CHALMERS
UNIVERSITY OF TECHNOLOGY

On-Chip Cache Coherency Evaluation for Space Applications: Snoop with AHB Bus vs. Directory with CHI NoC

Master's thesis in Embedded Electronic System Design

Alex Helmersson
Veronica Kimelman

Department of Microtechnology and Nanoscience
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2025

Master's thesis 2025

On-Chip Cache Coherency Evaluation for Space Applications: Snoop with AHB Bus vs. Directory with CHI NoC

Alex Helmersson

Veronica Kimelman

Department of Microtechnology and Nanoscience
Chalmers University of Technology
Gothenburg, Sweden 2025

On-Chip Cache Coherency Evaluation for Space Applications: Snoop with AHB Bus vs. Directory with CHI NoC
Alex Helmersson
Veronica Kimelman

Supervisor: Ahsen Ejaz, Department of Computer Science and Engineering
Company advisor: Martin Rönnbäck, Frontgrade Gaisler
Examiner: Per Larsson-Edefors, Department of Microtechnology and Nanoscience

Master's Thesis 2025
Department of Microtechnology and Nanoscience
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

On-Chip Cache Coherency Evaluation for Space Applications: Snoop with AHB Bus vs. Directory with CHI NoC
Alex Helmersson
Veronica Kimelman
Department of Microtechnology and Nanoscience
Chalmers University of Technology

Abstract

With the end of frequency scaling as a solution for more efficient processors, bigger multi-core systems are getting increasingly popular, including in the space domain. To achieve bigger high-performing multi-core systems, more efficient on-chip data transfer systems are required which in turn are dependent on systems scaling. Increasing the size of a multi-core system introduces more complexity in upholding coherency. Different protocols and hardware mechanisms have been used for supporting coherent systems, such as snooping and directory-based, which can be supported by different interconnects. Although NoC-based interconnects and directory-based mechanisms have been shown to successfully scale multi-core systems, past research lacks a comparison of the coherence overhead introduced in NoC versus bus interconnects.

This thesis project has performed an evaluation and comparison of two protocols for upholding coherency, AHB and CHI, used by Frontgrade Gaisler in their current and future state-of-the-art radiation hardened microprocessors. Two evaluation frameworks have been developed, one for each protocol, with configurable simulation stimuli targeted for cache coherency. The limited experiments we managed to perform in this project identified scenarios and topologies where the more expensive CHI system will not deliver a higher and more scalable performance compared to the more simple AHB system. This provides a valuable foundation for the next step, that is performing quantitative analysis of the topologies where CHI delivers superior performance as the system scales to a larger number of cores. However, due to technical challenges we leave that part for future work. Improvements such as increasing parallelism by using non-blocking components and sliced and distributed last-level-cache, as well as using more advanced NoC routers with reduced latency should be part of the future work that demonstrates the advantages of CHI over AHB system. Performance gains in the evaluated CHI system could still be seen when using a coherency protocol that allows for a write-back policy instead of write-through.

Keywords: Cache coherency, Network-on-Chip, AMBA AHB, AMBA CHI

Acknowledgements

We want to thank our academical supervisors Ahsen Ejaz, Mehrzad Nejat, and Madhavan Manivannan as well as our supervisors at Frontgrade Gaisler Martin Rönnbäck and Krishna Kalipurath Radhakrishnan. We are very thankful for the invaluable help and support through the entire project, and also grateful for the long discussions and many laughs along the way.

We also want to thank our examiner Per Larsson-Edefors, who gave us valuable encouragement throughout the project and helpful finishing touches on the report.

Alex Helmersson & Veronica Kimelman, Gothenburg, October 2025

Contents

Acronyms

AHB Advanced High-Performance Bus.
AMBA Advanced Microcontroller Bus Architecture.

CHI Coherent Hub Interface.

DUT Device Under Test.

EUMMSS Efficient Uncore Mechanisms for Multicore Space Systems.

HN Home Node.

I Invalid.
IP Intellectual Property.

LRU Least Recently Used.

NoC Network-on-Chip.

QEMU Quick Emulator.

RN Request Node.

SC Shared Clean.
SD Shared Dirty.
SN Subordinate Node.
SoC System-on-Chip.

UC Unique Clean.
UD Unique Dirty.
UVM Universal Verification Methodology.

1
Introduction

When increasing a processor's clock frequency became too power inefficient, increasing the number of cores became the new efficient way of increasing processor performance [1]. Adding on the widely adopted system design choice of using a private L1 cache for each core [2] [3], cache coherency protocols have become a relevant topic. These protocols ensure all cores have a consistent and synchronized view of the data in memory.

In modern cache coherence, mainly two cache coherency types exist: snoop- and directory-based [4]. The decision between the two often depends on system specifics but can be generalized to the directory approach having better scalability than the snoop approach [5]. Therefore, the directory approach is more commonly used in larger rather than smaller multi-core systems.

In the space industry, the scaling of System-on-Chip technology, driven by the increasing number of cores, introduces challenges in designing the SoC's uncore system, meaning all processor parts excluding the core, particularly concerning interconnect topology and cache coherency. State-of-the-art space-qualified processors like Frontgrade Gaisler's quad-core GR740 [6] and their still-under-development octa-core GR765 [7] incorporate a mechanism to solve these problems by having all of the processor cores and their L1 write-through caches connected to a shared AMBA AHB bus. The shared system-wide bus allows for snooping to detect memory updates.

Gaisler in a collaboration with Chalmers University of Technology is currently developing a SoC prototype which integrates Chalmers' CHI based last-level-cache and Network-on-Chip with Gaisler's SoC. Gaisler develops systems and IP cores for space applications, therefore they design their hardware by prioritizing dependability and fault tolerant systems to counter the effects of space radiation on electrical circuits[8][9]. They are now faced with the challenge of determining the threshold at which the number of processors would make their new directory-based approach better than their old snoop bus approach.

This project aims to develop an evaluation platform for evaluating Gaisler's SoC implementations and to utilize the platform for comparing the new directory-based cache coherency with their baseline snoop-based cache coherency. A performance analysis of the new and baseline SoC architectures will be performed to provide feedback to Gaisler about the benefits and trade-offs of adopting a directory-based cache coherency approach in their multi-core SoCs. The project becomes relevant for comparing the directory and snoop based approaches, comparing system-wide buses

against NoC solutions, and contribute to continued research of hardware solutions for space applications.

1.1 Related Work

There are many examples of how the effort of scaling multi-cores systems has been successfully done with both NoC based interconnect [10] and directory based cache coherency mechanisms [11]. The Intel Teraflops Research Chip [12], a prototype 80-core processor, uses a mesh network to connect the cores which includes one level of a cache. However, its system does not use cache coherency. This to avoid the communication overhead in maintaining coherency.

Limited research has been conducted on multi-core systems where the L1 caches' coherency is maintained by a NoC. One existing research design is the STORM Multiprocessor System-on-chip (MPSoC) [13], developed to investigate the cost directory based cache coherency mechanism has on a NoC interconnected MPSoC. STORM is designed to support up to 256 cores, each with a cache that is directly connected to the NoC interconnect. The cost was found to be well within acceptable limits. Different cache designs have also been evaluated on the system [14], while it was emphasized in [13] and [14] that NoC-based platform design papers often neglect the importance of cache coherency. Neither of the two STORM-related studies, despite their efforts to highlight this research gap, provides a comparison of the cache coherence overhead in a NoC versus a bus interconnect.

A paper comparing interconnect options for cache coherency is [15] where a chip made up of tiles is proposed. Each tile consists of a core, a private L1 and a dedicated slice of a shared L2. Coherency between the L1s is maintained with two different mechanisms, a snooping bus-based network and a directory based packet-switched network. The authors in [15] argue against the idea that a larger sized multiprocessor, such as one with 16 cores, automatically requires packet-switched networks with complex routers and a directory-based coherence protocol. This design has the disadvantage of the directory-based protocol suffering from indirection, the request has to first be processed by the directory before the L2. For every protocol transaction multiple messages are therefore required, consuming additional time and energy for the request to complete. Because of the packet-switched network, each message has to also pass multiple routers in the network, causing more drawbacks to the system.

To reduce energy consumption, [15] proposes a segmented bus-architecture with filters to suppress broadcasts, leading to no loss in performance compared to its packet-switched counterpart. However, the research paper does not address the implications of distributing a sliced L2 cache across tiles, in contrast to many other architectures that employ a unified L2 cache shared by all cores.

For space systems, one approach ensuring that on-board computers can survive the harsh environment is the use of fault-tolerant hardware [16]. Previously mentioned related work have a limitation in the space field as they do not need to consider this critical requirement in space environments. Although multi-processor systems

provide better performance and more features, their complexity makes it harder to ensure fault tolerance [17].

At the same time, multi-core processors are also inherently redundant in terms of processors, I/O and memory ports. This can be leveraged when developing software and hardware for fault-tolerant computing techniques. In fault-containment, the practice of isolating error-prone areas in order to stop a fault spreading to other areas, one option is to isolate each core. However, as described in [16], also isolating the cores' memory controllers and I/O ports can lead to a negative impact on performance.

A configurable isolation technique is proposed in [17], where a partial partitioned design for a multi-core processor is developed. The cores and their resources are divided into multiple groups, each containing a configurable number of cores. Each group shares resources such as an interconnect and L2 bank. To support fault-tolerant techniques such as Triple Modular Redundancy, a primary core and its redundant core can not be a part of the same group. The internal interconnect connecting the L1s of a group to their shared L2 bank is made up of a shared bus with a ring topology. The L1 coherency is ensured by snooping the shared bus.

Although [17] addresses one gap that previously mentioned research overlooked in the context of space applications, it does not explore other interconnect systems and cache coherence mechanism for performance gains. Related work is mentioned in which the coherence mechanism is designed with fault tolerance techniques incorporated, an approach that could benefit both performance and reliability in harsh space environments. However, this was not included in the final design of [17].

1.2 Purpose and Goal

In order to make an adequate design decision on what coherency mechanism and interconnect topology to use, the designer has to have knowledge about how design choices will impact the overall system. Latency, bandwidth and congestion are factors that vary across different topologies. Having a higher latency and smaller bandwidth can limit the system performance. Coherency mechanisms come at the expense of overhead in terms of bandwidth or resources. Directory based coherency mechanisms generate more coherence traffic and require more sophisticated NoCs. Therefore there is an interest to know to what extent the overhead will affect the overall performance.

The previously mentioned project between Chalmers and Gaisler, Efficient Uncore Mechanisms for Multicore Space Systems (EUMMSS), is aiming for a SoC based on a 2D mesh NoC topology [18]. The EUMMSS project focuses on evaluating performances and recommending strategies for improving scalability in Gaisler's upcoming multicore systems which are composed of multiple coherent subsystems. Each subsystem consists of multiple cores and their L1 cache together with a shared L2 cache. However as the intra-subsystem communication in the EUMMSS design is still based on a bus network, comparing the performance of this mixed-topology SoC with a fully bus network-based SoC would not allow for a meaningful performance comparison.

This research project will explore evaluation methods for a coherent multi-core system, with the purpose of providing designers with a framework for performance evaluation. The goal is to be able to evaluate, compare, and give recommendations regarding an existing bus-interconnected SoC (for example the GR765) and a NoC-interconnected SoC in terms of their cache coherency mechanisms. The challenge in this is to implement an evaluation suite and find a good enough method that tests coherency and compares bus interconnect versus NoC, with minimal interference of other factors.

Metrics that will be measured by the proposed evaluation suite are listed below:

- Memory access time
- Interconnect bandwidth
- Coherence protocol overhead

These metrics should also be benchmarked using configuration parameters defined for the evaluation suite, such as the number of processors and cache hit and miss ratios. This provides designers with a clear indication of the threshold, defined by the number of processor cores, at which each topology is most suitable.

1.3 Thesis Outline

The remainder of this thesis is organized as follows:

Chapter 2: describes the project's technical background.

Chapter 3: describes the method used in order to get the research's result.

Chapter 4: presents the design of the project's evaluation frameworks.

Chapter 5: presents the produced results of the evaluations.

Chapter 6: discusses the results.

Chapter 7: gives final remarks and conclusions of the project's results, as well as suggestions for future work.

2

Technical Background

This chapter describes the background information that is required for the reader to understand the thesis project. It begins with an introduction to key concepts related to processor memory hierarchy design, followed by an overview of cache coherence mechanisms and interconnect systems. Finally, this chapter introduces the two AMBA protocols (AHB and CHI) which will be compared for the project.

2.1 Memory Hierarchy and Cache Design

One of the biggest challenges since the earliest days of computing is providing programmers with unlimited amounts of fast memory [19, p. 388]. Modern applications require an expanding memory space to provide the processor with data and instructions at a high speed [20, p. 193]. But, the speed gap between the processor and main memory is increasing. To bridge the gap, a memory hierarchy of multiple levels of caches with various sizes and access time is employed. The memory hierarchy is based on speed, size and cost. When the distance between processor and memory increases, the size and access time of that memory increases while the cost per bit decreases [20, p. 195].

When implementing a memory hierarchy one takes advantage of the principle of locality. Defined in [19, p. 388-389], the principle of locality states that programs access a relatively small portion of their address space at any instant of time. Furthermore there are two different types of locality; temporal and spatial. The temporal locality principle states that if a data location is referenced it will likely be referenced again soon. The spatial locality principle states that if a data location is referenced, data locations with nearby addresses will likely be referenced soon. It can be seen from the natural structure of a program that locality will be an inherent property. Temporal locality can be seen in loops, where instructions and data are likely to be accessed repeatedly. In high-level sequential programs, instructions are accessed sequentially, showing the spatial locality principle. Spatial locality in programs can also be seen in data accesses, for example through the sequential access to elements in an array or record type.

The goal with a memory hierarchy is to provide the programmer with as much memory as is available with a technology cost as low as possible, while also providing memory access at the speed that is offered by the fastest memory [19, p. 389]. The data in the entire system is similarly hierarchical, a level that is closer to the processor is in general a subset of any level further away, and all of the system's

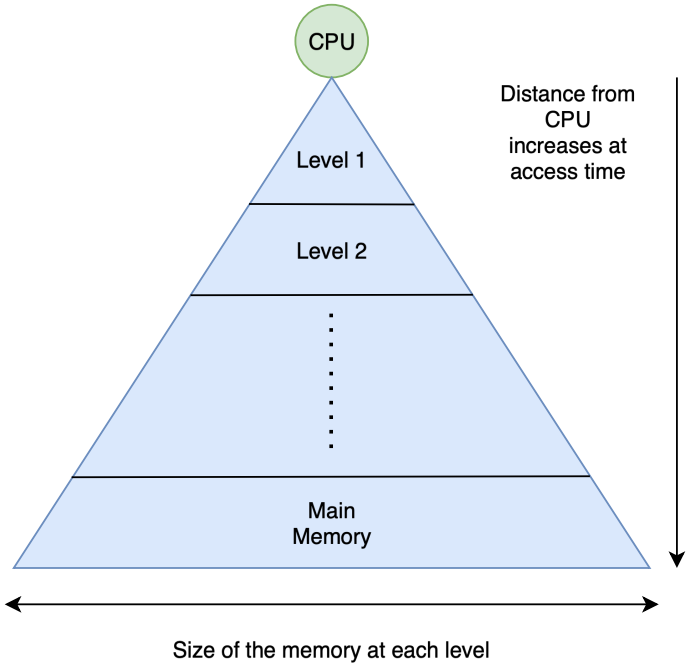


Figure 2.1: Structure of a memory hierarchy. Figure reproduced from [19].

data is stored at the lowest level. Figure 2.1 presents this structure with a triangular diagram where upper levels are smaller and faster than the lower levels.

The higher levels above the main memory in the hierarchy are referred to as caches. The cache closest to the processor is commonly known as the Level 1 (L1) cache, the second one - if it exists - as the L2 cache, and so on. As mentioned, each cache level stores a subset of the memory levels below it. To define one part of data in this subset, we are using the definition of a cache line from [19, p. 390]. A line is the minimum unit of information that can either be present or not present in the cache. The number of lines in a cache corresponds to the number of entries in that cache. When a processor performs a memory request, it will either want to read or write to a specific location in the system's memory area. The memory hierarchy is built in a way so that the processor accesses the highest level of cache, particularly the line corresponding to the requested location in the main memory. As per the previous definition, this unit of information can either be present in the L1 cache or not. If the requested line is present it will result in a hit and when not it will result in a miss. A miss in a L_N cache will generate a propagation of the memory request to the L_{N+1} level of memory.

To know whether the requested data is in a cache or not a set of tags are added to the cache entry line. The tag of a corresponding cache line contains the address of the data held in that line. The tag does not need to contain the whole address size, as the number of lines in a cache is a subset of the main memory's size. It only needs to contain the upper portion of the address, the part that is not used to index the lines of the cache.

Other control bits can also be added to the entry of a cache line, such as a valid or dirty bit. These control bits indicate whether the data stored at a specific tag is

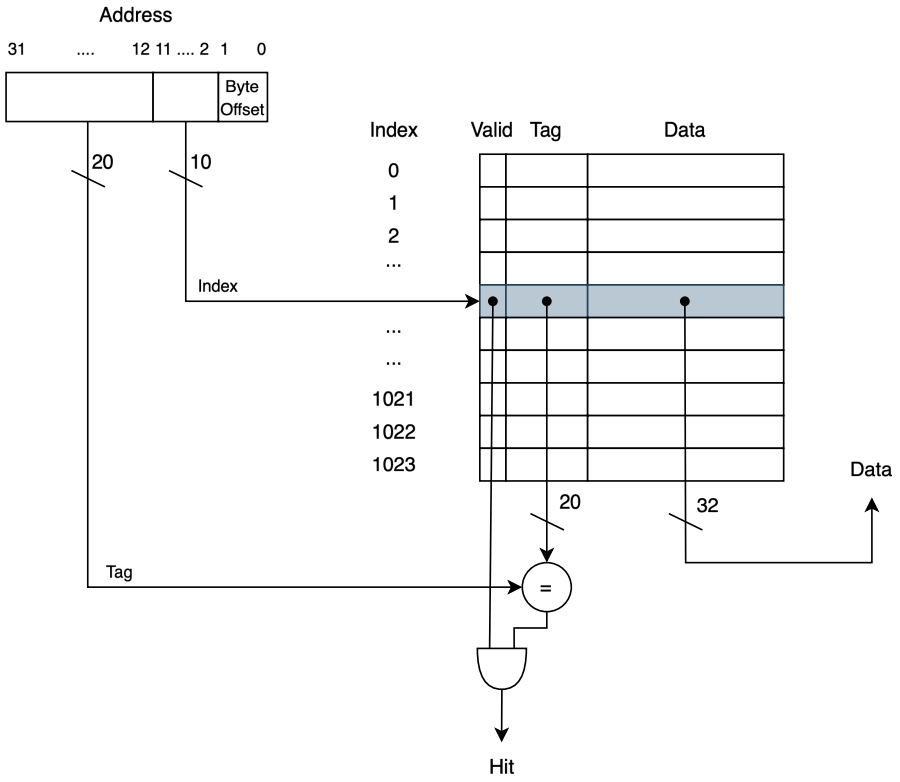


Figure 2.2: Simple cache implementation with valid, tag and data bits per cache line. Figure reproduced from [19]

valid or not. An example of how a simple 4KiB cache is implemented with a valid and tag bit entry field, together with the hardware logic required to determine a hit is shown in figure 2.2. In this figure the structure of the lines is a direct mapping of the cache. This means that each line in the memory only has one location where it may be placed in the cache [19, p. 401].

2.1.1 Cache Associativity

The principle of where a line can be placed in the cache is called associativity [19, p. 416-418]. If a cache is fully associative a line in the memory can be associated with any entry in the cache. When a cache is not fully associative it is set associative, and it is usually given in the name how many locations in the cache a line can be associated with. For example, in a two-way set associative cache each line can be associated with two locations of that cache. Therefore a one-way set associative cache is a direct mapped cache. A set associative cache is made up of sets, where the number specifying the degree of associativity is the number of lines in each set.

