.

[51] J. Kim, W. J. Dally and D. Abts Jun. 2007, "Flattened Butterfly: A Cost-efficient Topology for High-radix Networks", *SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 126–137, Jun. 2007.

[52] R. Balasubramonian, N. P. Jouppi and N. Muralimanohar 2011, "Multi-Core Cache Hierarchies", *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–153, 2011.

[53] C. H. O. Chen, S. Park, T. Krishna, S. Subramanian, A. P. Chandrakasan and L. S. Peh Mar. 2013, "SMART: A single-cycle reconfigurable NoC for SoC applications", in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2013, pp. 338–343.

[54] N. Muralimanohar and R. Balasubramonian Jun. 2007, "Interconnect Design Considerations for Large NUCA Caches", *SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 369–380, Jun. 2007.

[55] Z. Li, J. S. Miguel and N. E. Jerger Mar. 2016, "The runahead network-on-chip", in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, Mar. 2016, pp. 333–344.

[56] J. W. Goodman, F. J. Leonberger, S.-Y. Kung and R. A. Athale Jul. 1984, "Optical Interconnection for VLSI Systems", *Proceedings of the IEEE*, vol. 72, pp. 850–866, Jul. 1984.

[57] M. Horowitz, C.-K. K. Yang and S. Sidiropoulos Jan. 1998, "High-Speed Electrical Signaling: Overview and Limitations", *IEEE Micro*, vol. 18, no. 1, pp. 12–24, Jan. 1998.

[58] R. Ho, K. W. Mai and M. A. Horowitz 2001, "The Future of Wires", *Proceedings of the IEEE*, vol. 89, no. 4, pp. 490–504, 2001.

[59] I. Papakonstantinou, D. R. Selviah, R. C. A. Pitwon and D. Milward Aug. 2008, "Low-Cost, Precision, Self-Alignment Technique for Coupling Laser and Photodiode Arrays to Polymer Waveguide Arrays on Multilayer PCBs", *IEEE Transactions on Advanced Packaging*, vol. 31, no. 3, pp. 502–511, Aug. 2008.

[60] D. A. B. Miller 2009, "Device Requirements for Optical Interconnects to Silicon Chips", *Proceedings of the IEEE*, vol. 97, no. 7, pp. 1166–1185, 2009.

[61] M. Haurylau, G. Chen, H. Chen, J. Zhang, N. A. Nelson, D. H. Albonesi, E. G. Friedman and P. M. Fauchet Nov. 2006, "On-Chip Optical Interconnect Roadmap: Challenges and Critical Directions", *IEEE Journal of Selected Topics in Quantum Electronics*, vol. 12, no. 6, pp. 1699–1705, Nov. 2006.

[62] G. Chen, H. Chen, M. Haurylau, N. A. Nelson, D. H. Albonesi, P. M. Fauchet and E. G. Friedman 2007, "Predictions of CMOS Compatible On-Chip Optical Interconnect", *Integration, the VLSI Journal*, vol. 40, no. 4, pp. 434–446, 2007.

[63] G. Chen, H. Chen, M. Haurylau, N. A. Nelson, D. H. Albonesi, P. M. Fauchet and E. G. Friedman Jun. 2006, “On-Chip Copper-Based vs. Optical Interconnects: Delay Uncertainty, Latency, Power, and Bandwidth Density Comparative Predictions”, in *Interconnect Technology Conference, 2006 International*, Jun. 2006, pp. 39–41.

[64] J. Lienig 2013, “Electromigration and Its Impact on Physical Design in Future Technologies”, in *Proceedings of the 2013 ACM International Symposium on International Symposium on Physical Design*, ser. ISPD ’13, New York, NY, USA: ACM, 2013, pp. 33–40.

[65] K.-H. Koo, H. Cho, P. Kapur and K. C. Saraswat 2007, “Performance Comparisons Between Carbon Nanotubes, Optical, and Cu for Future High-Performance On-Chip Interconnect Applications”, *Electron Devices, IEEE Transactions on*, vol. 54, no. 12, pp. 3206–3215, 2007.

[66] Q. Li, N. Ophir, L. Xu, K. Padmaraju, L. Chen, M. Lipson and K. Bergman May 2012, “Experimental characterization of the optical-power upper bound in a silicon microring modulator”, in *2012 Optical Interconnects Conference*, May 2012, pp. 38–39.

[67] N. Ophir, C. Mineo, D. Mountain and K. Bergman Jan. 2013, “Silicon Photonic Microring Links for High-Bandwidth-Density, Low-Power Chip I/O”, *IEEE Micro*, vol. 33, no. 1, pp. 54–67, Jan. 2013.

[68] B. G. Lee, B. A. Small, Q. Xu, M. Lipson and K. Bergman 2007, “Characterization of a 4x4 Gb/s Parallel Electronic Bus to WDM Optical Link Silicon Photonic Translator”, *Photonics Technology Letters, IEEE*, vol. 19, no. 7, pp. 456–458, 2007.

[69] *International Technology Roadmap for Semiconductors*, http://www.itrs2.net/itrs-reports.html, 2007. (visited on 22/08/2016).

[70] Intel Corporation 2009, *Intel Xeon Processor W3580*, 2009. [Online]. Available: `http://ark.intel.com/products/39723/Intel-Xeon-Processor-W3580-8M-Cache-3%7B%5C_%7D33-GHz-6%7B%5C_%7D40-GTs-Intel-QPI` (visited on 22/08/2016).

[71] Intel Corporation 2015, *Intel Xeon Processor E7-8867 v3*, 2015. [Online]. Available: `http://ark.intel.com/products/84681/Intel-Xeon-Processor-E7-8867-v3-45M-Cache-2%7B%5C_%7D50-GHz` (visited on 23/08/2016).

[72] C. Nitta, M. Farrens and V. Akella 2012, “Evaluating the Energy Efficiency of Microring Resonator-based On-chip Photonic Interconnects”, University of California, Davis, Tech. Rep., 2012.

[73] J. D. Meindl 2003, “Interconnect Opportunities for Gigascale Integration”, *Micro, IEEE*, vol. 23, no. 3, pp. 28–35, 2003.

[74] J. U. Knickerbocker, P. S. Andry, B. Dang, R. R. Horton, M. J. Interrante, C. S. Patel, R. J. Polastre, K. Sakuma, R. Sirdeshmukh, E. J. Sprogis, S. M. Sri-Jayantha, A. M. Stephens, A. W. Topol, C. K. Tsang, B. C. Webb and S. L. Wright 2008, “Three-Dimensional Silicon Integration”, *IBM Journal of Research and Development*, vol. 52, no. 6, pp. 553–569, 2008.

[75] A. Carpenter, J. Hu, O. Kocabas, M. Huang and H. Wu Jun. 2012, "Enhancing effective throughput for transmission line-based bus", in *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, Jun. 2012, pp. 165–176.

[76] M. Stucchi, S. Cosemans, J. Van Campenhout, Z. Tokei and G. Beyer Dec. 2013, "On-chip optical interconnects versus electrical interconnects for high-performance applications", *Microelectronic Engineering*, vol. 112, pp. 84–91, Dec. 2013.

[77] A. Naeemi, J. Xu, A. V. Mule', T. K. Gaylord and J. D. Meindl Apr. 2004, "Optical and Electrical Interconnect Partition Length Based on Chip-to-Chip Bandwidth Maximization", *Photonics Technology Letters, IEEE*, vol. 16, no. 4, pp. 1221–1223, Apr. 2004.

[78] A. F. Benner, M. Ignatowski, J. A. Kash, D. M. Kuchta and M. B. Ritter Jul. 2005, "Exploitation of optical interconnects in future server architectures", *IBM Journal of Research and Development*, vol. 49, no. 4.5, pp. 755–775, Jul. 2005.

[79] A. N. Udipi, N. Muralimanohar, R. Balsubramonian, A. Davis and N. P. Jouppi Jun. 2011, "Combining memory and a controller with photonics through 3D-stacking to enable scalable and energy-efficient systems", in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, Jun. 2011, pp. 425–436.

[80] P. Grani, R. Proietti and S. J. B. Yoo May 2016, "Scalable and Energy-Efficient AWGR-based Computing Node : Performance under PARSEC Benchmark Workload", *2016 IEEE Optical Interconnects Conference (OI)*, pp. 3–4, May 2016.

[81] W. Bogaerts, P. De Heyn, T. Van Vaerenbergh, K. De Vos, S. Kumar Selvaraja, T. Claes, P. Dumon, P. Bienstman, D. Van Thourhout and R. Baets 2012, "Silicon Microring Resonators", *Laser & Photonics Reviews*, vol. 6, no. 1, pp. 47–73, 2012.

[82] C. J. Nitta, M. K. Farrens and V. Akella Nov. 2013, "On-Chip Photonic Interconnects: A Computer Architect's Perspective", en, *Synthesis Lectures on Computer Architecture*, vol. 8, no. 5, pp. 1–111, Nov. 2013.

[83] C. Nitta, M. Farrens and V. Akella Feb. 2011, "Addressing System-Level Trimming Issues in On-Chip Nanophotonic Networks", in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, Feb. 2011, pp. 122–131.

[84] Z. Zhou, B. Yin and J. Michel Nov. 2015, "On-chip light sources for silicon photonics", *Light: Science & Applications*, vol. 4, no. 11, e358, Nov. 2015.

[85] M. Lipson Dec. 2005, "Guiding, Modulating, and Emitting Light on Silicon - Challenges and Opportunities", *Lightwave Technology, Journal of*, vol. 23, no. 12, pp. 4222–4238, Dec. 2005.

[86] H. Liu, T. Wang, Q. Jiang, R. Hogg, F. Tutu, F. Pozzi and A. Seeds Jul. 2011, "Long-wavelength InAs/GaAs quantum-dot laser diode monolithically grown on Ge substrate", *Nat Photon*, vol. 5, no. 7, pp. 416–419, Jul. 2011.

[87] N. Kirman, M. Kirman, R. K. Dokania, J. F. Martinez, A. B. Apsel, M. A. Watkins, D. H. Albonesi, A. B. Apsel, M. A. Watkins and D. H. Albonesi Dec. 2006, “Leveraging Optical Technology in Future Bus-based Chip Multiprocessors”, in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, IEEE Computer Society, Dec. 2006, pp. 492–503.

[88] H. Subbaraman, X. Xu, A. Hosseini, X. Zhang, Y. Zhang, D. Kwong and R. T. Chen Feb. 2015, “Recent advances in silicon-based passive and active optical interconnects”, *Opt. Express*, vol. 23, no. 3, pp. 2487–2511, Feb. 2015.

[89] A. V. Krishnamoorthy, R. Ho, X. Zheng, H. Schwetman, J. Lexau, P. Koka, G. Li, I. Shubin and J. E. Cunningham Jul. 2009, “Computer Systems Based on Silicon Photonic Interconnects”, *Proceedings of the IEEE*, vol. 97, no. 7, pp. 1337–1361, Jul. 2009.

[90] A. W. Poon, X. Luo, F. Xu and H. Chen 2009, “Cascaded Microresonator-Based Matrix Switch for Silicon On-Chip Optical Interconnection”, *Proceedings of the IEEE*, vol. 97, no. 7, pp. 1216–1238, 2009.

[91] Q. Wang, J. Lu and S. He Dec. 2002, “Optimal Design Method of a Low-Loss Broadband Y Branch with a Multimode Waveguide Section”, EN, *Applied Optics*, vol. 41, no. 36, p. 7644, Dec. 2002.

[92] J. Z. Huang, R. Scarmozzino and R. M. Osgood Sep. 1998, “A new design approach to large input/output number multimode interference couplers and its application to low-crosstalk WDM routers”, *IEEE Photonics Technology Letters*, vol. 10, no. 9, pp. 1292–1294, Sep. 1998.

[93] P. Koka, M. O. McCracken, H. Schwetman, C.-H. Chen, X. Zheng, R. Ho, K. Raj and A. V. Krishnamoorthy 2012, “A Micro-Architectural Analysis of Switched Photonic Multi-Chip Interconnects”, in *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, 2012, pp. 153–164.

[94] D. Vantrease 2010, “Optical Tokens in Many-Core Processors”, Ph.D. Dissertation (supervisor: Mikko Lipasti), University of Wisconsin-Madison, 2010, p. 122.

[95] J. Ahn, M. Fiorentino, R. G. Beausoleil, N. Binkert, A. Davis, D. Fattal, N. P. Jouppi, M. McLaren, C. M. Santori, R. S. Schreiber, S. M. Spillane, D. Vantrease and Q. Xu 2009, “Devices and Architectures for Photonic Chip-Scale Integration”, *Applied Physics A*, vol. 95, no. 4, pp. 989–997, 2009.

[96] Y. Pan, P. Kumar, J. Kim, G. Memik, Y. Zhang and A. Choudhary 2009, “Firefly”, in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, New York, New York, USA: ACM Press, 2009, pp. 429–440.

[97] Y. Pan, J. Kim and G. Memik Jan. 2010, “FlexiShare: Channel Sharing for an Energy-Efficient Nanophotonic Crossbar”, in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, Jan. 2010, pp. 1–12.

[98] A. Joshi, C. Batten, Y.-J. Kwon, S. Beamer, I. Shamim, K. Asanovic and V. Stojanovic May 2009, “Silicon-Photonic Clos Networks for Global On-Chip Communication”, in *Networks-on-Chip, 2009. NoCS 2009. 3rd ACM/IEEE International Symposium on*, May 2009, pp. 124–133.

[99] Y. A. Vlasov and S. J. McNab Apr. 2004, “Losses in Single-Mode Silicon-on-Insulator Strip Waveguides and Bends”, *Opt. Express*, vol. 12, no. 8, pp. 1622–1631, Apr. 2004.

[100] T. L. Koch Oct. 2006, “Opportunities and Challenges in Silicon Photonics”, in *Lasers and Electro-Optics Society, 2006. LEOS 2006. 19th Annual Meeting of the IEEE*, Oct. 2006, pp. 677–678.

[101] F. Xia, L. Sekaric and Y. Vlasov 2007, “Ultracompact Optical Buffers on a Silicon Chip”, *Nature Photonics*, vol. 1, pp. 65–71, 2007.

[102] Z. Yu, X. Jin, J. Chen, G. Wang and D. R. Selviah Aug. 2015, “Microring-based tunable optical delay lines for optical time-division multiplexers”, in *Lasers and Electro-Optics Pacific Rim (CLEO-PR), 2015 11th Conference on*, vol. 3, Aug. 2015, pp. 1–2.

[103] M. T. Hill, H. J. S. Dorren, T. de Vries, X. J. M. Leijtens, J. H. den Besten, E. Smalbrugge, Y. S. Oei, J. J. M. Binsma, G. D. Khoe and M. K. Smit 2004, “A Fast Low-Power Optical Memory based on Coupled Micro-Ring Lasers”, *Nature*, vol. 432, pp. 206–209, 2004.

[104] N. McKeown Apr. 1999, “The iSLIP Scheduling Algorithm for Input-queued Switches”, *IEEE/ACM Trans. Netw.*, vol. 7, no. 2, pp. 188–201, Apr. 1999.

[105] M. Duser and P. Bayvel 2002, “Analysis of a Dynamically Wavelength-Routed optical Burst Switched Network Architecture”, *Journal of Lightwave Technology*, vol. 20, no. 4, pp. 574–585, 2002.

[106] G. Kurian, J. E. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. C. Kimerling and A. Agarwal 2010, “ATAC: a 1000-Core Cache-Coherent Processor with On-Chip Optical Network”, in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT ’10, New York, NY, USA: ACM, 2010, pp. 477–488.

[107] G. Hendry, E. Robinson, V. Gleyzer, J. Chan, L. P. Carloni, N. Bliss and K. Bergman 2011, “Time-Division-Multiplexed Arbitration in Silicon Nanophotonic Networks-on-Chip for High-Performance Chip Multiprocessors”, *Journal of Parallel and Distributed Computing*, vol. 71, no. 5, pp. 641–650, 2011.

[108] G. Hendry, J. Chan, S. Kamil, L. Oliker, J. Shalf, L. P. Carloni and K. Bergman 2010, “Silicon Nanophotonic Network-on-Chip Using TDM Arbitration”, in *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, 2010, pp. 88–95.

[109] D. Vantrease, R. Schreiber, M. Monchiero, M. McLaren, N. P. Jouppi, M. Fiorentino, A. Davis, N. Binkert, R. G. Beausoleil and J. H. Ahn Jun. 2008, “Corona: System Implications of Emerging Nanophotonic Technology”, in *Computer Architecture, 2008. ISCA 2008. 35th International Symposium on*, Jun. 2008, pp. 153–164.

