Developing a Library of Verified Cache Coherence Protocols

Reece Carr

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2021

Abstract

This report introduces and describes the details of our implementation of a library of verified cache coherence protocols. The design of these protocols has been motivated by various literature and public specifications.

The primary motivation in this project was to provide a collection of reusable, verified, and documented protocols for further academic and industrial use. We have provided protocols derived from the standard MOESI family of protocols as well as extending these by using three different communication patterns for forward messaging. Additionally we have provided an implementation of a subset of the cache coherence protocol used in the Arm AMBA CHI interconnect. Finally, we have made attempts at optimising this protocol by reducing the number of necessary coherence communications in the system.

Acknowledgements

I give my thanks to my advisor Professor Vijay Nagarajan, and associates Nicolai Oswald and Vasilis Gavrielatos, for all of their assistance and guidance.

I also wish to give credit to my friends and family for all of the support that has been given to me throughout all of my studies.

Table of Contents

Chapter 1

Introduction

1.1 Prologue

The concept of hardware parallelism has become deeply embedded into the design of today's common computer architectures. The limitations of power scaling [23] and the fall of moore's law [36], are practical complications that have lead to the implementation of multi-core technologies [21]. These systems require support for shared physical memory in hardware so as to efficiently share essential resources between multiple processing units. This must be done in a manner that is invisible to the programmer, and yet guarantees correct instruction execution [32]. This guarantee is provided by the combination of a memory consistency model and cache coherence protocol.

The memory consistency model can be viewed as an interface for the developer. It lays the foundation for what behaviour can be expected from a complex memory system [29, 2]. Meanwhile, cache coherence is a primary necessity for ensuring the memory consistency model is enforced. It is further responsible for ensuring that data values are correctly updated in local caches to ensure synchronisation of data [20]. The main design objective of cache coherence protocols is to achieve synchronisation with the lowest costs in terms of latency and hardware usage [33].

1.2 Current Issues

There are many different existing cache coherence protocols, each of which has different behaviour and varying levels of intricacies in the design of the models under which they are implemented. The design of such protocols is a very difficult task as concurrent operations can be hard to identify, and failing to correctly handle such concurrency can lead to reduced performance or even entire system failures.

A bug in the coherence protocol used in the CCI-400 interconnect resulted in Samsung S4 mobile phones shipping with concurrency disabled [3], resulting in reduced performance and increased power consumption. A more serious case involves a coherence bug in the TLB of the AMD Barcelona chip which resulted in the chip locking up entirely [38, 18]. These examples clearly illustrate how difficult it is to design and verify

coherence protocols correctly.

Tools such as ProtoGen [31] have been developed to alleviate some of the difficulties of these designs. These tools take in an atomic model of a cache coherence protocol and generate a complete non-atomic protocol as output. This certainly has it's benefits, however, these tools still require that any given input specification must be correct in the first place.

Furthermore, industrial specifications such as Arm AMBA CHI [9] and Gen-Z [13] provide designs for highly optimised coherent interconnects, yet currently none of these coherence protocols have been verified in a public space. These specifications are also very complex, often several hundred pages of dense documentation, and as such it can be difficult to discern the important aspects relating to the coherence systems of these interconnects.

1.3 Objectives

The objectives of this project are then derived from the previously described issues. We set out to provide a library of verified protocols, including protocols derived from the Arm AMBA CHI specification. These can then be used for further work such as latency simulation, hardware synthesis, or even further extending the library to contain a larger variation of protocols for industrial use.

1.4 Contributions

Our contributions as part of this project are as follows:

- Implementing five common MOESI protocols with three variations on message forwarding.
- Modelling the AMBA CHI coherence protocol in ProtoGen and Roc3. Implementing a subset of transitions and states so as to provide a simple protocol that is comparable to the MOESI protocols.
- Verifying the AMBA CHI coherence protocol in public space for the first time.
- Optimising an implementation of the CHI coherence protocol for enhanced performance.
- Ensuring correctness of all produced protocols through the use of the Murφ tool and invariant checking.

In total the produced library contains 20 verified cache coherence protocols. We hope that this library will continue to grow in the near future, and that the protocols produced here will be used in future research.

Chapter 2

Background

2.1 Memory Consistency

The memory consistency model acts as the contract between the system hardware, compiler, and the application programmer. It provides a guarantee on the ordering of instructions within a parallel system, giving the programmer expectations for the behaviour that can occur [22].

The strictest memory model is known as Sequential Consistency (SC). SC has the requirement that all operations on memory appear to adhere to some global order, and that the resulting behaviour aligns with the original program order at all issuing processors [17].

There are a variety of different existing memory models such as Release Consistency (RC), Total Store Order (TSO), and many others [2, 29, 25]. These exist so as to give varying degrees of freedom for optimisation, at the cost of requiring programmers to enforce synchronisation manually [22].

2.2 Invariants

Branching slightly from these memory models, is the concept of invariants. Invariants are a set of rules rather than a specification like memory models. These rules are used to enforce correctness for a shared memory system.

For this project we use the Single-Writer Multiple-Reader (SWMR) invariant for the aforementioned purposes. This requires that at any given time a cache line is either write-able by a single cache, or is read-able by zero or more caches [39]. This may seem like a tight constraint however this is a common method for ensuring all data accesses result in the usage of the most up-to-date data values [28].

Previous work has shown that it is possible to relax these constraints when more information is available about the environment and ordering of actions being performed [19]. This does not however, lead to any substantial gains in performance, and as such these methods are not considered within this project.

2.3 Cache Coherence

Cache coherence protocols are responsible for enforcing memory consistency within a shared memory system. This is often accomplished using hardware known as coherence controllers [39]. These controllers are finite state machines which are attached to each memory unit that requires coherence.

Coherence protocols can be described as either snooping or directory protocols. This refers to the method used to send and monitor messages in the memory system. These two classes can be described as follows:

- Snooping - A snooping protocol will broadcast messages to all connected coherence controllers, ensuring all units are informed on every coherence transition. This requires that all coherence units monitor the broadcast bus, and must keep their own coherence controllers up to date [32, 20].

- Directory - A directory protocol instead uses an additional coherence unit to store meta data about the global state of a cache line. This information may include validity, cleanliness, ownership, and a sharer vector which tracks the current sharers. Requests are sent to this central directory which serializes instructions and restricts communications to only the necessary components [39, 20].

Snooping protocols often require higher bandwidth interconnects in order to broadcast to all coherence controllers, whilst directory protocols can suffer from higher latency due to the additional communication hops. The most important distinction is that snooping protocols do not scale well for larger interconnects in general due to bandwidth limitations.

A very simple cache controller may look like that presented in Figure 2.1. This is a snippet of the atomic cache controller for a MI directory protocol. Here it can be seen that a cache line is either invalid (I) or in a modified state (M).

	Load	Store	GetM
I	Send GetM to Dir, await Data, /M	Send GetM to Dir, await Data, /M	
M	Hit	Hit	Send data to Req, update Dir, /I

Table 2.1: Atomic MI Protocol Cache Controller

Only a single cache may hold the line in a valid state at any given time. A cache is required to contact the directory and invalidate the current owner before entering the modified state. The directory will forward the request to the current owner, or directly respond to the requestor if no such owner exists. Clearly this is a very simplistic model and would not be used in a real system due to the strict limitations on data sharing and lack of parallelism.

2.4 MOESI Protocols

The MOESI protocols are a family of invalidation-based cache coherence protocols [20]. Invalidation-based means that sharers are invalidated upon an incoming write-request as opposed to being updated with the new value as with an update-based protocol. These protocols support write-back caches which reduces the communications necessary on the interconnect as modified data is not required to be immediately written to main memory [39, 20].

The available states are as follows:

- Invalid (I): The cache does not contain a valid copy of the cache line. Data must be requested from the directory before a read or write is viable.
- Shared (S): The cache is permitted to read the cache line. Copies of the cache line may be present in multiple caches. A write requires a request to the directory, and the invalidation of any other copies.
- Exclusive (E): The cache contains an exclusive copy of the data. The data can be read and written to, without any communication to the directory or other caches. A write involves a silent upgrade to M state.
- Owned (O): The cache contains dirty data and copies may be shared with other caches. The cache can read the data, but must invalidate sharers before a write operation. Must be written back to memory on eviction.
- Modified (M): The cache is permitted to read and modify the cache line. Any other copies of the data in the system are invalid. Must be written back to memory on eviction.

An individual protocol will make use of some subset of these states. For example, the MSI protocol uses only the Invalid, Shared, and Modified states. This limits the flexibility of the coherence protocol in order to reduce complexity and power consumption.

These protocols function as a basis for our modelling in this project as they can be quite simple whilst also commonly being used in industry. Examples of commercial multiprocessors that use MOESI protocols are the Silicon Graphics Origin 2000 [27] and Intel QuickPath Interconnect (QPI) [14]. Both are variations on the common MESI coherence protocol. The usage of such protocols by high profile companies leads us to believe there is value in the creation of a library which can be easily incorporated into industry level processors.