The reason for different associative line placement schemes is their effect on cache performance, particularly regarding their hit and miss rate as well as their resulting access time. A direct mapped cache has a fast access time on a hit as there is only one location where the line could be in, while a fully associative cache has a higher hit rate than the other two alternatives because of its entirely flexible line placement availability [20, p. 200-201]. The disadvantage of this flexibility is the prohibitively large overhead of the logic needed for finding a cache line and handling replacements [19, p. 418]. In a set associative structure, accesses to each set is direct mapped but a line mapped to a set can be placed anywhere in the set.

2.1.2 Write Policies

Other aspects needs to be taken into consideration for performance as well when designing a cache. Handling writes comes with the complexity of upholding memory hierarchy consistency when overwriting data as compared to read operations. When writing to the L1 cache the data would be inconsistent with the lower level caches or main memory if the first level cache does not propagate the writes. In order to ensure this consistency the cache can utilize specific write schemes. Two widely used schemes are called write-through and write-back [19, p. 408]. The former method always updates both the current cache and the next lower level of the memory hierarchy during a write. The latter, with the help of a dirty bit in its cache entry, only writes the new data to the requested line in the current cache, and it does not update the corresponding line in the next level until that line has been replaced in the current cache. The write-back method can improve performance, especially in those cases when write requests can be generated faster than the time to handle writes in the lower levels of memory.

Even with reducing the amount of lower level memory accesses for writes using write-back, each write will generally still take a considerably longer amount of clock cycles, compared to non-memory instructions, having to access the memory which stalls the processor. A solution to this bottleneck is to use a write buffer [19, p. 408]. A write buffer stores the requested write data while it is waiting to be written to the next memory level. This decreases the instances of when the processor has to stall because of numerous write requests. The processor would only have to stall when the write buffer is full.

Handling a write miss can also be done in specific ways, leading to different performance. The most common strategy during a write miss is to allocate a line in the cache, fetch that line from the lower level memory and overwrite it to the allocated location in the higher level cache. This is called write allocate [19, p. 409]. Another strategy, no write allocate, propagates the write request to the next level memory and overwrites the data currently stored on the requested location but it does not allocate a line in the cache for that data. This way, the replacement of useful data that may be needed later is avoided.

2.1.3 Replacement Policies

Reducing miss rate is also taken into consideration when deciding on which line to replace. In a direct mapped cache, there is only one fixed place for where a line can be written and in turn which line to overwrite. In an associative cache there is a variety of where the requested line can be placed and therefore the cache is now given a choice on which line to replace. In a fully associative cache all lines can be replaced while in a set associative cache any line in the requested set can be replaced [19, p. 421-422].

The procedure to decide on which line to replace is called replacement policy. The simplest policy is random selection [20, p. 202]. Another example of a replacement policy is the Least Recently Used (LRU). The LRU policy relies on the principle of locality that, as described earlier, states that data which have been requested is likely to be requested again. LRU will therefore choose the line that was least recently used when replacing a line on a miss.

2.2 Cache Coherency

A multiprocessor system consists of multiple processors sharing a common physical memory area. This introduces coherency problems for the individual caches of the processors, as two different processors can end up seeing two distinct values. Figure 2.1 gives a simple example of the cache coherence problem in a system with L1 write-through caches.

Time Step	Event	Cache contents for P1	Cache contents for P2	Memory contents for address X
0				0
1	P1 reads X	0		0
2	P2 reads X	0	0	0
3	P1 stores 1 into x	1	0	1

Table 2.1: Example of two processors in a cache coherent system with a write-through policy. In the last row, two caches store different values of the same memory line, showcasing the cache coherence problem.

A memory system is considered coherent when three properties, defined in [19, p. 476], are satisfied. The properties are as follows:

1. If a processor P is reading location X after writing to X, with no writes to X by another processor between the two operations by P, the value returned should have been written by P. If P1 reads after the time step 3 in table 2.1 it should see the value 1.
2. A read by processor P1 to location X after processor P2 has written to X should return the written value by P2. This is if the read and write operation have sufficient time in between and no other write to X has occurred between

the operations. This means that in the example of table 2.1 a mechanism is needed so that the value of P2's cache is eventually updated from 0 to 1.

3. Writes to the same location have to be serialized. This means that two writes to the same location by any two processors are seen in the same order by all processors. For example, in table 2.1, P1 and P2 cannot simultaneously write into location X at the same cycle, since they may write different values which breaks coherency.

To maintain cache coherency, multiprocessors often implement a hardware protocol. These protocols are called cache coherence protocols [19, p. 477]. A key functionality of a coherency protocol is being able to track the state of any sharing of a data line. In the following two subsections, the snoop and directory protocols are introduced.

2.2.1 Snooping Protocol

In a snoop coherence mechanism multiple processors together with their L1 caches are connected on a shared bus [20, p. 249]. Any read request is handled in the same way as in a single-processor system; a hit will immediately produce a response with the requested data while a miss has to first fetch the data from the L2. However for write requests, additional actions are required to uphold coherency. When a write is made to a line already available in the cache, the lower level of the memory hierarchy is updated as normal. All other caches of the same level in the system, which also has a copy of the same memory line needs to invalidate that line by clearing their valid bit. The same procedure happens if a write is made to a cache where the requested line doesn't exist but with the addition of the line first being fetched from the lower level memory.

If two processors each wants to write to the same line simultaneously, one of them will get their write request granted before the other. The other processor, which didn't get its request granted, will invalidate that line in its cache. To be able to finish the requested write operation the other processor has to now, in a write allocate protocol, obtain a new copy of the line which will now contain the updated data from the first processor's write. This example showcases how the method of invalidating cache lines enforces the third property of coherency defined previously [19, p. 477].

The way each cache knows if it has to invalidate a line that has been written to, by another cache, is by monitoring the shared bus. This method is called snooping [20, p. 250]. Each time a write is requested by any processor's cache on the bus, all other caches will snoop the bus and compare the address with their currently stored lines, to decide if any invalidation needs to be done.

2.2.2 Directory Protocol

Instead of distributing the status of shared memory lines, a directory-based coherency protocol offers a centralized structure to track the status of all memory lines. This is done in a directory [19, p. 479]. A directory protocol has advantages over snooping with respect to scalability and by removing unnecessary overhead

related to snoop operations. This is because the directory will only signal to the relevant caches that need to update their cache lines.

The directory keeps track of the status of each memory line. A directory can either be memory-centric or cache-centric. In a memory-centric directory, each memory line is located in the directory together with their status in each cache. In a cache-centric directory, the content of each cache is located in the directory.

An example of cache line status that a directory can track is presented later in section 2.4.2. The status of a cache line depends mainly on two factors, the number of copies of that line in the coherent cache system and the values of the valid and dirty bits in the line's cache entry.

The action that needs to be taken by the cache given by the directory can be the same as in a snoopy protocol (for example invalidate the cache line), as that is implemented by the cache. A directory based cache protocol defines the mechanism of requesting specific caches of taking necessary action to uphold cache coherency.

2.3 Interconnect Architectures

Interconnect systems are an important part in any computer system. Even with a low miss rate and penalty time in an efficiently designed memory system, half of the execution time can still be spent on transferring the data between memory and processor [20, p. 309]. It is therefore essential to keep the latency short when moving data through the memory hierarchy. Having sufficient bandwidth is also important in order to avoid contention among memory requests.

The interconnect networks are responsible for transferring instructions and data between the processor core and its surrounding components. These systems can be built with different architectures, suitable for different system's requirements. The question of interconnect design becomes increasingly important in a multi-core system as a design fitting for a single core system may not scale well when adding cores.

The following sections will introduce the theories behind a bus and a NoC interconnect.

2.3.1 Bus Interconnect

A bus interconnect is a shared medium between processors and components, also called nodes [20, p. 322-323]. A bus connects nodes, allowing them to communicate with each other. Since the bus is shared between the nodes, a requesting node has to be given exclusive access before it can send its data. Granting a node access to the bus is handled by a bus arbiter. The requesting nodes which have not been granted access must wait until the bus is available. To do this, the connected nodes all monitor the bus activity.

The exclusive access to a bus does not have to last for the whole duration of an operation. If a processor sends a read request to a cache connected on the bus, the memory access time could stall other nodes accessing the bus even though no

transfer of data is taking place. To increase the bus utilization, the bus arbiter can grant other nodes access during the operation of another node's request. To support this functionality a bus transaction is divided into a request and a response phase. The bus would then only be exclusively occupied during one of these two phases. From the previous example, other requests can be granted access on the bus during the time in which the cache fetches the requested line. Once the cache wants to send back the requested read data, it will have to request access from the arbiter to send its response over the shared bus.

Splitting the transaction into different phases adds latency to each transaction. There is a trade-off between this longer latency and the increase in bandwidth. The number of nodes in a bus interconnected system will also have an effect on both the transmission latency and the bus bandwidth as more nodes leads to longer wires and higher capacitive load on the bus. Higher capacitance on the bus' input signal will lead to slower rise and fall times, which degrades the bus bandwidth. With a low number of connected nodes, a bus interconnect network can be preferred because of its low complexity.

2.3.2 Network-on-Chip

As mentioned, the bus architecture can quickly lead to communication bottlenecks when scaling an interconnected multi-core system. The need for a more generalized interconnect solution is desirable. A prominent solution is the NoC, which is a data-routing network consisting of communication links and routing nodes, everything implemented on-chip [10].

The NoC idea is based on network theory. By exploiting parallel operations in one communication network, performance can be increased. The system can benefit from an increased throughput by naturally going from a bus architecture to a network design such as switching out wires for pipelines and bus-bridges to routing nodes [10].

It is not only the ideas of the physical architecture of a network that are applicable to a NoC, but also the widely accepted abstraction models for network communication; OSI. The authors of [10] divides the research area of NoC into four groups: 1) system, 2) network adapter, 3) network and 4) link. Figure 2.3 shows a comparison between the abstraction model layers of NoC research and the OSI layers, highlighting the data-flow through the network and the components that are part of each layer.

The *system layer* encompasses the system application and architecture. For example cores, memory and I/O peripherals. In this area, network implementation details are often still hidden.

The *network adapter layer* separates the cores from the network. It handles the control of end-to-end data flow by encapsulating transactions generated by the cores nodes into packets. Packets additionally contain destination information and other relevant information needed by the routing strategy of the network protocol. This is the highest level that is aware of the network protocol.

The *network layer* defines the topology and implements the network protocol. This level handles inter-communication of the network so it controls the node-to-node

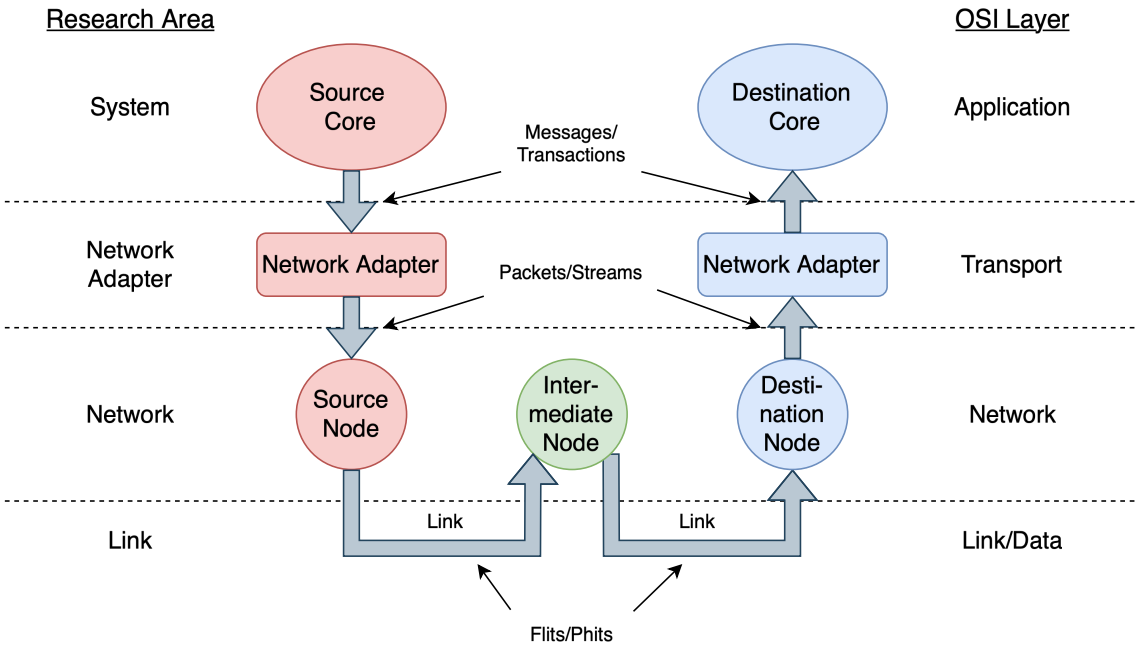


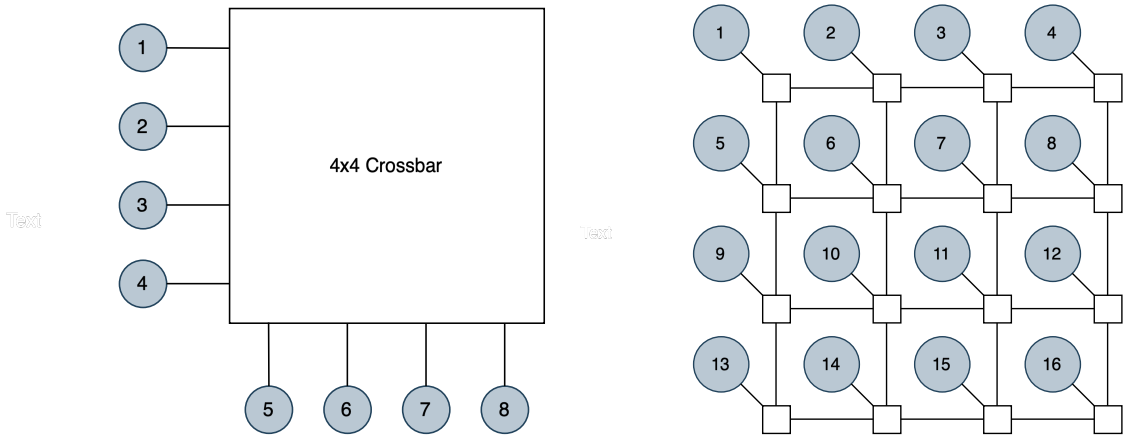
Figure 2.3: Model diagram comparing the research areas within NoC to the OSI model. Picture reproduced from [10].

data flow. It receives packets from the layer above and routes them forward through its links and routing nodes.

The *link layer* is the lowest layer. It consists of the smallest units of the transferred information, such as flits (flow control units) and phits (physical units). Flits are the flow control units that make up packets. While flits are the smallest control units, phits are the minimum size datagrams that can be transmitted in one link transmission. In highly serialized links, flits can be made of a sequence of phits.

The interconnect topology defined in the link layer varies between implementations. Common topologies are mesh and crossbar, which are both direct network. To understand what a direct network is one has to understand the role of a network node [21, p. 47]. A network node can be a terminal that acts as a source or sink for packet transfers. It can also be a switch node that forwards packets from input ports to output ports. In a direct network every network node acts as both a terminal and switch. Packets are forwarded directly between terminal nodes. In the opposite situation, an indirect network, a node can only act as one function (terminal or switch). Packets are therefore indirectly forwarded between terminals through switch nodes in an indirect network.

A crossbar switch network is a direct network with input and output ports. An N-by-N crossbar can connect N number of nodes in its network, each with their dedicated input and output ports [20, p. 323]. Figure 2.4a shows an example of this topology. A node is connected to the input and output ports depending on the expected actions it will perform on the NoC. If it should both send and receive requests through the NoC the node would have to be connected both through an input and output port.



(a) Crossbar **(b)** Mesh

Figure 2.4: Examples of crossbar and mesh topology. The gray circles are protocol nodes and the white squares are routers. Figures reproduced from [22].

An N-by-N crossbar can be implemented with $N \cdot N$ number of internal switches. The switches in the crossbar are responsible for transporting the packets to the correct destination. These switches are not to be confused with a switch node in a NoC defined in [21]. Instead, they are part of the internal implementation of one crossbar switch. They can for example be implemented as muxes. Each internal switch can connect the horizontal paths with the vertical path with the help of a routing mechanism. Point-to-point connection can be made between any N distinct pairs of nodes connected to the NoC.

The bandwidth of a crossbar NoC scales linearly with the number of nodes. Latency can also be affected as the length of the paths also increases linearly with the number of nodes. A NoC and especially one with a crossbar topology is resource heavy as the number of switches increases quadratically in relation to the number of nodes. These factors make a crossbar more fit for a limited number of nodes as the scalability is limited by cost per resource requirements.

In a mesh network each processor node is directly linked to a router or switch, which can be seen in figure 2.4b. The two can be implemented as one network node which is connected to its direct horizontal and vertical neighbors. A mesh network is more scalable than a crossbar as adding network nodes does not add to the complexity of internal switching in the same way it does in a crossbar.

A key reason for the added efficiency in a NoC, compared to a bus, is the fact that communication resources are shared [21, p. 13]. Instead of needing dedicated channels between each node, a NoC has a collection of shared router nodes connected by shared channels. This is especially beneficial in a larger multi-processor system.

2.4 AMBA Protocols

Advanced Microcontroller Bus Architecture (AMBA) specifications are a series of standardized interfaces and protocols for various applications developed by ARM

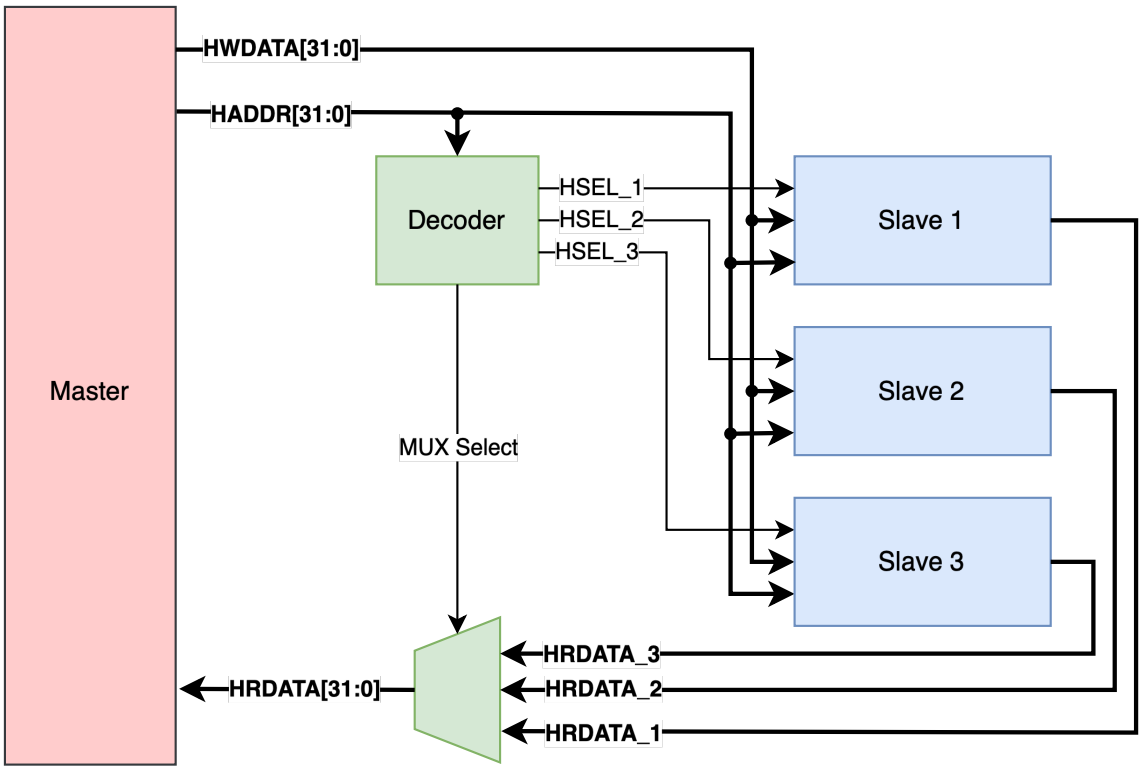


Figure 2.5: AHB system with one master and three slaves. Figure reproduced from [23].