[110] D. Vantrease, N. Binkert, R. Schreiber and M. M. H. Lipasti 2009, “Light Speed Arbitration and Flow Control for Nanophotonic Interconnects”, in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture - Micro-42*, New York, New York, USA: ACM Press, 2009, pp. 304–315.

[111] G. Hendry, S. Kamil, A. Biberman, J. Chan, B. G. Lee, M. Mohiyuddin, A. Jain, K. Bergman, L. P. Carloni, J. Kubiatowicz, L. Oliker and J. Shalf May 2009, "Analysis of Photonic Networks for a Chip Multiprocessor using Scientific Applications", in *Networks-on-Chip, 2009. NoCS 2009. 3rd ACM/IEEE International Symposium on*, May 2009, pp. 104–113.

[112] A. Shacham and K. Bergman 2007, "Building Ultralow-Latency Interconnection Networks Using Photonic Integration", *Micro, IEEE*, vol. 27, no. 4, pp. 6–20, 2007.

[113] A. Shacham, B. G. Lee and K. Bergman 2005, "A Scalable, Self-Routed, Terabit Capacity, Photonic Interconnection Network", in *High Performance Interconnects, 2005. Proceedings. 13th Symposium on*, 2005, pp. 147–150.

[114] M. J. Cianchetti, J. C. Kerekes and D. H. Albonesi 2009, "Phastlane: A Rapid Transit Optical Routing Network", in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA 2009, New York, NY, USA: ACM, 2009, pp. 441–450.

[115] H. Matsutani, M. Koibuchi, H. Amano and T. Yoshinaga Feb. 2009, "Prediction Router: Yet Another Low Latency On-Chip Router Architecture", in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, Feb. 2009, pp. 367–378.

[116] N. E. Jerger, M. Lipasti and L.-S. Peh 2007, "Circuit-Switched Coherence", *Computer Architecture Letters*, vol. 6, no. 1, pp. 5–8, 2007.

[117] R. Das, O. Mutlu, T. Moscibroda and C. R. Das 2010, "Aérgia: Exploiting Packet Latency Slack in On-Chip Networks", in *ACM SIGARCH computer architecture news*, ACM, vol. 38, 2010, pp. 106–116.

[118] C. A. D. Adi, H. Matsutani, M. Koibuchi, H. Irie, T. Miyoshi and T. Yoshinaga 2010, "An Efficient Path Setup for a Photonic Network-on-Chip", in *Networking and Computing (ICNC), 2010 First International Conference on*, 2010, pp. 156–161.

[119] U. Y. Ogras and R. Marculescu 2006, "Prediction-Based Flow Control for Network-on-Chip Traffic", in *Design Automation Conference, 2006 43rd ACM/IEEE*, 2006, pp. 839–844.

[120] R. Hesse and N. Enright Jerger 2015, "Improving DVFS in NoCs with Coherence Prediction", in *Proceedings of the International Symposium on Networks on Chip*, ser. NOCS '15, New York, NY, USA: ACM, 2015, 24:1–24:8.

[121] Y. S.-C. Huang, K. C.-K. Chou and C.-T. King 2012, "Application-Driven End-to-End Traffic Predictions for Low Power NoC Design", *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. PP, no. 99, pp. 1–10, 2012.

[122] K. Wen, D. Calhoun, S. Rumley, X. Zhu, Y. Liu, L. W. Luo, R. Ding, T. B. Jones, M. Hochberg, M. Lipson and K. Bergman Aug. 2014, "Reuse Distance Based Circuit Replacement in Silicon Photonic Interconnection Networks for HPC", in *2014 IEEE 22nd Annual Symposium on High-Performance Interconnects*, Aug. 2014, pp. 49–56.

[123] M. E. Acacio, J. Gonzalez, J. M. Garcia and J. Duato 2002, “Owner Prediction for Accelerating Cache-to-Cache Transfer Misses in a cc-NUMA Architecture”, in *Supercomputing, ACM/IEEE 2002 Conference*, 2002, pp. 49–61.

[124] S. Kaxiras and C. Young 2000, “Coherence Communication Prediction in Shared-Memory Multiprocessors”, in *High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on*, 2000, pp. 156–167.

[125] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill and D. A. Wood 2003, “Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared-Memory Multiprocessors”, in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, 2003, pp. 206–217.

[126] A. K. Abousamra, R. G. Melhem and A. K. Jones May 2012, “Deja-Vu Switching for Multiplane NoCs”, in *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*, May 2012, pp. 11–18.

[127] Y. Demir and N. Hardavellas Apr. 2015, “Towards Energy-Efficient Photonic Interconnects”, in *SPIE OPTO*, H. Schröder and R. T. Chen, Eds., International Society for Optics and Photonics, Apr. 2015, 93680T.

[128] S. Ma, N. E. Jerger and Z. Wang Feb. 2012, “Supporting Efficient Collective Communication in NoCs”, in *IEEE International Symposium on High-Performance Comp Architecture*, IEEE, Feb. 2012, pp. 1–12.

[129] E. Blem, J. Menon and K. Sankaralingam 2013, “Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures”, in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–12.

[130] D. Abts, N. D. Enright Jerger, J. Kim, D. Gibson and M. H. Lipasti 2009, “Achieving Predictable Performance Through Better Memory Controller Placement in Many-core CMPs”, in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA ’09, New York, NY, USA: ACM, 2009, pp. 451–461.

[131] B. G. Lee, A. V. Rylyakov, W. M. J. Green, S. Assefa, C. W. Baks, R. Rimolo-Donadio, D. M. Kuchta, M. H. Khater, T. Barwicz, C. Reinholm, E. Kiewra, S. M. Shank, C. L. Schow and Y. A. Vlasov Feb. 2014, “Monolithic Silicon Integration of Scaled Photonic Switch Fabrics, CMOS Logic, and Device Driver Circuits”, *J. Lightwave Technol.*, vol. 32, no. 4, pp. 743–751, Feb. 2014.

[132] A. Biberman, K. Preston, G. Hendry, N. Sherwood-Droz, J. Chan, J. S. Levy, H. Wang, M. Lipson and K. Bergman 2011, “CMOS-Compatible Scalable Photonic Switch Architecture Using 3D-Integrated Deposited Silicon Materials for High-Performance Data Center Networks”, in *Optical Fiber Communication Conference/National Fiber Optic Engineers Conference 2011*, Optical Society of America, 2011, OMM2.

[133] L. Eeckhout 2010, “Computer Architecture Performance Evaluation Methods”, *Synthesis Lectures on Computer Architecture*, Synthesis lectures in computer architecture, vol. 5, no. 1, pp. 1–145, 2010.

[134] J. S. Emer and D. W. Clark 1998, “A Characterization of Processor Performance in the VAX-11/780”, in *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ser. ISCA ’98, New York, NY, USA: ACM, 1998, pp. 274–283.

[135] J. P. Shen and M. H. Lipasti 2013, *Modern Processor Design: Fundamentals of Superscalar Processors*. Long Grove, IL, USA: Waveland Press, 2013.

[136] A. R. Alameldeen and D. A. Wood Feb. 2003, “Variability in Architectural Simulations of Multi-Threaded Workloads”, in *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, Anaheim, CA, USA: IEEE Computer Society, Feb. 2003, pp. 7–18.

[137] A. R. Alameldeen and D. A. Wood Jul. 2006, “IPC Considered Harmful for Multiprocessor Workloads”, *Micro, IEEE*, vol. 26, no. 4, pp. 8–17, Jul. 2006.

[138] C. Bienia, S. Kumar, J. P. Singh and K. Li Jan. 2008, “The PARSEC Benchmark Suite: Characterization and Architectural Implications”, Princeton University, Tech. Rep. TR-811-08, Jan. 2008.

[139] N. Barrow-Williams, C. Fensch and S. Moore 2009, “A Communication Characterisation of Splash-2 and Parsec”, in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, 2009, pp. 86–97.

[140] W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery 1992, *Numerical recipes in C: the Art of Scientific Computing*. Cambridge, UK: Cambridge University Press, 1992.

[141] C. Bienia and K. Li 2009, “PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors”, Princeton, NJ, Tech. Rep., 2009, pp. 1–9.

[142] A. Van Laer, W. Wang and C. Emmons Oct. 2015, “Inefficiencies in the Cache Hierarchy: A Sensitivity Study of Cacheline Size with Mobile Workloads”, in *Proceedings of the 2015 International Symposium on Memory Systems - MEMSYS ’15*, New York, New York, USA: ACM Press, Oct. 2015, pp. 235–245.

[143] S. Williams, A. Waterman and D. Patterson Apr. 2009, “Roofline”, *Communications of the ACM*, vol. 52, no. 4, p. 65, Apr. 2009.

[144] K. Ding, Chen and Kennedy 2000, “The Memory of Bandwidth Bottleneck and its Amelioration by a Compiler”, in *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, 2000, pp. 181–189.

[145] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill and D. A. Wood Aug. 2011, “The gem5 Simulator”, *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[146] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi and S. K. Reinhardt 2006, “The M5 simulator: Modeling Networked Systems”, *IEEE Micro*, vol. 26, pp. 52–60, 2006.

[147] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill and D. A. Wood 2005, "Multifacets General Execution-Driven Multiprocessor Simulator (gems) Toolset", *SIGARCH Comput. Archit. News*, vol. 33, pp. 1–8, 2005.

[148] P. Sweazey and A. J. Smith 1986, "A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus", in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ser. ISCA '86, Los Alamitos, CA, USA: IEEE Computer Society Press, 1986, pp. 414–423.

[149] A. Butko, R. Garibotti, L. Ost and G. Sassatelli Jul. 2012, "Accuracy Evaluation of GEM5 Simulator System", in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, Jul. 2012, pp. 1–7.

[150] A. Hansson, N. Agarwal, A. Kolli, T. Wenisch and A. N. Udipi Mar. 2014, "Simulating DRAM controllers for Future System Architecture Exploration", in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, Mar. 2014, pp. 201–210.

[151] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga and N. Paver Mar. 2014, "Sources of Error in Full-System Simulation", in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, Mar. 2014, pp. 13–22.

[152] J. Chan, G. Hendry, A. Biberman, K. Bergman and L. P. Carloni Mar. 2010, "PhoenixSim: A Simulator for Physical-Layer Analysis of Chip-Scale Photonic Interconnection Networks", in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, Mar. 2010, pp. 691–696.

[153] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep and A. Agarwal 2010, "Graphite: A Distributed Parallel Simulator for Multicores", in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, 2010, pp. 1–12.

[154] S. Bartolini, L. Lusnig and E. Martinelli Sep. 2013, "Olympic: A Hierarchical All-Optical Photonic Network for Low-Power Chip Multiprocessors", in *Digital System Design (DSD), 2013 Euromicro Conference on*, Sep. 2013, pp. 56–59.

[155] D. Sanchez, G. Michelogiannakis and C. Kozyrakis May 2010, "An Analysis of On-chip Interconnection Networks for Large-scale Chip Multiprocessors", *ACM Trans. Archit. Code Optim.*, vol. 7, no. 1, 4:1–4:28, May 2010.

[156] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh and A. Gupta 1995, "The SPLASH-2 Programs: Characterization and Methodological Considerations", in *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ser. ISCA '95, New York, NY, USA: ACM, 1995, pp. 24–36.

[157] A. Jaleel, M. Mattina and B. Jacob Feb. 2006, "Last level cache (LLC) performance of data mining workloads on a CMP - a case study of parallel bioinformatics workloads", in *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, Feb. 2006, pp. 88–98.

[158] N. Agarwal, T. Krishna, L. S. Peh and N. K. Jha Apr. 2009, "GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator", in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, Apr. 2009, pp. 33–42.

[159] C. Nitta 2012, "Design and Analysis of Large Scale Nanophotonic On-Chip Networks", Ph.D. Dissertation (supervisor: Matthew Farrens), University of California Davis, 2012, pp. 1–145.

[160] G. Passas, M. Katevenis and D. Pnevmatikatos Apr. 2012, "Crossbar NoCs Are Scalable Beyond 100 Nodes", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 4, pp. 573–585, Apr. 2012.

[161] P. Surapong and M. Glesner May 2011, "On-chip Efficient Round-Robin Scheduler for High-Speed Interconnection", in *2011 22nd IEEE International Symposium on Rapid System Prototyping*, May 2011, pp. 199–202.

[162] C. Kim, D. Burger and S. W. Keckler Dec. 2002, "An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches", *ACM SIGOPS Operating Systems Review*, vol. 36, no. 5, pp. 211–222, Dec. 2002.

[163] A. Van Laer, M. R. Madarbux, P. M. Watts and T. M. Jones 2014, "Towards Zero Latency Photonic Switching in Shared Memory Networks", in *Workshop on SiPhotonics at HIPEAC*, 2014, pp. 1–8.

[164] R. E. Kessler, R. Jooss, A. Lebeck and M. D. Hill Apr. 1989, "Inexpensive Implementations of Set-Associativity", *SIGARCH Comput. Archit. News*, vol. 17, no. 3, pp. 131–139, Apr. 1989.

[165] J. Chan, G. Hendry, K. Bergman and L. P. Carloni 2011, "Physical-Layer Modeling and System-Level Design of Chip-Scale Photonic Interconnection Networks", *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 10, pp. 1507–1520, 2011.

[166] X. Zheng, F. Liu, J. Lexau, D. Patil, G. Li, Y. Luo, H. Thacker, I. Shubin, J. Yao, K. Raj, R. Ho, J. E. Cunningham and A. V. Krishnamoorthy 2011, "Ultra-Low Power Arrayed CMOS Silicon Photonic Transceivers for an 80 Gbps WDM Optical Link", in *Proc. and the National Fiber Optic Engineers Conf. Optical Fiber Communication Conf. and Exposition (OFC/NFOEC)*, 2011, pp. 1–3.

[167] S. Beamer, C. Sun, Y.-J. Kwon, A. Joshi, C. Batten, V. Stojanović and K. Asanović Jun. 2010, "Re-Architecting DRAM Memory Systems with Monolithically Integrated Silicon Photonics", *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 129–140, Jun. 2010.