2.5 Atomicity

The majority of the complications with designing and verifying cache coherence protocols is due to real systems not being atomic. This means that when transitioning from one state to another, it is in fact possible that a message may arrive during this time. Depending on the implementation this may either stall (resulting in a loss in performance) or may need to be responded to immediately. As always there are trade-offs for different designs, and no single design can possibly fit every product requirement.

Let us consider the following example. A cache currently holds a dirty cache line and wishes to evict it. This cache must send a request to the directory to inform it of the eviction. The issue with a physical implementation of a coherence protocol is that before the directory receives this message it may already have received a request for this data. This results in the directory requesting the data from the cache that is currently in the process of evicting. At this point in time the cache with the data is required to respond to the request, as not doing so will result in a system deadlock. The message flow for this example is given in Figure 2.1.

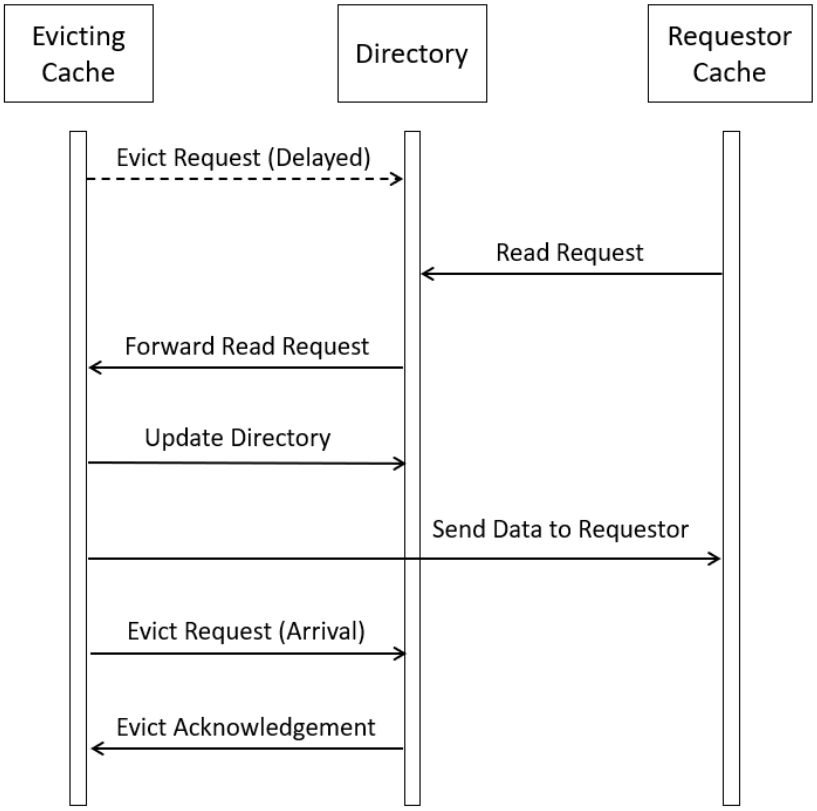

Figure 2.1: Cache Coherence Non-Atomic Evict Flow

These situations appear much more often than one would expect, and can be incredibly difficult to catch. The states in between transitions are referred to as transient states and often dominate the models for coherence protocols. In order to avoid this complication, designers use Stable States Protocols (SSPs) as the starting point for protocol design.

These SSPs contain only the stable states which can be entered at the end of a transition. The transient states would then be derived manually from this specification. This however, is not a simple task and still requires a vast amount of time. As such, a tool named ProtoGen has been created to handle the automatic generation of complete coherence protocols from a given SSP.

2.6 Tools

2.6.1 Murφ

Murφ is a tool for describing and verifying finite-state concurrent systems such as coherence protocols. The language is heavily verbose which is one of the reasons we use other tools for the description and generation of protocols. Instead, we strictly use Murφ for the verification of our protocols.

Verification is performed by generating and storing states in a hash-table, and uses state space reduction techniques to reduce the amount of computation required [15]. This process uses a set of invariants alongside state space search techniques to check for violations of any invariants. At the point this occurs an error would be returned. Similarly Murφ also checks for deadlocks during this process, which would be discovered when a node has no successor.

2.6.2 ProtoGen

The ProtoGen tool [31] can be used to take an atomic stable state protocol (SSP) of a directory protocol and generate the complete protocol with transient states. This removes the complications incurred by concurrency when attempting to design coherence protocols.

The input atomic SSP is written in the ProtoGen domain specific language, and converted into a collection of finite state machines (FSMs) for describing the cache and directory controllers. An example of the input for a very simple coherence protocol is given in Figure 2.2. The example shows the transition from a cache in state I to state S, with the GetS request being sent on the request bus and the corresponding await for the data acknowledgement.

```
Architecture cache {

    Stable{I, S, M}

    // I ////////////////////////////////////////////////////
    Process(I, load, State){
        msg = Request(GetS, ID, directory.ID);
        req.send(msg);

        await{
            when GetS_Ack:
                cl=GetS_Ack.cl;
                State = S;
                break;
        }
    }
```

Figure 2.2: Snippet of MSI SSP Written in ProtoGen DSL

The generation of complete protocols takes note of the linearization of messages which is performed at the directory, and combines this with message renaming to convey the ordering of messages to caches. The complete protocol is then output as FSMs into a Murφ file. This file includes a framework for verification of the protocol which can be given to the Murφ model checker [15] to ensure correctness of the generated protocol.

2.6.3 Roc3

Roc3 is another tool which can be used for the creation of cache coherence protocols [30]. The input to Roc3 is a complete protocol specified in a DSL unlike the ProtoGen tool which takes in an SSP. This means that extra work is required in the manual implementation of a protocol however this is still much preferable to implementing directly with Murφ.

A short snippet of the Roc3 DSL is given in Figure 2.3. This shows the transitions from the stable I state to the transient states for the load and store operations in the CHI coherence protocol. It can be seen that no await block is used, and that it is required to state all transitions including transient states.

```
////////// Cache ////////////////////////////////////
machine cache[3] {


    startstate: I;

    //////// I ////////////////////////////////////
    (I, *load, I_RS){
        Dir[0]!ReadShared@req;
    }

    (I, *store, I_CU){
        Dir[0]!CleanUnique@req;
    }
```

Figure 2.3: Snippet of Non-Atomic CHI Protocol Written in Roc3 DSL

The advantage of such a tool is that this permits us to verify the complete specification of a protocol without any automatic generation of transient states. Roc3 also produces a Murφ file which can be verified as previously mentioned.

Chapter 3

Message Forwarding Communication Patterns

This section presents a brief introduction to three methods for handling requests which require messages to be forwarded between components in a shared-memory system. The abstract models, from which our work is based on, are adapted from Lametti 2010 [26]. Our work expands on this by providing verified implementations using each communication model, as well as further discussing the relative merits in this report.

Each communication pattern has varying advantages and disadvantages, and some have even been used together in real systems. The details here are relevant to both the MOESI protocols and the CHI implementation that is provided in the developed coherence protocol library.

3.1 Strict Request-Response

The first method of handling requests for data that require communication to additional cache, is known as strict request-response. In this model, the requestor sends a request to the directory, which then responds with the ID of the cache that contains the data. The requestor is then required to send a direct request to the data holder, who then responds with the data and updates the directory. A diagram is given in Figure 3.1.

It should be clear that such a method of communication is highly inefficient. Firstly, a total of four sequential latencies are observed during this transaction. Secondly, this model requires additional complexity at both the directory controller and the cache controllers for handling multiple messages each during this transaction. Both of these issues can be fixed by making simple alterations to the communication pattern, as is the goal of the next two methods.

This is surprisingly the most difficult of the three methods to implement. This is due to the additional complexity regarding acquiring the owner ID before re-sending the request to the owner. Our main struggles surround the design of the cache controller due to this multi step process, and the difficulties of catching all possible types of

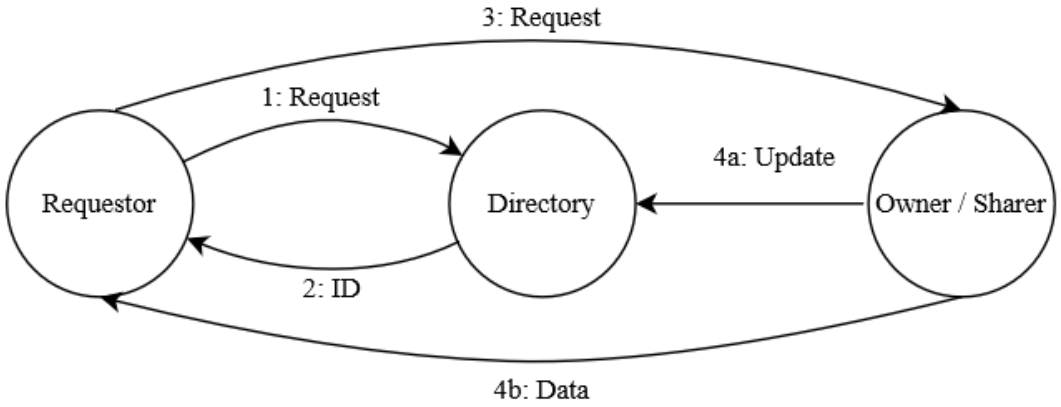

Figure 3.1: Communication model for a strict request-response cache coherence protocol. Adapted from Lametti 2010 [26].

incoming messages. We shall see with the following methods that this is unnecessary and can easily be circumvented through better design.