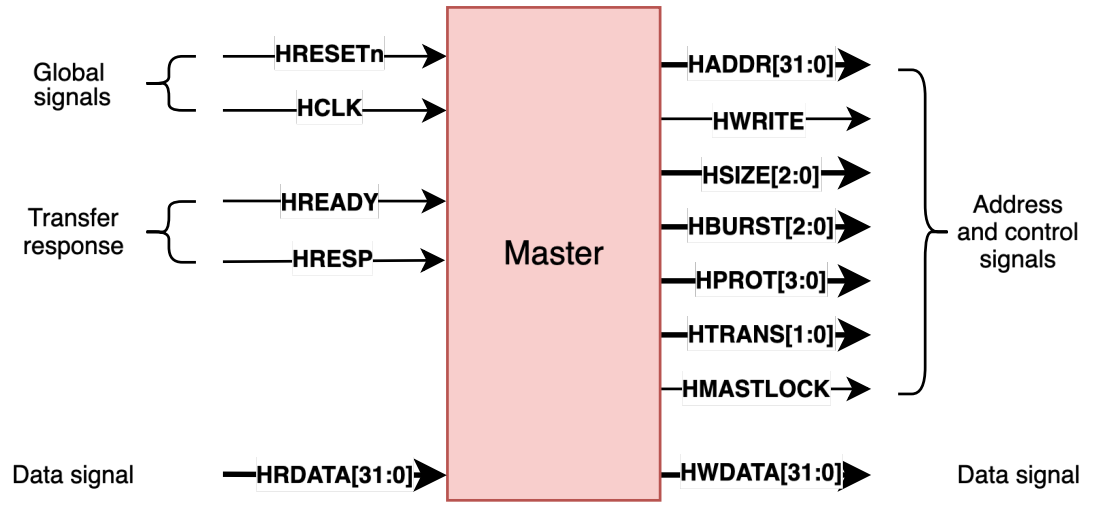
Limited. Two AMBA specifications are introduced in this section. The first, Advanced High-Performance Bus (AHB), is a protocol for a high-speed bus interconnect. The second, Coherent Hub Interface (CHI), is a directory-based coherence protocol.

2.4.1 Advance High-Performance Bus

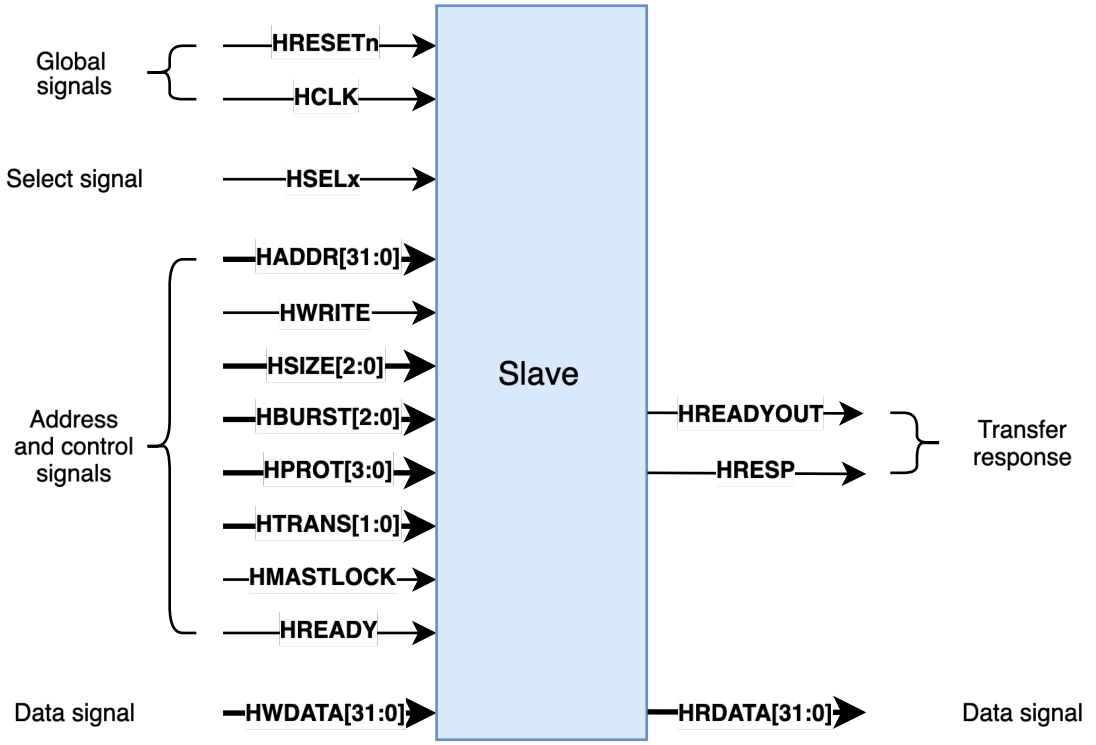
The AHB protocol defines the interface between components called masters and slaves and the interconnect between them [23]. The interconnect protocol supports high-bandwidth components such as internal memory devices, external memory interfaces and high-bandwidth peripherals. Figure 2.5 shows an example block diagram of how an AHB system can look like with one master, three slaves, an address decoder and a slave-to-master multiplexor. AHB supports multiple masters as well, where an arbiter and extra routing signals from all masters to the slaves are added.

The master is responsible for initializing a read or write transmission and providing the address and control information for the corresponding operation. The slaves respond to the request initialized by the master with the completion, extension or failure of the bus transfer. The interface of masters and slaves is shown in figures 2.6a and 2.6b.

Any transfer on an AHB bus has two phases, address and data [23]. The address phase usually last for one clock cycle if it is not extended by the previous bus transfer. The data phase can last for multiple clock cycles. The duration of the



(a) AHB Master



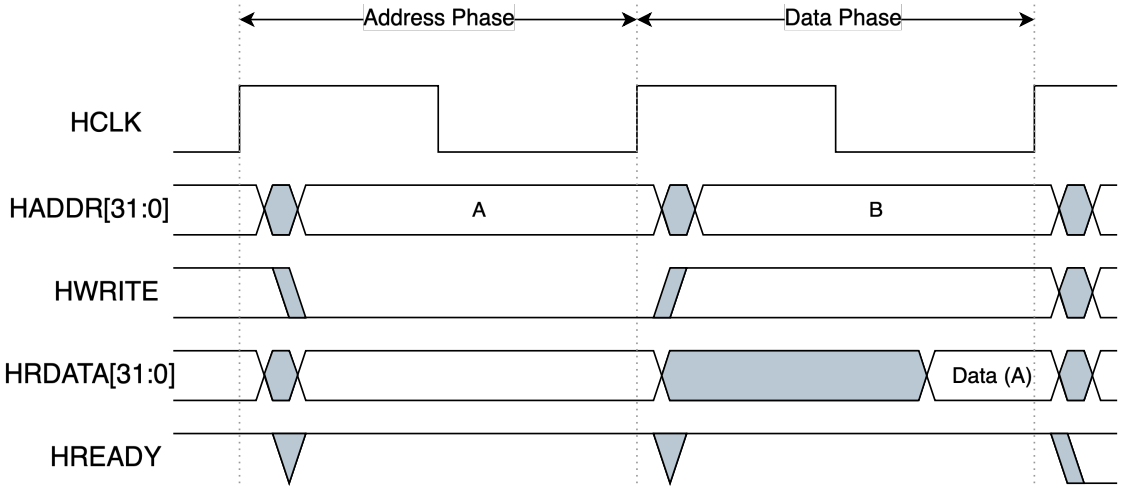
(b) AHB Slave

Figure 2.6: Interface signals for AHB master and slave. Figures reproduced from [23].

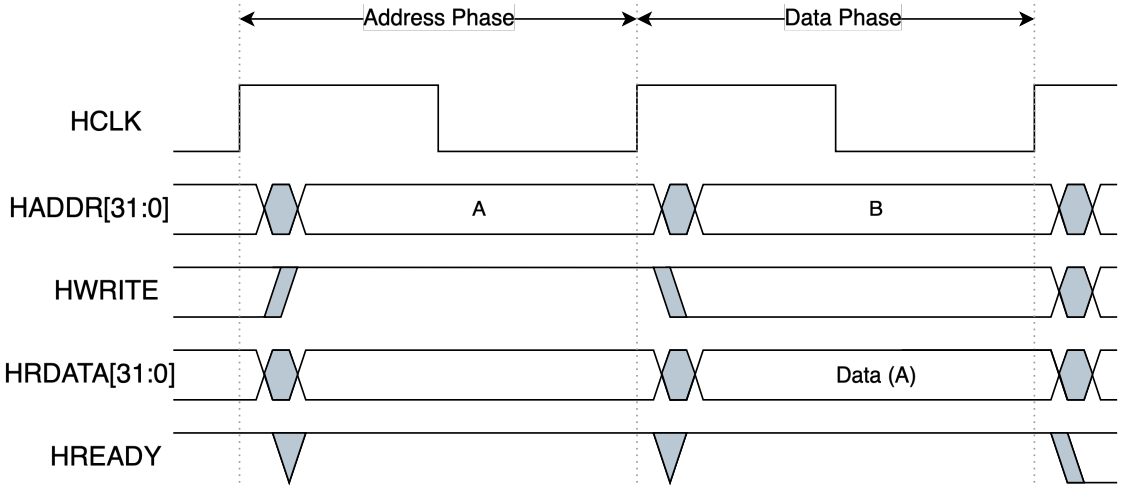
data phase is controlled with the **HREADY** signal by the slave. Figure 2.7a and 2.7b shows two examples, one read respectively one write transfer. Both examples have transfers that last for one clock cycle. The **HWRITE** signal controls which operation is taking place and it is set by the master during the address phase. If **HWRITE** is a logical 1 a write will take place and the master should broadcast its write data after the next rising edge of **HCLK** on the write data bus **HWDATA**. If **HWRITE** is a logical 0 a read will take place and the slave must generate the read data on the read data bus **HRDATA** in the same clock cycle as **HREADY** is a logical 1 in the data phase.

An example of a multi-cycle data phase is shown in figure 2.7c. When the slave samples **HWRITE** in the second rising edge of **HCLK** it can start to drive its **HREADYOUT** response depending on how many clock cycles the operation will take. The interconnect of the AHB system is responsible for combining the **HREADYOUT** from all slaves into one single **HREADY** which controls the overall transfer [23].

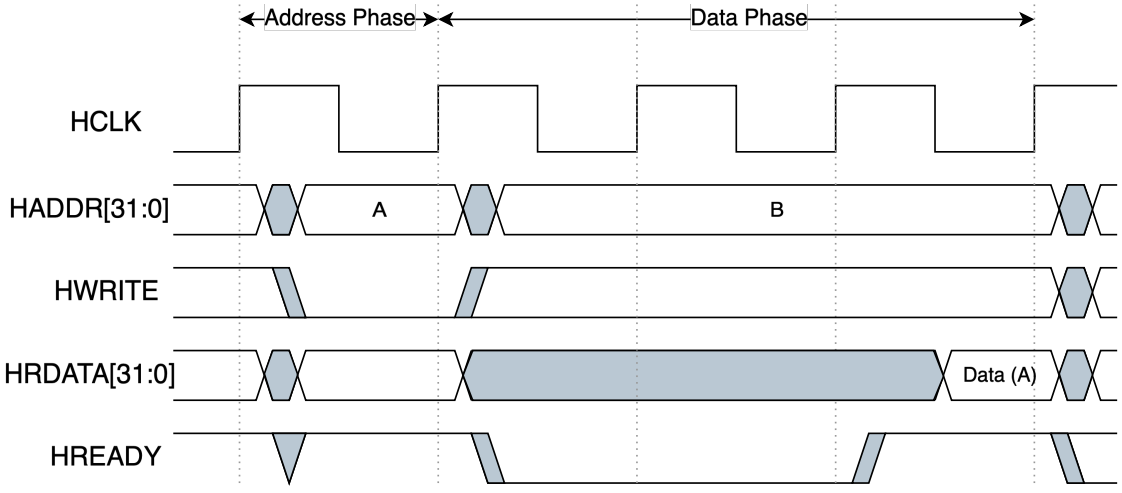
What can additionally be seen in figure 2.7 is the pipelined structure of the bus. The address phase of any transfer will take place in the data phase of the previous transfer [23]. The overlapping of the address and data being sent allows the high-performance of the AHB bus while simultaneously giving the slave enough time to response to the request of the master.



(a) Read transfer without wait cycles



(b) Write transfer without wait cycles



(c) Read transfer with two wait cycles

Figure 2.7: AHB transfers. Figures reproduced from [23].

2.4.2 Coherent Hub Interface

ARM Limited has developed a coherence protocol called CHI, which is a topology-independent interconnect system specification [22]. The CHI protocol itself can be viewed as the system level from figure 2.3, and it describes the interfaces of the protocol layer, network layer and link layer which are implemented by the chosen topology. The protocol layer described in the specification corresponds to the network adapter layer.

The CHI protocol's main connected components are the Request Node (RN), Home Node (HN) and Subordinate Node (SN). A requester, connected to an RN interface will initiate a transaction on the NoC. The requester can also be called RNF for Fully Coherent Requester Node, if it includes a hardware-coherent cache. The HN will handle coordination between all transaction requests on the NoC such as snoops, cache and memory accesses. The SN will receive transactions from the HN and complete them accordingly. The SN is the most downstream agent in the network system, meaning it is furthest away from the processors' RN nodes.

The interconnect itself can be implemented with optional topology as a fabric of connected switches. CHI also supports other network concepts such as Quality of Service (QoS).

Packets are sent through the specific channels; request (REQ), response (RSP), data (DAT) and snoop (SNP). Each channel is bidirectional, they are divided into RX and TX links, as can be seen in figure 2.8. For example, the RN initiates memory access requests through the NoC, so the REQ channel would go from the RN's TX port to the NoC's RX port.

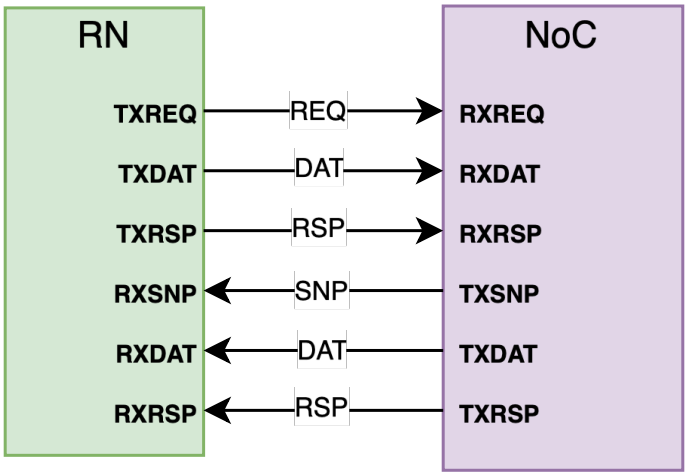


Figure 2.8: Interface for an RN node towards the interconnect. Figure reproduced from [22].

Each channel has a defined packet format with multiple fields. These can be seen defined for every channel in figure 2.9, with descriptions in table 2.2. The packet is sent on the **FLIT** link. The name **FLIT** is used as the CHI protocol considers packets as a single flit. The packet itself contains flow control information such as

target and source identification, QoS, opcode, and data (if the packet is sent on the DAT channel).

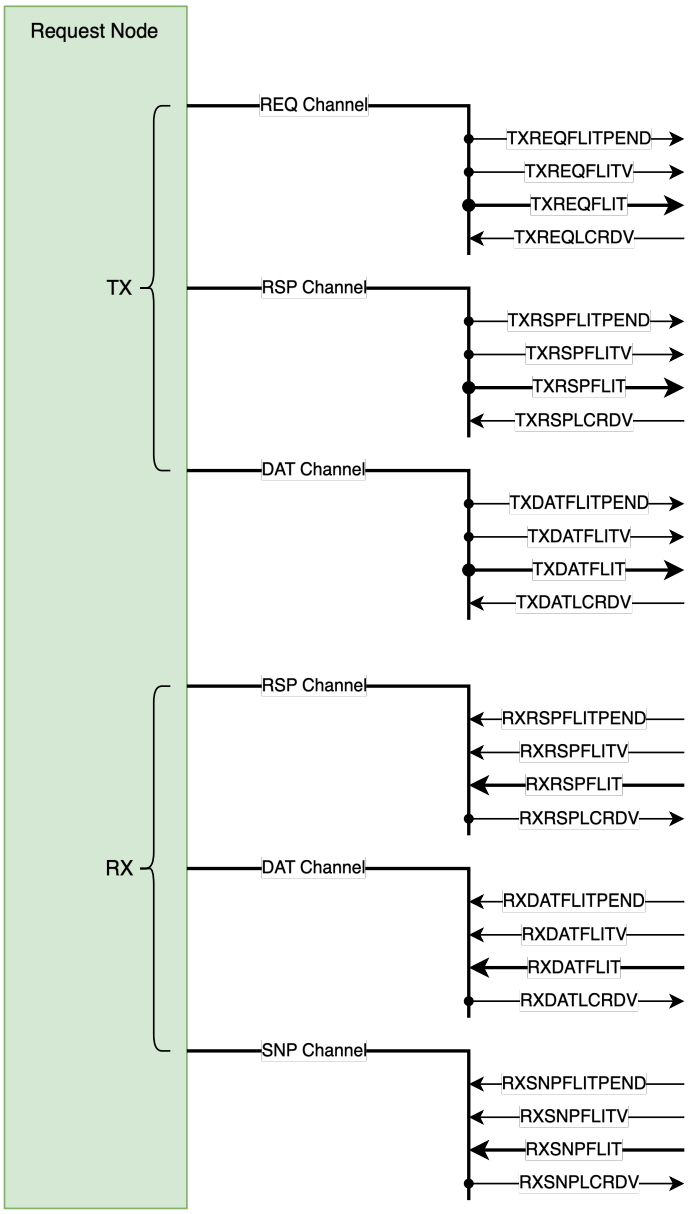


Figure 2.9: Relationship between ports, channels and links. Figure reproduced from [22].

Signals	Description
FLITPEND	Packet is pending. Indicates that a packet could be sent the next clock cycle.
FLITV	Packet is valid. Transmitter sets this as high if **FLIT[(W-1):0]** is valid.
FLIT[(W-1):0]	Packet with a width of **W**.
LCRDV	Link-Credit is valid. A credit is a guarantee that a Packet will be accepted from the receiver.

Table 2.2: Packet interface signals.

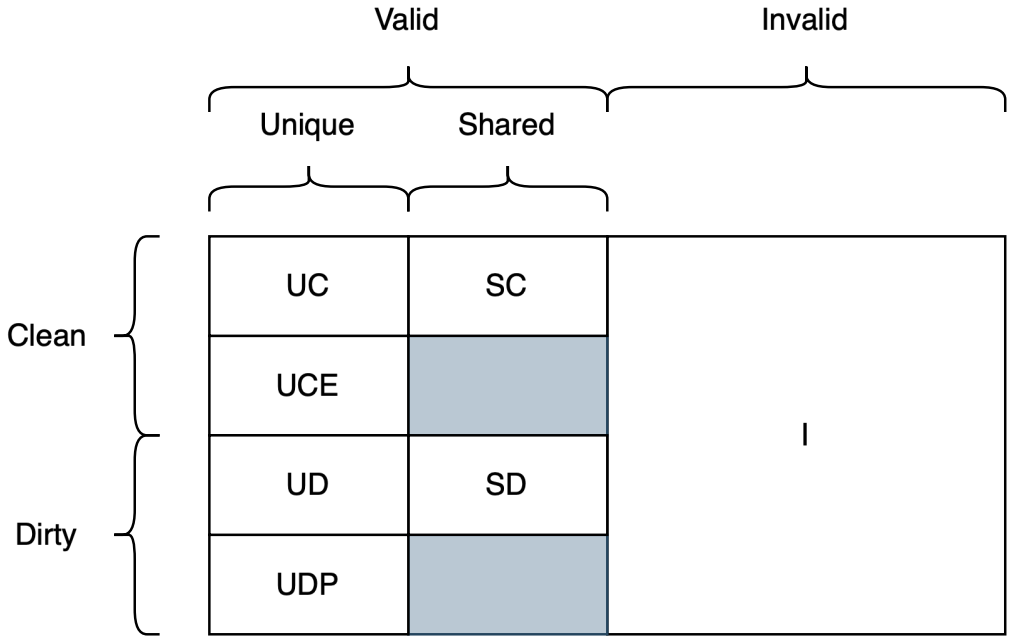


Figure 2.10: Cache state model of the CHI protocol. Figure reproduced from [22].