7
Appendix

7.1 Coherence predictor - header file

```
#ifndef GENERALPREDICTOR_H
#define GENERALPREDICTOR_H

#include <assert.h>
#include <vector>
#include <string>
#include <stdint.h>
#include <unordered_map>
#include "EntropyCounter.hh"

class GeneralPredictor {

        // Properties
        private:
        unsigned NumTiles; // Number of nodes in the
           network
                unsigned AddressLength; // Number of bits
                    in the address
        unsigned StartBit; // First bit of the address-
           chunk that will be used to do the LUT indexing
        unsigned EndBit; // Last bit of that address-
           chunk
                uint32_t IndexMask; // Bit mask to
                   extract the bits used to do the
                   indexing
        uint32_t MSBTagMask; // Bit mask to extract the
           most significant bits of the tag
```

```
                uint32_t LSBTagMask; // Bit mask to
                   extract the least significant bits of
                   the tag
                bool PartialTag; // Bool to indicate
                   whether or not we extract a partial
                   tag mask
                uint32_t PartialTagMask; // Actual
                   partial tag bit mask
                unsigned PartialStartBit; // Start bit
                   for partial tag mask to create the
                   partial tag
                unsigned PartialEndBit; // End bit for
                   partial tag mask to create the partial
                    tag
                unsigned ValidBit;// Location of the
                   valid bit
        unsigned SequenceBit;// Location of the sequence
           bit
        unsigned SharerBit;// Location of the start of
           the sharersList
        unsigned DirectoryBit;// Location of the
           directory bit
        unsigned RequestTypeBit;// Current type of access
            to this block - LD/ST/IFETCH/EVICT
        unsigned PredictedSourceBit;// Prediction
        unsigned PredictedDestinationBit;// Prediction
                unsigned ValidPredictionBit; // Indicates
                    whether the prediction was actually
                   valid
                unsigned TimeStampBit; // Used to create
                   an arrival time distribution
                unsigned EntryLength;

                // LUT
                struct SetEntry {
                        std::string Address; // Tag
                        std::vector<uint64_t>
                           PredictionInfo; // Actual
                           prediction info
                        SetEntry* Previous; // Pointer to
                            previous entry
                        SetEntry* Next; // Pointer to
                           next entry
                };
                struct Set {
                        std::unordered_map<uint32_t,
                           SetEntry*> Entries; // Actual
                           container which holds the
                           SetEntries --> key = tag,
                           value = set_entry
                        SetEntry* Head; // Head and tail
                           are the book-ends of the
                           actual container and are only
                           used as pointers, not to hold
                           actual info
                        SetEntry* Tail;
                };
                std::unordered_map<uint32_t, Set> LUT; //
                    LUT containing the entries used for
                   path setup KEY = index (uint32_t),
                   VALUE = Set

                unsigned Granularity;
                unsigned Associativity; // If this is
                   zero, this is a directly mapped LUT

                std::string PredictorType; // Predictor
                   organization
                std::string ReplacementPolicy; //
                   Replacement policy used, right now
                   only two types --> FIFO and LRU

       // Associative setup specific
        unsigned EntriesPerSet;
        unsigned TagBit;
        unsigned LUTSize;
                unsigned NumberSets;

                // Statistics about inter-arrival time of
                    messages
                std::unordered_map<uint64_t, uint32_t>
                   StatsMessageInterArrivalTimes; //
                   Distribution of message arrival time
                uint32_t StatsArrivedMessages; // Number
                   of messages used in the distribution
                uint64_t StatsMaxInterArrivalTime; //
                   Maximal inter-arrival time

       // Statistics about the cache indexing
        std::vector<uint32_t> StatsNumberUsesPerSet;
        std::vector<uint32_t> StatsNumberConflictsPerSet;
                uint32_t StatsEntryHits; // Number of
                   accesses to the LUT that hit
                uint32_t StatsEntriesTouched; // Number
                   of LUT accesses
```

```
// Stats about the partial tag
   characteristics
uint32_t StatsEqTagDiffAddr; // Number of
    entries that had the same tag but a
   different address

//Stats about the type of predictions
   being made KEY = pred type (str) VALUE
    = occurrence
std::unordered_map<std::string, uint32_t>
    StatsPredictionType;
// Verbosity
bool Verbosity;
bool VerbosityPrintErrors;

enum SharerBitValues
{
        SB_noSharer = 0,
        SB_normalSharer, // used to '1'
        SB_currentEvictor, // '3' -->
           evicting copy of address
        SB_receivedInv, // '5' --> has
           been requested to invalidate a
            copy of address
        SB_upgrader // '7' --> upgrading
           own copy of address
};

enum StateBitValues
{
        State_noRecording = 0, // '0'
        State_requestedByLLC, // '1'
        State_presentInLLC, // '2'
        State_presentInOneL1, // '3'
        State_presentInMultL1, // '4'
        State_eviction // '5'
};

enum RequestType
{
        RT_LD = 1,
        RT_IFETCH, // '2'
        RT_ST, // '3'
        RT_EVICT, //'4'
        RT_UPGRADE, // '5'
        RT_L2_EVICT, // '6'
        RT_null
};
```

```
                // Real-time adaptive bitmask
                EntropyCounter* entropyCounter;
                bool FindOptimalBitMask; // Do we try and
                    find an optimal bitmask?

        public:

        // Functions
        public:
                GeneralPredictor(unsigned num_tiles,
                   unsigned start_bit,unsigned LUT_size,
                   unsigned associativity, unsigned
                   granularity, bool verbosity, std::
                   string predictor_type, std::string
                   replacement_pol);
                void setPartialTagMask(unsigned
                   p_start_bit, unsigned p_end_bit);
                void resetStats();
                void recordMessage(std::string address,
                   unsigned source, unsigned destination,
                    uint64_t time);
                std::pair<bool, bool> checkPrediction(
                   unsigned source_tile, unsigned
                   destination_tile,std::string address);
                std::pair<int, int>
                   predictOpticalConnection(std::string
                   address, unsigned source, unsigned
                   destination, uint64_t time,std::string
                    previous_message);
                uint32_t changeSequenceType(std::string
                   address, std::string message_type, std
                   ::string request_type, unsigned
                   source_node);
                void changeVerbosity (bool new_verbosity)
                {
                        Verbosity = new_verbosity;
                        VerbosityPrintErrors =
                           new_verbosity;
                }
                void startTrackingBitMask() {
                   FindOptimalBitMask = true; }
                std::vector<uint32_t> getUsesPerSet() {
                   return StatsNumberUsesPerSet; }
                std::vector<uint32_t> getConflictsPerSet
                   () { return StatsNumberConflictsPerSet
                   ; }
                uint32_t getEntryHits() { return
                   StatsEntryHits; }
                uint32_t getEntriesTouched() { return
                   StatsEntriesTouched; }
                uint32_t getEntriesEqTagDiffAddr() {
                   return StatsEqTagDiffAddr; }
                uint32_t getReplacements();
                std::unordered_map<std::string, uint32_t>
                    getPredTypes() { return
                   StatsPredictionType; }
                // Message inter-arrival time
                   distribution functions
                void printDistributionToFile(std::string
                   file_name);
                void resetTimings(uint64_t current_time);
                    // To be called when the
                   TimeStampBits need to be reset, after
                   fast forwarding
                void resetTimeStamp(std::string address,
                   uint64_t current_time);
                void addToDistribution(std::string
                   address, uint64_t time);
                std::string getPredProperties();
                std::string getPartialTagProperties();
                // Information functions
                const unsigned getAssociativity() {
                   return Associativity; }
                const unsigned getStartBit() { return
                   StartBit; }
                const bool usesPartialTag() { return
                   PartialTag; }
                const unsigned getPartialTagLength()
                {
                        assert(PartialTag);
                        return (PartialStartBit -
                           PartialEndBit);
                }
                const unsigned getPartialTagStartBit()
                {
                        assert(PartialTag);
                        return PartialStartBit;
                }

        private:
                // LUT addressing related functions
                void findEntry(std::string address);
                void makeNewSet(uint32_t set_number);
                bool isEntryPresentInSet(std::string
                   address, uint32_t set_number, uint32_t
```

```
    tag);
bool isEntryPresent(std::string address);
bool spaceInSet(uint32_t set_number);
void addEntryToSet(std::string address,
   uint32_t set_number, uint32_t tag);
std::string findVictim(uint32_t
   set_number);
void clearVictim(uint32_t set_number, std
   ::string victim_address);
long unsigned convertHexToInt(std::string
    address);
uint32_t extractIndex(std::string address
   );
uint32_t extractTag(std::string address);
uint32_t getSetNumber(uint32_t index);
// Actual recording of functions
void recordSharers(uint32_t source_tile,
   uint32_t destination_tile, unsigned
   source, unsigned destination, uint64_t
    time, std::string address);

// Helper functions
uint32_t convertNodeID(unsigned node_id);
uint32_t returnDirectory(std::string
   address, uint32_t set_number, uint32_t
    tag);
int returnUniqueSharer(std::string
   address, uint32_t set_number, uint32_t
    tag);
std::vector<unsigned> returnSharers(std::
   string address, uint32_t set_number,
   uint32_t tag);
unsigned getNumSharers(std::string
   address, uint32_t set_number, uint32_t
    tag);
void removeSharer(std::string address,
   uint32_t set_number, uint32_t tag,
   unsigned sharer);
int returnEvictor(std::string address,
   uint32_t set_number, uint32_t tag);
void setAsEvictor(std::string address,
   uint32_t set_number, uint32_t tag,
   unsigned source_node);
void hasReceivedInv(std::string address,
   uint32_t set_number, uint32_t tag,
   unsigned node);
int getNextSharerToInv(std::string
   address, uint32_t set_number, uint32_t
```

```
                    tag);
                int getNextSharerToInvAck(std::string
                   address, uint32_t set_number, uint32_t
                    tag);
                unsigned getNumSharersToInv(std::string
                   address, uint32_t set_number, uint32_t
                    tag);
                void setUpgrader(std::string address,
                   uint32_t set_number, uint32_t tag,
                   unsigned node);
                int getUpgrader(std::string address,
                   uint32_t set_number, uint32_t tag);
                void setUpgraderAsNormalSharer(std::
                   string address, uint32_t set_number,
                   uint32_t tag,unsigned node);
                uint32_t convertAccessType(std::string
                   type);

};


#endif // GENERALPREDICTOR_H
```