3.2 Intervention Forwarding

The previously discussed forwarding mechanism required a total of five messages to be sent between components. This presents an opportunity for improvement, as often the goal of cache coherence protocols is to minimise the number of communications in a networked system.

The first simple improvement can be achieved by having the directory directly contact the node which holds the current data. This removes the need for the requestor to have further involvement after requesting the data from the directory. This communication pattern is known as intervention-forwarding. A diagram of this method is given in Figure 3.2.

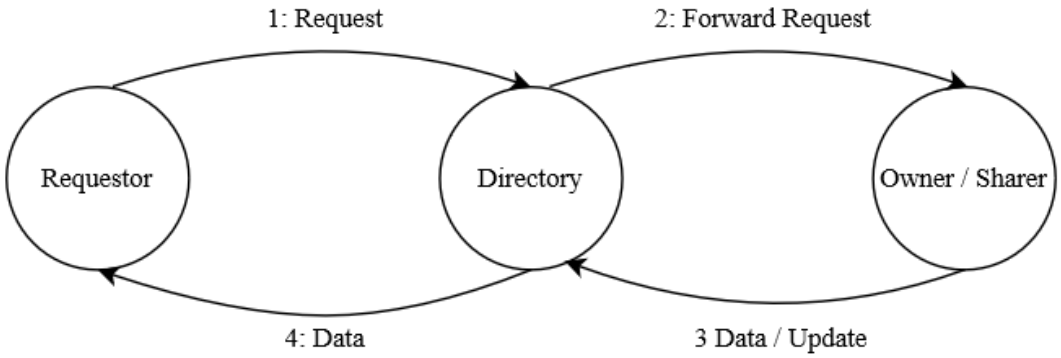

Figure 3.2: Communication model for an intervention-forwarding cache coherence protocol. Adapted from Lametti 2010 [26].

It can be seen that the directory no longer responds to the requestor with the identifier for the node that the requestor would need to contact. Instead the simpler route of communication removes complexity from the cache controller and the directory, as well as reducing the number of total messages sent from five to four.

The implementation of this method is much simpler since the cache controller is now only required to send a single request, and receive a single data response. It should be noted that an additional advantage of this approach is that invalidation acknowledgements can be handled at either the directory (before sending data to the requestor), or at the requestor itself. Our implementations continue to use the prior method since this is the common pattern used with other protocols, however we shall see that real systems may opt to use the later method as shown in Chapter 4.

3.3 Reply Forwarding

The next issue with the previous communication patterns is the number of sequential messages being sent and the overall latency visible from this. Therefore the next goal is to reduce this as much as possible. Since forward messaging requires communication between three separate components, it is impossible to lower the number of sequential messages below three.

It should be clear that the previous method had an unnecessary pause in communication, as the directory receives the data from the node that holds the current data, and forwards it to the requestor. Instead a simple optimisation is to have the node directly send the data to the cache in parallel with sending an update to the directory.

This is a communication pattern known as Reply Forwarding, and is actually the method used for protocols given in the primer by Nagarajan et al [39]. A diagram of this communication pattern can be seen in Figure 3.3.

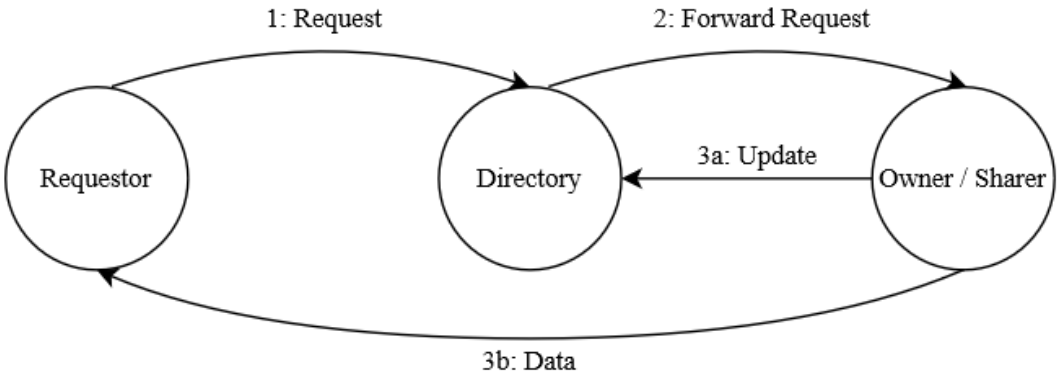

Figure 3.3: Communication model for a reply-forwarding cache coherence protocol. Adapted from Lametti 2010 [26].

The primary benefit of this method is the low latency derived from the minimised communications on the network. This is likely to be the most important factor for picking

this model as one of the objectives of coherence protocols is to minimise communication latency. The limitation is that invalidation acknowledgements cannot be handled at the directory as discussed previously. In most cases this is likely not a concern, however large interconnects may wish to conceal this complexity from requestor nodes.

This is the most common, and most well documented of the three methods discussed in this chapter. Therefore, our implementations are derived directly from those presented in the primer by Nagarajan et al [39]. These were the simplest protocols to implement, partly due to the extensive documentation, but also due to the reduced number of steps leading to the designs being easier to follow during implementation.

3.4 Summary

In this short chapter, we have presented and discussed three models for forwarding messages in coherence protocols. These methods make use of very simply optimisations that can remove unnecessary communication. At a large scale, these optimisations can have a drastic effect on the overall performance for a shared memory system.

It is clear that the strict-response method is simply inferior to either of the other methods, however in our library we include a set of MOESI protocols using all three methods for completeness. Additionally, we shall see in the next chapter that the intervention-forwarding and reply-forwarding models can also be used in tandem within a real system.

Chapter 4

Amba CHI Cache Coherence

As open specifications have recently become a more common proposition, we include a model from a publicly available and high profile specification. The Arm AMBA Coherent Hub Interface (CHI) specification presents the requirements for an interface for connecting high performance fully coherent processors and memory controllers [8].

This chapter focuses on the details of the coherence protocol for the CHI interconnect and presents our implementation of a subset of the protocol, along with modifications for optimisations.

4.1 Introduction

4.1.1 Context

Designed by Arm, this free public specification presents the requirements and the design details for the CHI architecture. The specification is written so as to allow designers to make decisions regarding the balance of high-performance, power consumption and efficiency [9]. The cache coherence protocol clearly takes inspiration from the common MOESI protocols, however additional states are included so as to permit partial writing of data and ownership without valid data. This is intended to save bandwidth and reduce the number of writes to main memory.

The CHI interconnect can be found on many of today's System On Chip (SOC) products. There is a wide variety of uses of this interconnect within Arm products such as the Cortex A65AE for automotive applications [6], Cortex A76 for mobile computing [7], and the CoreLink DMC-620 for high performance computing servers [5].

4.1.2 Specification

The CHI specification [8] is a large specification that allows for a lot of design decisions regarding the usage of states, transactions, and the fine grain design of the directory. Since the specification is too large for us to model in entirety, we have taken to modelling subsets that we believe best encompass the main features available for the

CHI protocol. Additionally, some specific items have been left out entirely due to the details being far too complex to handle with the time or tools available. For instance, we do not handle any form of atomic transaction or partial data for these exact reasons.

The specification presents transactions with stable states however it also includes details on how to handle requests during transitions. As such, we have taken the information provided and modelled the full protocols with Roc3. This allows us to contrast the complete protocol derived from the specification with that generated by ProtoGen. Our aim was to prove that ProtoGen can be used to model coherence protocols with higher complexity, and to contrast the automated design to our manual work for verification purposes.

In this specification the design and details of the directory are purposely left vague so as to allow for various configurations of the protocol. Therefore, our designs are based on the (often vague) notes presented and examples of transactions found throughout the specification.

4.2 States

The CHI specification gives details on a set of distinct states that are permitted to be used within products built from this specification. It provides seven states which can be best described as combinations of the following elements:

- Validity: If a cache line is present in the cache then it is valid.
- Uniqueness: A cache line is unique if only a single cache contains the data.
- Dirtiness: A cache line is clean if the data has not been modified with regards to main memory.
- Granularity: A cache line can either be fully coherent, partially coherent, or entirely empty. This refers to the number of valid bytes in the cache line.

The set of states produced by categorising according to these elements is given in Figure 4.1. We have given equivalent MOESI state names next to the CHI counterparts where these are available.

It can be seen that the majority of these states can be matched to a corresponding state in the MOESI protocols. There are only two states which are exceptions to this. Both the UCE and UDP states contain properties that are not expressed within the MOESI protocols. The UCE state allows for a cache to enter a unique state without currently containing any valid data. This permits a cache to perform a write without requiring data to be sent, hence reducing the load on the network. The UDP state allows for a write to modify segments of the cache line rather than to overwrite the entirety of it.

