Implementing Cache Coherence through Manager-Client Pairing

A Project Report

submitted by

REENA E

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

June 2016

THESIS CERTIFICATE

This is to certify that the thesis titled **Implementing Cache Coherence through Manager-Client Pairing**, submitted by **Reena E**, to the Indian Institute of Technology, Madras, for the award of the degree of **MASTER OF TECHNOLOGY**, is a bona fide record of the research work done by her under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. V Kamakoti
Project Guide
Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 21^{st} June 2016

ACKNOWLEDGEMENTS

I would like to express my earnest gratitude to my adviser Prof.V.Kamakoti whose energy, passion and support has been a tremendous source of inspiration throughout my project. I would like to extend my sincere thanks to my co-adviser G.S.Madhusudhan for guiding me throughout the project. I would like to express my special thanks and deepest gratitude to Rahul Bodduna for his valuable suggestions and guidance throughout the project. Finally, I would like to extend my thanks to all my friends because of whom my graduate experience has been one that I would cherish forever.

ABSTRACT

Multi-core processors dominate the microprocessor industry as the scaling of single core processor performance is rapidly reaching saturation. Designing multi-core processors presents various new challenges. Especially, the design of memory subsystem is a huge challenge because of the complications involved in coherence management. As the cores become more numerous and more diverse, heterogeneous hierarchical coherence protocols are required. Hence, the design, verification and evaluation of advanced memory subsystems become very difficult.

This dissertation uses the Manager-Client pairing method as an attempt to overcome these challenges and enable rapid construction of coherence hierarchy for multi-core processors. Manager-Client pairing provides a standardized coherence communication interface that provides protocol encapsulation. This reduces the complexity in the designing and verification of hierarchical coherence protocols. Each tier can be verified and evaluated in isolation. As a result, bug-free design with protocol heterogeneity can be created without much difficulty. This enables the architects to focus on performance enhancement rather than on debugging and verifying correctness of coherence implementation. The implementation of flat MOESI protocol using Manager-Client pairing and Tilelink protocol is presented in this dissertation. This can be replicated to construct the coherence hierarchy tiers. The memory hierarchy constructed is integrated with the I-class processor of Shakti processor series and the functional correctness of the design is verified.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

MCP	Manager-Client Pairing
BSV	Bluespec System Verilog
HDL	Hardware Description Language
ASIC	Application Specific Integrated Circuit
FPGA	Field Programmable Gate Array
SOC	System on Chip
RTL	Register Transfer Level
IP	Internet Protocol
CPU	Central Processing Unit

CHAPTER 1

Introduction

In the recent years, there has been a shift in focus towards multi-core processors due to the limitations of performance-scaling in single-core design. Parallelism in terms of multi-threading in multi-core processors improve performance by faster execution, better system utilization and lower power consumption. Cache hierarchy is often used with multi-core processor design in several applications for optimal performance. The use of shared memory presents numerous challenges. With multiple caches sharing single memory, data traffic becomes huge and it may present a performance bottleneck. Hierarchical clustered cache design is one possible solution to this problem. Grouping cores and their caches in clusters reduces network congestion by localizing traffic among several hierarchy levels, potentially enabling much higher scalability.

In a shared memory system, cache coherence is required for consistent view of memory across all the cores. As the cores become more diverse, there is a need for sophisticated coherence management. In hierarchical clustered cache design, each cluster may operate with a different coherence protocol depending on its application. With multiple levels of cache, this leads to a hierarchy of coherence protocols. New states have to be introduced to manage multiple coherence protocols in single design. The complexity of the design increases exponentially with increase in number of coherence protocol states, which makes the testing and verification of the design very difficult. This limits the scalability of the shared memory design.

The challenges in designing, testing and verifying advanced hierarchical cache coherence protocols can be overcome by the use of standardized coherence communication interface that provides protocol encapsulation. This dissertation presents the implementation of this standardized interface by adapting the Manager-Client pairing technique.

1.1 Cache Coherence

In a shared memory environment, the private processor caches may contain copies of data which may be dirty with respect to main memory. Cache coherence handles the management and distribution of data in such cases. Without coherence, the consistency model of architecture could be violated. That is when a private processor cache commits a store which is not being observed in the local cache of other processors, it breaks the consistency among the copies of data stored in the private processor caches. This affects the fundamental way in which the processors communicate with each other in a shared memory environment. The loads and stores performed by every processor should be observed by every other processor for functional correctness and cache coherence ensures that this behavior is maintained.

Cache coherence in hardware is accomplished through the addition of state bits to the data in cache, which indicates the coherence state of the data. The coherence state associated with the data depends on the coherence protocol used. The basic coherence states are invalid, shared and modified. Invalid state indicates whether the copy of data in cache is valid or not. Shared state of data implies that one or more of other caches also contain the copy of the same data. When a private processor cache commits a store, that copy of data is stored in modified state. These provide the basic mechanisms required to maintain coherency in the caches. A processor can read from its local cache only if the data is valid. Also, when a local copy of data is written by the processor, all other shared copies are invalidated. And the copy written is put in modified state. The two major classes are coherence protocols are broadcast based protocols and directory based protocols. In the directory based coherence management, the coherence state of all the data in the local caches is maintained in a directory. Whenever a processor has to perform read or write, the request is sent to the directory and the operation is performed if the required permissions are granted by the directory. In a broadcast based protocol, read or write miss is broadcasted through a shared bus and every local cache is snooped to check if the requested data is present in them. Also, when a processor performs a store on a shared copy of data in its local cache, invalidation request is sent to other caches to invalidate the other copies of that data. The invalidation of the shared copies in the event of a write operation by the processor is done in invalidation based

protocols. There is another variant of coherence protocol which handles write-hit in a different manner. Instead of invalidating the shared copies, the written value is broadcasted to all the caches that share this data and they are updated with the new value. This protocol is referred to as the update-based protocol.