7.2 Coherence predictor - code file

```
#include "GeneralPredictor.hh"
#include <stdlib.h>     /* strtol */
#include <math.h> /* for ceil, floor */
#include <stdio.h>
#include <algorithm>
#include <fstream>  /* for print */
#include <sstream> /* for string stream */

// Constructor
GeneralPredictor::GeneralPredictor(unsigned num_tiles,
   unsigned start_bit, unsigned LUT_size, unsigned
   associativity, unsigned granularity, bool verbosity,
   std::string predictor_type, std::string
   replacement_pol)
{
        // Setup which bits correspond to which
           information in the LUT entries
        NumTiles = num_tiles;
        Associativity = associativity;
        TagBit = 0;
        SequenceBit = TagBit + 1;
        SharerBit = SequenceBit + 1;
        ValidBit = SharerBit + NumTiles;
```

```
DirectoryBit = ValidBit + 1;
TimeStampBit = DirectoryBit + 1;
RequestTypeBit = TimeStampBit + 1;
PredictedSourceBit = RequestTypeBit + 1;
PredictedDestinationBit = PredictedSourceBit + 1;
ValidPredictionBit = PredictedDestinationBit + 1;

// Setup the LUT
EntryLength = ValidPredictionBit + 1; // Size is
   one extra
LUTSize = LUT_size;

// Set some options depending on whether or not
   this is set-associative LUT or  directly
   mapped one
if (predictor_type == "directly_mapped"){
        NumberSets = LUTSize;
        EntriesPerSet = 1;
        float gran = log(LUT_size)/log(2);
        Granularity = floor(gran);
} else if (predictor_type == "fully_associative")
    {
        NumberSets = 1;
        EntriesPerSet = LUTSize;
        Granularity = granularity;
} else if (predictor_type == "set_associative"){
        NumberSets = LUTSize / associativity;
        EntriesPerSet = associativity;
        Granularity = granularity;
} else {
        printf("Not a valid LUT organization");
        exit(1);
}
PredictorType = predictor_type;
ReplacementPolicy = replacement_pol;

// Decide upon the address-chunk to be used for
   the index
AddressLength = 32;
assert(start_bit <= 32);
StartBit = AddressLength - start_bit;
int possible_end_bit = StartBit - Granularity +
   1; // + 1 to include the EndBit as well
EndBit = std::max(0, possible_end_bit);
IndexMask = 0;
MSBTagMask = 0;
LSBTagMask = 0;
printf("Start bit %u and end bit %u, possible end
    bit %i (granularity %u) \n", StartBit, EndBit
   , possible_end_bit, Granularity);
// Going from MSB to LSB
for (unsigned i = (AddressLength-1); (int) i >= 0
    ; i--) {
        if ((i <= StartBit) && (i >= EndBit) ) {
                // Part of the index
                IndexMask += pow(2,i); // Set
                   this bit to 1
        } else {
                // Part of the tag
                if (i < EndBit) {
                        LSBTagMask += pow(2,i);
                           // Set this bit to 1
                } else {
                        assert(i > EndBit);
                        MSBTagMask += pow(2,i);
                }
        }
}
// Partial tag mask setting --> need to be set
   via specific function
PartialTag = false;
PartialTagMask = 1;
PartialStartBit = 0;
PartialEndBit = 0;
// Verbosity levels
Verbosity = verbosity;
VerbosityPrintErrors = verbosity;

// Make entropy counter but do not use it
unsigned saturation_counters = 4;
unsigned saturation_messages = 10000;
FindOptimalBitMask = false;
if ( FindOptimalBitMask) {
        entropyCounter = new EntropyCounter(
           AddressLength,saturation_counters,
           saturation_messages, Granularity,
           StartBit, Verbosity);
} else {
        entropyCounter = NULL;
}
printf("Made a %s LUT (replacement policy %s)
   with %i entries, %i fields per entry \n",
   predictor_type.c_str(),ReplacementPolicy.c_str
   (),LUT_size, EntryLength);
printf("Bitmask used for indexing is %u (start
   bit %u, granularity %u) \n", IndexMask,
            StartBit, StartBit - EndBit);
        printf("Tag mask for MSB = %u, tag mask for LSB =
            %u \n", MSBTagMask, LSBTagMask);

        resetStats();
}

void
GeneralPredictor::setPartialTagMask(unsigned p_start_bit,
    unsigned p_end_bit)
{
        PartialTag = true;
        PartialTagMask = 0;
        PartialStartBit = p_start_bit;
        PartialEndBit = p_end_bit;
        unsigned tag_length = AddressLength - (StartBit -
            EndBit);
        for (unsigned i = (tag_length - 1); (int) i >= 0;
            i--) {
                if ((i <= PartialStartBit) && (i >=
                    PartialEndBit)) {
                        PartialTagMask += pow(2,i);
                }
        }
        printf("Partial mask = %u, partial start bit = %u
            & partial end bit = %u, tag length = %u \n",
           PartialTagMask, PartialStartBit, PartialEndBit
           , tag_length);
}
void
GeneralPredictor::resetStats()
{
        StatsNumberUsesPerSet.clear();
        StatsNumberUsesPerSet.resize(NumberSets, 0);
        StatsNumberConflictsPerSet.clear();
        StatsNumberConflictsPerSet.resize(NumberSets, 0);
        StatsEntryHits = 0;
        StatsEntriesTouched = 0;
        StatsMessageInterArrivalTimes.clear();
        StatsArrivedMessages = 0;
        StatsMaxInterArrivalTime = 0;
        StatsEqTagDiffAddr = 0;
        StatsPredictionType.clear();
}

//////////////////////////////////////////////
// BASIC HELPER FUNCTION TO FIND ENTRY IN LUT
/////////////////////////////////////////////
```

```
void
GeneralPredictor::findEntry(std::string address)
{
        // Get set number
        uint32_t index = extractIndex(address);
        uint32_t set_number = getSetNumber(index);
        uint32_t tag = extractTag(address);

        if (Verbosity) {
                printf("LUT: Finding an entry for address
                    %s (index %u, set %u, tag %u) \n",
                   address.c_str(), index, set_number,
                   tag);
        }

        // Is this entry present in the LUT?
        bool set_present = LUT.count(set_number) > 0 ?
           true : false;
        bool entry_present = false;
        if (set_present) {
                entry_present = isEntryPresentInSet(
                   address, set_number, tag);
        }

        if (entry_present) {
                // Entry is present, do nothing
                if (Verbosity) {
                        printf("LUT: Address was already
                           present in LUT -- nothing
                           needed to be done \n");
                }
                StatsEntryHits++;
                if (FindOptimalBitMask) {
                        // Update the cycle count in the
                           entropy counter --> no new
                           address
                        entropyCounter->updateCycleCount
                           ();
                }
        } else {
                // Is the set present?
                if (set_present) {
                        // Make new entry or find victim
                        if (spaceInSet(set_number)) {
                                if (Verbosity) {
                                        printf("LUT: Set
                                           was present
                                           and stil space
                                            in set \n");
                                }
                                // There is still space
                                   in set
                                // Add entry to Set
                                addEntryToSet(address,
                                   set_number, tag);
                        } else {
                                if (Verbosity) {
                                        printf("LUT: Set
                                           was present
                                           but no space
                                           in set (%zi
                                           entries) --
                                           need to find
                                           victim \n",LUT
                                           [set_number].
                                           Entries.size()
                                           );
                                }
                                // Find and clear the
                                   victim
                                std::string
                                   victim_address =
                                   findVictim(set_number)
                                   ;
                                clearVictim(set_number,
                                   victim_address);
                                // Update statistics
                                StatsNumberConflictsPerSet
                                   [set_number]++;
                                // Add entry to Set
                                addEntryToSet(address,
                                   set_number, tag);
                        }
                } else {
                        // Not even the set is present
                        // Make the Set
                        if (Verbosity) {
                                printf("LUT: Set not yet
                                   present, so need to
                                   make set \n");
                        }
                        makeNewSet(set_number);
                        addEntryToSet(address, set_number
                           , tag);
                        assert(LUT.size() <= NumberSets);
```

```
                            // No more sets than there is
                            space
                }

                if (FindOptimalBitMask) {
                        // Update the entropy counter as
                           this a new entry
                        entropyCounter->updateCounters(
                           address);
                }
        }

        // If partial tag -- update the address in the
           entry
        if (PartialTag) {
                // Considering multiple addresses can
                   have the same partial tag we need to
                   update the address in setEntry
                // @TODO add stats
                if (LUT[set_number].Entries[tag]->Address
                    != address) {
                        StatsEqTagDiffAddr++;
                }
                LUT[set_number].Entries[tag]->Address =
                   address;
        }
        // Entry has either been added or has been
           touched so now it is time to do replacement
           admin
        if (ReplacementPolicy == "LRU") {
                // This address was the LRU so needs to
                   become closest the new head
                // Downgrade the previous sub-head
                assert(LUT[set_number].Entries.count(tag)
                    > 0);
                if (Verbosity) {
                        printf("REPL: Original situation
                           in set %u (%zi entries) (from
                           head to tail): Tail -> ",
                           set_number, LUT[set_number].
                           Entries.size());
                        SetEntry* looking_at = LUT[
                           set_number].Tail->Previous;
                        while (looking_at->Address != LUT
                           [set_number].Head->Address) {
                                printf(" Addr %s ->",
                                   looking_at->Address.
                                   c_str());
                                SetEntry* prev =
                                   looking_at->Previous;
                                looking_at = prev;
                        }
                        printf("Head \n");
                }

                SetEntry* previous_sub_head = LUT[
                   set_number].Head->Next;
                SetEntry* new_sub_head = LUT[set_number].
                   Entries[tag];
                if (Verbosity) {
                        printf("addr %s Prev subhead = %s
                           , new subhead =  %s \n",
                           address.c_str(),
                           previous_sub_head->Address.
                           c_str(), new_sub_head->Address
                           .c_str());
                }
                if (previous_sub_head->Address == address
                   ) {
                        // Not changing LRU state
                        if (Verbosity) {
                                printf("REPL: Address %s
                                   was already most LRU
                                   in set %u (next = %s,
                                   previous %s) \n",
                                   address.c_str(),
                                   set_number,
                                   previous_sub_head->
                                   Next->Address.c_str(),
                                    previous_sub_head->
                                   Previous->Address.
                                   c_str());
                        }
                } else {
                        if (entry_present) {
                                // The entry was already
                                   present in the LUT
                                // So first the existing
                                   connections need to be
                                    broken
                                SetEntry*
                                   above_new_sub_head =
                                   new_sub_head->Previous
                                   ;
                                SetEntry*
                                   under_next_sub_head =
                                   new_sub_head->Next;
                                above_new_sub_head->Next
                                   = under_next_sub_head;
                                under_next_sub_head->
                                   Previous =
                                   above_new_sub_head;
                                if (Verbosity) {
                                        printf("REPL: new
                                            sub head was
                                           already
                                           present so
                                           broke all
                                           connections \n
                                           ");
                                }
                        }
                        // Make the new entry the sub-
                           head
                        previous_sub_head->Previous =
                           new_sub_head; // Previous of
                           sub-head WAS the head, but
                           will now become the new entry
                        new_sub_head->Previous = LUT[
                           set_number].Head;
                        LUT[set_number].Head->Next =
                           new_sub_head;
                        new_sub_head->Next =
                           previous_sub_head;
                        if (Verbosity) {
                                printf("REPL: Address %s
                                   has been marked as
                                   most recently used in
                                   set %u (%s was
                                   previous sub-head) \n"
                                   ,address.c_str(),
                                   set_number,
                                   previous_sub_head->
                                   Address.c_str());
                        }
                }
                if (Verbosity) {
                        printf("REPL: New situation in
                           set %u (%zi entries) (from
                           tail to head): Tail -> ",
                           set_number, LUT[set_number].
                           Entries.size());
                        SetEntry* looking_at = LUT[
                           set_number].Tail->Previous;
                        while (looking_at->Address != LUT
                           [set_number].Head->Address) {
                                printf(" Addr %s ->",
                                   looking_at->Address.
                                   c_str());
                                SetEntry* prev =
                                   looking_at->Previous;
                                looking_at = prev;
                        }
                        printf("Head \n");
                }

        } else if (ReplacementPolicy == "FIFO") {
                // The entry only needs to become the sub
                   -head IF it was new
                if (!entry_present) {
                        // Downgrade the previous sub-
                           head
                        SetEntry* previous_sub_head = LUT
                           [set_number].Head->Next;
                        assert(LUT[set_number].Entries.
                           count(tag) > 0);
                        assert(previous_sub_head != NULL)
                           ;
                        assert(previous_sub_head->
                           Previous != NULL);
                        previous_sub_head->Previous = LUT
                           [set_number].Entries.at(tag);
                           // Previous of sub-head WAS
                           the head, but will now become
                           the new entry

                        // Make the new entry the sub-
                           head
                        LUT[set_number].Entries[tag]->
                           Previous = LUT[set_number].
                           Head;
                        LUT[set_number].Head->Next = LUT[
                           set_number].Entries[tag];
                        LUT[set_number].Entries[tag]->
                           Next = previous_sub_head;
                        if (Verbosity) {
                                printf("REPL: Address %s
                                   (tag %u) has been
                                   marked as last
                                   addition in set %u \n"
                                   ,address.c_str(),tag,
                                   set_number);
                                }
                        }
                } else {
                        printf("No replacement policy specified")
                           ;
                        exit(EXIT_FAILURE);
                }
                // Update the stats
                StatsNumberUsesPerSet[set_number]++;
                StatsEntriesTouched++;
}

// Function to make new set
void
GeneralPredictor::makeNewSet(uint32_t set_number)
{
                // Make the set
                Set new_set;
                SetEntry* head = new SetEntry;
                SetEntry* tail = new SetEntry;
                head->Previous = NULL;
                head->Next = tail;
                head->Address = "HEAD";
                tail->Next = NULL;
                tail->Previous = head;
                tail->Address = "TAIL";
                new_set.Head = head;
                new_set.Tail = tail;

                // Push the set back into the LUT
                LUT.insert(std::make_pair(set_number,new_set));
                if (Verbosity) {
                        printf("LUT: Created new set with set
                           number %u \n", set_number);
                }
}

// Function to check a specific set for an address
bool
GeneralPredictor::isEntryPresentInSet(std::string address
     , uint32_t set_number, uint32_t tag)
{
                bool entry_present = LUT[set_number].Entries.
                     count(tag) > 0 ? true : false;
                return entry_present;
}

// Function to check whether an address has an entry in
```

```
   the LUT
bool
GeneralPredictor::isEntryPresent(std::string address)
{
        uint32_t set_number = getSetNumber(extractIndex(
           address));
        bool set_present = LUT.count(set_number) > 0 ?
           true : false;
        bool entry_present = false;
        int32_t tag = extractTag(address);
        if (set_present) {
                entry_present = isEntryPresentInSet(
                   address, set_number, tag);
        }
        if (Verbosity) {
                printf("LUT: Address %s (index %u, set
                   number %u, tag %u) --> set present: %s
                    entry present: %s \n", address.c_str
                   (), extractIndex(address), set_number,
                    tag, set_present ? "YES": "NO",
                   entry_present ? "YES" : "NO");
        }
        return entry_present;
}

// Fuction to check whether there is space in the set for
    a new entry
bool
GeneralPredictor::spaceInSet(uint32_t set_number)
{
        bool space_in_set = LUT[set_number].Entries.size
           () < EntriesPerSet ? true : false;
        return space_in_set;
}

// Function to add a new entry to a set
void
GeneralPredictor::addEntryToSet(std::string address,
   uint32_t set_number, uint32_t tag)
{
        SetEntry* set_entry = new SetEntry;
        set_entry->Address = address; // We still just
           store the complete address, easier
        set_entry->PredictionInfo = std::vector<uint64_t>
            (EntryLength, 0);
        set_entry->Next = NULL;
        set_entry->Previous = NULL;
        LUT[set_number].Entries.insert(std::make_pair(tag
           , set_entry));
        if (Verbosity) {
                printf("LUT: Added new entry for address
                   %s to set %u (still empty) \n",
                   address.c_str(), set_number);
        }
}

// Function to find a victim
std::string
GeneralPredictor::findVictim(uint32_t set_number)
{
        assert(LUT[set_number].Entries.size() ==
           EntriesPerSet);
        // Tail-entry is the one up for removal
        std::string victim_address = LUT[set_number].Tail
           ->Previous->Address;
        if (Verbosity) {
                printf("REPL: Victim in set %u is %s \n",
                    set_number, victim_address.c_str());
        }
        assert(LUT[set_number].Tail->Previous != LUT[
           set_number].Head);
        return victim_address;
}

// Function to remove a victim from the set
void
GeneralPredictor::clearVictim(uint32_t set_number, std::
   string victim_address)
{
        uint32_t victim_tag = extractTag(victim_address);
        SetEntry* victim = LUT[set_number].Entries[
           victim_tag];
        if (Verbosity) {
                printf("REPL: Removing the links from the
                    victim %s (tag %u) \n",
                   victim_address.c_str(), victim_tag);
        }
        assert(victim->Next == LUT[set_number].Tail);
        // Connect the previous and next pointers
        assert(victim != NULL);
        assert(victim->Previous != NULL);
        assert(victim->Previous->Next != NULL);
        assert(victim->Next != NULL);
        if (Verbosity) {
                printf("REPL: Victim's previous = %s
                   victim's next = %s \n", victim->
```

```
                        Previous->Address.c_str(), victim->
                        Next->Address.c_str());
          }
          SetEntry* head_side_victim = victim->Previous;
          SetEntry* tail_side_victim = victim->Next;
          head_side_victim->Next = tail_side_victim;
          tail_side_victim->Previous = head_side_victim;
          victim->Previous = NULL;
          victim->Next = NULL;
          victim->Address = "null";
          victim->PredictionInfo.clear();
          // Erase the entry
          delete victim;
          LUT[set_number].Entries.erase(victim_tag);
          assert(LUT[set_number].Tail->Previous ==
             head_side_victim);
}

// Function to convert a string containing a hex number
   to a long int
long unsigned
GeneralPredictor::convertHexToInt(std::string address)
{
          long unsigned address_unsigned = std::stoul(
             address.c_str(), NULL, 16);
          return address_unsigned;
}

// Function to extract the index
uint32_t
GeneralPredictor::extractIndex(std::string address)
{
          uint32_t address_unsigned =  (uint32_t)
             convertHexToInt(address);
          uint32_t index = IndexMask & address_unsigned;
          index >>= EndBit;
          if (Verbosity) {
                    printf("IND: Address %s gives index %u (%
                       u & %u) \n", address.c_str(), index,
                       IndexMask, address_unsigned);
          }
          return index;
}

// Function to extract the tag
uint32_t
GeneralPredictor::extractTag(std::string address)
{
          uint32_t address_unsigned = (uint32_t)
             convertHexToInt(address);
          // MSB part of the tag
          uint32_t ms_tag = MSBTagMask & address_unsigned;
          unsigned actual_granularity = StartBit - EndBit;
          ms_tag >>= actual_granularity; // Shift to the
             right, to close the 'index-gap'
          // LSB part of the tag
          uint32_t ls_tag = LSBTagMask & address_unsigned;
          uint32_t tag = ms_tag | ls_tag;
          if (Verbosity) {
                    printf("TAG: Address %s give tag %u (
                       ms_tag %u and ls_tag %u) \n", address.
                       c_str(), tag, ms_tag, ls_tag);
          }
          if (PartialTag) {
                    tag &= PartialTagMask;
                    tag >>= PartialEndBit;
                    if (Verbosity) {
                            printf("TAG: shortened tag to %u
                               \n", tag);
                    }
          }
          return tag;
}

// Function to get set number
uint32_t
GeneralPredictor::getSetNumber(uint32_t index)
{
          //uint32_t set_number = floor(index/EntriesPerSet
             );
          uint32_t set_number = index;
          if (PredictorType == "fully_associative") {
                    set_number = 0;
          }
          return set_number;
}


////////////////////////////////////
// METHODS USED TO MAKE A PREDICTION
////////////////////////////////////

// Function to simply record a message -- source and
   destination are added to the sharersList
void
GeneralPredictor::recordMessage(std::string address,
   unsigned source, unsigned destination, uint64_t time)
{
          if (Verbosity) {
                    printf("REC: Recording message from %u to
                       %u for address %s at time %zu \n",
                       source, destination, address.c_str(),
                       time);
          }
          // Find the entry for this address
          findEntry(address);

          // Convert source and destination to their
             tileIDs
          uint32_t source_tile = convertNodeID(source);
          uint32_t destination_tile = convertNodeID(
             destination);

          // Record the sharers
          recordSharers(source_tile,destination_tile,source
             ,destination,time,address);
}

// Actual recording of the sharers
void
GeneralPredictor::recordSharers(uint32_t source_tile,
   uint32_t destination_tile, unsigned source, unsigned
   destination, uint64_t time, std::string address)
{
          assert(isEntryPresent(address));
          // Get the setNumber and tag
          uint32_t set_number = getSetNumber(extractIndex(
             address));
          uint32_t tag = extractTag(address);

          // First message for the L2
          uint32_t valid_entry =  LUT[set_number].Entries[
             tag]->PredictionInfo[ValidBit];
          if ((valid_entry == 0) && (destination >=
             NumTiles) && (destination < 2*NumTiles)) { //
             For directory
                    LUT[set_number].Entries[tag]->
                       PredictionInfo[SharerBit + source_tile
                       ]= SB_normalSharer;
          LUT[set_number].Entries[tag]->PredictionInfo[
             SharerBit + destination_tile] =
             SB_normalSharer;
                    LUT[set_number].Entries[tag]->
                       PredictionInfo[ValidBit] = 1;
                    LUT[set_number].Entries[tag]->
                       PredictionInfo[DirectoryBit] =
                       destination_tile;
          } else if ((valid_entry == 0) && !(destination >=
              2*NumTiles) && !(destination < 3*NumTiles)) {
                    if (VerbosityPrintErrors){
                            printf("Messages coming by not
                               destined for the directory but
                                the entry has not been
                               initialized yet!");
                    }
          } else {
                            // Check this is not overwriting
                               the evictor counter + never
                               write the directory/L2 away as
                                a sharer
                    if ((source < NumTiles) && (LUT[
                       set_number].Entries[tag]->
                       PredictionInfo[SharerBit + source_tile
                       ] != SB_currentEvictor) && (LUT[
                       set_number].Entries[tag]->
                       PredictionInfo[SharerBit + source_tile
                       ] != SB_receivedInv)

                                                                                      &&
                        LUT[set_number].Entries[tag]->
                               PredictionInfo[SharerBit +
                               source_tile] = SB_normalSharer
                               ;
                    }

                    if ((destination < NumTiles) && (LUT[
                       set_number].Entries[tag]->
                       PredictionInfo[SharerBit +
                       destination_tile] != SB_currentEvictor
                       ) && (LUT[set_number].Entries[tag]->
                       PredictionInfo[SharerBit +
                       destination_tile] != SB_receivedInv)

                                                        && (
                       LUT[set_number].Entries[tag]->
                       PredictionInfo[SharerBit +
                       destination_tile] != SB_upgrader)) {
                            LUT[set_number].Entries[tag]->
                               PredictionInfo[SharerBit +
                               destination_tile] =
                               SB_normalSharer;
                    }

                    LUT[set_number].Entries[tag]->
                       PredictionInfo[ValidBit] = 1;
            }

            LUT[set_number].Entries[tag]->PredictionInfo[
               ValidBit] = 1;
            LUT[set_number].Entries[tag]->PredictionInfo[
               TagBit] = tag;

            if (Verbosity) {
                    printf("REC: Recording sharers %u and %u
                       at time %zu/%zu \n", source_tile,
                       destination_tile, time, LUT[set_number
                       ].Entries[tag]->PredictionInfo[
                       TimeStampBit]);
            }
    }


     // Function to change the sequence type
    uint32_t
```

```
GeneralPredictor::changeSequenceType(std::string address,
    std::string message_type, std::string request_type,
   unsigned source_node)
{
        assert(isEntryPresent(address));