In our models we use all of these states except the Partial Dirty state and respectively we have no notion of partial data. This was a simplification made due to limitations of our tools for handling partial data as well as removing complexity so as to meet deadlines. Therefore the addition of partial data is left as a potential extension for a future project.

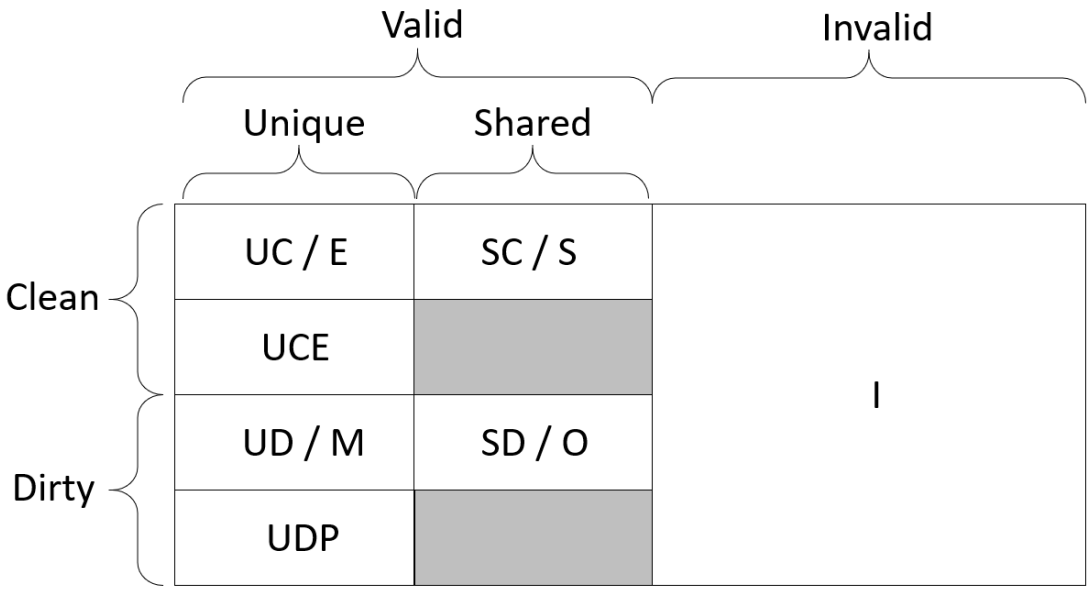

Figure 4.1: States Within CHI Coherence Protocol. Adapted from CHI Specification 2019 [8].

For the sake of consistency, as well as to better contrast this protocol with the more common MOESI protocols, the MOESI counterpart names will be used for the rest of this document.

4.3 Message Types

The majority of the complexity comes from the plethora of transactions permitted within the CHI specification. As this specification is written to permit a variety of components to be connected to the interface, there are a wide variety of messages that are permitted to be sent over the interconnect. The specification dictates that the directory must support all available request types, however as this project is time limited and restricted in scope, our implementation uses a minimised subset of transactions that we feel best demonstrate the CHI interconnect.

There are multiple categories of request messages given in the specification. The categories are presented here so as to allow the reader to understand the relation between messages that fit into the same category, and to help understand our design decisions which are presented briefly after. The types of messages available are:

- Responses: Response messages can either be a completion acknowledgement, data response, or a combined completion and data response. These can be sent by either the directory or a cache as a response to a request.
- Read: Equivalent to Get messages in the MOESI protocols. These are responsible for obtaining data for a requestor and may result in state changes at either the directory or the owner if appropriate.
- Write: Equivalent to Put messages in the MOESI protocols. These are necessary

for updating main memory and the directory and for evicting dirty data. Note that if data is not dirty then it can simply be dropped and an evict message sent to the directory.

- Cache Maintenance Operation (CMO): Allow a cache to change the states of other cache in the system without expecting any data or permitting a change to its own state. For example, a CleanInvalid request ensures all cache are invalidated and that any dirty data is written back to memory.
- Snoop: Responsible for passing requests from the directory to a cache. Responses are always sent to the directory and may include data.
- Forwarding Snoop: Responsible for forwarding requests from the directory to a cache. Responses can be sent to both the requestor and the directory and may include data.

At the end of most request transactions a completion acknowledgement message must also be sent to the directory to ensure correct ordering of instructions. CHI permits an ordering attribute to be used to allow for more control over the ordering of requests, however this is excluded from our models due to the complexity this introduces.

4.4 Transactions

As stated prior, the specification presents a large number of options for transactions and system design. This section presents the transactions used within the provided library, as well as discussing the reasoning behind these design choices.

Load

One of the essential cache operations is that of a load instruction. As described earlier, a large number of read requests are given within the specification. From these options, we have decided to use the ReadShared request.

The ReadShared request simply requires that coherent data is sent to the requestor. The response from the directory (or another cache in the case the message is forwarded) contains data as well as the response message determining the state the requestor must enter.

An example of this transaction is given in Figure 4.2. It can be seen that the request is forwarded to the owner by the directory. The owner then transitions to an O state, and sends data in an S state to the requestor. Additionally, the owner informs the directory that it has entered an O state and forwarded data in an S state. Finally, the transaction ends after the directory receives an acknowledgement from the requestor upon receiving data.

This read request has the most flexibility as it permits the directory to have full control over the resulting states of all caches, so long as the requestor receives coherent data and is informed to enter a valid state. This allows us to capitalise on the available states, so as to keep as many caches in a coherent state as possible. For example, a

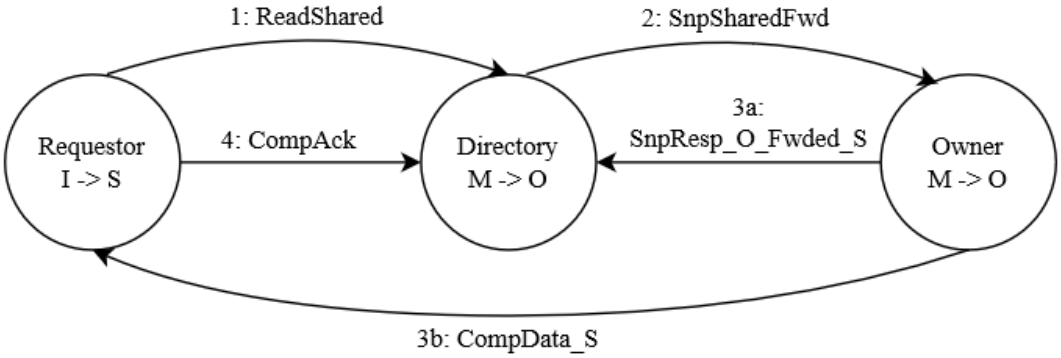

Figure 4.2: Possible Flow of a SnpReadShared Request.

cache may transition from M to O upon receiving the forwarded ReadShared request from the directory. This allows the data to be kept at the owner, instead of causing an invalidation. Such transitions are not always possible with the alternative read request options.

Store

A cache must also be able to perform a store operation. If a cache is in a state with exclusivity then it is always permitted to silently perform the store and upgrade to an M state. This means that no message is sent upon this transition. The directory is responsible for considering this case and managing coherence appropriately.

In any other case, the cache is required to acquire exclusivity from the directory through some method. This can be accomplished using certain read requests, however we have instead opted to use the CleanUnique dataless request to acquire exclusivity without requiring data to be received.

There are three possible scenarios for a store transaction depending on the initial state of the requestor. Regardless of this state, the communication patterns are always the same. The request is sent to the directory, and the directory must invalidate any sharers (excluding the requestor). After the invalidation acknowledgements are received at the directory, the directory will send a Comp_E completion message to the requestor. The rules for state transitions are as follows:

1. If the cache was in an I state then the directory will enter the UCE state. The requestor must enter the UCE state since no data is present at the cache and exclusivity has been acquired. This is presented in Figure 4.3.

2. If the cache was in an S state then the directory will enter the E state. The requestor must enter the E state since data was already present at the cache and exclusivity has been acquired. This is presented in Figure 4.4.

3. If the cache was in an O state then the directory will enter the M state. The requestor must enter the M state since modified data was already present at the cache and exclusivity has been acquired. This is presented in Figure 4.5.

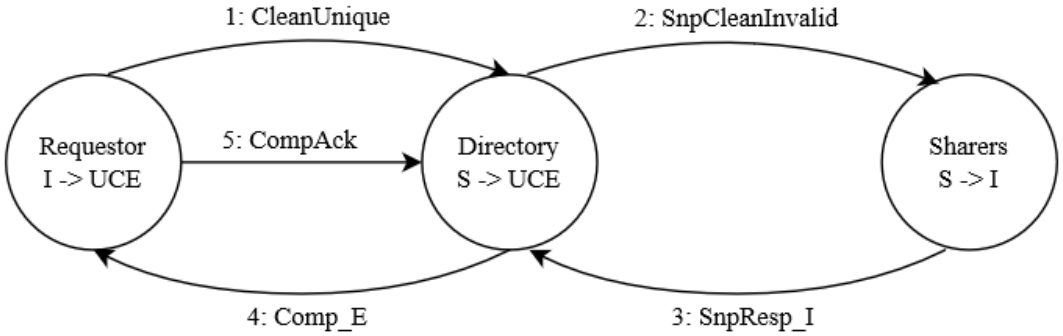

Figure 4.3: Transaction Flow of a Store From Requestor in I State.