CHI handles coherency with the help of a cache state model, seen in figure 2.10. The different boxes each represent the state of a particular cache line that has been accessed by a processor. The HN decides what action is required by one or more caches when a processor node accesses a cache line. The action is determined by the cache line's state. The characteristics (column and row values of the state model) are defined as follow:

Valid & Invalid Cache line is valid if it is present in the cache. Cache line is invalid if it is not present in the cache.

Unique & Shared Cache line is unique if it only exists in one cache. Cache line is shared if it exists in more than one cache.

Clean & Dirty Cache line is clean if it does not need to write to next level cache. Cache line is dirty if it has been modified compared to the next level cache. It is responsible for eventually updating the next level cache.

3

Methods

In this chapter the methodology of the thesis is presented.

3.1 System Architecture

To meet the purpose and goals defined for this project, a framework needs to be designed for performing evaluations. A system target architecture has been chosen, which makes up the Device Under Test (DUT) of the evaluation framework, for the purpose of having a multi-core system with L1 coherency. The performance metrics to be evaluated will be measured on the interconnect between the L1 caches. A high-level view of the system is shown in figure 3.1.

Two systems will be constructed following the architecture of the target system. One for upholding the cache coherence by snooping on an AHB bus and the other one by having a directory with a CHI NoC. It is desired to have the same components for both systems to the extent possible, except for the obvious case of the protocol-specific interconnect. The design is inspired by the memory hierarchy of the GR765 which consists of three levels [7].

As the research topic is focused on performance within the memory interconnect, it has been decided to exclude a processor component in the design. This approach is used because we want to push the interconnect protocols to their limit for bandwidth evaluation. As we are only focused on the interconnect, modeling and simulation of the full processor functionality is not required.

Instead of having a processor run software as simulation input for the system, a test stimuli generator will be developed. This will allow for configurable tests where desired scenarios can be reached, such as hits and misses. It is also desirable if the stimuli generator could compile real-world software into memory requests in order to also test the system with industry-used benchmarks.

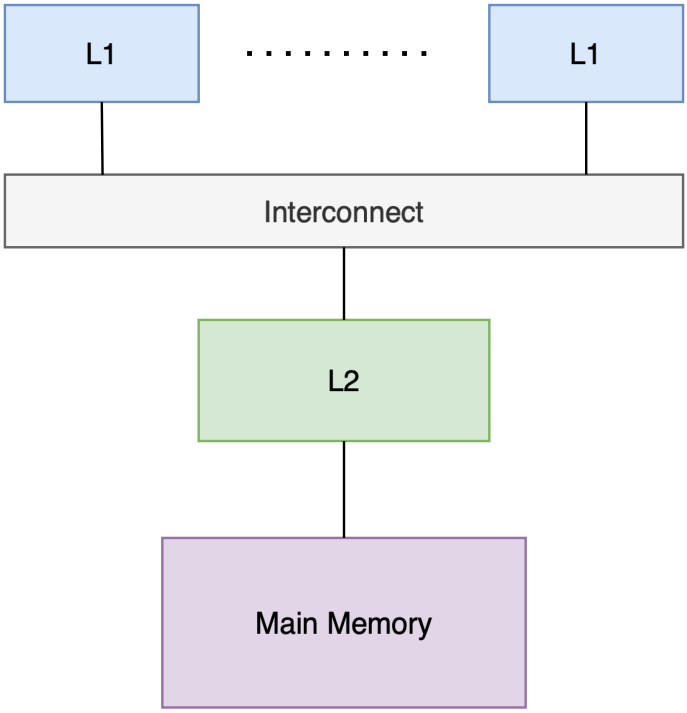


Figure 3.1: Target architecture for evaluation of system performance with coherency. The interconnect module will be evaluated.

3.2 Simulation and Tools

Many components are already available for the project through Gaisler's GRLIB library [9], EUMMSS and InfiniNode Technologies. QuestaSim will mainly be used for simulation and evaluation as GRLIB includes already set up projects with the tool.

To validate the functionality of the DUT and individual modules, SystemVerilog testbenches were written and simulated in both QuestaSim [24] and Verilator [25]. Since most modules written by Gaisler are in VHDL, and modules for EUMMSS in SystemVerilog, QuestaSim's mixed-language simulation will be utilized for simulating the functionality of the DUT. QuestaSim also includes the Universal Verification Methodology (UVM) library, which is a verification library created for simplifying and standardizing hardware verification [26]. In the thesis UVM will mainly be used for stimuli generation.

To facilitate the different tool support, a Makefile was implemented and connected to Gaisler's build system. Different make targets were created for simulating in Verilator or QuestaSim, and outside simulator config files were connected through the commands written in the make target. Different targets were also created depending on the components to be simulated.

4

Evaluation Framework

This chapter describes the design of the evaluation framework. The framework consists of two memory hierarchy systems; one with an AHB bus interconnect and one with a CHI NoC interconnect. A configurable L1 cache has been designed to support both the AHB and CHI protocol. For simulation, a UVM environment is connected to the Device Under Test (DUT) with different ways of generating test stimuli.

The names used for presenting components are the same which have been used in the provided libraries.

4.1 AHB System

The memory stack which makes up the DUT for the AHB evaluation suite can be seen in the red box in figure 4.1. The components below the L1 caches have not been designed for this project. They are already finished IPs that have been chosen and configured for this research project from the GRLIB library. The following subsections describe the IP blocks used in the project. The UVM environment from figure 4.1, seen in the blue box is described later in section 4.3.

4.1.1 Write-Through L1 Cache

Two different cache versions were developed; one for the AHB system supporting write-through, write-no-allocating, and a snooping based coherency system mimicking Gaisler's current L1 cache and one for the CHI system supporting write-back, write-allocate, and a directory based coherency system. The two different implementations were done to fully utilize the potential of each system.

To faciliate the idea of two different caches without having to redo much of the logic in both caches, a single cache module was written, and the SystemVerilog preprocessor directive `` `ifdef `` was used extensively throughout its design. With this directive it is possible to write code specific to one of the implementations while it is unseen in the other, simply by defining variables (e.g. `` `define SNOOP `` will make the compiler compile code to support snoop, `` `define WRITE_BACK `` will make the compiler compile code to support write-back). This makes common parts of the cache implementations and bug fixes available for both caches, while still separating the version specific parts of the module.

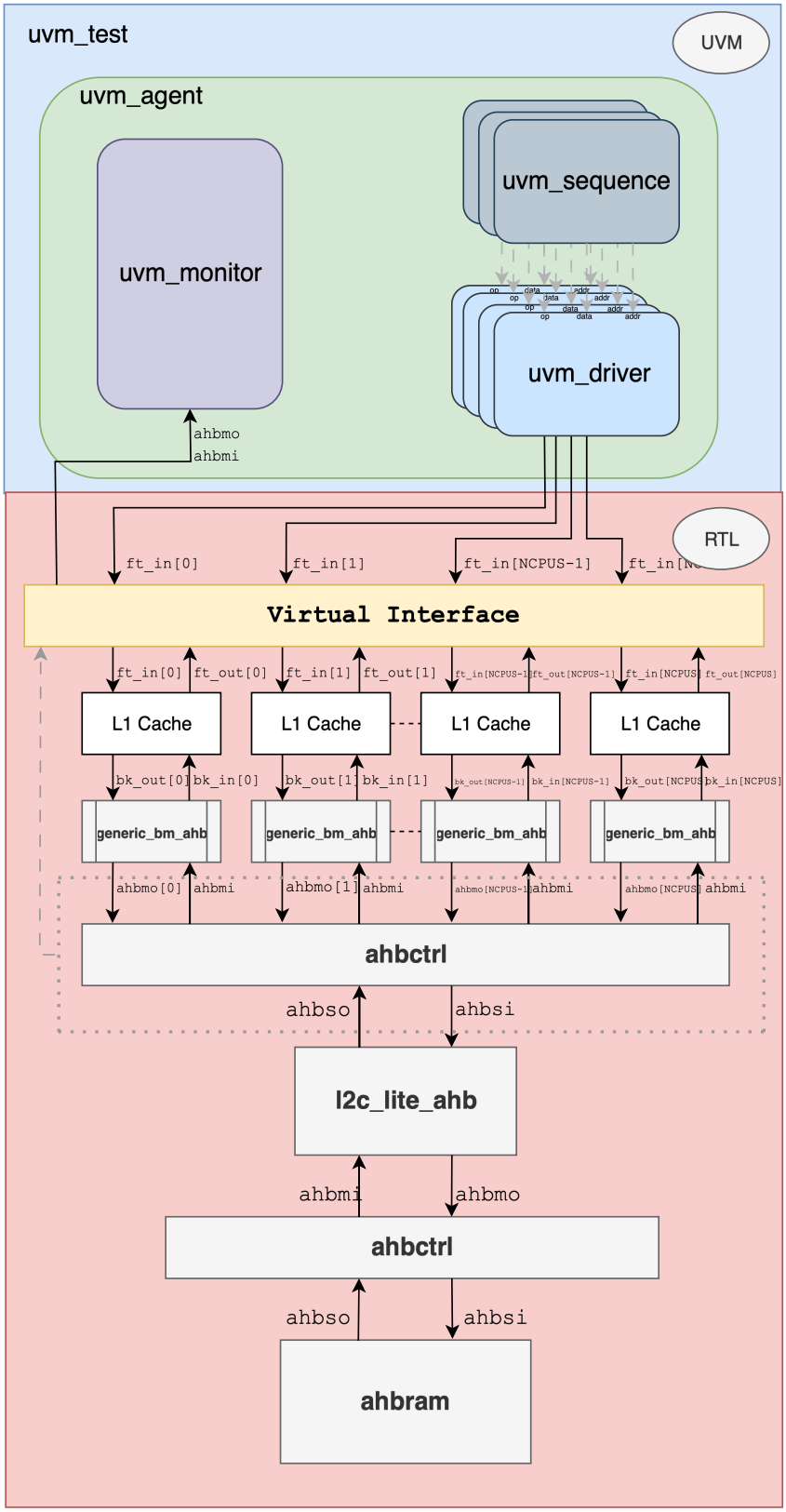


Figure 4.1: Block diagram of the evaluation environment for the AHB system.

To support write-through, write-no-allocate, and a snoop based coherency system a cache line status of valid, and LRU was implemented, shown in figure 4.2.

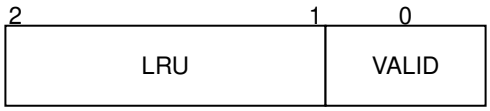


Figure 4.2: AHB cache control bits

The write-no-allocate part was implemented in the following way. If a write request produces a hit in the L1, the write data will be written to the requested cache line. If a write request produces a miss in the L1, the write request will be propagated to the L2 (together with the request's input size), without allocating the requested cache line to the L1. A write-allocate would have been possible to implement but could slow down the system and would complicate the design, as additional logic would have to be added to allocate a requested cache line to the L1 on a write miss. It would also not be in accordance to Gaisler's L1 cache implementation.

While the cache was implemented with big endian, the AHB RAM module was little endian. The cache was therefore changed to also support little endian by implementing a reverting function and applying it on the input bits from the frontend or the backend if corresponding endian bit of either the frontend or backend was set.

To mimic Gaisler's L1 cache a write-queue of size 4 was implemented for the AHB cache. This meant write requests could be accepted, queued, and considered done in a single clock cycle, letting the processor work on the next instruction while the cache solves the problem of transferring the write on the bus. The queue is as mentioned limited and when becoming full the cache has to stop accepting writes. This was solved by adding a state of `FULL` to the state machine; which the cache takes on if the queue is full and receives a write. In this implementation there is a coherency problem if a write in the queue is to the same address as a read miss. If the read is done before the write it will read the prior value before the write has written its value to the L2, contradiction the coherency rules presented in section 2.2. This is solved simply by waiting for the write queue to empty out before propagating a read on a read miss.

The cache's snooping mechanism works by checking the `ahbso` signal, seen in figure 4.1 to the L2 cache of the AHB bus. If a write request appears the tag bits of the requested address is matched against corresponding tag bits in the set index by the requests address index bits; if it matches, the cache line is invalidated.

One scenario we wanted to avoid was the caches being stuck invalidating read cache lines while snooping. This scenario could happen if the caches are snooping on the `ahbmo` of other caches. If cache 1 is reading a cache line and cache 2 wants to write to the same cache line, cache 1 will invalidate the read cache line upon receiving it. This is because the write request has not yet been accepted by the AHB arbiter. If caches instead snoop on the `ahbsi` they will only snoop on requests that has been accepted by the bus. This is the reason why snooping is done on the `ahbsi`.

Since write queue entries target specific sizes of the cache line, a write queue entry is not deleted if a snoop with equal block address is received. This is because it does not contradict the coherency rules, the write can instead be seen as happening after the snoop. The cache does not support locked requests.

4.1.2 AHB Generic Bus Master

The `generic_bm_ahb` IP, as shown in figure 4.1, translates the generic bus signals (used by many GRLIB IPs) into AHB signals and vice versa. The block outputs the AHB signals on the same clock cycle as the input is given, but since address and data transfers are split into different clock cycles in the AHB protocol, the IP block adds at least one clock cycle to the transfer.

4.1.3 AHBCTRL

The `AHBCTRL`, as shown in figure 4.1, is an IP block consisting of an AHB controller, a bus, and a user-defined number of bus lines. The block is used to connect AHB masters to AHB slaves and correctly route AHB traffic between the components. Through the IP's generic parameters the arbitration algorithm can be chosen to be either round-robin or fixed-priority; in this project it is configured to use round-robin.

There are two `AHBCTRL` entities in the design. The first one (in this project referenced as the main bus) is for connecting the L1 caches' backend to the L2 cache's frontend, and the second one is for connecting the L2 cache's backend to the main memory. It is therefore only the first bus that is snooped for cache coherency and the one that is evaluated for this research project.

The IP supports up to 15 masters. In our design, the number of masters is configured with the desired number of processor cores. Each master is given a grant ID by the AHB controller. As the bus only has one signal output to the masters, `ahbmi`, the masters uses their ID to know if the bus response is directed to them or not.

The AHB systems implemented in GRLIB supports bus width between 32-128. For a fair comparison between the interconnects there was a wish to make the input stimuli as similar as possible, including the data width. Because the NoC supports a data width of 512 bits, the AHB bus width was extended to that as well. This allowed both systems to receive the same input data size.

4.1.4 L2C Lite

The `l2c_lite_ahb` IP, as shown in figure 4.1, is used in this project as the L2 cache. It is a wrapper for the cache and its backend interface. The L2C Lite core uses Gaisler's generic bus interface towards its backend. These signals need to be translated to communicate with the AHB bus. The IP wrapper was chosen for easy integration instead of adding an additional `generic_bm_ahb` component.

The L2C Lite core from GRLIB supports bus widths between 32-128 bits. For our design this has been extended to also include 512 bits. The L2C Lite core uses

write-back as write policy, without a write buffer. The replacement policy can be chosen as either pseudo LRU or random, our instantiation uses pseudo LRU.

The cache can be configured for associativity as direct-mapped or as 2, 4, 8, 16, or 32 way set associative. Set and block size are also configurable. See the configuration done for our setup in table 4.1.

Config	Parameter Setting	Permitted Values
Set Associativity	4	1, 2, 4, 8, 16, 32
Set size (KiB)	64	1,2,4,8,16,32,...,1024
Block size (Bytes)	64	32, 64

Table 4.1: L2C Lite configuration

4.1.5 AHBRAM

The `ahbram_sim` IP, as shown in figure 4.1, has been chosen for the main memory RAM in our design. It is a non-syntheziable Single-Port RAM with the frontend interface of an AHB slave. Both read and write access have one wait state. As this is not the realistic case for on-chip memories, the wait time will be parameterized in our evaluation.

4.2 CHI System

The memory stack which makes up the DUT for the CHI evaluation suite can be seen in the red box in figure 4.3. The components below the L1 caches have not been designed for this project. They are already finished IPs that have been chosen, configured and in some cases extended for this research project from the GRLIB and EUMMSS project library. The following subsections describes the IP blocks used in the project.

4.2.1 Write-Back L1 Cache

To support the second cache implementation two new bits were added to the cache line state, which can be seen in figure 4.4. The cache supports the Invalid (I), Shared Clean (SC), Unique Clean (UC), and Unique Dirty (UD) states from the cache state model in figure 2.10. The shared bit tells whether or not the cache line could be or actually is present in another cache. If it can be shared with another cache it can not be written to (since the cache does not support Shared Dirty (SD) state), instead it must first be acquired from the HN in a unique state. The dirty bit tells whether the cache line has not been propagated to the HN after being written to. If so the cache needs to propagate a write to the HN before evicting or invalidating the cache line.

To facilitate write-back, when replacing cache lines, the cache checks the status bits of the cache line. If the status is valid and dirty the cache line needs to be propagated to the RN, and is then put on the `rnf_out` output, seen in figure 4.3

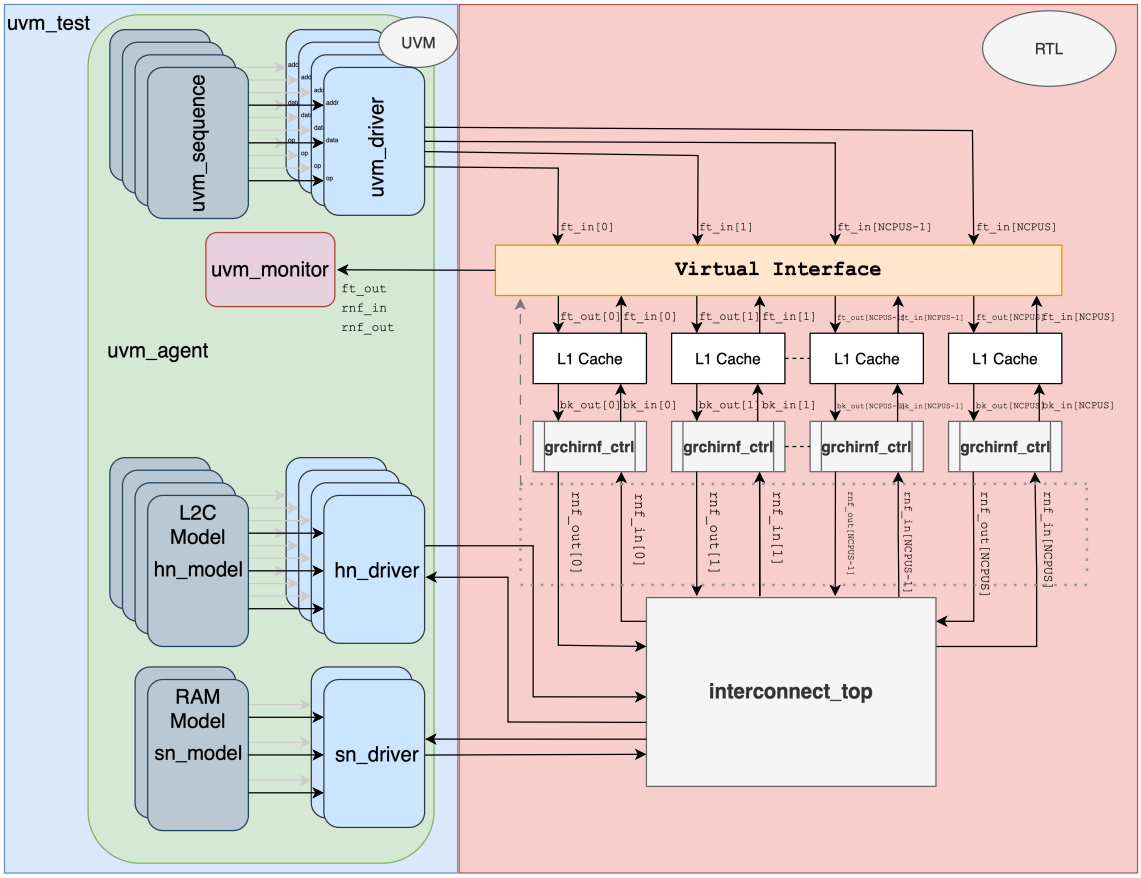


Figure 4.3: Block diagram of the evaluation environment for the CHI system.