        // Get number of the set this address maps on
        uint32_t set_number = getSetNumber(extractIndex(
           address));
        uint32_t tag = extractTag(address);

        // Change the sequence type
        uint32_t valid_entry = LUT[set_number].Entries[
           tag]->PredictionInfo[ValidBit];
        uint32_t sequence_bit = LUT[set_number].Entries[
           tag]->PredictionInfo[SequenceBit];
        unsigned num_sharers = getNumSharers(address,
           set_number, tag);

        if (valid_entry == 1) {
                // ------------- STATE 0 -------------
                if (sequence_bit == State_noRecording) {
                   // Nothing recorded yet -- change to
                   only present at directory
                        if (( message_type == "L1_REQ_C"
                           ) && ((request_type == "LD")
                           || (request_type == "ST") || (
                           request_type == "IFETCH"))) {
                           // LD 1
                                LUT[set_number].Entries[
                                   tag]->PredictionInfo[
                                   SequenceBit] =
                                   State_requestedByLLC;
                                LUT[set_number].Entries[
                                   tag]->PredictionInfo[
                                   RequestTypeBit] =
                                   convertAccessType(
                                   request_type);
                        } else if (request_type == "EVICT
                           ") {
                                LUT[set_number].Entries[
                                   tag]->PredictionInfo[
                                   SequenceBit] =
                                   State_requestedByLLC;
                                   // Eviction
                                LUT[set_number].Entries[
                                   tag]->PredictionInfo[
                                   RequestTypeBit] =
                                   convertAccessType(
                                   request_type);
                          } else {
                                // This should not be
                                   happening
                                if (VerbosityPrintErrors)
                                    {
                                        printf("
                                           Unexpected
                                           message type
                                           in sequence
                                           type state 0:
                                           message type %
                                           s, request
                                           type %s from %
                                           u for address
                                           %s \n",
                                           message_type.
                                           c_str(),
                                           request_type.
                                           c_str(),
                                           source_node,
                                           address.c_str
                                           ());
                                }
                          }
//  ------------- STATE 1 -------------
} else if (sequence_bit ==
   State_requestedByLLC) { // Change from
    not present in DIR to present in DIR
        if (message_type == "L2_REQ_C") {
            // LD1/ST1
                LUT[set_number].Entries[
                   tag]->PredictionInfo[
                   SequenceBit] =
                   State_requestedByLLC;
        } else if (message_type == "
            MEM_RES_D") { // LD1/ST1
                LUT[set_number].Entries[
                   tag]->PredictionInfo[
                   SequenceBit] =
                   State_presentInLLC;
        } else {
                // This should not be
                   happening
                if (VerbosityPrintErrors)
                    {
                        printf("
                                                    Unexpected
                                                    message type
                                                    in sequence
                                                    type state 1:
                                                    message type %
                                                    s, request
                                                    type %s from %
                                                    u for address
                                                    %s \n",
                                                    message_type.
                                                    c_str(),
                                                    request_type.
                                                    c_str(),
                                                    source_node,
                                                    address.c_str
                                                    ());
                                 }
                         }
                // ------------- STATE 2 -------------
                } else if (sequence_bit ==
                   State_presentInLLC) { // Change from
                   only present in DIR to only present in
                    1 L1
                        if (request_type == "L2_EVICT") {
                                LUT[set_number].Entries[
                                   tag]->PredictionInfo[
                                   SequenceBit] =
                                   State_eviction;
                                LUT[set_number].Entries[
                                   tag]->PredictionInfo[
                                   RequestTypeBit] =
                                   convertAccessType(
                                   request_type);
                        } else if (message_type == "
                           L2_RES_D") {     // LD1/ST1/LD4
                           /ST2
                                LUT[set_number].Entries[
                                   tag]->PredictionInfo[
                                   SequenceBit] =
                                   State_presentInOneL1;
                        } else if (message_type == "
                           L1_REQ_C") { // LD4/ST2
                                LUT[set_number].Entries[
                                   tag]->PredictionInfo[
                                   SequenceBit] =
                                   State_presentInLLC;
                                LUT[set_number].Entries[
                                   tag]->PredictionInfo[
                                   RequestTypeBit] =
                                   convertAccessType(
                                   request_type);
                        } else {
                                // This should not be
                                   happening
                                if (VerbosityPrintErrors)
                                    {
                                        printf("
                                           Unexpected
                                           message type
                                           in sequence
                                           type state 2:
                                           message type %
                                           s, request
                                           type %s from %
                                           u for address
                                           %s \n",
                                           message_type.
                                           c_str(),
                                           request_type.
                                           c_str(),
                                           source_node,
                                           address.c_str
                                           ());
                                }
                        }

                // ------------- STATE 3 -------------
                } else if (sequence_bit ==
                   State_presentInOneL1) { // Change from
                    only present in 1 L1 to present in
                   multiple L1s
                        if (request_type == "L2_EVICT") {
                                LUT[set_number].Entries[
                                   tag]->PredictionInfo[
                                   SequenceBit] =
                                   State_eviction;
                                LUT[set_number].Entries[
                                   tag]->PredictionInfo[
                                   RequestTypeBit] =
                                   convertAccessType(
                                   request_type);
                        } else if ( (message_type == "
                           L1_REQ_C") && (request_type ==
                            "LD") ) { // LD2
                                LUT[set_number].Entries[
                                   tag]->PredictionInfo[
                                   SequenceBit] =
                                   State_presentInOneL1;
                                LUT[set_number].Entries[
                                   tag]->PredictionInfo[
                                   RequestTypeBit] =
                                   convertAccessType(
                                   request_type);
                        } else if ( (message_type == "
                           L1_REQ_C") && (request_type ==
                           "IFETCH") ) { // LD2 but for
                           blocks which can never be
                           owned exclusively
                                LUT[set_number].Entries[
                                   tag]->PredictionInfo[
                                   SequenceBit] =
                                   State_presentInMultL1;
                                LUT[set_number].Entries[
                                   tag]->PredictionInfo[
                                   RequestTypeBit] =
                                   convertAccessType(
                                   request_type);
                        } else if ( (message_type == "
                           L1_REQ_C") && (request_type ==
                            "ST") ) {  // ST3
                                LUT[set_number].Entries[
                                   tag]->PredictionInfo[
                                   SequenceBit] =
                                   State_presentInOneL1;
                                LUT[set_number].Entries[
                                   tag]->PredictionInfo[
                                   RequestTypeBit] =
                                   convertAccessType(
                                   request_type);
                        } else if (message_type == "
                           L2_REQ_C") { //ST3 or LD2
                                if (LUT[set_number].
                                   Entries[tag]->
                                   PredictionInfo[
                                   RequestTypeBit] ==
                                   RT_ST) {
                                       LUT[set_number].
                                           Entries[tag]->
                                           PredictionInfo
                                           [SequenceBit]
                                           =
                                           State_presentInOneL1
                                           ;
                                } else if (LUT[set_number
                                   ].Entries[tag]->
                                   PredictionInfo[
                                   RequestTypeBit] ==
                                   RT_LD || LUT[
                                   set_number].Entries[
                                   tag]->PredictionInfo[
                                   RequestTypeBit] ==
                                   RT_IFETCH) { // LD2
                                        LUT[set_number].
                                           Entries[tag]->
                                           PredictionInfo
                                           [SequenceBit]
                                           =
                                           State_presentInMultL1
                                           ;
                                }
                        } else if (message_type == "
                           L1_RES_D") { // ST3
                                LUT[set_number].Entries[
                                   tag]->PredictionInfo[
                                   SequenceBit] =
                                   State_presentInOneL1;
                                removeSharer(address,
                                   set_number, tag,
                                   source_node);
                        } else if (request_type == "EVICT
                           ") {
                                LUT[set_number].Entries[
                                   tag]->PredictionInfo[
                                   SequenceBit] =
                                   State_eviction;
                                setAsEvictor(address,
                                   set_number, tag,
                                   source_node);
                                LUT[set_number].Entries[
                                   tag]->PredictionInfo[
                                   RequestTypeBit] =
                                   convertAccessType(
                                   request_type);
                        } else if (message_type == "
                           L2_RES_D") { // LD2
                                if (num_sharers == 1) {
                                        LUT[set_number].
                                           Entries[tag]->
                                           PredictionInfo
                                           [SequenceBit]
                                           =
                                           State_presentInOneL1
```

```
                                          ;
                              } else {
                                      LUT[set_number].
                                         Entries[tag]->
                                         PredictionInfo
                                         [SequenceBit]
                                         =
                                         State_presentInMultL1
                                         ;
                              }
                      } else {
                              // This should not be
                                 happening
                              if (VerbosityPrintErrors)
                                  {
                                      printf("
                                         Unexpected
                                         message type
                                         in sequence
                                         type state 3:
                                         message type %
                                         s, request
                                         type %s from %
                                         u for address
                                         %s \n",
                                         message_type.
                                         c_str(),
                                         request_type.
                                         c_str(),
                                         source_node,
                                         address.c_str
                                         ());
                              }
                      }
              // ------------- STATE 4 -------------
              } else if (sequence_bit ==
                 State_presentInMultL1) {
                      if (request_type == "L2_EVICT") {
                              LUT[set_number].Entries[
                                 tag]->PredictionInfo[
                                 SequenceBit] =
                                 State_eviction;
                              LUT[set_number].Entries[
                                 tag]->PredictionInfo[
                                 RequestTypeBit] =
                                 convertAccessType(
                                 request_type);
                      } else if ( (message_type == "
                         L1_REQ_C") && ( (request_type
                         =="LD") ||(request_type == "
                         IFETCH"))) { // LD3
                              LUT[set_number].Entries[
                                 tag]->PredictionInfo[
                                 SequenceBit] =
                                 State_presentInMultL1;
                              LUT[set_number].Entries[
                                 tag]->PredictionInfo[
                                 RequestTypeBit] =
                                 convertAccessType(
                                 request_type);
                      } else if ( message_type == "
                         L2_RES_D") { // LD3/ST4
                              LUT[set_number].Entries[
                                 tag]->PredictionInfo[
                                 SequenceBit] =
                                 State_presentInMultL1;
                      } else if ( (message_type =="
                         L1_REQ_C") && (request_type ==
                          "ST")) { // ST4
                              LUT[set_number].Entries[
                                 tag]->PredictionInfo[
                                 SequenceBit] =
                                 State_presentInMultL1;
                              LUT[set_number].Entries[
                                 tag]->PredictionInfo[
                                 RequestTypeBit] =
                                 convertAccessType(
                                 request_type);
                              setUpgrader(address,
                                 set_number, tag,
                                 source_node);
                      } else if ( (message_type =="
                         L1_REQ_C") && ( request_type
                         == "UPGRADE") ) { // ST5
                              LUT[set_number].Entries[
                                 tag]->PredictionInfo[
                                 SequenceBit] =
                                 State_presentInMultL1;
                              LUT[set_number].Entries[
                                 tag]->PredictionInfo[
                                 RequestTypeBit] =
                                 convertAccessType(
                                 request_type);
                              setUpgrader(address,
                                 set_number, tag,
                                 source_node);
```

```
} else if (message_type =="
   L2_RES_C") { // ST5
        LUT[set_number].Entries[
           tag]->PredictionInfo[
           SequenceBit] =
           State_presentInMultL1;
} else if (message_type == "
   L2_REQ_C") { // ST4/LD2/ST5
        LUT[set_number].Entries[
           tag]->PredictionInfo[
           SequenceBit] =
           State_presentInMultL1;
} else if (message_type == "
   L1_RES_D") { // LD2
        LUT[set_number].Entries[
           tag]->PredictionInfo[
           SequenceBit] =
           State_presentInMultL1;
} else if (message_type == "
   L1_RES_C") { //  ST4/ST5 -
   invalidation ack
        removeSharer(address,
           set_number, tag,
           source_node);
        int num_sharers_currently
            = getNumSharers(
           address, set_number,
           tag);
        if (num_sharers_currently
            > 1) {
                LUT[set_number].
                   Entries[tag]->
                   PredictionInfo
                   [SequenceBit]
                   =
                   State_presentInMultL1
                   ;
                if (Verbosity) {
                        printf("
                           PRED:
                           Transition
                            from
                           state
                           4 to 4
                            after
                            INV
                           ACK (%
                           i num
```

```
                                                    sharers

                                                    currently
                                                    ) \n",
                                                    num_sharers_curre
                                                    );
                                            }
                                    } else {
                                            LUT[set_number].
                                               Entries[tag]->
                                               PredictionInfo
                                               [SequenceBit]
                                               =
                                               State_presentInOneL1
                                               ;
                                            if (Verbosity) {
                                                    printf("
                                                    PRED:
                                                    Transition
                                                     from
                                                    state
                                                    4 to 3
                                                     after
                                                     INV
                                                    ACK (%
                                                    i num
                                                    sharers

                                                    currently
                                                    ) \n",
                                                    num_sharers_curre
                                                    );
                                            }
                                    }
                            } else if (request_type == "EVICT
                               ") {
                                    LUT[set_number].Entries[
                                       tag]->PredictionInfo[
                                       SequenceBit] =
                                       State_eviction;
                                    setAsEvictor(address,
                                       set_number, tag,
                                       source_node);
                                    LUT[set_number].Entries[
                                       tag]->PredictionInfo[
                                       RequestTypeBit] =
                                       convertAccessType(
                                       request_type);
```

```
                    } else {
                            // This should not be
                               happening
                            if (VerbosityPrintErrors)
                                {
                                    printf("
                                       Unexpected
                                       message type
                                       in sequence
                                       type state 4:
                                       message type %
                                       s, request
                                       type %s from %
                                       u for address
                                       %s \n",
                                       message_type.
                                       c_str(),
                                       request_type.
                                       c_str(),
                                       source_node,
                                       address.c_str
                                       ());
                            }
                    }
            // ------------- STATE 5 -------------
            } else if (sequence_bit == State_eviction
               ) {
                    if (LUT[set_number].Entries[tag
                       ]->PredictionInfo[
                       RequestTypeBit] == RT_EVICT) {
                        // L1 evict
                            if (message_type == "
                               L2_RES_C") {
                                    int
                                       num_sharers_currently
                                        =
                                       getNumSharers(
                                       address,
                                       set_number,
                                       tag);
                                    if (
                                       num_sharers_currently
                                        <= 2) {
                                            LUT[
                                               set_number
                                               ].
                                               Entries
                                               [tag
```

```
         ]->
         PredictionInfo
         [
         SequenceBit
         ] =
         State_presentInLI
         ;
      if (
         Verbosity
         ) {
              printf
                 (
                 "
                 PRED
                 :

                 Transitio

                 from

                 state

                 5

                 to

                 2

                 after

                 evict

                 ACK

                 (%
                 i

                 num

                 sharers

                 currently
                 )

                 \
                 n
                 "
                 ,
                                                    num_share
                                                    )
                                                    ;

                                         }
                              } else if (
                                 num_sharers_currently
                                  == 3) {
                                      LUT[
                                         set_number
                                         ].
                                         Entries
                                         [tag
                                         ]->
                                         PredictionInfo
                                         [
                                         SequenceBit
                                         ] =
                                         State_presentInOr
                                         ;
                                      if (
                                         Verbosity
                                         ) {
                                              printf
                                                 (
                                                 "
                                                 PRED
                                                 :

                                                 Transitio

                                                 from

                                                 state

                                                 5

                                                 to

                                                 3

                                                 after

                                                 evict

                                                 ACK

                                                 (%
                                                 i

                                                 num

                                                 sharers

                                                 currently
                                                 )

                                                 \
                                                 n
                                                 "

                                                 ,
                                                 num_share
                                                 )
                                                 ;

                                         }
                              } else {
                                         LUT[
                                         set_number
                                         ].
                                         Entries
                                         [tag
                                         ]->
                                         PredictionInfo
                                         [
                                         SequenceBit
                                         ] =
                                         State_presentInMu
                                         ;
                                         if (
                                         Verbosity
                                         ) {
                                              printf
                                                 (
                                                 "
                                                 PRED
                                                 :

                                                 Transitio

                                                 from

                                                 state

                                                 5
```

```
                                                      to

                                                      4

                                                      after

                                                      evict

                                                      ACK

                                                      (%
                                                      i

                                                      num

                                                      sharers

                                                      currently
                                                      )

                                                      \
                                                      n
                                                      "
                                                      ,
                                                      num_share
                                                      )
                                                      ;

                                              }
                                      }
                                      // Remove evictor
                                         from