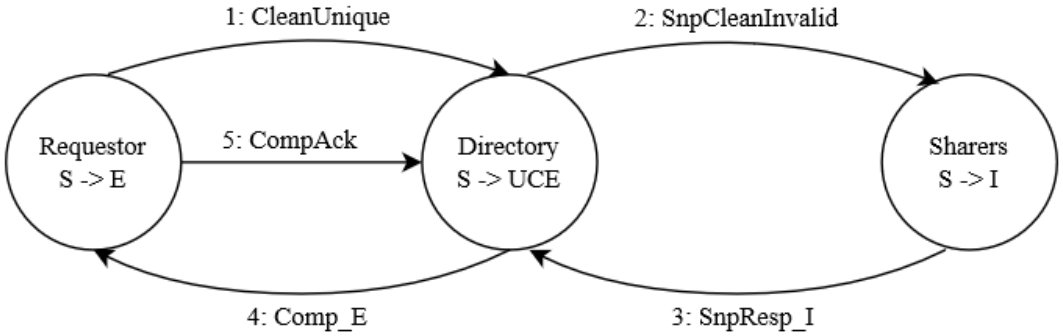

Figure 4.4: Transaction Flow of a Store From Requestor in S State.

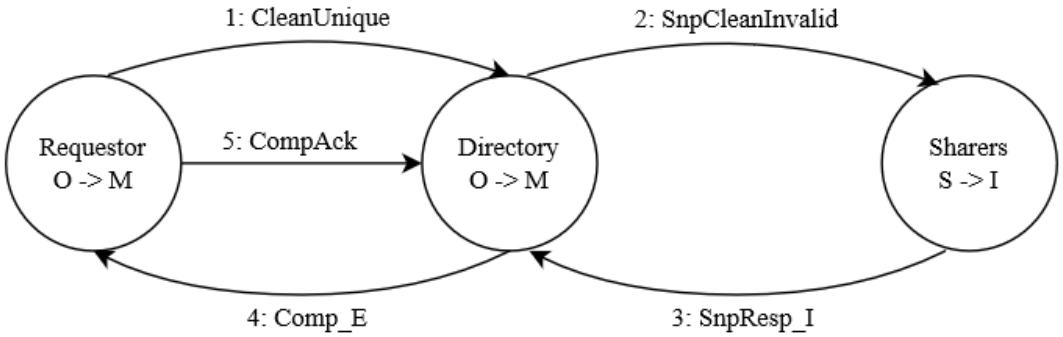

Figure 4.5: Transaction Flow of a Store From Requestor in O State.

This request leads to some of the most complex behaviour at the directory due to the possible resulting states. We consider this to be worthwhile since bandwidth is saved as no unnecessary data is sent over the network.

Replacement

A cache is also required to be able to replace a cache line so as to make room for a new cache line. There are two different approaches used for handling replacement in

the CHI specification according to the current cache line state.

The CHI specification permits a simple Evict message to be used if the data has not been modified. In this case, the cache simply sends an Evict message to the directory, and the directory updates the state and responds with a completion acknowledgement. The transaction flow for an Evict transaction from a cache that is the only sharer of a cache line is given in Figure 4.6.

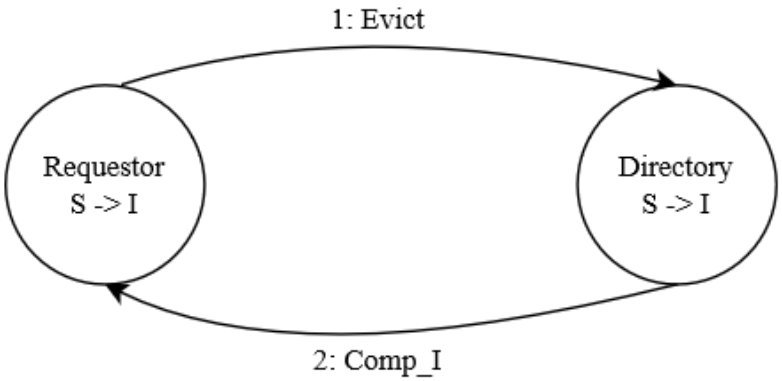

Figure 4.6: Transaction Flow of an Evict From Single Sharer.

Note that the Comp_I response is specified for use as an acknowledgement for an Evict request. The state changes are handled at the directory in the same manner as with a Put request in the MOESI protocols.

In the case that a cache contains modified data, the previous Evict message is not permitted. Instead, the cache must use a form of WriteBack request. These are special requests for writing modified data back to main memory. The specification includes variations that can invalidate all sharers, invalidate only the cache performing the write-back, or even leave this cache in it's current state but still update main memory.

We have chosen to use the WriteBackFull request due to the simplicity inherent with it's operation. This request entails that a cache enters the I state after the write-back is complete, and has no effect on the states of other caches in the system. The transaction flow for a WriteBackFull transaction is given in Figure 4.7.

The transaction begins with the request being sent from the requestor to the directory. The directory then responds with a CompDBIDResp message to indicate that the cache is permitted to write-back. The cache then sends the data in the form of a CBWRData message including the state information (M_PD meaning Modified and passing dirty data) and becomes invalidated. Finally, The directory would then transition into a stable state according to the state information provided.

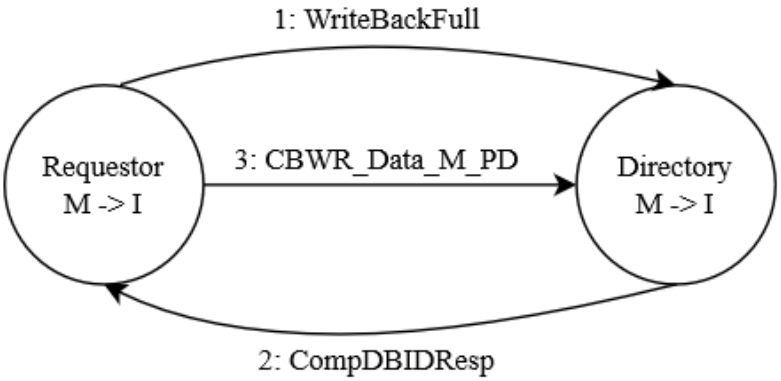

Figure 4.7: Transaction Flow of WriteBackFull Request.

4.5 Concurrency

So far this report has only discussed individual transactions without any additional concurrency. Meaning that it has not yet been discussed as to how additional parallel requests can interfere with existing requests. This section presents a brief discussion of how these messages are handled according to the specification and our implementation.

The specification makes use of stalling along with the additional completion message at the end of many types of transactions so as to prevent race conditions from happening. The directory will ignore any future requests until after the current request has been resolved. This does not however, mean that concurrent operations do not exist, as the following example will show.

Let Cache A begin a load operation, while Cache B begins a store operation. Cache A will put a ReadShared request on the bus, and Cache B will put a CleanUnique request on the bus. Let us say that the ReadShared request is processed first by the directory. Therefore the directory will stall all other incoming requests at this point, including the CleanUnique request. The directory then forwards the ReadShared request to Cache B, which is currently in a transient state due to beginning a store operation. The specification requires that cache B must respond to the request so as to prevent a deadlock from occurring. After the acknowledgement from Cache A reaches the directory, it can then move on to process the CleanUnique from Cache B. The flow of messages is illustrated in Figure 4.8.

This type of stalling behaviour at the directory limits the amount of available concurrency. It would be preferable for the directory to be freed immediately after it has completed any required coherence tasks. This is covered further in the next chapter which details attempts at optimising our implementation of the CHI coherence protocol.

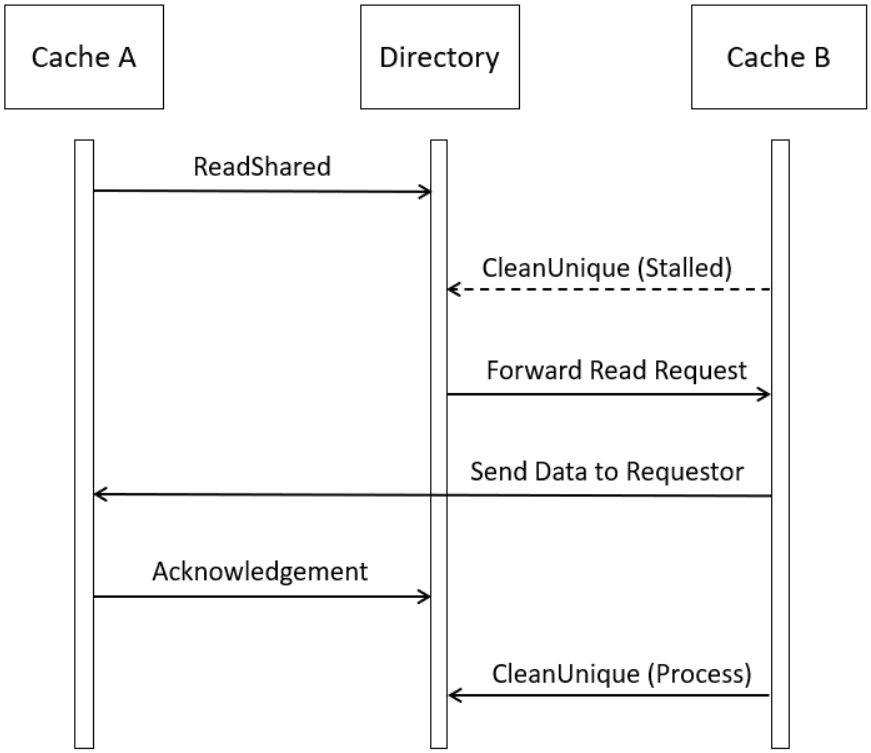

Figure 4.8: Example Concurrent Transaction Flow.