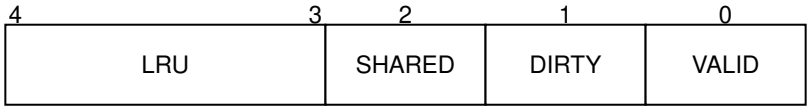


Figure 4.4: CHI cache control bits

waiting to be accepted by the NoC. In this cache implementation, read misses will wait until the write queue is empty before propagating (just like the write-through cache implementation), the same reasoning as discussed in section 4.1.1 applies to this case as well. In the write-back cache, write operations are only performed when a write-back condition has been met. This means the cache will wait for write-backs before propagating reads. Since there can only be one write-back at a time, the queue size is in practice always one.

Added to the `bk_in` and `bk_out` signals of the cache module, seen in figure 4.3, is also a cache line state two-bit signal, `cl_status`. Table 4.2 shows the encoding of each state.

State	cl_status
I	00
SC	01
UC	10
UD	11

Table 4.2: Bit encodings for the cache line status

The bk_in and bk_out signals both include a cl_status signal for regular read and write requests and a cl_snp_status signal for snoop requests. On write-backs the bk_out.cl_status tells the HN what state the cache line was in before propagating it. On read propagates the bk_out.cl_status describes to the HN the preferred cache line status when receiving the data. On a received data the bk_in.cl_status expresses the state of the received cache line. On a received snoop, the input bk_in.cl_snp_status states the changes necessary to a cache line if present in the cache. And on a snoop response the bk_out.cl_snp_status states the cache lines status after the snoop was handled. If a snoop makes it impossible for a cache line to stay in the dirty state it must be propagated. Signals for transferring a dirty cache line to the snoop response was therefore also added to bk_in and bk_out (a data signal group and a propagate valid signal).

Supporting write-allocate in this cache version is implemented by making the state machine go into reading a cache line on a write miss, and when a response is received, write to that new cache line locally in the cache. It is however important that the cache line is not present in another cache before writing to it, since this can violate the coherency rules if the cache line stays dirty while other caches are reading their version of the cache line. Hence, the propagated read needs to ask the HN for a cl_snp_status of a Unique state (10 or 11) so that the HN via snoops invalidates the cache line present in other caches.

Upon receiving a snoop the cache checks for conflicts with currently propagating requests. If there are no conflicts, the snoop algorithms checks the entries of the cache and changes the cache line status bits accordingly. On a conflict the snoop request is saved until the conflicting currently propagating requests have been supplied.

4.2.2 NoC

The NoC used in this project is a 2D mesh interconnect topology, seen in figure 4.3 as interconnect_top, with the number of nodes configurable. The design made for the evaluation system has different configurations which can be seen in figure 4.5.

The NoC has five channels: REQA, REQB, DAT, RSP and SNP. REQA is the dedicated REQ channel between the HN and all RNs, while REQB is a dedicated channel between the HN and the SN. Each flit sent on a channel includes the corresponding information shown in tables 4.3, 4.4, 4.5 or 4.6.

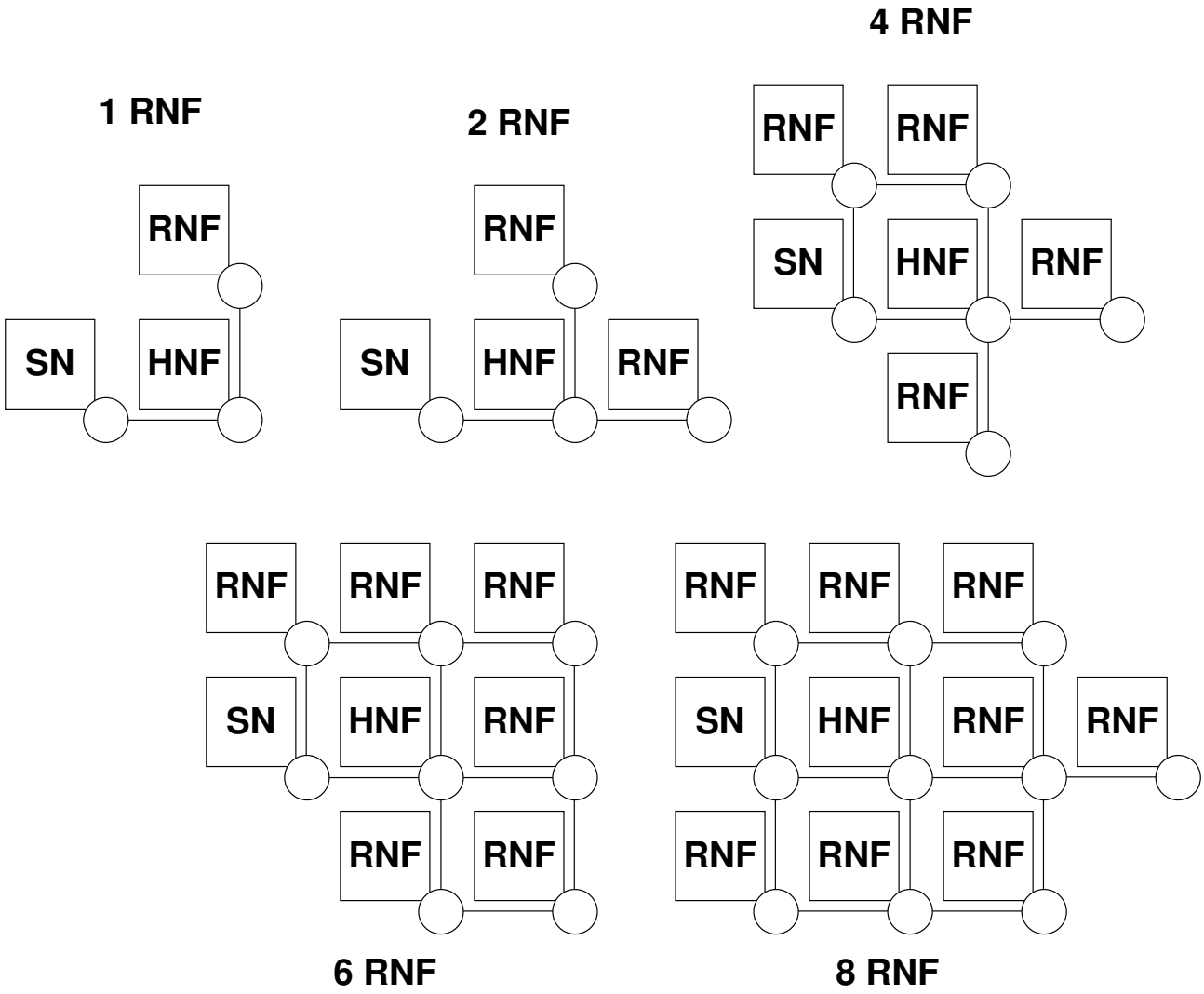


Figure 4.5: NoC nodes setups.

REQ Flit Field	Description
QoS[3:0]	Quality of Service.
TgtID[5:0]	Target ID
SrcID[5:0]	Source ID
TxnID[11:0]	Transaction ID
ReturnNID[5:0]	Return Node ID
ReturnTxnID[11:0]	Return Transaction ID
OpCode[6:0]	Request opcode. **REQA**: ReadShared, ReadUnique, WriteBackFull. **REQB**: ReadNoSnp, WriteNoSnp
Size[2:0]	Size of Read/Write data
Addr[47:0]	Memory address of requested data
SnpAttr	Snoop attribute associated with the transaction. [0]: Non-snoopable (HN to SN), [1]: Snoopable (RN to HN)
ExpCompAck	Indicates if a transaction includes a CompAck response. 0: No CompAck, 1: CompAck

Table 4.3: REQ flit fields.

RSP Flit Field	Description
QoS[3:0]	Quality of Service
TgtID[5:0]	Target ID
SrcID[5:0]	Source ID
TxnID[11:0]	Transaction ID
OpCode[6:0]	Response opcode. SnpResp, CompAck, CompDBIDResp
RespErr[1:0]	Response error
Resp[2:0]	Snoop response: Final state of the snooped cache line (I, I_PD, SC, SC_PD). RD/WR response: UC, UD, SC
DBID[11:0]	Data Buffer ID from sender. Value is used for TxnID for CompAck or CopyBackWriteData from RN

Table 4.4: RSP flit fields.

DAT Flit Field	Description
QoS[3:0]	Quality of Service
TgtID[5:0]	Target ID
SrcID[5:0]	Source ID
TxnID[11:0]	Transaction ID
HomeNID[5:0]	Home Node ID.
OpCode[6:0]	Data opcode. RN to HN: CopyBackWriteData, SnpRespData. HN to RN: CompData. HN to SN: NonCopyBackWriteData
RespErr[1:0]	Response error
Resp[2:0]	Indicates snooped data was dirty. Status is final state of the snooped cache line (I_PD, SC_PD).
DBID[11:0]	Data Buffer ID from sender. Value is used for TxnID for CompAck or CopyBackWriteData from RN
BE[63:0]	Byte Enable
Data[511:0]	Data payload transported.

Table 4.5: DAT flit fields

SNP Flit Field	Description
QoS[3:0]	Quality of Service.
SrcID[5:0]	Source ID
TxnID[11:0]	Transaction ID
TgtMask[5:0]	Bit mask of target nodes
OpCode[6:0]	Snoop opcode. SnpShared, SnpUnique, SnpClean-Invalid
Addr[47:0]	Snoop address

Table 4.6: SNP flit fields

4.2.3 Request Node

The RN IP, shown in figure 4.3, used in this project is an extended version of the grchirnf developed by Gaisler for the EUMMSS project. The RN acts as an interface between the L1 cache's Generic Bus Master ports and the NoC's CHI channel ports. It will react on either a read or write request from the L1 or a snoop request from the HN. When either of these has been triggered the RN will decide what response to send to the corresponding node with the help of an internal state machine. All transactions supported between the RN and HN can be seen in figure 4.6. Each arrow represents a flit being sent on that channel. The opcode of the flit can be seen as well.

At the start of the project, this IP only supported flit opcodes to set a cache line state to Invalid (I) (one of the states from the cache state model in figure 2.10). This had to be extended as the minimum cache states needed to support coherence

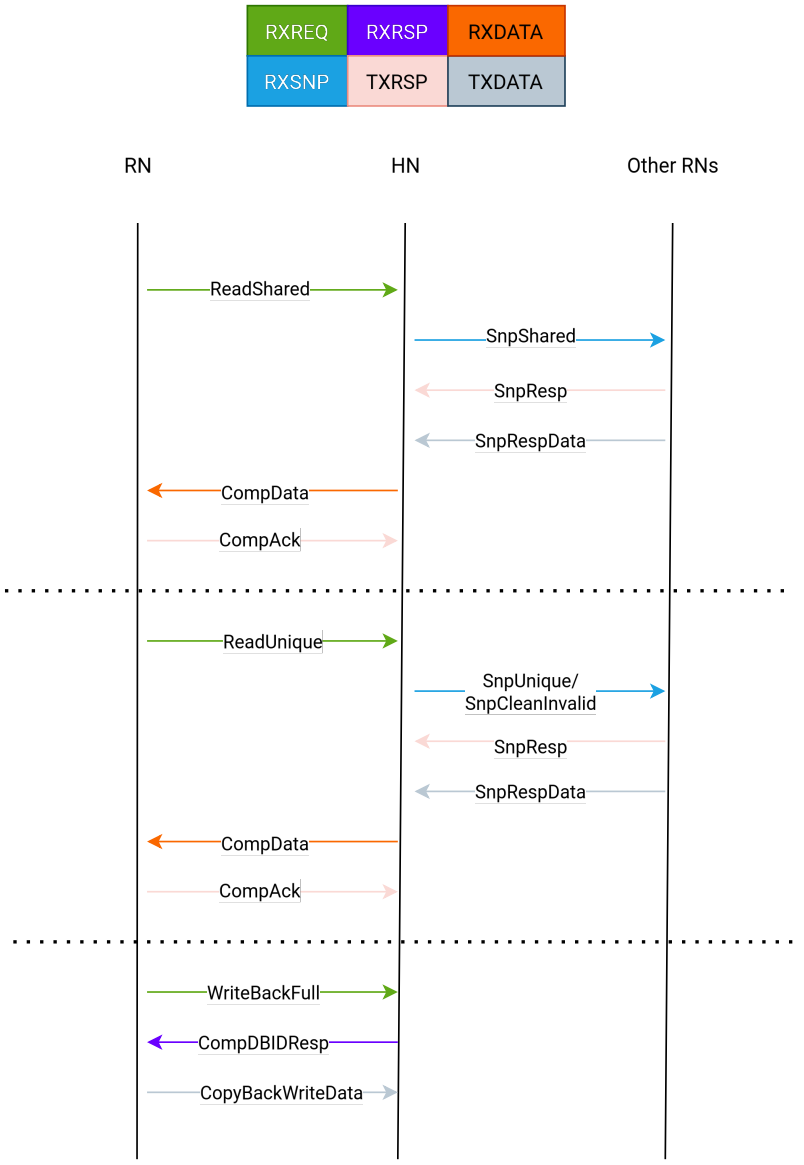


Figure 4.6: Full transactions for each CHI operation supported by the RN. Channel names are set from the RN's perspective. These scenarios assume a hit in HN and therefore the SN is excluded.

in a multi-core system are: I, Unique Clean (UC), Unique Dirty (UD) and Shared Clean (SC). Opcodes that were added in this project are: SnpShared, SnpUnique and ReadUnique. The rest were already implemented. In this subset version of the CHI protocol in which the RN support, SnpCleanInvalid and SnpUnique triggers the same action. The name of the cache state model in the opcodes (Unique and Shared) represents the cache line status that the sender requests from the receiver. A state diagram shows which opcode is used for each L1 and HN request in figure A.1.

Extension was also required for the communication between the L1 and RN regarding the cache line states. The L1 receives information on what a cache line should have as status from the RN. The RN itself receives this information from the HN through

the `Resp` field of a RSP or DAT flit. To support a multi-core system UC, UD, SC and SC_PD were added as options for this field. The SC_PD option of the `Resp` field clarifies to the HN that the cache line was dirty but has been updated to Shared. For a dirty cache line that is updated to Invalid, the already existing option I_PD is used.

4.2.4 Home Node and Subordinate Node

The HN and L2 cache, seen in figure 4.3 in the green box, are modeled with UVM and SystemC components, while the SN and RAM memory are modeled with only UVM components. These components were not developed for this research project, they were provided to us by InfiniNode Technologies.

To integrate the components with the CHI evaluation environment, the RTL side of the design instantiates CHI channel interfaces which are then connected to the UVM SN and HN Drivers through the virtual interface, also seen in figure 4.3. In the DUT these interface signals are then connected to the NoC module.

4.3 UVM Test Setup

The evaluation of the metrics defined in section 1.2 are performed through Universal Verification Methodology (UVM) [26]. This methodology was chosen partly because it is a base library in SystemVerilog and for its flexibility and scalability in test cases due to functionalities such as constrained random verification. This is done through SystemVerilog constructs that allow the tester to have a random distribution of its test stimuli with desired constraints.

UVM components relies on Transaction Level Model (TLM) verification. TLM is a modeling design style where systems and components that interact on a signal-level are represented through a higher abstraction level. The communication and lower-level activities are represented by protocol-specific data transactions.

An evaluation environment has been set up with UVM for testing both systems based on AHB and CHI. As UVM incorporate methods of object oriented programming, six custom classes have been created, each extending its own UVM class based on its use. Because the frontend of our L1 cache uses a Generic Bus Master interface, all UVM components needed to create and initate test stimuli are reusable for both AHB and CHI evaluations, without much modification. Only the monitor, the component monitoring the interconnect transactions, has significant functional differences depending on which system is being evaluated.

The main priority for the UVM environment has been to evaluate the metrics defined in this research project. Therefore there is no functional verification made simultaneously as metric evaluations are being run.

The following subsections will introduce each custom class and their role in the evaluation framework.

4.3.1 Sequence Generator

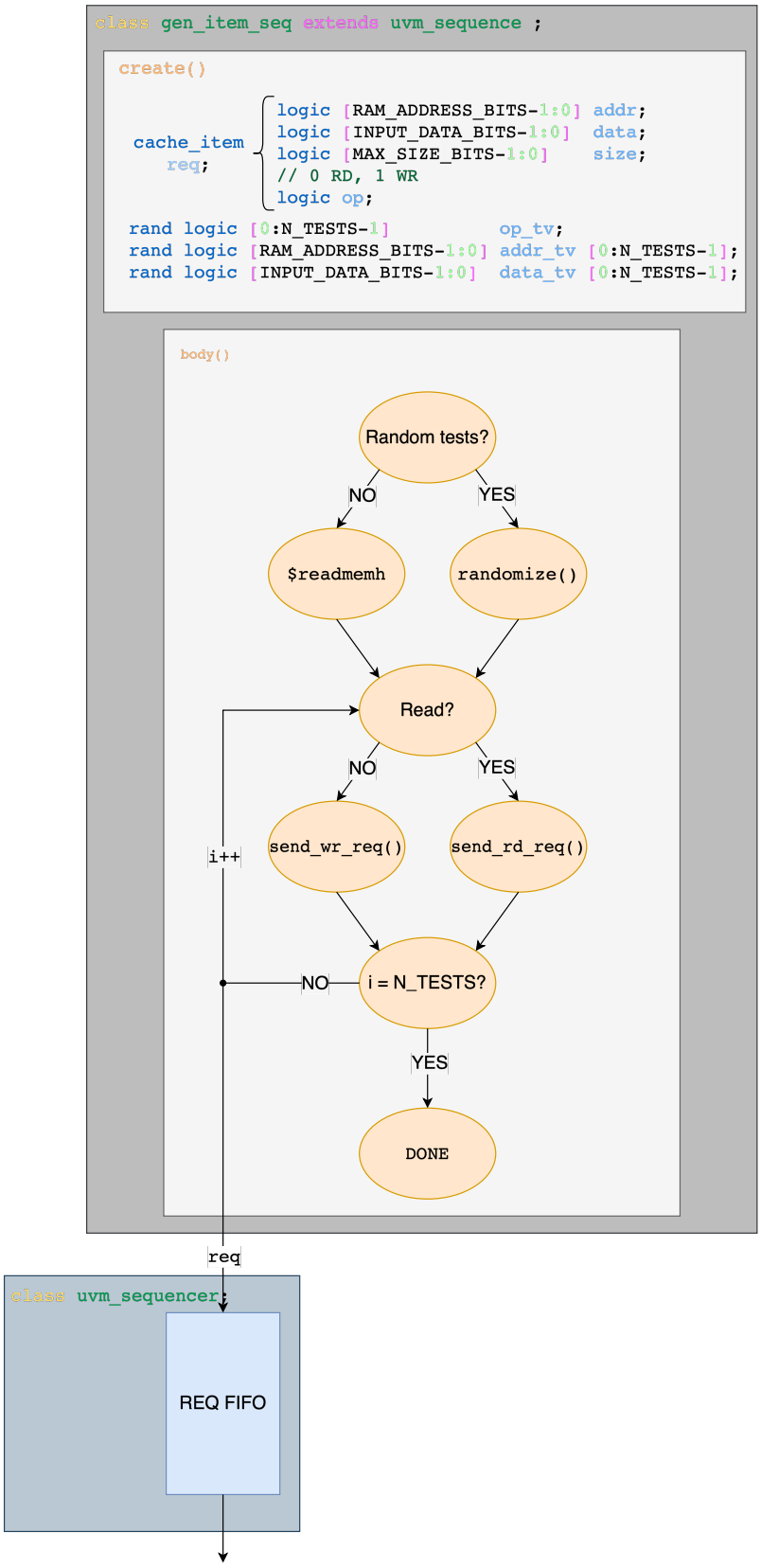


Figure 4.7: UVM Sequence Class

The Sequence Generator class `gen_item_seq` extends the `uvm_sequence` class, and its design is shown in figure 4.7. The Sequence is an object that describes the behavior of generating test stimuli. The class `gen_item_seq` contains global variables for a memory request: `addr`, `data`, `size`, `op`. The last variable holds the value of the memory operation, a logic `0` for read and a logic `1` for write. The variables are stored in an object `req`, representing the request. The object `req` is of a class `cache_item`, which extends `uvm_sequence_item`.