                                         sharersList
                                      int evictor =
                                         returnEvictor(
                                         address,
                                         set_number,
                                         tag);
                                      if (evictor !=
                                         -1) { // Added
                                         to C++
                                              removeSharer
                                                 (
                                                 address
                                                 ,
                                                 set_number
                                                 , tag,
                                                  (
                                                 unsigned
                                                 )
                                                 evictor
                                                 );
                                      }
                              } else {
                 // This should not be happening
                                      if (
                                         VerbosityPrintErrors
                                         ) {
                                              printf("
                                                 Unexpected

                                                 message
                                                  type
                                                 in
                                                 sequence
                                                  type
                                                 state
                                                 5 (L1
                                                 evict)
                                                 :
                                                 message
                                                  type
                                                 %s,
                                                 request
                                                  type
                                                 %s
                                                 from %
                                                 i for
                                                 address
                                                  %s \n
                                                 ",
                                                 message_type
                                                 .c_str
                                                 (),
                                                 request_type
                                                 .c_str
                                                 (),
                                                 source_node
                                                 ,
                                                 address
                                                 .c_str
                                                 ());
                                      }
                              }
                      } else if (LUT[set_number].
                         Entries[tag]->PredictionInfo[
                         RequestTypeBit] == RT_L2_EVICT
                         ) { //L2 evict
                              if ( message_type == "
                                 MEM_RES_C") {
                                      LUT[set_number].
                                         Entries[tag]->
                                         PredictionInfo
                                         [SequenceBit]
                                         =
                                         State_noRecording
                                         ;
                              } else if ( (message_type
                                  == "L1_RES_C") || (
                                 message_type == "
                                 L1_RES_D") ) {
                                      removeSharer(
                                         address,
                                         set_number,
                                         tag,
                                         source_node);
                                      LUT[set_number].
                                         Entries[tag]->
                                         PredictionInfo
                                         [SequenceBit]
                                         =
                                         State_eviction
                                         ;
                              } else if ( (message_type
                                  == "L2_RES_C") || (
                                 message_type == "
                                 L2_RES_D") ) {
                                      LUT[set_number].
                                         Entries[tag]->
                                         PredictionInfo
                                         [SequenceBit]
                                         =
                                         State_eviction
                                         ;
                              } else if (message_type
                                 =="L2_REQ_C") {
                                      LUT[set_number].
                                         Entries[tag]->
                                         PredictionInfo
                                         [SequenceBit]
                                         =
                                         State_eviction
                                         ; // Inv
                                         messages
```

```
                                } else {
                                        if (
                                           VerbosityPrintErrors
                                           ) {
                                                printf("
                                                   Unexpected

                                                   message
                                                    type
                                                   in
                                                   sequence
                                                    type
                                                   state
                                                   5 (L2
                                                   evict)
                                                   :
                                                   message
                                                    type
                                                   %s,
                                                   request
                                                    type
                                                   %s
                                                   from %
                                                   i for
                                                   address
                                                    %s \n
                                                   ",
                                                   message_type
                                                   .c_str
                                                   (),
                                                   request_type
                                                   .c_str
                                                   (),
                                                   source_node
                                                   ,
                                                   address
                                                   .c_str
                                                   ());
                                        }
                                }
                        }
                }
        } else {
                // This should not be happening
                if (VerbosityPrintErrors) {
                        printf("Message looking for
                           connection for non-initialized
                            address!");
                }
        }
        uint32_t new_sequence_bit = LUT[set_number].
           Entries[tag]->PredictionInfo[SequenceBit];
        if (Verbosity){
                printf("PRED: Changing the sequence type
                   from %u to %u for message_type %s and
                   request_type %s  \n",sequence_bit,
                   new_sequence_bit,message_type.c_str(),
                   request_type.c_str());
        }
        return new_sequence_bit;
}

// Function to predict the next optical connection
// Returns a standard pair containing the predicted
   source and destination
std::pair<int, int>
GeneralPredictor::predictOpticalConnection(std::string
   address,unsigned source,unsigned destination,uint64_t
   time,std::string previous_message)
{
        // Find key_dec for this address
        assert(isEntryPresent(address));

        // Get number of the set this address maps on
        uint32_t set_number = getSetNumber(extractIndex(
           address));
        uint32_t tag = extractTag(address);

        // Setup connection
        uint32_t valid_entry = LUT[set_number].Entries[
           tag]->PredictionInfo[ValidBit];
        uint32_t sequence_bit = LUT[set_number].Entries[
           tag]->PredictionInfo[SequenceBit];
        // Predict from and to which tile the next
           message will come and go
        uint32_t source_tile = convertNodeID(source);
        uint32_t destination_tile = convertNodeID(
           destination);

        // These variables are integers now, during the
           actual prediction they can be set to -1,
           indicating there is no valid prediction to be
           made
        // Predicted source and destination will not be
           stored like this, but be used to set the
           PredictionValid field
```

```
int predicted_source = (int) destination_tile; //
     The next message will leave from where the
   current message is going to
int predicted_destination = -1; // Starts as -1,
   will be converted later on

// For statistics only -- to remember type of
   predictions made when imperfect predictions
   are made eg partial tag
std::string pred_type = "blob";

if (Verbosity) {
        printf("PRED: Predicting an optical
           connection -- current message [%u -> %
           u], sequence bit %u, currently %u
           sharers \n",source,destination,
           sequence_bit,getNumSharers(address,
           set_number, tag));
}
if ( (getNumSharers(address, set_number, tag) <
   1) && (valid_entry == 1) && (sequence_bit !=
   State_eviction)) {
        predicted_source = -1;
        pred_type = "single_sharer";
        if (Verbosity) {
                printf("PRED: Only 1 registered
                   sharer so no point in setting
                   up a connection (set_number %u
                   , state %i) \n",set_number,
                   sequence_bit);
        }
} else if (valid_entry == 1) {
        }
        // -------- STATE 1 --------------------
        if (sequence_bit == State_requestedByLLC)
            { // Requested by directory
                predicted_destination =
                   returnDirectory(address,
                   set_number, tag); // Memory
                   connection
                pred_type = "state1";
                if (Verbosity) {
                        printf("PRED: Requested
                           by directory so
                           destination will be
                           memory controller/
                           directory -- %i \n",
                           predicted_destination)
```

```
                    ;
          }
//-------- STATE 2 ----------------------
} else if( sequence_bit ==
   State_presentInLLC) { // Requested by
   L1 but only present in directory at
   the moment
          predicted_destination =
             returnUniqueSharer(address,
             set_number, tag);
          pred_type = "state2";
          if (Verbosity) {
                  printf("PRED: Requested
                     by L1 so destination
                     will be only other
                     registered sharer --
                     %i \n",
                     predicted_destination)
                     ;
          }
//-------- STATE 3 ----------------------
} else if (sequence_bit ==
   State_presentInOneL1) { // Only
   present in 1 L1
          if ( (previous_message == "
             L1_REQ_C") && (LUT[set_number
             ].Entries[tag]->PredictionInfo
             [RequestTypeBit] == RT_ST) ) {
              // ST3
                  // Next message will be
                     an invalidation from
                     directory L2
                  // to L1" -- choose the
                     other sharer not this
                     L1
                  std::vector<unsigned>
                     all_sharers =
                     returnSharers(address,
                      set_number, tag);
                  for (unsigned i = 0; i <
                     all_sharers.size(); i
                     ++){
                          if ( all_sharers[
                             i] !=
                             source_tile) {
                                  predicted_destinati
                                     = (
                                     int)
```

```
                                                    all_sharers
                                                    [i];
                                                 break;
                                         }
                                 }
                                 pred_type = "
                                    state3_ST3_inv";
                                 if (Verbosity) {
                                         printf("PRED:
                                            Next message
                                            will be the
                                            INV to the
                                            other sharer
                                            --  %i \n",
                                            predicted_destination
                                            );
                                 }
                         } else if ( (previous_message ==
                            "L1_REQ_C") && ( (LUT[
                            set_number].Entries[tag]->
                            PredictionInfo[RequestTypeBit]
                             == RT_LD) || (LUT[set_number
                            ].Entries[tag]->PredictionInfo
                            [RequestTypeBit] == RT_IFETCH)
                             ) ) { // LD2
                                 // Next message will be
                                    an invalidation from
                                    directory L2
                                 // to L1" -- choose the
                                    other sharer not this
                                    L1
                                 std::vector<unsigned>
                                    all_sharers =
                                    returnSharers(address,
                                     set_number, tag);
                                 for (unsigned i = 0; i <
                                    all_sharers.size(); i
                                    ++){
                                         if ( all_sharers[
                                            i] !=
                                            source_tile) {
                                                 predicted_destinati
                                                    = (
                                                    int)
                                                    all_sharers
                                                    [i];
                                                 break;
                                         }
```

```
        }
        pred_type = "
           state3_LD2_inv";
        if (Verbosity) {
                printf("PRED:
                   Next message
                   will be the
                   INV to the
                   other sharer
                   --  %i \n",
                   predicted_destination
                   );
        }
} else if ( previous_message == "
   L2_REQ_C") { // ST3
        // Next message will be
           the invalidation ACK
        std::vector<unsigned>
           all_sharers =
           returnSharers(address,
            set_number, tag);
        for (unsigned i = 0; i <
           all_sharers.size(); i
           ++){
                if ( all_sharers[
                   i] !=
                   destination_tile
                   ) {
                        predicted_destinati
                           = (
                           int)
                           all_sharers
                           [i];
                        break;
                }
        }
        pred_type = "
           state3_ST3_ack";
        if (Verbosity){
                printf("PRED:
                   Next message
                   will be the
                   INV ACK to the
                    original
                   requestor -- %
                   i \n",
                   predicted_destination
                   );
        }
} else if (previous_message == "
   L1_RES_D" ) { // ST3
        // Next message will be
           the UNB
        predicted_source =
           destination_tile;
        predicted_destination = (
           int) returnDirectory(
           address, set_number,
           tag);
        pred_type = "
           state3_ST3_unb";
} else if(previous_message == "
   L2_RES_D" ) { // LD1/3/4/ST2
        // Next message will be
           the UNB
        predicted_source =
           returnUniqueSharer(
           address, set_number,
           tag);
        predicted_destination = (
           int) returnDirectory(
           address, set_number,
           tag);
        pred_type = "
           state3_ST2_LD1_3_4_unb
           ";
} else if (previous_message == "
   L1_RES_C") { // ST4
        // Next message will be
           the UNB - set upgrader
            as
        // normal sharer now
        setUpgraderAsNormalSharer
           (address, set_number,
           tag,destination_tile);
        predicted_source =
           returnUniqueSharer(
           address, set_number,
           tag);
        predicted_destination = (
           int) returnDirectory(
           address, set_number,
           tag);
        pred_type = "
           state3_ST4_unb";
} else {
```

```
predicted_source = -1;
predicted_destination =
   -1; // JUST TEMPORARY
   HACK
pred_type = "
   state3_unkown";
if (VerbosityPrintErrors)
    {
        printf("PRED: do
           not know which
            prediction to
            make from
           state 3 (
           previous
           message %s
           from %i to %i
           for address %s
            - last
           transaction
           type %zu) \n",
                        previous_me
                           .
                           c_str
                           ()
                           ,
                           source
                           ,
                           destinati
                           ,
                           address
                           .
                           c_str
                           ()
                           ,
                           LUT
                           [
                           set_numbe
                           ].
                           Entries
                           [
                           tag
                           ]->
                           Predictio
                           [
                           RequestTy
                           ])
                           ;
                        }
                }

            //--------------- STATE 4
               --------------------------------
            } else if (sequence_bit ==
               State_presentInMultL1) { // Requested
               by a L1 but present in other L1(s)
                    if (previous_message == "L2_REQ_C
                       " ) { // LD2 or ST5/ST4
                            if ( (LUT[set_number].
                               Entries[tag]->
                               PredictionInfo[
                               RequestTypeBit] ==
                               RT_LD) || (LUT[
                               set_number].Entries[
                               tag]->PredictionInfo[
                               RequestTypeBit] ==
                               RT_IFETCH) ) {// LD or
                                IFETCH
                                    // Next message
                                       will be the
                                       transfer of
                                       the
                                    // exclusive L1"
                                       to the
                                       requestor L1
                                    // Next message
                                       will be the
                                       invalidation
                                       ACK
                                    std::vector<
                                       uint32_t>
                                       all_sharers =
                                       returnSharers(
                                       address,
                                       set_number,
                                       tag);
                                    for (unsigned i =
                                        0; i <
                                       all_sharers.
                                       size(); i++) {
                                            if (
                                               all_sharers
                                               [i] !=

                                               destination_tile
                                               ) {
                                                     predicted_d

                                                       =

                                                       (
                                                       int
                                                       )

                                                       all_share
                                                       [
                                                       i
                                                       ];

                                                     break
                                                       ;

                                            }
                                    }
                                    pred_type = "
                                       state4_LD2_data
                                       ";
                                    if (Verbosity) {
                                            printf("
                                               PRED:
                                               Next
                                               message
                                                will
                                               be the
                                                data
                                               transfer
                                                to
                                               the
                                               original

                                               requestor
                                                -- %
                                               i \n",
                                               predicted_destina
                                               );
                                    }
                            } else if ( (LUT[
                               set_number].Entries[
                               tag]->PredictionInfo[
                               RequestTypeBit] ==
                               RT_UPGRADE) || (LUT[
                               set_number].Entries[
                               tag]->PredictionInfo[
                               RequestTypeBit] ==
                               RT_ST)) { //ST5/ST4
                                    hasReceivedInv(
                                       address,
                                       set_number,
                                       tag,
                                       destination_tile
                                       ); // Mark as
                                       having
                                       received an
                                       INV
                                    if (
                                       getNumSharersToInv
                                       (address,
                                       set_number,
                                       tag) > 1) { //
                                        There are
                                       more sharers
                                       to invalidate
                                       ( > 1 because
                                       the upgrader
                                       does not have
                                       to be
                                       invalidated)
                                            predicted_destinati
                                                = (
                                               int)
                                               getNextSharerToIn
                                               (
                                               address
                                               ,
                                               set_number
                                               , tag)
                                               ;
                                            predicted_source
                                                = (
                                               int)
                                               returnDirectory
                                               (
                                               address
                                               ,
                                               set_number
                                               , tag)
                                               ;
                                            pred_type
                                                = "
                                               state4_ST4_5_inv
                                               ";
                                            if (
                                                       Verbosity
                                                       ) {
                                                            printf
                                                               (
                                                               "
                                                               PRED
                                                               :

                                                               Last

                                                               message

                                                               was

                                                               INV

                                                               but

                                                               more

                                                               INV

                                                               to

                                                               follow

                                                               as

                                                               there

                                                               are

                                                               %
                                                               u

                                                               sharers
                                                               ,

                                                               next

                                                               INV

                                                               to

                                                               %
                                                               i

                                                               (
                                                        ST4
                                                        /5)

                                                        \
                                                        n
                                                        "

                                                        ,
```

```
                  }
         } else { // No
            more sharers
            to invalidate
            so next up
            will be the
            invalidation
            count to the
            requestor
                  predicted_destinati
                     = (
                    int)
                    getUpgrader
                    (
                    address
                    ,
                    set_number
                    , tag)
                    ;
                  predicted_source
                     = (
                    int)
                    returnDirectory
                    (
                    address
                    ,
                    set_number
                    , tag)
                    ;
                 pred_type
                     = "
                    state4_ST4_5_ack
                    ";
                 if (
                    Verbosity
                    ) {
                        printf
                            (
                            "
                            PRED
                            :

                            All

                            sharers

                            have

                            been

                            INV
                            ,

                            next

                            will

                            be

                            the

                            INV

                            ack

                            from

                            %
                            i

                            (
                            ST4
                            /5)
```

```
                                                                      \
                                                                      n