4.6 Generated CHI Cache Coherence Models

We have used both the Roc3 and ProtoGen tools to produce verified models of the CHI protocol as described in this chapter. The intent was to use Roc3 for verifying the protocol as it is described within the CHI specification. This tool is used specifically to avoid any automatic generation or further optimisations during this process. Whereas the ProtoGen tool is used to make a comparison of how effective the tool is, as well as to produce a verified model which is easier to modify for the further work done in Chapter 5.

4.6.1 Comparison of ROC3 and ProtoGen Models

Before comparing the generated protocols, it is important to note that ProtoGen usually uses an optimisation process to reduce the number of redundant states. This means that identical states are formed into a single state. Due to the size of the state space of our CHI implementation, we were forced to disable these optimisations as the program would fail to terminate within a reasonable time frame.

With that being said, the model that has been generated by ProtoGen is still very similar to the model produced using Roc3. There are only a few small differences in the number of states at the cache controller due to the above optimisation not being utilised.

The exact figures for the number of states, stalls, and transitions for the Roc3 generated cache and directory controllers are given in Figure 4.1. The equivalent ProtoGen

generated model statistics are given in Figure 4.2.

	#States	#Stalls	#Transitions
Cache Controller	17	31	49
Directory Controller	34	125	64

Table 4.1: Roc3 CHI Coherence Protocol Statistics

	#States	#Stalls	#Transitions
Cache Controller	21	43	53
Directory Controller	36	141	67

Table 4.2: ProtoGen CHI Coherence Protocol Statistics

It can be seen that the ProtoGen generated cache controller has an additional 4 states when compared with the Roc3 cache controller. These are all additional transient states that are reached by invalidating a cache that is currently evicting. In the Roc3 model, regardless of the original state of the cache, the cache will enter a transient II^A state which awaits the confirmation for the evict before transitioning to the stable I state. The ProtoGen version does not generate such a transient state, and instead has a separate transient state for each initial state instead of using only the one.

For example, a cache in state M may issue a WriteBackFull, enter an initial transient state, and then be invalidated by a snoop request. The cache would then enter a specific MI^A state instead of the generic II^A state described before. This is applied to the UCE, E, O, and M states. Hence resulting in the additional 4 states.

The directory controller is almost identical to that generated for the ROC3 model. There are only two cases of states that have identical behaviour but have not been combined. Both are results of an invalidation acknowledgement being split into separate states when in reality the resulting states are identical. This would also likely be removed by the optimisation algorithm if it was being run.

4.6.2 General Comments

An important element of this protocol is that both load and store communications end with an acknowledgement message being sent to the directory. This is used to inform the directory that a transaction is complete, and that the cache is ready for the next request to be issued. The specification states that this is necessary for ensuring the ordering of messages is kept consistent between the components in the system.

The specification additionally uses a request-retry system for handling concurrent transaction requests. A buffer is used to store incoming requests that cannot currently be handled. Once this buffer has reached maximum capacity, any future requests result in a negative acknowledgement, otherwise known as a NACK. This is a common concept for coherence protocols, however this is beyond the scope of this project and as such has been left out from our implementations.

It is clear from the generated models that most of the complexity lies in the directory specification. This is due to the CHI specification using intervention-forwarding for

handling transactions that require invalidations. This means that the transitions at a cache are reliant only on the immediate response from the directory. The cache will not receive invalidation acknowledgements unlike with the MOESI protocols. Instead, these tasks are given to the directory to manage.

Unfortunately the specification is less explicit with the methods for handling these tasks than we expected. Our models have been inferred from various transaction flow examples given within the specification, however no direct explanation has been found. This does not lead to any issues with correctness, however the ambiguity means that our models are only one example of an implementation, whereas other solutions to handling these issues may exist.

4.7 Correctness

The primary focus of this part of the project was to verify the correctness of the CHI protocol as it is described in the Arm Amba CHI specification. This was done in a similar manner to the correctness checking for the MOESI protocols as discussed previously.

The Murφ model checker was again used to check for deadlocks in the protocols; of which none were found. It was then run with the invariants as given in Appendix A Figure A.2, to ensure that SWMR was upheld. Since the models pass under these constraints it can be assumed that these models are correct.

Of course this only applies to the subset of the CHI specification that has been implemented into our models. As this is a subset of what is available in the specification, we cannot guarantee correctness when using other combinations of states and transactions. Other transactions and states may result in more or less flexibility in the protocol design at the cost of higher complexity, particularly within the directory controller as shown by our implementations.

Chapter 5

Optimising The CHI Coherence Protocol

This chapter delves into attempts made at optimising our CHI protocol implementations as discussed in Chapter 4. The first objective was to design an implementation using an unordered bus for communication, so as to remove the ordered-bus constraint. The second objective was to remove as many unnecessary acknowledgement messages as possible.

The work done in this chapter is also relevant to other existing cache coherence protocols as the optimisations are presumed to be transferable to other acknowledgement-heavy coherence protocols.

5.1 Creating an Unordered-Bus Implemenation

It can be seen that the CHI protocol as presented in the specification, makes use of acknowledgement messages at the end of all transactions other than evicts and write-backs. These acknowledgements entail further stalls during a transaction, since the directory is unable to continue processing later requests during this time. This leads us to the following hypothesis.

Hypothesis 1 (H1): *It is possible to use an unordered-bus for the CHI coherence protocol as described in this report.*

This is our belief due to the additional acknowledgements at the end of all transactions providing a linearization of messages between the cache and the directory.

This model requires only a single change in the input SSP against the model presented previously. When a cache receives a forward-snoop request, it is required to send an update to the directory and data (if it contains any) to the requestor. Since it is possible that the update message is delayed, it is possible for the acknowledgement from the requestor to arrive at the directory before the update message. A diagram of the flow of messages is provided in Figure 5.1.

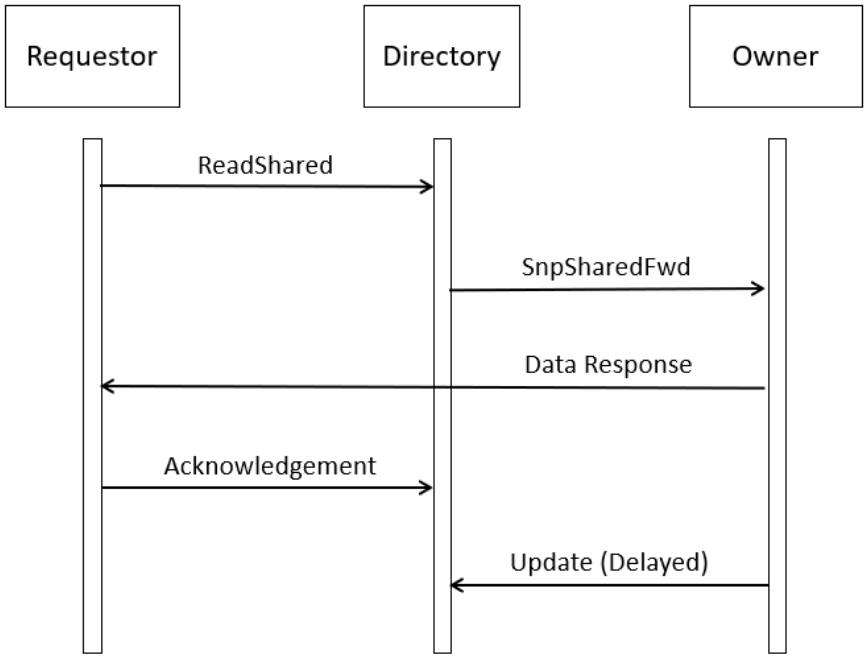

Figure 5.1: Message Flow of Delayed Update on Unordered-Bus.

This requires a very minor change in the SSP to account for this ordering. It is enough to adapt the SSP so as to allow for both orderings of these messages. Note that this is not an issue on an ordered-bus since the order is determined at the time when the messages are placed on the bus, which means that the update is always ordered before the acknowledgement.

5.2 Reducing Acknowledgements

The second objective for optimizing the original CHI coherence implementation was to attempt to remove as many of the final acknowledgements as possible. Here, we are strictly discussing the acknowledgements sent by a requestor at the end of a transaction. The motivation for this is to reduce the latency of communications within the interconnect, similar to the optimisations made in section 3.