An object of the class `uvm_sequence_item` is needed for the functionalities of printing, copying and comparing items correct. All items will also receive a transaction ID, allowing for easier debugging and tracking of transactions [26]. Having the test stimuli item of type `uvm_sequence_item` is also needed for utilizing their TLM functionalities.

The Sequence class also contains test vectors as global variables, one each for `addr`, `data`, and `op`. The test vectors can be configured with two parameters: `RAND_TEST` and `N_TESTS`. These parameters are globally available for all UVM components.

The behavior of `gen_item_seq` is as follows. If `RAND_TEST` is set, the test vectors will be randomly generated with a length of `N_TESTS`. If `RAND_TEST` is not set, the test vectors will be read from locally stored files with the `$readmemh` function. The sequence will then go through all elements in `op_tv`, the test vector for memory access operation. Depending on if a read or write access request should be generated, the sequence will set the corresponding parameters of the object `req`.

All iterations of the object `req` will be put on a FIFO, sending the request information to the Driver. The FIFO itself is instantiated by a `uvm_sequencer` class. Each Sequence is assigned its own Sequencer object.

When all test items have been generated the Sequence will stop its run and let the UVM testbench know that it has completed.

4.3.2 DUT Driver

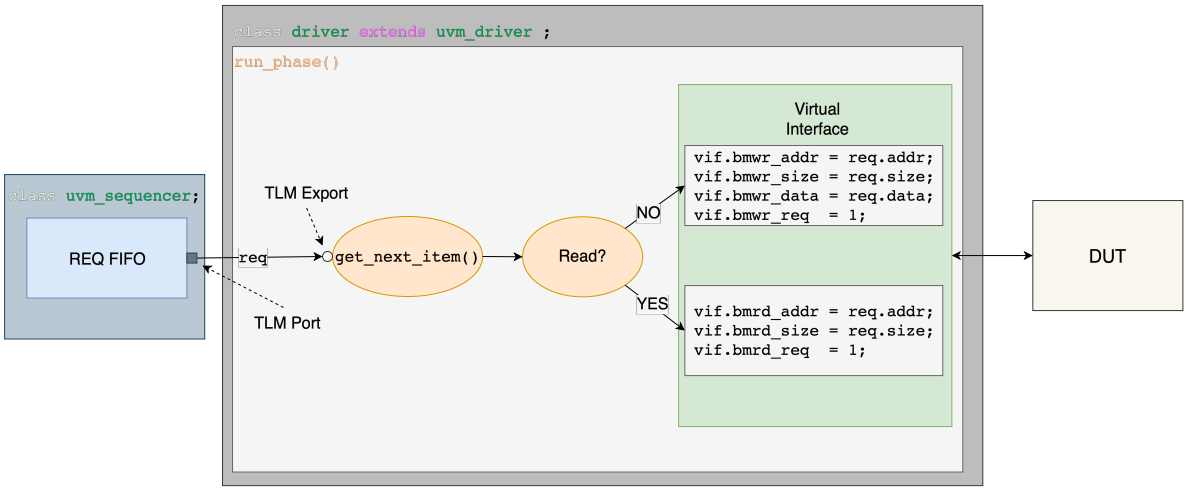


Figure 4.8: UVM Driver class

The Driver class `driver` extends the `uvm_driver` class, and its design can be seen in figure 4.8. The Driver's role is to drive the TLM values received from the Sequence as signals for the DUT's input port. It has access to a virtual interface instantiation from the testbench, which is directly connected to the DUT's inputs and outputs.

The Driver is connected to a Sequencer FIFO and will either continuously grab the first item in a `forever` loop, or it will be given a latency before grabbing the next item. The latency time can be configured, and its purpose is to simulate different request frequencies.

The Driver has to have the knowledge of how a transaction should occur according to the protocol on the interface, as it acts as the other side of the protocol transaction. In this case the protocol is the Generic Bus Master Interface from GRLIB. For this protocol the `bmrd_req` and `bmwr_req` signals has to stay high on a request until they are granted access by the corresponding `bmrd_req_granted` or `bmwr_req_granted`.

As the Driver only drives the input signals to the DUT, and does not perform any other evaluation, it does not need to signal to the testbench that it is done. It will finish with the simulation finish.

4.3.3 Interconnect Monitor

The Monitor class `monitor` extends the `uvm_monitor` class. It is the only class in the built UVM framework that has small functional differences for the AHB versus CHI evaluation. Just as the Drivers, the Monitor has access to the virtual interface. The virtual interface includes both the frontend of the L1 cache and the respective interconnect's interface signals. The connections for the virtual interface can be seen in figure 4.1 and 4.3, where the current interconnect signals are circled with the dotted gray lines.

The Monitor will, for the duration of the UVM simulation, monitor the AHB bus signals for all masters in the AHB environment. For the CHI environment, it will instead monitor the channel signals of the RNs and HNs.

In the AHB environment the Monitor has its own mechanism for keeping track of which master has been granted access on the bus, through their grant IDs. With this ID it can count how long a master is waiting on the bus. Wait scenarios are for being granted access on the bus and for the memory operations in either the L2 cache or RAM.

For the CHI environment, the Monitor needs to keep track of source, target, and transaction IDs of each transaction being made. All RNs receive their response to their own port, compared with the AHB system where the response signal is shared by all masters. This simplifies the evaluation method for the Monitor.

The monitor can be configured to count memory access times from different starting and end points. Starting points can be for example when a memory request is done on the L1's frontend or when the request is put on the interconnect.

4.3.4 Testbench

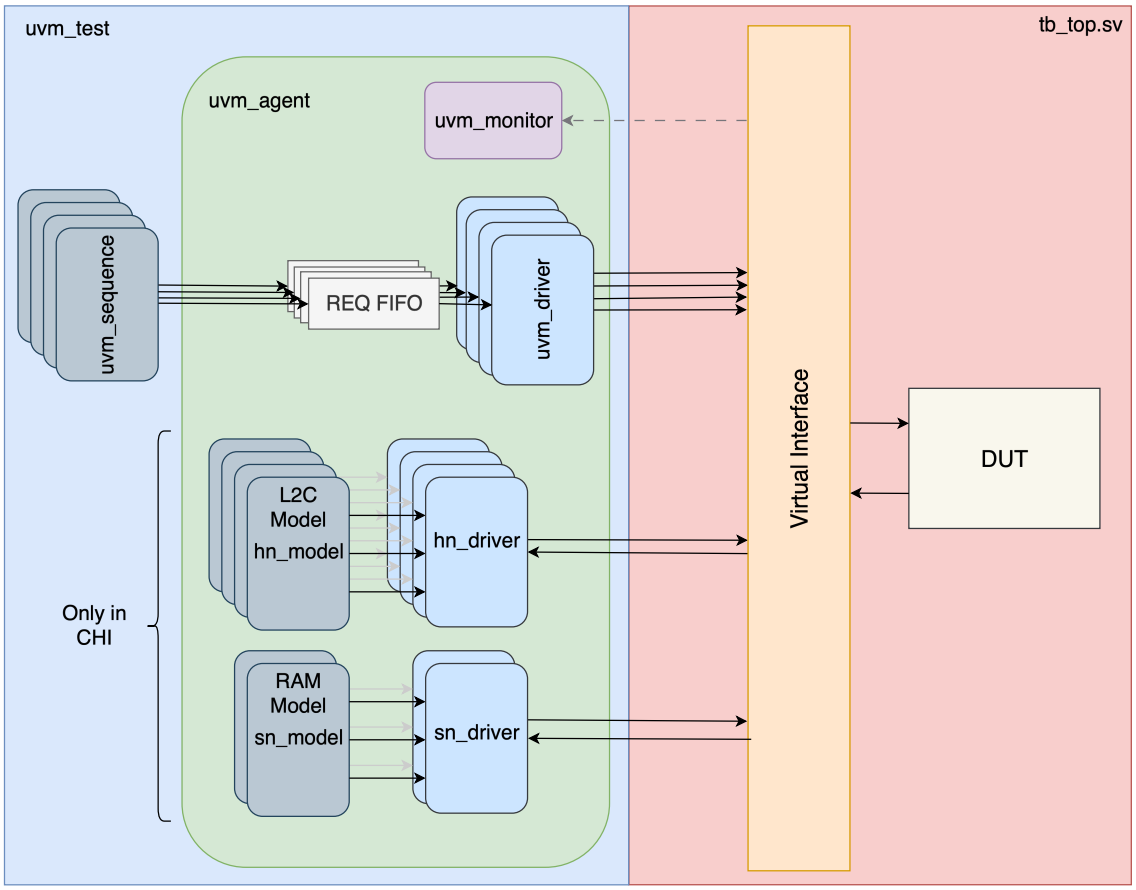


Figure 4.9: UVM Testbench class

The Testbench class `testbench` extends the `uvm_test`. The Testbench acts as the environment for the whole UVM simulation. It is the topmost component in the UVM hierarchy and it is represented by the blue box in figure 4.9. As the `testbench` primarily instantiates and connect components with the virtual interface, the figure is showing a high level view of the class.

The `testbench` class is created from a SystemVerilog simulation file `tb_top.sv`. This file is represented by the red box in the above figure. The DUT in `tb_top.sv` is either the topmost AHB or CHI system. Through a UVM database it receives the instantiated signals of the virtual interface. It will thereafter pass the required signals to each respective components.

It will also instantiate a UVM Agent class `agent` and Sequnces of `gen_item_seq`. The Agent will instantiate and connect all other UVM components and their TLM connections.

The Testbench has to keep track of all components that are performing action directly effecting the evaluation. These components are the Sequences and the Monitor. Each of them will send a done flag to the Testbench when they have finished their operation for all test cases. The Testbench will finish its simulation execution when all flags are set.

It is in the Testbench that designers can create new Sequences if one would like to evaluate other test cases than memory access time.

4.4 Test Input Data

To evaluate the test suites two memory access traces were composed. One targeting specific memory access patterns via targeted memory address requests, and one utilizing benchmarking software to create a real-world memory access trace. The targeted memory addresses were chosen to calculate the interconnect latencies of different specific scenarios, while the software was taken from the OBPMark benchmark open-source repository [27].

4.4.1 Targeted Memory Accesses

Targeted memory accesses were used to run and calculate memory latency for specific cache hit/miss scenarios. The following four scenarios were important: L2 cache hit, L2 cache miss, and for CHI specifically; L2 cache hit with snoop (to change cache line state of other caches to either Unique or Shared), and L2 cache miss with write-back (if the cache line is not present in L2, and upon being received by the memory it will replace a dirty entry in the L2). These four presented scenarios are not emphasized as different types of memory requests, like Read, Write, ReadShared, WriteBackFull. This is because the memory requests scenarios can all be categories under the four presented scenarios from a memory latency perspective.

Firstly, to achieve an L2 cache miss, one simply needs to change the address on every request. A new address will always miss in L1 and in L2.

To achieve an L2 hit, if the L1 does not have a lower associativity than the L2, which was the case in this study, multiple L1 caches (at least 3) have to be used. Sending the same address request to all L1 caches will first yield misses in the L1s. One will miss in the L2 and the rest will hit. The first hit in the L2 will have to change the state of the first request from Unique to Shared, but the third request for the same cache line is already in the Shared state and hence can return the data immediately; a cache hit.

Notice that the L2 cache hit with snoop was found on the second request to the same address. The last scenario is a bit trickier, but can be found if by issuing multiple write requests, from two or more caches, to addresses with the same index and offset bits, but different tag bits. On the first round of requests, the first L1 cache will issue a ReadUnique, get the cache line and write to it locally. The second cache will then issue a ReadUnique, the HN will issue a SnpUnique to retrieve the now dirty cache line from the first L1, and then give the cache line to the second L1. If now the first L1 cache issues a request for an address with a different tag but the same index as the first request, and the associativity of the L2 is set to 1, the L2 has to replace its dirty cache line with one from memory, issue a snoop to the second L1 cache to invalidate its dirty cache line, and write-back its own dirty cache line to memory when memory gives the new cache line. And that is how targeted memory accesses was able to generate L2 cache miss with write-back.

4.4.2 OBPMark Benchmarks

The other set of tests were done by utilizing performance benchmarking C code, developed specifically for spacecraft on-board data processing applications, OBPMark [28]. This application was chosen to get a realistic scenario of an application running on the coherent systems. There has been a lack of standardization regarding benchmarking space applications. OBPMark and its use case in this project could contribute towards a standardized performance analysis.

To facilitate software tests on the systems, either a processor was needed to run the full software, or the software had to be turned into a memory trace, since without a processor only memory accesses can be simulated on the systems. Since there is no processor in the systems the second option was chosen. When turning software into memory traces, picking memory instructions directly from software is not possible. This is due to jump instructions in the code (facilitating subrutines/functions). Questions arised like what function runs where, how often, and in what order. Instead, turning software into memory traces needed to be done by simulating the software on hardware and picking out the memory accesses whilst it is running. This process is problematic for three reasons: The memory trace output is dependent on the hardware the software is simulated on, if the software is multithreaded a threading system is needed, and if a booting system/operating system is used, picking what memory accesses belongs, or do not belong, to the simulated software is difficult.

The first problem was tackled by simulating the software on a Quick Emulator (QEMU) instance emulating Gaisler's NOEL-V system. There is an argument for not using Gaisler's hardware since it might clutter the result for strictly research purposes, but since Gaisler are the stakeholders of the project, and since the research question includes "for space application", how this problem was tackled is arguably bringing more relevancy to the results, rather than less.

Moving on to the second problem, multithreading is necessary to simulate the software over multiple cores/cache instances, which is present in the systems of the project. OBPMark does include several compilation options for multithreading alternatives (OpenMP, OpenCL, CUDA etc.). Since these libraries are mainly created for GPU applications Gaisler's bare metal NOEL-V compiler does not support them out of the box. Two options then remained relevant: Gaisler's RTEMS for NOEL-V, and Gaisler's Linux for NOEL-V. At first glance the RTEMS seemed like a better option, due to it not starting unnecessary background threads (less likely to have irrelevant memory access clutter the memory trace). However support for the multithreading libraries were not present in either NOEL-V RTEMS or the NOEL-V Linux compiler. Since editing and compiling compilers to support these threading libraries is very much out of scope for the project at hand, an alternative route was chosen: to write a threaded version of the OBPMark (based on the other multithreaded versions of OBPMark) with the Linux `pthreads.h` library. This was chosen over RTEMS task system for the reason of having earlier experience with `pthreads.h` and not RTEMS task system. The library was used to first create a threadpool, where the threadpool size was chosen to 10 (to cover 1-8 cores), and the task queue size was chosen to 13 (to cover all threads).

In order to run a Linux application it is necessary to have a kernel to run it on. The source code for the kernel was given by Gaisler's own Linux system and compiled with the buildroot system. Once compiled the `initd` program was replaced with the test software; this to minimize the unnecessary background tasks and hence minimize the amount of irrelevant memory accesses.

Continuing on to the third problem, QEMU has the benefit of external plugins, which can be programmed by users themselves. Among other things QEMU plugins provide a way to monitor and log hardware events of the simulated hardware instance. A plugin to monitor and log memory access events was therefore created. When simulating the hardware the QEMU option `"-smp"` was used to change the number of simulated cores between 1-4. Each QEMU event in the system was automatically connected to a core number from a plugin's perspective, and so the separation of memory accesses between cores could easily be derived from the plugin itself. To separate the memory accesses of the main OBPMark test from kernel initialization code, and OBPMark preparation and wrap-up code, the OBPMark software was edited to include two memory accesses to a separate constant address, which was irrelevant to the test; one access before the main test, and one after. The constant memory address was then checked via the `readelf binutils` binary, which was present via Gaisler's Linux kernel framework, and put as the starting and the ending condition for the monitoring of the memory accesses in the QEMU plugin. The address was also checked to be accessed exactly twice every simulation.

To finalize, the OBPMark test simulated was 1.1-image, with the settings: width-1024, height-1024, frames-1.

5

Results

This chapter presents the results which have been produced by running the UVM simulations, with both target request generation and OBPMark stimuli. The metrics defined in section 1.2 have been the basis for the tests. These metrics are:

- Memory access time
- Interconnect bandwidth
- Coherence protocol overhead

The following sections present the results for each metric for the AHB and CHI systems, respectively. All sections conclude with a comparison of the results of the two systems. Simulation time is only measured for requests in the RTL components.

For a memory access request, the measurement, if not mentioned otherwise, starts when the request is injected at the frontend to the L1. The request is considered complete when the L1 outputs its response on its frontend. When considering snoop in CHI, it is instead started when the HN outputs the snoop requests, and ends when it receives the last snoop response.

Unless specified the tests for memory access time and bandwidth were done with target memory access tests.

5.1 Memory Access Time

All evaluation tests were performed with this metric in mind as described in section 4.3.3. The goal was to see how much time is spent on accessing the memory through the different interconnects.

AHB Evaluation

In a single-core system, the AHB bus grants the master immediate access. The master is never required to wait any additional time after making its request, which is not the case in a multi-core system.

For a multi-core system, if a master makes a request on the bus while another master has already been granted, the requesting master would have to wait for an additional duration of time, until itself receives the grant from the bus.

For example, in a double-core system, two masters 1 and 2 both make a memory request on the bus. The AHB arbiter has a round-robin selection process so if both

masters have their request made on the same clock cycle the bus will grant master 1 the first access. Master 2 has to first wait for the completion of master 1's request, which has been measured to either 14+δ (L2 miss) or 5 (L2 hit) cycles, where δ is the main memory's access time. It then has to wait for the completion of its own request. This represents the maximum time a master has to wait if the bus is occupied by another master's request, and this behavior is linear with the number of cores.

So the maximum time for any master to wait in a fully congested multi-core system with N cores is: $N \cdot (14 + \delta)$ (all misses), and the minimum is $N \cdot 5$ (all hits). These functions are shown in figure 5.1 where tests for a fully congested bus were run, meaning that all caches were trying to occupy the bus as fast as possible for the whole duration of the simulation.

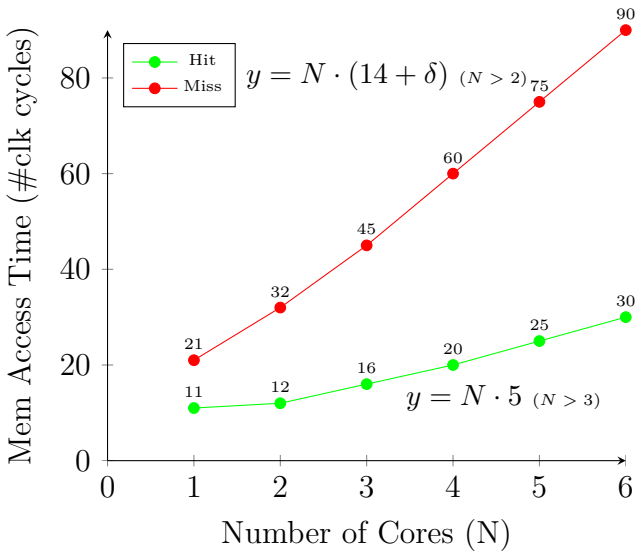


Figure 5.1: AHB Memory Access Time per Number of Cores. The plotted values are for $\delta = 1$.

The numbers 14 and 5 comes from the time spent in the bottleneck component of the AHB system; the L2. This is presented in figure 5.2. Note that the AHB bus handles request in a pipelined structure, so the same request can have its address already processed by the L2 while its data is being handled by the frontend AHB. For example, while one request is occupying the bus, another is being processed by the L2 cache, and another is being processed in the L1. Essentially when all L1 caches are trying to send requests the bottleneck blocking requests from propagating further in the system will be the L2 cache. From figure 5.2 we see that the bottleneck (L2 cache) takes 5 clock cycles for a hit, and 12+2+δ clock cycles for a miss; meaning requests propagate every 5 clock cycles on hits, and $14 + \delta$ clock cycles on misses. This means the system completes one request every 5 clock cycles on average for all hits, and one request every $14 + \delta$ for all misses. And each individual cache completes a request every $5 \cdot N$ clock cycles for all hits, and every $(14 + \delta) \cdot N$ clock cycles for all misses.