                                                                      "
                                                                      ,
                                                                      predicte
                                                                      )
                                                                      ;
                                                            }
                                                    }
                                            }
                                    } else if ( previous_message == "
                                       L1_RES_D") { // LD2
                                            // If the previous
                                               message was the WB,
                                               next will be
                                            // UNB
                                            if (destination_tile ==
                                               returnDirectory(
                                               address, set_number,
                                               tag)) {
                                                    predicted_destination
                                                        = (int)
                                                       returnDirectory
                                                       (address,
                                                       set_number,
                                                       tag);
                                                    pred_type = "
                                                       state4_LD2_unb
                                                       ";
                                            } else { // Next message
                                               will be the WB to the
                                               L2 (predict the second
                                                L1_RES_D)
                                                    predicted_destination
                                                        = (int)
                                                       returnDirectory
                                                       (address,
                                                       set_number,
                                                       tag);
                                                    predicted_source
                                                       = source_tile;
                                                    pred_type = "
                                                       state4_LD2_writeback
                                                       ";
                                                    if (Verbosity) {
                                                            printf("
                           PRED:
                           Next
                           message
                            will
                           be the

                            writeback
                            from
                           not-
                           requestor
                            L1 (%
                           i) to
                           the L2
                            -- %i
                            \n",
                           predicted_source
                           ,
                           predicted_destina
                           );
                } 
        }
} else if (previous_message == "
   L1_REQ_C") { // LD3/ST4/ST5
        if (LUT[set_number].
           Entries[tag]->
           PredictionInfo[
           RequestTypeBit] ==
           RT_ST) { // ST -- ST4
                // Next message
                   will be the
                   first
                   invalidation
                predicted_destination
                    = (int)
                   getNextSharerToInv
                   (address,
                   set_number,
                   tag);
                predicted_source
                   = (int)
                   returnDirectory
                   (address,
                   set_number,
                   tag);
                pred_type = "
                   state4_ST4_inv
                   ";
                if (Verbosity) {
```

```
                                        printf("
                                           PRED:
                                           Next
                                           message
                                            will
                                           be the
                                            first

                                           invalidation
                                            (ST4)
                                            \n");
                              }
                    } else if (LUT[set_number
                       ].Entries[tag]->
                       PredictionInfo[
                       RequestTypeBit] ==
                       RT_UPGRADE) { //
                       UPGRADE -- ST5
                              // Next message
                                 will be the
                                 first
                                 invalidation
                              predicted_destination
                                  = (int)
                                 getNextSharerToInv
                                 (address,
                                 set_number,
                                 tag);
                              predicted_source
                                 = (int)
                                 returnDirectory
                                 (address,
                                 set_number,
                                 tag);
                              pred_type = "
                                 state4_ST5_inv
                                 ";
                              if (Verbosity) {
                                        printf("
                                           PRED:
                                           Next
                                           message
                                            will
                                           be the
                                            first

                                           invalidation
                                            (ST5)
```

```
                                                \n");
                                        }
                                } else if ( (
                                   previous_message == "
                                   L1_REQ_C") && ( (LUT[
                                   set_number].Entries[
                                   tag]->PredictionInfo[
                                   RequestTypeBit] ==
                                   RT_LD) || (LUT[
                                   set_number].Entries[
                                   tag]->PredictionInfo[
                                   RequestTypeBit] ==
                                   RT_IFETCH) ) ) { //
                                   LD3
                                        predicted_destination
                                           = source_tile
                                          ;
                                        pred_type = "
                                           state4_LD3_data
                                           ";
                                        if (Verbosity) {
                                               printf("
                                                  PRED:
                                                  Next
                                                  message
                                                   will
                                                  be the
                                                   data
                                                  transfer
                                                   from
                                                  the L2
                                                   to
                                                  the
                                                  original

                                                  requestor
                                                   (LD3)
                                                   \n");
                                        }
                                }
                        } else if (previous_message == "
                           L2_RES_D") { // ST4/LD3 -- we
                           need to distinguish between
                           them
                                if (LUT[set_number].
                                   Entries[tag]->
                                   PredictionInfo[
                                   RequestTypeBit] ==
                                   RT_ST) { // ST
                                        // Next message
                                           will be the
                                           invalidation
                                        //
                                           acknowledgments

                                        predicted_source
                                           = (int)
                                           getNextSharerToInvAck
                                           (address,
                                           set_number,
                                           tag);
                                        predicted_destination
                                           = (int)
                                           getUpgrader(
                                           address,
                                           set_number,
                                           tag);
                                        pred_type = "
                                           state4_ST4_ack
                                           ";
                                        if (Verbosity) {
                                               printf("
                                                  PRED:
                                                  Next
                                                  message
                                                   will
                                                  be the
                                                   first
                                                   INV
                                                  request
                                                   (ST4)
                                                   \n");
                                        }
                                } else if ( (LUT[
                                   set_number].Entries[
                                   tag]->PredictionInfo[
                                   RequestTypeBit] ==
                                   RT_LD) || (LUT[
                                   set_number].Entries[
                                   tag]->PredictionInfo[
                                   RequestTypeBit] ==
                                   RT_IFETCH) ) { // LD
                                   or IFETCH
                                               // This
                                                  was a
                                                  LD3 so
```

```
     there
     will
   be no
   next
//
   connection

predicted_source
     = -1;
pred_type
     = "
   state4_LD3
   ";
if (
   Verbosity
   ) {
        printf
           (
           "
           PRED
           :

           previous

           message

           L2_RES_D
           ,

           access

           type

           is

           LD
           /
           IFETCH

           so

           sequence

           type

           LD3

           \
                                                                                n
                                                                                "
                                                                                )
                                                                                ;
                                                                    }
                                                   }
                                          } else if (previous_message == "
                                             L1_RES_C") { // ST4/ST5
                                                   if (getNumSharers(address
                                                      , set_number, tag) ==
                                                      0) { // All sharers
                                                      have ack"ed their
                                                      invalidation so next
                                                      up will be the UNB
                                                           predicted_source
                                                              =
                                                              destination_tile
                                                              ;
                                                           predicted_destination
                                                               = (int)
                                                              returnDirectory
                                                              (address,
                                                              set_number,
                                                              tag);
                                                           pred_type = "
                                                              state4_ST4_5_unb
                                                              ";
                                                           if (Verbosity) {
                                                                   printf("
                                                                      PRED:
                                                                      All
                                                                      sharers
                                                                       have
                                                                      INV
                                                                      acked
                                                                      so
                                                                      next
                                                                      up is
                                                                      the
                                                                      UNB to
                                                                       %i \n
                                                                      ",
                                                                      predicted_destina
                                                                      );
                                                           }
                                                   } else {
                                                           predicted_source
```

```
                   = (int)
                   getNextSharerToInvAck
                   (address,
                   set_number,
                   tag);
                predicted_destination
                    = (int)
                   returnDirectory
                   (address,
                   set_number,
                   tag);
                pred_type = "
                   state4_ST4_5_ack
                   ";
                if (Verbosity) {
                        printf("
                           PRED:
                           Next
                           will
                           be the
                            INV
                           ack
                           from %
                           i \n",
                           predicted_destina
                           );
                }
        }
} else if (previous_message == "
   L2_RES_C") { // ST5
        // Next messages will be
           the inv acks
        predicted_source = (int)
           getNextSharerToInvAck(
           address, set_number,
           tag);
        predicted_destination = (
           int) getUpgrader(
           address, set_number,
           tag);
        if (predicted_destination
            == -1) { // Ordinary
           eviction by one of the
            sharers
                pred_type = "
                   state4_ST5_evict
                   ";
                if (Verbosity) {
```

```
                                                    printf("
                                                       PRED:
                                                       No
                                                       next
                                                       mesage
                                                       , one
                                                       of the

                                                       sharers

                                                       evicted
                                                        the
                                                       line \
                                                       n");
                                            }
                                    } else {
                                            pred_type = "
                                               state4_ST5_ack
                                               ";
                                            if (Verbosity) {
                                                    printf("
                                                       PRED:
                                                       Next
                                                       message
                                                        will
                                                       be the
                                                        first
                                                        INV
                                                       request
                                                        (ST5)
                                                        \n");
                                            }
                                    }
                            } else {
                                    predicted_source = -1;
                                    predicted_destination =
                                       -1; // JUST TEMPORARY
                                       HACK
                                    pred_type = "
                                       state4_unkown";
                                    if  (VerbosityPrintErrors
                                       ) {
                                            printf("PRED: do
                                               not know which
                                                prediction to
                                                make from
                                               state 4 (
                                               previous
```

```
                                                    message %s
                                                    from %i to %i)
                                                     \n",
                                                    previous_message
                                                    .c_str(),
                                                    source,
                                                    destination);
                                          }
                                  }

                        } else if (sequence_bit == State_eviction
                           ) { // In the midst of an eviction
                                  if (LUT[set_number].Entries[tag
                                     ]->PredictionInfo[
                                     RequestTypeBit] == RT_EVICT) {
                                      // L1 eviction
                                          predicted_destination = (
                                             int) returnEvictor(
                                             address, set_number,
                                             tag);
                                          pred_type = "eviction_L1"
                                             ;
                                  } else if (LUT[set_number].
                                     Entries[tag]->PredictionInfo[
                                     RequestTypeBit] == RT_L2_EVICT
                                     ) { // L2 eviction
                                          if (previous_message == "
                                             L2_REQ_C") {
                                                  hasReceivedInv(
                                                     address,
                                                     set_number,
                                                     tag,
                                                     destination_tile
                                                     ); // Mark as
                                                     having
                                                     received an
                                                     INV
                                          }

                                          if ( (previous_message ==
                                              "L2_RES_C") || (
                                             previous_message =="
                                             L2_RES_D") ){
                                                  // Next message
                                                     will be the WB

                                                     acknowledgment
                                                  // from memory
```

```
                    predicted_destination
                        = (int)
                       returnDirectory
                       (address,
                       set_number,
                       tag);
                    pred_type = "
                       eviction_L2_wback
                       ";
                    if (Verbosity) {
                            printf("
                               PRED:
                               Last
                               message
                                was
                               WB,
                               next
                               message
                                will
                               be the
                                WB
                               ack
                               from %
                               i \n",
                               predicted_destina
                               );
                    }
              } else if ((
                 previous_message == "
                 L2_REQ_C")) {
                    // Previous
                       message was an
                        invalidation
                       request
                    if (
                       getNumSharersToInv
                       (address,
                       set_number,
                       tag) > 0) { //
                        There are
                       more sharers
                       to invalidate
                            predicted_destinati
                                = (
                               int)
                               getNextSharerToIn
                               (
                               address
```

```
    ,
    set_number
    , tag)
    ;
predicted_source
     = (
    int)
    returnDirectory
    (
    address
    ,
    set_number
    , tag)
    ;
pred_type
     = "
    eviction_L2_inv
    ";
if (
    Verbosity
    ) {
           printf
               (
               "
               PRED
               :

               Last

               message

               was

               INV

               but

               more

               INV

               to

               follow

               as

               there
```

```
                                                                                     are

                                                                                     %
                                                                                     i

                                                                                     sharers
                                                                                     ,

                                                                                     next

                                                                                     INV

                                                                                     to

                                                                                     %
                                                                                     i

                                                                                     \
                                                                                     n
                                                                                     "
                                                                                     ,

                                                                         }
                                                                 } else { // No
                                                                    more sharers
                                                                    to invalidate
                                                                    so next up
                                                                    will be the
                                                                    invalidation
                                                                    acknowledgments

                                                                          predicted_source
```

```
       = (
      int)
      getNextSharerToI
      (
      address
      ,
      set_number
      , tag)
      ;
   predicted_destinati
       = (
      int)
      returnDirectory
      (
      address
      ,
      set_number
      , tag)
      ;
   pred_type
       = "
      eviction_L2_ack
      ";
   if (
      Verbosity
      ) {
           printf
              (
              "
              PRED
              :

              All

              sharers

              have

              been

              INV
              ,

              next

              will

              be
```

```
                                                            the

                                                            INV

                                                            ack

                                                            from

                                                            %
                                                            i

                                                            \
                                                            n
                                                            "
                                                            ,
                                                            predicte
                                                            )
                                                            ;

                                                    }
                                            }
                                    } else if ((
                                      previous_message == "
                                      L1_RES_C") || (
                                      previous_message == "
                                      L1_RES_D")) {
                                            // Previous
                                               message was an
                                                invalidation
                                               ack
                                            if (getNumSharers
                                               (address,
                                               set_number,
                                               tag) == 0) {
                                               // All sharers
                                                have ack"ed
                                               their
                                               invalidation
                                               so next up
                                               will be the
                                               writeback
                                                    predicted_destinati
                                                       = (
                                                      int)
                                                      returnDirectory
                                                      (
                                                      address
```

```
                                                                                   ,
                                                                                   set_number
                                                                                   , tag)
                                                                                   ;
                                                                               pred_type
                                                                                    = "
                                                                                   eviction_L2_wb
                                                                                   ";
                                                                               if (
                                                                                   Verbosity
                                                                                   ) {
                                                                                          printf
                                                                                              (
                                                                                              "
                                                                                              PRED
                                                                                              :

                                                                                              All