This is easily done for the ordered-bus, as the directory provides serialization which is then enforced by the order of which messages are placed onto the bus. This protocol has been verified and included within the provided library. The question then remains of whether it is possible to produce such a protocol when using an unordered-bus.

Hypothesis 2 (H2): *It is possible to remove the additional acknowledgements at the end of all transactions when using an unordered-bus, so long as messages are linearized at the directory and cache.*

This is inferred from the directory being able to linearize messages even without these acknowledgements due to the required update message from the requestor. Though this seems to be a reliable method of handling message ordering, it is also required

for us to rename messages as typically done by the ProtoGen engine [31]. This is necessary for ensuring message ordering is made clear at the cache, so that correct order of transitions and communications can be discerned from the order that messages have been linearized by the directory.

It must be stressed that some of the functions of automation provided by ProtoGen had been disabled due to the immense state size of these protocols. Therefore, it is left to us to manually handle the renaming of messages where necessary.

The rest of this section presents a detailed explanation on the concurrency related issues that arise when trying to develop such a protocol, and presents our solutions as mentioned briefly above.

The difficulties of this optimisation are best explained by returning to the original ordered-bus implementation. When the final acknowledgement is removed, so too is the stall at the directory after receiving the update from the owner. Therefore, the directory will immediately be able to proceed with future requests, even before the data reaches the requestor. The issue here, is that a future request may result in a forward-snoop to the requestor discussed in the previous context. The requestor may then receive the forward-snoop before receiving any data from the previous transaction. A message flow diagram is given in Figure 5.2.

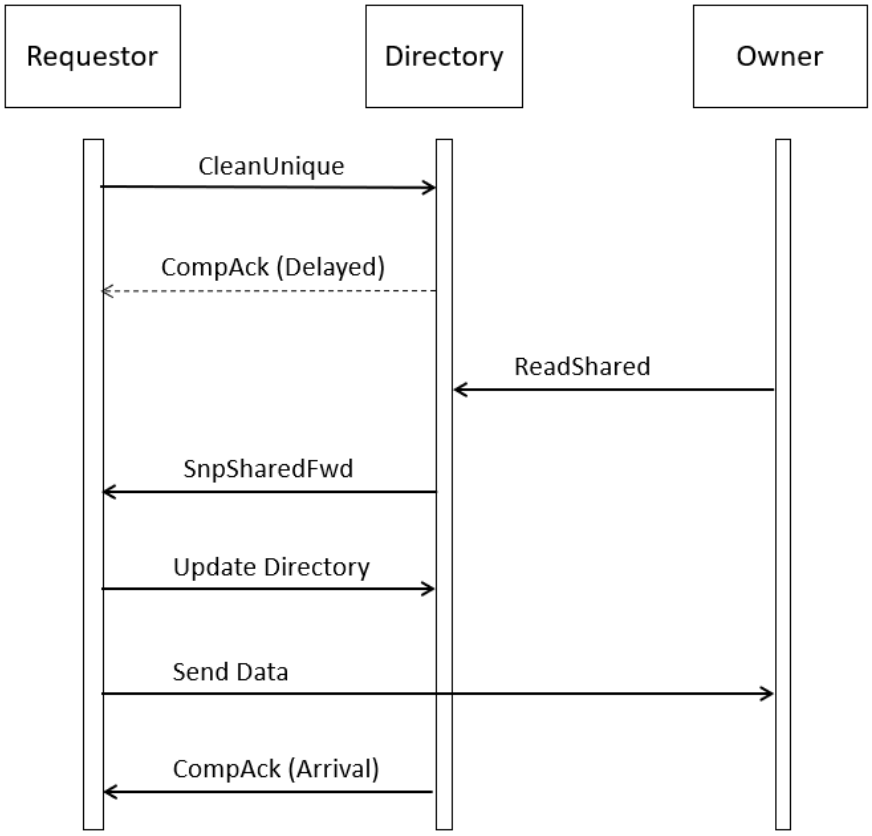

Figure 5.2: Message Flow of Delayed Completion Acknowledgement on Unordered-Bus.

This is not an issue for the ordered-bus since messages are linearized when they get

placed on the bus. This is however, clearly an issue when using an unordered-bus since the messages may be processed out of order. The objective is to then find a way to enforce the ordering of these messages in some other manner.

The devised solution involves attaching the directory state to particularly hazardous forward-snoop requests. For example, a SnpSharedFwd from a directory in M state has been manually renamed to SnpSharedFwd_M. This additional information is enough to identify whether a snoop occurs before or after data was sent to a requestor who was previously in O state. If the directory state is that of the state the requestor would have entered, then it can be inferred that the snoop was ordered after the data or completion response. This means that the recipient of the snoop is able to determine the correct state to enter upon processing received messages. The state transitions at the requestor in the described scenario are modelled in Figure 5.3.

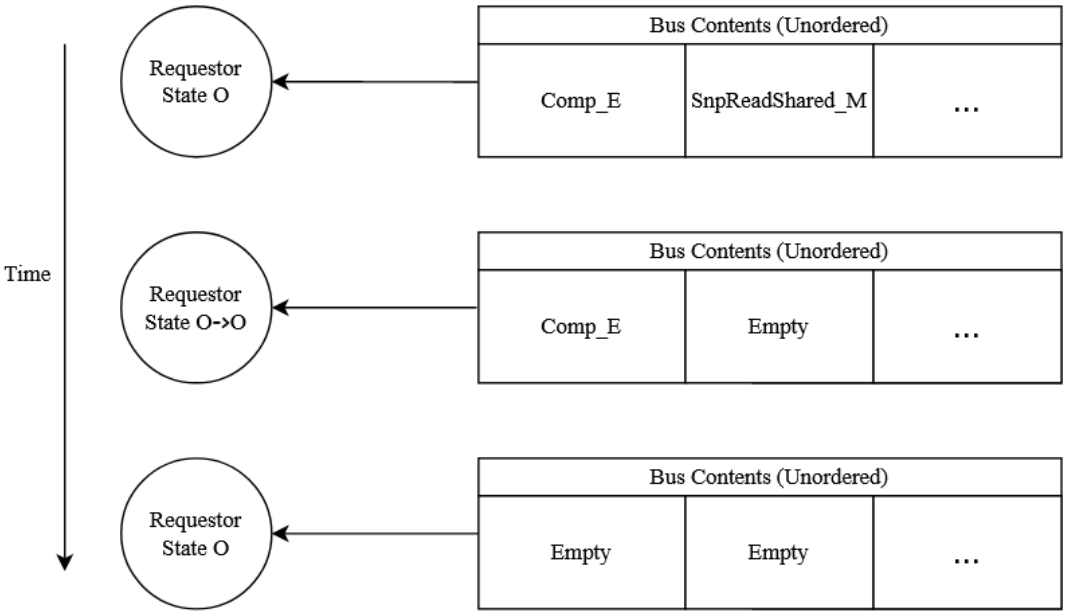

Figure 5.3: State Transitions at Requestor For Out-of-Order Write Response and Incoming Read Request on an Unordered-Bus.

This method of renaming messages has been used to handle multiple transactions within the revised SSP for the unordered-bus model, and allows for the complete removal of the additional acknowledgements. Correctness has been guaranteed by using the same Murφ invariants as described in previous sections. Therefore we have proven our hypothesis to be true.

5.3 Summary

In this short chapter, we have shown attempts at optimising our original CHI protocol implementation. These attempts were successful in removing the ordered-bus constraint, and reducing the latency for multiple forms of communication.

These designs have been implemented using the ProtoGen tool, and the generated models are included within the produced library. Correctness has been guaranteed using the same method as described in 4.7, and so our objectives for implementing optimisations have been completed.

Chapter 6

Conclusion

6.1 Summary

This project initially began with the goal of simply implementing the models given in the primer written by Nagarajan et al [39]. This was a stepping stone in providing the foundation for our coherence protocol library. From this point we then moved on to experimenting with variations on the forward messaging models. These simple conceptual modifications required a large amount of time to ensure correctness due to the difficulties of concurrent communications. The end of this stage signalled completion of a major part of this project, yet there was still so much more that could be done.

Next the decision was made to pursue existing industry specifications, and hence leading to including the CHI coherence protocol in our library. The design and implementation of the CHI protocol was significantly more difficult than was originally expected. The specification was at times ambiguous, and the magnitude of options presented lead to debate over what could even be considered an implementation of this protocol. After finally having a concrete vision and a design of the protocol created, we were able to model the complete protocol with transient states, and generate a verified model.

This then lead to the final part of the project. As the initial designs of the CHI protocol had visible limitations, it was felt that attempts should be made to optimise our implementation. This involved the removal of unnecessary acknowledgements and other adaptations to permit an unordered-bus to be used. These changes lead to unexpected bugs in the generated models due to new concurrent operations that were previously not considered. After clearing up these issues, this signalled the end of the project.