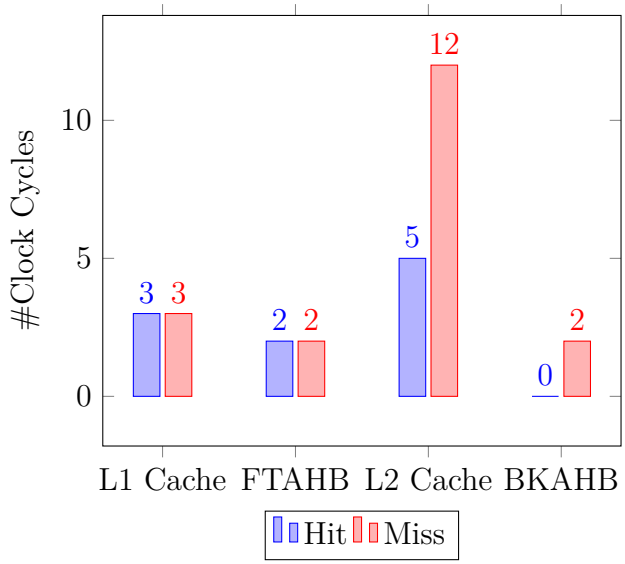


Figure 5.2: Cycles spent between components in the AHB system. Note that FT refers to frontend and BK as backend.

CHI Evaluation

The CHI system does not require a request to wait to be accepted by the interconnect. All RNs will immediately have their requests processed by the NoC. Since the CHI system has a single HN, only capable of processing one request at a time, at the NoC port to the HN, all packets are serialized.

The number of clock cycles measured for each component in the CHI system is presented in figure 5.3.

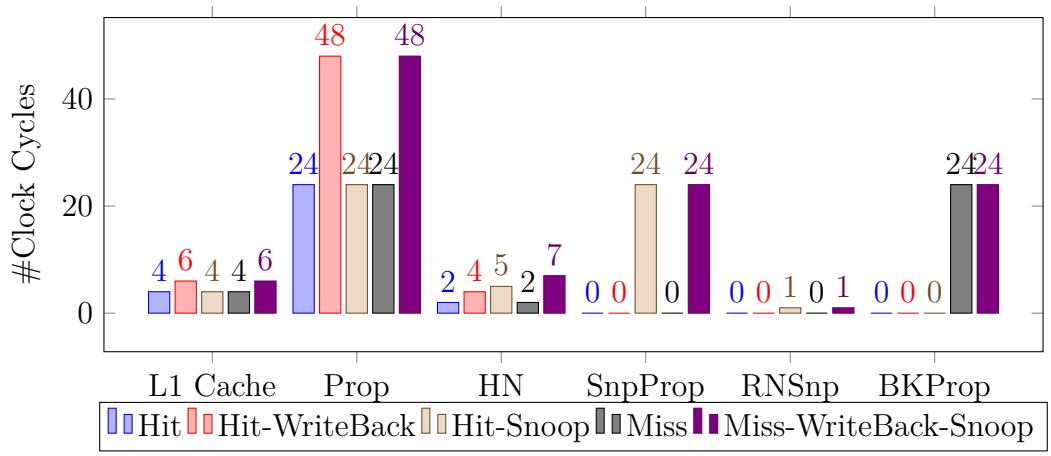


Figure 5.3: Cycles spent between components in the CHI system. The write-back operations refer to the L1 write-backs.

Figure 5.4 shows the propagation delay of the NoC for a request hit in the HN.

With the CHI system's setup of one HN and one SN, and in which the RN, and the HN both handle one request at a time, we can assume that with enough RNs congesting the NoC we will arrive at a linear function just as with the AHB system.

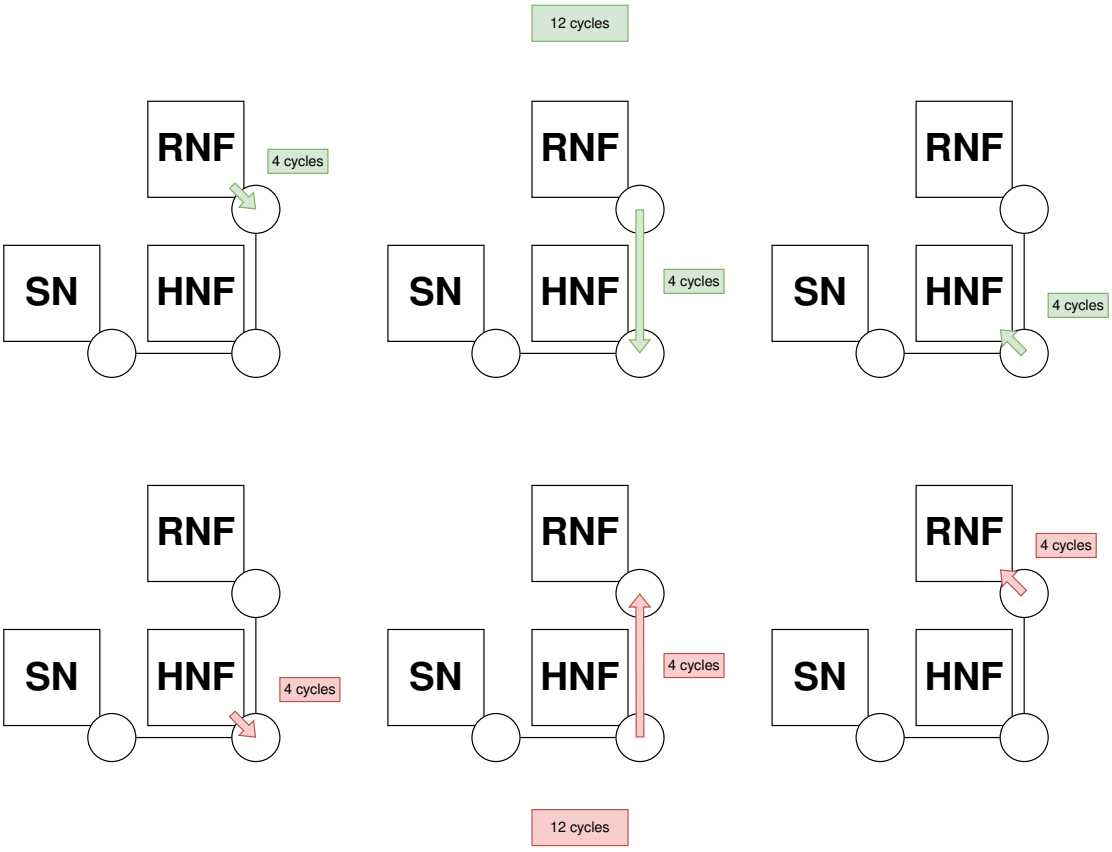


Figure 5.4: NoC propagation of an RN request hit in the HN

The function describes the average latency of a single memory access, for each individual core, depending on the number of cores (N) accessing the NoC. The function consists of three multiplied factors which will be explained below.

Table 5.1 describes the relevant parameters of the linear function.

Parameter	Description
$T_{cache2HN}$	Average L1 to HN Memory Request Round-Trip Time
δ	Main Memory Access Time
T_{HN2SN}	Round-Trip Time of the HN's Request to the SN
α	Total delay of an L2 miss
M	Miss Rate
WB	Write-Back rate
T_{SNP}	Average Maximum Snoop Round-Trip Time
SNP	Snoop Rate
γ	Total delay of a snoop

Table 5.1: Parameters of the CHI Memory Access Time linear function

The linear function is as follows:

$$T_{mem}(N) = N \cdot (4 + T_{cache2HN} + \alpha \cdot M + \gamma \cdot SNP) \cdot (1 + WB)$$

where

$$\alpha = T_{HN2SN} + \delta$$

$$\gamma = T_{SNP} + 2$$

The first factor N is for the number of cores in the system. As the memory access time for each core increases with the number of cores, the total delay will be proportional to N.

The second factor, $(4 + T_{cache2HN} + \alpha \cdot M + \gamma \cdot SNP)$, is a weighted average of each scenario that could occur in the NoC during a memory request, with different amounts of delays added. The scenarios included are an HN access, an L2 miss, and snoops. Each memory request will generate an HN access. The additional delay of 4 cycles is seen from the communication protocol of a CHI transaction between the requesting RN and the HN, shown in figure 4.6. It is the minimum time the HN spends processing a request before accepting the next request. The total delay of a snoop includes an additional delay of 2 clock cycles for processing the snoop CHI transaction seen in figure 4.6.

The third factor, $(1 + WB)$, is the additional delay in case a write-back occurs in the L1. This delay will be the same as the second factor as a read request (either ReadShared or ReadUnique) will have to be performed according to the CHI protocol.

Comparison

As can be seen from the two previous sections, both the AHB and CHI systems have a memory access time that increases linearly to the amount of cores in the system. An overview of the measured values for both systems can be seen in figure 5.5. These values representing the average memory access time was retrieved from simulation runs with OBPMark memory traces. 5000 instructions, consisting only of load and stores, were run per core.

The AHB performs linearly, meaning each additional core adds a constant amount of time to each memory access. The reason is that the L2 slave of the AHB bus cannot handle requests in parallel. The CHI system also performs linearly since it is not able to utilize the NoC's parallel data paths; because the HN and RN components used in the project are not able to handle requests in parallel.

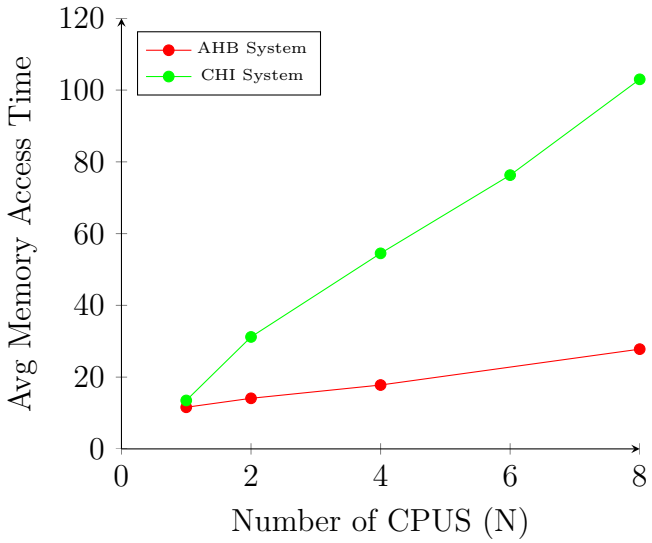


Figure 5.5: Memory access time latency, per core, depending on the number of cores in the system

It is clear from figure 5.5 that the CHI system has a longer memory access time than the AHB system in this project. In figure 5.4 it was shown that the propagation delay of a request from the RN to the HN was 12 cycles, and an additional 12 cycles for the HN to send the data back on a hit. For the AHB evaluation, the same scenario had a total of 2 cycles latency (5 cycles if the L2 was occupied). This shows that for a simple system, with a single core, the additional overhead of a NoC only adds to the latency without any benefits. In that case a simple AHB system performs better.

5.2 Interconnect Bandwidth

The bandwidth in our evaluation is the throughput of the whole system. It describes how many memory requests the interconnect can handle per clock cycle.

The average bandwidth of the system is defined as the average request per clock cycle. For a multi-core system this is calculated as:

$$BW = \frac{TotalRequests}{\#cycles} = \frac{1}{\frac{\#cycles}{TotalRequests}} = \frac{N}{\frac{\#cycles}{TotalRequests} \cdot N} = \frac{N}{T_{mem}}$$

where T_{mem} is the average memory access time per request, for each individual core, the same average from the previous section.

The following evaluations for AHB and CHI have been done from the same simulations performed for the previous section.

AHB Evaluation

Taking the two memory access times, presented in section 5.1, and the hit and miss rates, we get the average memory access time per request for the AHB system:

$$T_{mem\_hit} = 5 \cdot N \cdot (1 - M)$$

$$T_{mem\_miss} = (14 + \delta) \cdot N \cdot M$$

where $(1 - M)$ is the hit rate and M is the miss rate. With this, a model of the bandwidth is given as:

$$BW_{AHB} = \frac{N}{N \cdot (5 \cdot (1 - M) + (14 + \delta) \cdot M)} = \frac{1}{5 \cdot (1 - M) + (14 + \delta) \cdot M}$$

CHI Evaluation

Since the CHI system has a single point of coherence (a blocking HN), it cannot exploit parallelism. At the NoC port to the HN, all packets are serialized. The L1 cache and RN are also blocking, so a simple equation that considers all requests to be served sequentially is sufficient in this case to model the system bandwidth. The following equation models the CHI system's bandwidth:

$$BW_{CHI} = \frac{1}{(4 + T_{cache2HN} + \alpha \cdot M + \gamma \cdot SNP) \cdot (1 + WB)}$$

Comparison

A comparative graph between the two systems' calculated bandwidth can be seen in figure 5.6. These measurements were taken with the OBPMark test runs, with 5000 memory requests generated per core.

The main reason for why the CHI system has a lower bandwidth than AHB is the fact that the HN can only handle a single request at a time and that the RNs are

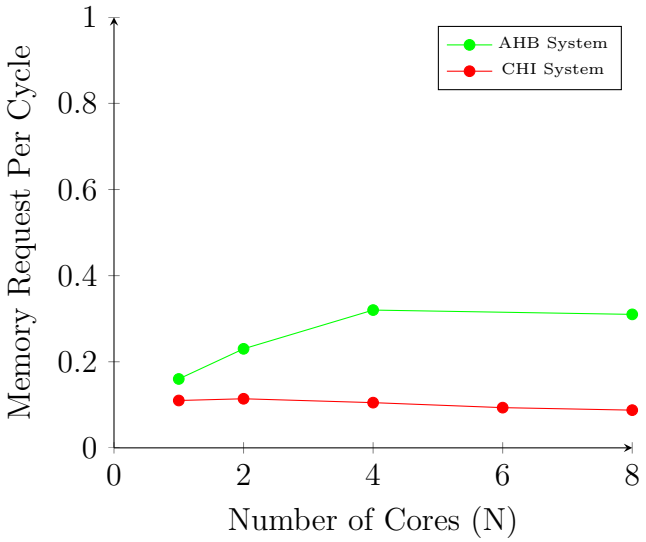


Figure 5.6: System throughput, depending on number of cores in the system

thereby blocking during most transactions. The system is not able to utilize the separated data channels of the NoC for parallelism. As the bandwidth is the inverse of latency, the same arguments made in section 5.1 can be made for the system throughput.

With a higher number of cores the bandwidth of the CHI system drops, as the latency is increasing linearly. The reason behind the decrease in bandwidth is longer distances. As more nodes are added to the NoC, the average length to the HN increases, which is visualized in figure 4.5. However, when looking at the AHB graph in figure 5.6, there is an increase in the bandwidth when going from one core up to four. This is because before four cores, there is not enough L1 misses in the memory trace to congest the AHB interconnect fully. At four cores the AHB bus has become fully congested.

Just as for the latency, an increase in snoop requests for CHI also affects the bandwidth with the current set up. There was an even higher increase of snoop requests when running the OBPMark stimuli tests for eight cores, compared to four cores, where one half of the cores had duplicated memory traces of the other half. This adds to the lowered bandwidth of the CHI system, which will be further discussed in section 5.3. Note that this would not be the case for real world scenarios, but could be seen in these types of simulations.

5.3 Coherence Protocol Overhead

The overhead of the snooping protocol in the AHB system is an additional L1 miss. As the CHI protocol introduces specific snoop requests, the latency of these transactions causes an overhead. An overview of the snoop requests will be presented for the CHI system before going into the comparison.

All results presented in this section are produced from simulations runs with test

input generated from OBPMark. 5000 memory requests are generated for each core. Simulations for this metric was ran for 1, 2, 4, 6 and 8 number of cores. The memory traces have been extracted from the QEMU plugin as described in section 4.4.2. When running simulations with 1, 2 and 4 number of cores, the memory traces from the QEMU cores were directly mapped to the cores in our evaluation. For the simulation with 6 number of cores, the memory traces of QEMU cores 1-4 were mapped to core 1-4 of our evaluation, and core 5 and 6 had their memory trace mapped from QEMU cores 1 and 2. For a simulation run with 8 cores the memory trace mapping were duplicated, the QEMU cores 1-4 were mapped to both halves of cores in our evaluation; 1-4 and 5-8. This mapping for over 4 cores was done because the maximum number of cores in the QEMU simulation is 4.

CHI Evaluation

For the CHI protocol overhead evaluations, only time for snoop requests was measured. The duration of a snoop operation starts when the HN has generated its request. The total time is measured for all RNs receiving the snoop request. The operation is finished when the last RN acknowledges its completion.

When running the OBPMark test, the amount of snoop requests generated in CHI, for systems utilizing up to eight cores, was extracted and is presented in table 5.2.

#Cores	1	2	4	6	8
Generated Snoop Requests	74	383	1482	3126	4832
Cycles Spent on Snoop Requests	1924	9962	42532	9666	154033
Test Total cycles	45362	87725	190801	321053	456644

Table 5.2: Snoops overhead during OBPMark software tests.

The reason for such an increase in number of snoop requests is the fact that multiple cores are running the same program with duplicated memory traces. As many address ranges overlap, the more coherent cores the system has, the more snoop requests it generates. We believe this is one reason why there is a reduction of the throughput for the CHI system presented in the previous section.

The ratio of total number snoop requests being generated to the total amount of instructions being run is shown in figure 5.7. The total number of instructions run is equal the number of memory requests, 5000 per core. As we can see the percentage is also increasing when increasing the amount of cores in the system being evaluated, while the number of instructions remain unchanged.

We can therefore see that the CHI protocol has a significant amount of additional transactions being made over the NoC.

Comparison

It's not only the way the coherence protocol generates snoop request that has an effect on the performance, but the systems' write policies, which can be controlled by the protocol, also make a difference. To have coherency within an AHB system,

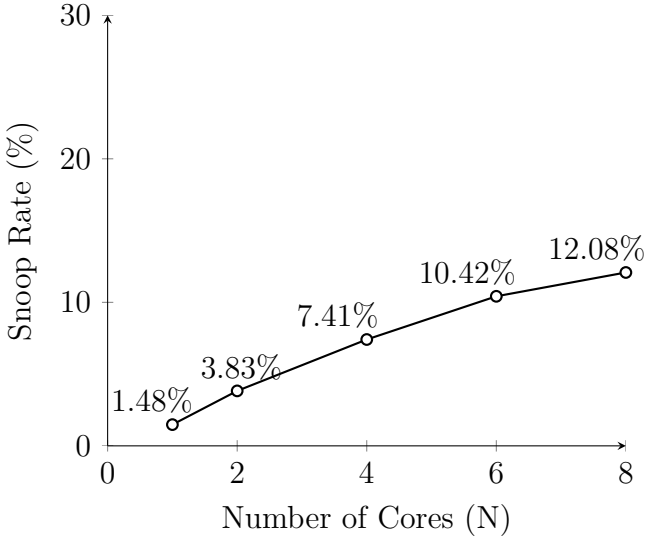


Figure 5.7: Ratio of Snoop Requests to Total Instructions

the coherent caches have to have a write-through policy, always writing new data to the next level cache. If they instead would have a write-back policy, only writing data to the next level cache when the data has been deemed dirty, it would be impossible for all L1 caches to share all memory content.

In the AHB system we are restricted to only using write-through. In the CHI protocol we can utilize either write-through or write-back, as the directory manages the coherency. To showcase how the write policy has an effect on the systems' throughput, we will take a look at the frequency of requests being generated towards the L2 cache in both systems.

There are two reasons for why a memory request would be generated to the L2, either if a read or a write request in the L1 has generated a miss. For the read misses we can see in figure 5.8 that the green graphs are similar for both AHB and CHI, their rate values as well as their behavior for the increase in cores. CHI, however, can still be seen with a lower rate even in this scenario. This is because the L1 in CHI is write-allocate. It allocates cache lines that have been written to, which means if there is later a read request on that cache line, it will return as a hit. For the AHB system, the L1 cache is write-no-allocate. Therefore the AHB system will have misses in this same scenario, as the cache lines were never allocated in the L1.