                                                                                              sharers

                                                                                              have

                                                                                              INV

                                                                                              acked

                                                                                              so

                                                                                              next

                                                                                              up

                                                                                              is

                                                                                              the

                                                                                              WB

                                                                                              to

                                                                                              %
                                                                                              i

                                                                                              \
                                                                                              n
                                                                                              "
                                                                                              ,
                                                                                            predicte
                                                                                              )
                                                                                              ;

                                                                               }
                                                                       } else {
                                                                               predicted_source
                                                                                    = (
                                                                                   int)
                                                                                   getNextSharerToI
                                                                                   (
                                                                                   address
                                                                                   ,
                                                                                   set_number
                                                                                   , tag)
                                                                                   ;
                                                                               predicted_destinati
                                                                                    = (
                                                                                   int)
                                                                                   returnDirectory
                                                                                   (
                                                                                   address
                                                                                   ,
                                                                                   set_number
                                                                                   , tag)
                                                                                   ;
                                                                               pred_type
                                                                                    = "
                                                                                   eviction_L2_ack
                                                                                   ";
                                                                               if (
                                                                                   Verbosity
                                                                                   ) {
                                                                                          printf
                                                                                              (
                                                                                              "
                                                                                              PRED
                                                                                              :

                                                                                              Next

                                                                                              will

                                                                                              be

                                                                                              the

                                                                                              INV
```

```
                                                                      ack

                                                                      from

                                                                      %
                                                                      i

                                                                      \
                                                                      n
                                                                      "
                                                                      ,
                                                                      predicte
                                                                      )
                                                                      ;

                                                            }
                                                    }
                                            }

                                    }

        } else {
                if (VerbosityPrintErrors) {
                        printf("Message trying to setup
                           connection for non-initialized
                            address! \n");
                }
                predicted_destination = -1;
                pred_type = "uninitialised_address";
        }

        if (Verbosity) {
                printf("PRED: Setting up a connection
                   from %i to %i \n",predicted_source,
                   predicted_destination);
        }
        //speculative_size = 72; // TBD: needs to be
           determined by the FSM
        //request = Request(predicted_source,
           predicted_destination,time,true,"SPECULATIVE",
           address,speculative_size);
        //int request = 0;
        // Check whether we can the results away
        if ( (predicted_source == -1) || (
           predicted_destination == -1) ) {
                LUT[set_number].Entries[tag]->
                   PredictionInfo[ValidPredictionBit] =
                   0;
                // Does not matter what was in the actual
                    prediction fields because we won't
                   look at them
                LUT[set_number].Entries[tag]->
                   PredictionInfo[PredictedSourceBit] =
                   0;
                LUT[set_number].Entries[tag]->
                   PredictionInfo[PredictedDestinationBit
                   ] = 0;
        } else {
                LUT[set_number].Entries[tag]->
                   PredictionInfo[ValidPredictionBit] =
                   1;
                LUT[set_number].Entries[tag]->
                   PredictionInfo[PredictedSourceBit] =
                   predicted_source;
                LUT[set_number].Entries[tag]->
                   PredictionInfo[PredictedDestinationBit
                   ] = predicted_destination;
        }

        // Write away the statistics
        if ( StatsPredictionType.count(pred_type) > 0) {
                StatsPredictionType[pred_type]++;
        } else {
                StatsPredictionType.insert(std::pair<std
                   ::string, uint32_t>     (pred_type, 1)
                   );
        }

        std::pair<int,int> connection_info;
        connection_info = std::make_pair(predicted_source
           , predicted_destination);

        return connection_info;
}

///////////////////////////////
// CHECK PREDICTION FUNCTION
///////////////////////////////

// Check a prediction [source_correct,destination_correct
   ]
std::pair<bool,bool>
GeneralPredictor::checkPrediction(unsigned source_tile,
   unsigned destination_tile,std::string address)
{
```

```
bool source_correct;
bool destination_correct;
// Find the key for this address
bool entry_present = isEntryPresent(address);
uint32_t set_number = getSetNumber(extractIndex(
   address));
uint32_t tag = extractTag(address);

if (!entry_present) { // No entry for this
   address
        source_correct = false;
        destination_correct = false;
        if (Verbosity) {
                printf("PRED: Checking prediction
                   , message is from %i to %i, no
                    entry for this address in LUT
                    \n",source_tile,
                   destination_tile);
        }
} else {
        if (LUT[set_number].Entries[tag]->
           PredictionInfo[ValidPredictionBit] ==
           1) {
                if ((uint32_t) source_tile == LUT
                   [set_number].Entries[tag]->
                   PredictionInfo[
                   PredictedSourceBit]) {
                        source_correct = true;
                } else {
                        source_correct = false;
                }

                if ( (uint32_t) destination_tile
                   == LUT[set_number].Entries[tag
                   ]->PredictionInfo[
                   PredictedDestinationBit]) {
                        destination_correct =
                           true;
                } else {
                        destination_correct =
                           false;
                }
                if (Verbosity) {
                        printf("PRED: Checking
                           prediction, message is
                            from %i to %i, was
                           predicted from %zu to
                           %zu  \n",
                                                                source_tile
                                                                   ,
                                                                   destination_tile
                                                                   ,LUT[
                                                                   set_number
                                                                   ].
                                                                   Entries
                                                                   [tag
                                                                   ]->
                                                                   PredictionInfo
                                                                   [
                                                                   PredictedSourceBi
                                                                   ],LUT[
                                                                   set_number
                                                                   ].
                                                                   Entries
                                                                   [tag
                                                                   ]->
                                                                   PredictionInfo
                                                                   [
                                                                   PredictedDestinat
                                                                   ]);
                                        }
                                } else {
                                        source_correct = false;
                                        destination_correct = false;
                                        if (Verbosity) {
                                                printf("PRED: Checking
                                                   prediction but no
                                                   valid prediction for
                                                   this entry \n");
                                        }
                                }
                        }
                        std::pair<bool,bool> correct_prediction;
                        correct_prediction = std::make_pair(
                           source_correct, destination_correct);
                        return correct_prediction;
}

///////////////////////////
// GENERAL HELPER FUNCTIONS
///////////////////////////

// Convert nodeID to tileID
// Ouput needs to be signed because might be stored in
   LUT
uint32_t
```

```
GeneralPredictor::convertNodeID(unsigned node_id)
{
        uint32_t tile;
        if (node_id < NumTiles) { // L1
                tile = node_id;
        } else if((node_id >=  NumTiles) && (node_id < 2*
           NumTiles)) { // L2
                tile = node_id - NumTiles;
        } else if((node_id >= 2*NumTiles) && (node_id <
           3*NumTiles)) { // Directory
                tile = node_id - 2*NumTiles;
        } else { // DMA
                tile = 0;
        }
        return tile;
}

// Return the directory for this address
uint32_t
GeneralPredictor::returnDirectory(std::string address,
   uint32_t set_number, uint32_t tag)
{
        return LUT[set_number].Entries[tag]->
           PredictionInfo[DirectoryBit];
}

// Return the unique sharer for this address
int
GeneralPredictor::returnUniqueSharer(std::string address,
    uint32_t set_number, uint32_t tag)
{
        // Return all SharerFields that are not equal to
           zero
        std::vector<unsigned> possible_destinations;
        for (unsigned i = SharerBit; i < ValidBit; i++) {
                if (LUT[set_number].Entries[tag]->
                   PredictionInfo[i] != 0) {
                        possible_destinations.push_back(i
                           -SharerBit); // This loop
                           needs to return the tileID's,
                           not the LUT ID's
                }
        }

        uint32_t comp_value = SB_normalSharer;
        int sharer = -1;
        for (unsigned i= 0; i < possible_destinations.
           size(); i++) {
                unsigned index_in_LUT = SharerBit +
                   possible_destinations[i];
                if (LUT[set_number].Entries[tag]->
                   PredictionInfo[index_in_LUT] ==
                   comp_value) {
                        if ( possible_destinations.size()
                            == 1) { // If there is only 1
                            tile, this is the unique
                           sharer and no need to compare
                           to directory
                                sharer =
                                   possible_destinations[
                                   i];
                                break;
                        } else if (possible_destinations[
                           i] != (unsigned) LUT[
                           set_number].Entries[tag]->
                           PredictionInfo[DirectoryBit])
                           {
                                sharer =
                                   possible_destinations[
                                   i];
                                break;
                        }
                }
        }
        if (Verbosity) {
                printf("HELPER: %i is unique sharer \n",
                   sharer);
        }
        return sharer;
}

// Return a list of all sharers for this address
std::vector<unsigned>
GeneralPredictor::returnSharers(std::string address,
   uint32_t set_number, uint32_t tag)
{
        // Return all SharerFields that are not equal to
           zero
        std::vector<unsigned> possible_destinations;
        for (unsigned i = SharerBit; i < ValidBit; i++) {
                if (LUT[set_number].Entries[tag]->
                   PredictionInfo[i] != 0) {
                        possible_destinations.push_back(i
                           -SharerBit); // This loop
                           needs to return the tileID's,
                           not the LUT ID's
                }
        }

        std::vector<unsigned> sharers;
        for (unsigned i = 0; i < possible_destinations.
           size(); i++) {
                unsigned index_in_LUT = SharerBit +
                   possible_destinations[i];
                if (LUT[set_number].Entries[tag]->
                   PredictionInfo[index_in_LUT] ==
                   SB_normalSharer) {
                        sharers.push_back(
                           possible_destinations[i]);
                }
        }
        assert(sharers.size() <= possible_destinations.
           size());
        if (Verbosity) {
                printf("HELP: %zu possible sharers, only
                   %zu normal sharers \n",
                   possible_destinations.size(), sharers.
                   size());
        }
        return sharers;
}

// Get the number of sharers for this address
unsigned
GeneralPredictor::getNumSharers(std::string address,
   uint32_t set_number, uint32_t tag)
{
        // Return all SharerFields that are not equal to
           zero
        std::vector<unsigned> possible_destinations;
        for (unsigned i = SharerBit; i < ValidBit; i++) {
                if (LUT[set_number].Entries[tag]->
                   PredictionInfo[i] != 0) {
                        possible_destinations.push_back(i
                           -SharerBit); // This loop
                           needs to return the tileID's,
                           not the LUT ID's
                }
        }
        return possible_destinations.size();
}

// Remove a sharer
void
```

```
GeneralPredictor::removeSharer(std::string address,
   uint32_t set_number, uint32_t tag, unsigned sharer)
{
        LUT[set_number].Entries[tag]->PredictionInfo[
           SharerBit + sharer] = SB_noSharer;
        if (Verbosity) {
                printf("HELP: Removed %u from the
                   sharersList for %u \n",sharer,
                   set_number);
        }
}

// Return the sharer which sent out an eviction message
int
GeneralPredictor::returnEvictor(std::string address,
   uint32_t set_number, uint32_t tag)
{
        int evictor = -1;
        for (unsigned i = SharerBit; i < ValidBit; i++) {
                if (LUT[set_number].Entries[tag]->
                   PredictionInfo[i] == SB_currentEvictor
                   ) {
                        evictor = i - SharerBit;
                        break;
                }
        }
        if ((evictor == -1)&& VerbosityPrintErrors) {
                 printf("Error -- no evictor to be found"
                    );
        }
        return evictor;
}

// Set as evictor
void
GeneralPredictor::setAsEvictor(std::string address,
   uint32_t set_number, uint32_t tag, unsigned
   source_node)
{
        LUT[set_number].Entries[tag]->PredictionInfo[
           SharerBit + source_node] = SB_currentEvictor;
        if (Verbosity) {
                printf("PRED: Marking %i as in the midst
                   of an eviction for %i \n",source_node,
                   set_number);
        }
}
```

```
// Mark as having received an invalidation message
void
GeneralPredictor::hasReceivedInv(std::string address,
   uint32_t set_number, uint32_t tag, unsigned node)
{
        LUT[set_number].Entries[tag]->PredictionInfo[
           SharerBit + node] = SB_receivedInv;
        if (Verbosity) {
                printf("PRED: Marking %i as having
                   received an invalidation for %i \n",
                   node,set_number);
        }
}

// Get the sharer which will need to receive an inv
   message next
int
GeneralPredictor::getNextSharerToInv(std::string address,
    uint32_t set_number, uint32_t tag)
{
        int node_to_inv = -1;
        for (unsigned i = SharerBit; i < ValidBit; i++) {
                if ( (LUT[set_number].Entries[tag]->
                   PredictionInfo[i] != SB_receivedInv)
                   && (LUT[set_number].Entries[tag]->
                   PredictionInfo[i] != SB_noSharer) && (
                   LUT[set_number].Entries[tag]->
                   PredictionInfo[i] != SB_upgrader)) {
                   // This node is a sharer but  has not
                   been invalidated yet and is not the
                   evictor nor the upgrader
                        node_to_inv = i - SharerBit;
                        break;
                }
        }

        if ((node_to_inv == -1) && VerbosityPrintErrors)
           {
                        printf("Error -- no next sharer
                           to be found to receive an INV"
                           );
        }
        return node_to_inv;
}

// Get the sharer which will send out an inv ack next
int
GeneralPredictor::getNextSharerToInvAck(std::string
   address, uint32_t set_number, uint32_t tag)
{
        int node_to_inv = -1;
        for (unsigned i = SharerBit; i < ValidBit; i++) {
                if (LUT[set_number].Entries[tag]->
                   PredictionInfo[i] == SB_receivedInv){
                   // This node has not yet acknowledged
                   its invalidation
                        node_to_inv = i - SharerBit;
                        break;
                }
        }
        if ((node_to_inv == -1) && VerbosityPrintErrors)
           {
                        printf("Error -- no next sharer
                           to be found which will ACK an
                           INV");
        }
        return node_to_inv;
}

// Get the number of nodes that still need to receive an
   invalidate message
unsigned
GeneralPredictor::getNumSharersToInv(std::string address,
    uint32_t set_number, uint32_t tag)
{
        unsigned sharers_received_inv = 0;
        for (unsigned i = SharerBit; i < ValidBit; i++) {
                if (LUT[set_number].Entries[tag]->
                   PredictionInfo[i] == SB_receivedInv){
                   // This node has not yet acknowledged
                   its invalidation
                        sharers_received_inv++;
                }
        }
        unsigned num_sharers = getNumSharers(address,
           set_number, tag);
        unsigned num_nodes_to_inv = num_sharers -
           sharers_received_inv;
        return num_nodes_to_inv;
}

// Set the upgrader
void
GeneralPredictor::setUpgrader(std::string address,
   uint32_t set_number, uint32_t tag, unsigned node)
{
        LUT[set_number].Entries[tag]->PredictionInfo[
           SharerBit + node] = SB_upgrader;
        if (Verbosity) {
                printf("PRED: Marking %i as in the midst
                   of an upgrade for %i \n",node,
                   set_number);
        }
}

// Return the upgrader
int
GeneralPredictor::getUpgrader(std::string address,
   uint32_t set_number, uint32_t tag)
{
        int upgrade_node = -1;
        for (unsigned i = SharerBit; i < ValidBit; i++) {
                if (LUT[set_number].Entries[tag]->
                   PredictionInfo[i] == SB_upgrader) {
                        upgrade_node = i - SharerBit;
                        break;
                }
        }

        if (upgrade_node == -1) {
                if (VerbosityPrintErrors) {
                        printf("Error -- no upgrader to
                           be found");
                }
        }
        return upgrade_node;
}

// Change the upgrader node to a normal node
void
GeneralPredictor::setUpgraderAsNormalSharer(std::string
   address, uint32_t set_number, uint32_t tag, unsigned
   node)
{
        LUT[set_number].Entries[tag]->PredictionInfo[
           SharerBit + node] = SB_normalSharer;
        if (Verbosity) {
                printf("PRED: Marking %i completed its
                   upgrade and normal sharer now for %i \
                   n",node,set_number);
        }
}
```

```
// Convert access type to int
uint32_t
GeneralPredictor::convertAccessType(std::string type)
{
        uint32_t int_type = RT_null;

        if (type == "LD") {
                int_type = RT_LD;
        } else if ( type == "IFETCH") {
                int_type = RT_IFETCH;
        } else if ( type == "ST") {
                int_type = RT_ST;
        } else if (type == "EVICT") {
                int_type = RT_EVICT;
        } else if (type == "UPGRADE") {
                int_type = RT_UPGRADE;
        } else if ( type ==  "L2_EVICT") {
                int_type = RT_L2_EVICT;
        }
        return int_type;
}

////////////////////////////////
// MESSAGE DISTRIBUTION
////////////////////////////////
void
GeneralPredictor::addToDistribution(std::string address,
   uint64_t time)
{
        // Get set number
        assert(isEntryPresent(address));
        uint32_t set_number = getSetNumber(extractIndex(
           address));
        uint32_t tag = extractTag(address);
        uint64_t inter_arrival_time = time  - LUT[
           set_number].Entries[tag]->PredictionInfo[
           TimeStampBit];

        assert(time-LUT[set_number].Entries[tag]->
           PredictionInfo[TimeStampBit] >= 0);

        if (StatsMessageInterArrivalTimes.count(
           inter_arrival_time) > 0) {
                // We have already seen this value before
                   , just increase the counter
                StatsMessageInterArrivalTimes[
                    inter_arrival_time]++;
        } else {
                std::pair<uint64_t, uint32_t> key_pair;
                key_pair = std::make_pair(
                   inter_arrival_time, 1);
                StatsMessageInterArrivalTimes.insert(
                   key_pair);
        }
        if (inter_arrival_time > StatsMaxInterArrivalTime
           ) {
                if (Verbosity) {
                        printf("PRED: new max = %zu for %
                           s @ %zu \n",
                           inter_arrival_time, address.
                           c_str(), time);
                }
                StatsMaxInterArrivalTime =
                   inter_arrival_time;
        }

        StatsArrivedMessages++;
        if (Verbosity) {
                printf("PRED: inter-arrival time for %s
                   is %zu @%zu\n", address.c_str(),
                   inter_arrival_time, time);
        }
        // Now update the timestamp bit
        // Record the time the last message (this message
           ) in the sequence was seen
        LUT[set_number].Entries[tag]->PredictionInfo[
           TimeStampBit] =  time;
}

void
GeneralPredictor::printDistributionToFile(std::string
   file_name)
{
        // Convert the unorder map to a vector
        std::vector<uint32_t> inter_arrival_times;
        printf("PRED: reserved %zu spaces for vect \n",
           StatsMaxInterArrivalTime+1);
        inter_arrival_times.resize(
           StatsMaxInterArrivalTime+1);
        for (auto it = StatsMessageInterArrivalTimes.
           begin(); it != StatsMessageInterArrivalTimes.
           end(); it++){
                uint64_t time_to_record = it->first;
                uint32_t occurrence = it->second;
                if (time_to_record >= inter_arrival_times
                   .size()) {
                        inter_arrival_times.resize(
                           time_to_record + 1, 0);
                        printf("Should not occur - time=%
                           zu, occ=%u \n",time_to_record,
                           occurrence);
                }
                inter_arrival_times[time_to_record] =
                   occurrence;
        }
        // Print to file
        std::ofstream output;
        output.open(file_name.c_str(), std::ofstream::app
           ); // Append to file
        // Loop over the distribution
        output << "dist = " << StatsArrivedMessages << "
           messages" ;
        for (unsigned i = 0; i < inter_arrival_times.size
           (); i++){
                output << ";"  << inter_arrival_times[i];
        }
        output << "\n";
        output.close();

}

void
GeneralPredictor::resetTimings(uint64_t current_time)
{
        // Loop over all sets
        for (auto set_it = LUT.begin(); set_it != LUT.end
           (); set_it++) {
                // Loop over all entries in the set
                for (auto in_set_it = set_it->second.
                   Entries.begin(); in_set_it != set_it->
                   second.Entries.end(); in_set_it++) {
                        // Set the TimeStampBit to the
                           current time
                        in_set_it->second->PredictionInfo
                           [TimeStampBit] = current_time;
                }
        }
        printf("PRED: Resetting the inter-arrival times \
           n");
        StatsMessageInterArrivalTimes.clear();
        StatsMaxInterArrivalTime = 0;
}
```

```
void
GeneralPredictor::resetTimeStamp(std::string address,
   uint64_t current_time)
{
        assert(isEntryPresent(address));
        uint32_t set_number = getSetNumber(extractIndex(
           address));
        uint32_t tag = extractTag(address);
        LUT[set_number].Entries[tag]->PredictionInfo[
           TimeStampBit] = current_time;
        if (Verbosity) {
                printf("PRED: setting timeStampBit of %s
                   to %zu \n", address.c_str(),
                   current_time);
        }
}

std::string
GeneralPredictor::getPredProperties()
{
        std::ostringstream string_stream;
        if (PredictorType == "set_associative"){
                string_stream << "NW="<< Associativity ;
                string_stream << "SB="<< StartBit;
        } else {
                string_stream << PredictorType;
        }
        return string_stream.str();
}

std::string
GeneralPredictor::getPartialTagProperties()
{
        std::ostringstream string_stream;
        if (!PartialTag) {
                string_stream << "null";
        } else {
                string_stream << "PEndBit=" <<
                   PartialEndBit;
                string_stream << ";PStartBit=" <<
                   PartialStartBit;
        }
        return string_stream.str();
}

uint32_t
GeneralPredictor::getReplacements()
```

```
{
        uint32_t total_replacements = 0;
        for (uint32_t repl_set :
           StatsNumberConflictsPerSet) {
                total_replacements += repl_set;
        }
        return total_replacements;
}
```