As the goal of the project was to provide a collection of verified cache coherence protocols, it is our belief that we have more than achieved that goal. We have developed a better understanding of the CHI coherence protocol, and even provided fully verified protocols with additional optimisations. This library is now available for future usage, and will hopefully see further extensions in the near future.

6.2 Future Work

Cache coherence is a topic that is still very relevant to the design of new systems and is also open to a myriad of different research opportunities. Continuing from the work presented here, there are a few key areas of interest that we believe would be suitable for future projects.

Since coherence protocols are intended to operate as efficiently as possible, it would be worthwhile to transfer the generated models into a simulation suite. Simulation has often been a topic of research for cache coherence [4, 37, 24] however no such study has had as large a library of verified protocols to compare. This could expose more interesting behaviour within these protocols, as well as clearly defining the benefits of each coherence protocol against differing workloads.

Another similar route would be to apply hardware synthesis to the models in our library. Similar work has been done by various authors in order to perform evaluation on coherence protocols using FPGA's and other hardware [35, 10, 1]. Simulations can often hide performance details of protocols as they cannot perfectly model the true physical system. Therefore, this route becomes necessary for fully testing and comparing realized designs.

Additionally, it would be interesting to continue to model other existing industry specifications. The current architecture focus on hardware accelerators means that cache coherence is becoming prevalent once more in the design of these coherent interfaces. Investigating these interconnects could lead to new insights on current trends, as well as providing more resources for academic and industrial research. Our initial research hints that the work done by the OpenCAPI [34, 16] and Gen-Z [13, 11, 12] groups could be a logical next step in this direction.

Appendix A

Supplied Invariants

```
  invariant "exclusive store check"
    forall a:Address do
        forall m1:Machines do
        forall m2:Machines do
        ( m1 != m2
          & MultiSetCount(i:g_perm[m1][a], g_perm[m1][a][i] = store) > 0)
        ->
          MultiSetCount(i:g_perm[m2][a], g_perm[m2][a][i] = store) = 0
        endforall
        endforall
    endforall;

invariant "store excludes load check"
    forall a:Address do
        forall m1:Machines do
        forall m2:Machines do
        ( m1 != m2
          & MultiSetCount(i:g_perm[m1][a], g_perm[m1][a][i] = store) > 0)
        ->
          MultiSetCount(i:g_perm[m2][a], g_perm[m2][a][i] = load) = 0
        endforall
        endforall
    endforall;
```

Figure A.1: ProtoGen Automatically Generated Invariants

```
invariant "Single-Writer-Multiple-Reader (SWMR)"
    forall c1:cacheIndexType do
    forall c2:cacheIndexType do
      ( c1 != c2
      & (caches[c1].state = cache_M |
         caches[c1].state = cache_E |
         caches[c1].state = cache_UCE ))
      ->
      ( caches[c2].state != cache_M &
        caches[c2].state != cache_E &
        caches[c2].state != cache_UCE &
        caches[c2].state != cache_O &
        caches[c2].state != cache_S)
    endforall
    endforall;

-- Domain specific knowledge
invariant "[Model opt.] M, E, I, and UCE imply empty sharers list"
   forall dir:DirIndexType do
      ( Dirs[dir].state = Dir_M
        | Dirs[dir].state = Dir_E
        | Dirs[dir].state = Dir_I
        | Dirs[dir].state = Dir_UCE
      )
      ->
      ( MultiSetCount(i:Dirs[dir].sharers, true) = 0 )
    endforall;

invariant "[Model opt.] S or I implies undefined owner"
     forall dir:DirIndexType do
      ( Dirs[dir].state = Dir_S
        | Dirs[dir].state = Dir_I )
      ->
      ( IsUndefined(Dirs[dir].owner) )
    endforall;
```

Figure A.2: Manually Defined Invariants For CHI Protocol

Bibliography

[1] D. Abts, S. Scott, and D. J. Lilja. So many states, so little time: verifying memory coherence in the cray x1. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 10 pp.–, 2003.

[2] Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996.

[3] Brian Klug Anand Lal Shimpi. Samsung updates exynos 5 octa (5420), switches back to arm gpu. https://www.anandtech.com/show/7164/samsung-exynos-5-octa-5420-switches-back-to-arm-gpu.

[4] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems (TOCS)*, 4(4):273–298, 1986.

[5] ARM. The arm corelink dmc-620 dynamic memory controller. https://developer.arm.com/ip-products/system-ip/memory-controllers/corelink-dmc-620.

[6] ARM. The arm cortex-a65ae. https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a65ae.

[7] ARM. The arm cortex-a76. https://developer.arm.com/ip-products/processors/cortex-a/cortex-a76.

[8] ARM. Amba 5 chi architecture specification. *Issue D*, August 2019.

[9] ARM. Amba 5. https://developer.arm.com/architectures/system-architectures/amba/amba-5, 2020.

[10] Ch Attada Sravanthi, Rajasekhara Rao, K Krishnam Raju, and L Rambabu. Implementation of mesi protocol using verilog. 2019.

[11] Matheus Cavalcante, Andreas Kurth, Fabian Schuiki, and Luca Benini. Design of an open-source bridge between non-coherent burst-based and coherent cache-line-based memory systems. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*, pages 81–88, 2020.

[12] Gen-Z Consortium. Core specification version 1.1, 2020.

[13] Gen-Z Consortium. Gen-z about section. https://genzconsortium.org/about-us/, 2020.

[14] Intel Corporation. An introduction to the intel quickpath interconnect. *Document Number 320412–001US*, 2009.

[15] David L Dill. The mur ϕ verification system. In *International Conference on Computer Aided Verification*, pages 390–393. Springer, 1996.

[16] Johan De Gelas. Opencapi unveiled: Amd, ibm, google, xilinx, micron and mellanox join forces in the heterogenous computing era. `https://www.anandtech.com/show/10759/opencapi-unveiled-amd-ibm-google-more`, 2016.

[17] James R Goodman. Cache consistency and sequential consistency. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1991.

[18] Mark Hachman. Amd phenom, barcelona chips hit by lock-up bug. `https://www.extremetech.com/extreme/81580-amd-phenom-barcelona-chips-hit-by-lockup-bug`.

[19] Blake A Hechtman and Daniel J Sorin. The limits of concurrency in cache coherence. In *Proceedings of the Workshop on Duplicating, Deconstructing and Debunking (WDDD'12)*, 2012.

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

[21] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The art of multiprocessor programming*. Newnes, 2020.

[22] Mark D Hill. Multiprocessors should support simple memory consistency models. *Computer*, 31(8):28–34, 1998.

[23] Mark Horowitz and William Dally. How scaling will change processor architecture. In *2004 IEEE International Solid-State Circuits Conference (IEEE Cat. No. 04CH37519)*, pages 132–133. IEEE, 2004.

[24] Luma Fayeq Jalil, Maha Abdulkareem H Al-Rawi, and Abeer Diaa Al-Nakshabandi. Cache coherence protocol design and simulation using ies (invalid exclusive read/write shared) state. *Baghdad Science Journal*, 14(1), 2017.

[25] Pete Keleher, Alan L Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. *ACM SIGARCH Computer Architecture News*, 20(2):13–21, 1992.

[26] Silvia Lametti. Cache coherence techniques. *Dec*, 1:39, 2010.

[27] James Laudon and Daniel Lenoski. The sgi origin: a ccnuma highly scalable server. *ACM SIGARCH Computer Architecture News*, 25(2):241–251, 1997.

[28] Albert Meixner and Daniel J Sorin. Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. *IEEE Transactions on Dependable and Secure Computing*, 6(1):18–31, 2008.

[29] David Mosberger. Memory consistency models. *ACM SIGOPS Operating Systems Review*, 27(1):18–26, 1993.

[30] Kyriakos Katsamaktsis; Dr. Marco Elver; Dr. Vijayanand Nagarajan. Roc3. `https://github.com/icsa-caps/roc3`, 2017.

[31] Nicolai Oswald, Vijay Nagarajan, and Daniel J Sorin. Protogen: automatically generating directory cache coherence protocols from atomic specifications. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 247–260. IEEE, 2018.

[32] David A Patterson and John L Hennessy. *Computer Organization and Design ARM Edition: The Hardware Software Interface*. Morgan kaufmann, 2016.

[33] Per Stenstrom. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, 1990.

[34] Jeffrey Stuecheli, Bart Blaner, CR Johns, and MS Siegel. Capi: A coherent accelerator processor interface. *IBM Journal of Research and Development*, 59(1):7–1, 2015.

[35] Taeweon Suh. *Integration and Evaluation of Cache Coherence Protocols for Multiprocessor SOCS*. PhD thesis, Georgia Institute of Technology, 2006.

[36] Thomas N Theis and H-S Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41–50, 2017.

[37] Milo Tomasevic and Veljko Milutinovic. A simulation study of snoopy cache coherence protocols. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, volume 1, pages 427–436. IEEE, 1992.

[38] Alexander Wolfe. Amd's quad-core barcelona bug revealed. `https://www.informationweek.com/it-leadership/amds-quad-core-barcelona-bug-revealed/d/d-id/1062350?`

[39] Vijay Nagarajan; Daniel J. Sorin; Mark D. Hill; David A. Wood;. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2020.