For write misses we see a much bigger difference. AHB has in this case a 100% rate of write requests generating a request to the L2 cache. This is expected because of its write-through policy. While for CHI the rate is significantly lower because of our utilization of write-back.

The total rate, given by the blue plots, is in favor for the CHI system as fewer transactions propagated out to the interconnect leads to less overhead on the overall system. The difference in the total rate between AHB and CHI is explained by their write policies, given that they have a similar rate for read misses. This showcases the improvements possible with the availability of changing to a system supporting

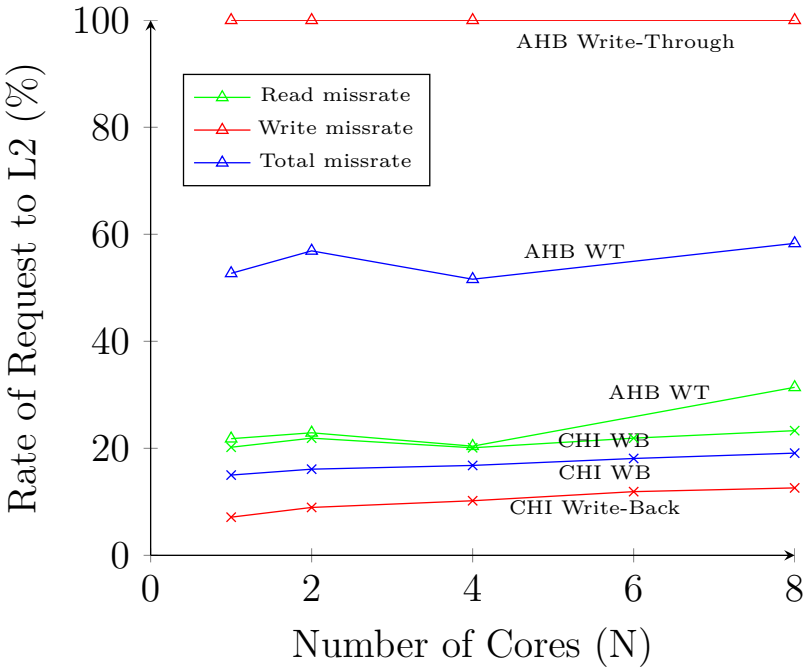


Figure 5.8: Rate of when a request is generated towards the L2, compared to total instructions (all instructions being memory requests).

a write-back policy.

The access time of requests has also been measured from the moment that a request has been accepted by the bus and NoC (no waiting time included). These can be seen in figure 5.9 for AHB and figure 5.10 for CHI.

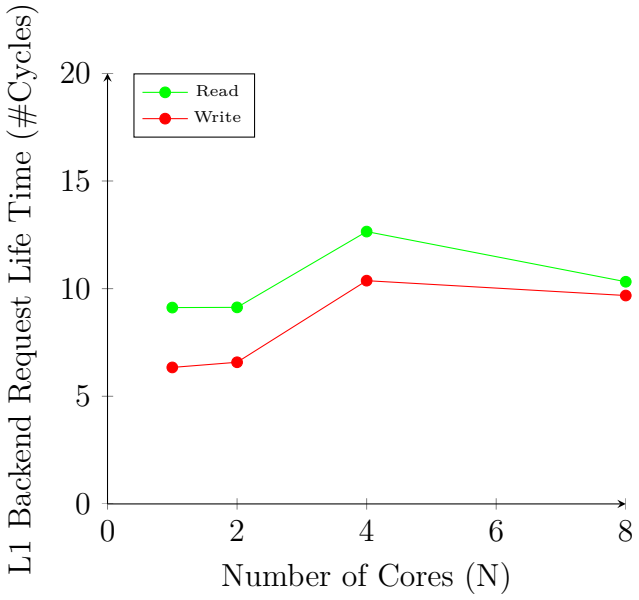


Figure 5.9: Bus Access Time per Request, Main Memory Latency not included

The AHB bus is fully congested at four cores, when its maximum bus access time

latency is reached. The drop after is explained by the duplicated memory traces leading to fewer misses in the L2. This means that less time, per request, is spent sending read requests from the L2 to RAM.

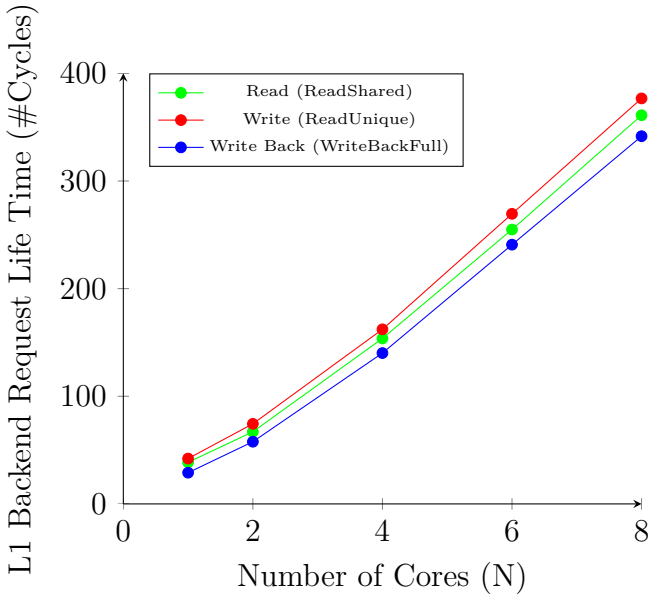


Figure 5.10: NoC Access Time per Request, Main Memory Latency not included

Looking at figure 5.10, the CHI system will still see wait times in this measurement, not because of the NoC but because of the blocking HN. As before, these are increasing linearly. The high number of clock cycles is explained by the NoC's high routing latency, blocking HN and our simulations having duplicated memory traces for eight cores, resulting in more snoop request generated.

6

Discussion

This chapter will discuss the results found in the previous chapter and possible improvements.

6.1 Coherence Protocols

The idea of the research project was to compare a coherent AHB bus-interconnected system with a coherent CHI NoC-interconnected system. However, the research suffered by, on the CHI side, being limited to components that could not handle requests from different sources in parallel, these being the HNs and RNs. Individual request sent in parallel from individual RNs, and individual requests handled in parallel in the HN would clearly increase the bandwidth of the CHI system. The evaluation that was run on the CHI system ended up performing similar to an exclusive bus interconnect, even though it was connected through a NoC. Since both the CHI system and the AHB system performed linearly, the CHI system as expected performed worse with a higher latency and lower bandwidth (processed memory request per clock cycles). This would, for most software traces, not be the case in a CHI system with parallel request handling. Instead, would the CHI system perform logarithmically and for a larger number of cores it would beat the AHB system.

Comparing write-back and write-through is not optimal, but since the AHB system could only have a write-through cache in order to be coherent, and having a CHI system without a write-back cache would mean only utilizing the invalid state of the protocol, both cache versions were necessary and the comparison needed to be done. When comparing, there is now clearly evidence for a potential performance increase in figure 5.8.

6.2 Design Choices

Regarding the write-queue, as mentioned in section 4.1.1 and in section 4.2.1 the queues in the different caches differ in size. The reasoning behind this is because Gaisler's write-through uses a queue with size 4, and as discussed in section 4.2.1 it is essentially irrelevant to increase the queue size more than 1 in the CHI system. Considering this, the performance of the AHB system increases somewhat in cases where memory accesses does not miss in L1 as often. In these cases, new requests

to the L1 can be handled while the L1 cache backend is propagating a write; this ends when a miss on a new request in the L1 is encountered.

In the project real processors were not simulated. This means instructions other than memory accesses are not simulated. In a real scenario there would be more waiting for the cache while the processor handles addition, subtraction, jumps etc. This means the AHB write queue would increase the AHB systems performance more since its L1 cache would be able to handle write requests while the processor was executing other instructions.

Considering the software traces run on the evaluation suites, they were run only for 5000 memory instructions. When picking the memory instructions from QEMU the entire extracted memory trace from the benchmark comprised of more than 1 million memory instructions, meaning not even 1% of the benchmark was actually run on the evaluation suite. The executed benchmark was also only considered for only one set of parameter inputs, with the lowest parameter values possible to reduce the number of memory instructions. There were also more benchmarks available in the OBPMark suite which code was written for, but only the first benchmark, 1.1-image, was run. Considering this the memory traces of different tests and different settings could end up looking differently from the one run in this project.

6.3 Improvement and Future Work

Two main improvements are clear from the results and discussion:

- A NoC with lower routing latency
- RN and HN components that support parallel request

On top of this, it would be beneficial to divide up the address range between multiple HNs connected to the NoC.

The first improvement is truly essential. Looking at figure 5.3 in all cases without a main memory access, more than 75% of the clock cycles are spent propagating a request. This is not sustainable. A decrease in propagation delay would lead to a much lower memory access latency and much higher system throughput for all number of cores in the CHI system.

The second (and third) improvement would make the CHI latency and system throughput non-linear instead of linear; making it an actual possibility to beat AHB for the higher number of cores.

Both these improvements are required and essential for finding a threshold number of cores for which scaling with CHI would be preferable than AHB, which was the goal of the project.

The performance increase considering these improvements would e.g. be seen in following cases: The CHI graph in figure 5.5 would not increase linearly. And with parallelism the CHI bandwidth shown in figure 5.6 would increase with the number of cores instead. We would therefore in these two figures see the CHI graph intersecting the AHB graphs, giving a result of when scaling with CHI would be more suitable.

The NoC access time per request shown in figure 5.10, would be significantly lower with the improvements in place.

Also, to fully take advantage of the parallelism which the NoC offers, the designs need to include either an HN that can handle requests in parallel or add multiple HNs to the design. As of now with only one non-parallel HN, essentially only one request is propagating through the NoC, since all other requests which are not being handled by the HN are being blocked at the input of the HN until they can be handled. This makes the NoC non-contributory in this project's CHI design; the NoC is now doing what a bus does, but with added NoC latency.

Adding on multiple HNs would mean dividing up the address range between multiple HNs. The HNs would essentially act as one and the same memory area, while physically connected to different nodes; thus being able to handle requests in parallel. However this setup was never established fully because of limited time.

Even with the limited CHI system, it could still be seen that a L1 write-back policy improves the system throughput, as fewer requests are propagated to the L2. For future comparison, it would be valuable to measure the performance gains of the write-back policy with all improvements implemented.

7

Conclusion

To conclude the report, the main contribution made during the project has been an evaluation platform for cache coherency. The platform is ready with two systems; one for testing snooping with an AHB bus and one for testing a directory with a CHI NoC. It is highly configurable in terms of using RTL or UVM modules, and for different HDLs. Test stimuli have been produced with two different procedures; custom-generated memory requests for targeted memory accesses and memory traces extracted from software programs (in this case OBPMark).

Components supporting parallelism were not available during the project. This meant that the evaluation goals were not met as it was not possible to fully evaluate the CHI protocol. Further testing needs to be performed with components that support the parallel aspects of the CHI protocol, such as a non-blocking RN and HN. When these components have been developed, the proper environment to test them in is ready.

As expected, the limitation of a CHI system with a single point of coherence caused the system's performance to have a linear development with the number of cores in the system. Because of the linearity it performed worse, for all number of cores, than the AHB system because of the interconnect's higher complexity. What could be seen however, was a possibility of improvement in system throughput if a system can go from write-through to write-back policy, having NoC parallelism and lower propagation delay in the NoC.

Bibliography

[1] Apple Inc., *Concurrency Programming Guide: Concurrency and Application Design*, 2023, accessed December 22, 2024. [Online]. Available: https://developer.apple.com/library/archive/documentation/General/Conceptual/ConcurrencyProgrammingGuide/ConcurrencyandApplicationDesign/ConcurrencyandApplicationDesign.html#//apple_ref/doc/uid/TP40008091-CH100-SW7

[2] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manuals*. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[3] Custom PC, "Inside AMD Zen 4 Ryzen CPU Architecture," *Custom PC*, 2023. [Online]. Available: https://www.custompc.com/inside-amd-zen-4-ryzen-cpu-architecture

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2011.

[5] University of Illinois at Urbana-Champaign, "Computer Architecture," https://courses.engr.illinois.edu/cs533/, 2023, accessed December 14, 2024.

[6] Frontgrade Gaisler, "GR740 Quad-Core LEON4FT Microprocessor Overview Presentation," [PowerPoint slides], Available at https://www.gaisler.com/doc/gr740/GR740-OVERVIEW.pdf.

[7] F. Malatesta, R. Weigand, and J. Andersson, "GR765: SPARC and RISC-V Multiprocessor System-on-Chip," in *2023 European Data Handling & Data Processing Conference (EDHPC)*, 2023, pp. 1–4.

[8] Frontgrade Gaisler, "NOEL-V," https://www.gaisler.com/products/noel-v, 2025, accessed January 13, 2025.

[9] ——, "GRLIB documentation," https://download.gaisler.com/products/GRLIB/doc/grip.pdf, 2025, accessed January 13, 2025.

[10] T. Bjerregaard and S. Mahadevan, "A Survey of Research and Practices of Network-on-Chip," *ACM Computing Surveys (CSUR)*, vol. 38.1, 2006.

[11] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal, "Directory-based cache coherence in large-scale multiprocessors," *Computer*, vol. 23, no. 6, pp. 49–58, 1990.

[12] G. De Micheli, C. Seiculescu, S. Murali, L. Benini, F. Angiolini, and A. Pullini, "Networks on Chips: from Research to Products," in *Proceedings of the 47th Design Automation Conference*, 2010, pp. 300–305.

[13] G. Girão, B. C. de Oliveira, R. Soares, and I. S. Silva, "Cache coherency communication cost in a NoC-based MPSoC platform," ser. SBCCI '07. Association for Computing Machinery, 2007, p. 288–293.

[14] I. S. Silva, B. C. de Oliveira, and G. Girão, "Cache alternatives concerning cache coherence in NoC-based MPSoC platform," in *2010 First IEEE Latin American Symposium on Circuits and Systems (LASCAS)*, 2010, pp. 176–179.

[15] A. N. Udipi, N. Muralimanohar, and R. Balasubramonian, "Towards scalable, energy-efficient, bus-based on-chip networks," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12.

[16] "Survey and future directions of fault-tolerant distributed computing on board spacecraft," *Advances in Space Research*, vol. 58, no. 11, pp. 2352–2375, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0273117716304689

[17] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith, "Configurable isolation: building high availability systems with commodity multi-core processors," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, p. 470–481, Jun. 2007. [Online]. Available: https://doi.org/10.1145/1273440.1250720

[18] Rymdstyrelsen, "EUMMSS: Efficient uncore mechanisms for multicore space systems," Available at https://www.rymdstyrelsen.se/innovation/beviljade-bidrag/pirf-2021/eummss/, accessed December 13, 2024.

[19] J. L. Hennessy and D. A. Patterson, *Computer Organization and Design: The Hardware Software Interface*, RISC-V, second. ed. Morgan Kaufmann, 2021.

[20] M. Annavaram, M. Dubois, and P. Stenström, *Parallel Computer Organization and Design*. Cambridge University Press, 2012.

[21] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Elsevier, 2004.

[22] ARM Limited, *AMBA CHI Architecture Specification*, 2024, accessed January 20, 2025. [Online]. Available: https://developer.arm.com/documentation/ihi0050/g/?lang=en

[23] ——, *AMBA AHB Protocol Specification*, 2021, accessed January 20, 2025. [Online]. Available: https://developer.arm.com/documentation/ihi0033/latest/

[24] Siemens EDA / Siemens Digital Industries Software, "Questasim fact sheet," https://sintecs.eu/webdata/uploads/2024/08/Siemens-SW-QuestaSim-FS-85329-D5.pdf, 2024, accessed October 18, 2025.

[25] W. Snyder, "Overview — verilator documentation," https://verilator.org/guide/latest/overview.html, 2024, accessed October 18, 2025.

[26] Accellera Systems Initiative, *Universal Verification Methodology (UVM) 1.2 User's Guide*, 2015. [Online]. Available: https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf

[27] OBPMark, "On-board processing benchmarks," Available at https://obpmark.github.io/ (2005/06/12), accessed April 2, 2025.

[28] D. Steenari, L. Kosmidis, I. Rodriguez-Ferrandez, A. Jover-Alvarez, and K. Förster, "OBPMark (on-board processing benchmarks)-open source computational performance benchmarks for space applications," pp. 14–17, 2021.

A

Appendix 1

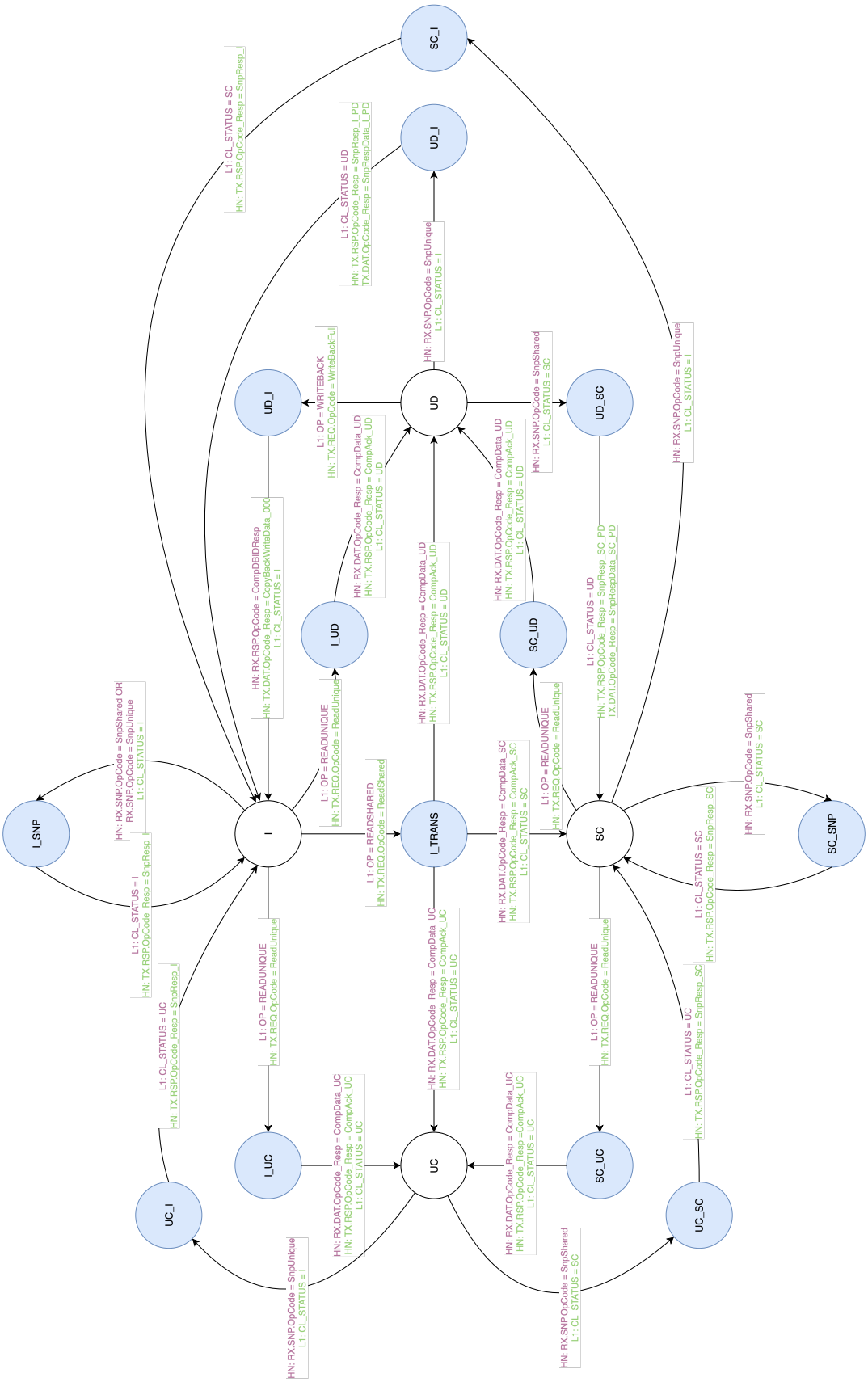


Figure A.1: State machine of RN for communications with L1 and HN