The choice of a coherence protocol primarily depends on the application. Different protocols can be best suited for different hardware implementations. In this dissertation, broadcast based MOESI coherence protocol is used.

1.1.1 MOESI Protocol

As already mentioned, we will be using broadcast based MOESI protocol, which will handle write-hit by invalidation. Initially, when the data is fetched from the memory, it is stored in Exclusive(E) state, as it will be the only copy of that data among the caches. When this data is read by some other cache, it is no longer an exclusive copy. So, the state is changed to Owner(O) and the cache which read this data will store it in Shared(S) state. When the data in Shared(S) state is read by other caches, its state does not change.

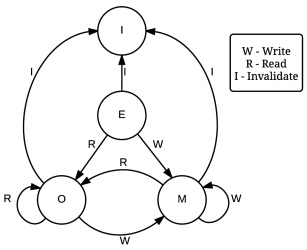

Figure 1.1: State transition beginning with data in E state

Reading of data in Owner(O) state, by other caches does not change the state of the local copy. However, writing of data in Owner(O) state or Shared(S) state, by the processor, changes its state to Modified(M) and the invalidate request is sent to other shared copies. Writing of data in Modified(M) state, will cause it to remain in modified state. When data in Exclusive(E) state is written, its state is updated as Modified(M), but invalidate request is not sent out as there are no shared copies to invalidate. The data in any coherence state, when invalidated, is put in Invalid(I) state. This transition of coherence states is shown in Figures 1.1 and 1.2.

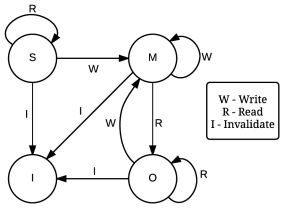

Figure 1.2: State transition beginning with data in S state

1.2 Manager Client Pairing

Manager-Client pairing(MCP) is a method used to resolve the issues of heterogeneous integration of hierarchical coherence protocols. It provides a standardized interface definition, to rapidly construct and integrate coherence hierarchies. This reduces the complexity of the hierarchical coherence design by taking the native functioning of a given component protocol and mapping these to corresponding MCP methods. Therefore, the details of the component protocol's implementation gets abstracted away from the composition of the hierarchy, hidden behind an interface layer.

A coherence realm in the memory hierarchy consists of a single upper interface at

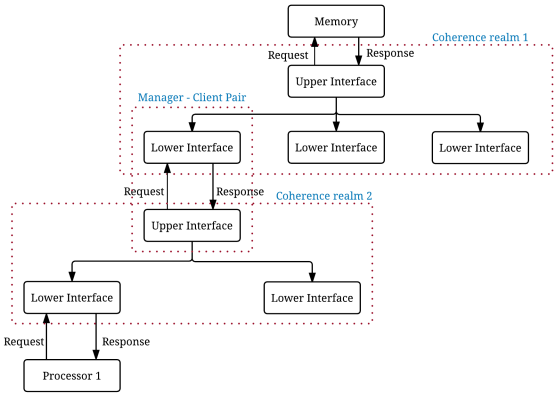

Figure 1.3: Construction of coherence hierarchy using MCP

the Manager and multiple lower interfaces at the Clients as shown in Figure 1.3. The interfaces are designed such that the upper interface of one coherence realm can plug into the lower interfaces of any other coherence realm without any additional modification, and creating Manager-Client pairs at the junction points. This allows us to construct hierarchies with ease. The responsibility of coherence management is localized in the hierarchical tiers. The verification and validation can be done in isolation in all of these component parts of the coherence hierarchy. The processor and the main

memory can be mapped seamlessly to the dangling interfaces of the MCP, completing the memory hierarchy.

1.3 Organization

The remainder of the dissertation is arranged as follows:

Chapter 2 presents the Manager-Client pairing framework.

Chapter 3 presents a brief overview of Bluespec and its features. Bluespec is the HDL used to design the memory hierarchy.

Chapter 4 provides the implementation details of MCP.

Chapter 5 presents the conclusion and possible future work.

CHAPTER 2

Manager-Client Pairing

As we move towards multi-core processors, there is an increasing need for sophisticated coherence management. When cache hierarchies are used in the design, we also need to introduce coherence management hierarchies to achieve data consistency. The hierarchical coherence protocol based design increases the complexity in terms of the ease of designing, testing and verification. Also, this complexity increases exponentially with the addition of more coherence states in the protocol. This in turn, strains the interaction between the hierarchy tiers.

Manager-Client Pairing (MCP) is a method to design hierarchical coherence protocols by formally defining and limiting interactions between different levels of the hierarchy. This enables composition in the coherence hierarchy design, which makes the designing of several complex designs much faster by restricting the complexity of design to a small set of protocol component parts.

2.1 Problem

During the recent years, there has been a shift in focus towards multi-core processors because of the advantage it has over single-threaded designs in terms of power and performance. The addition of cache hierarchies in multi-core processors makes the process of fetching data from the memory faster compared to having single level of cache. However, cache hierarchy presents a problem of scalability, mainly due to the burden of coherent data management.

Data coherence management is a major challenge in hardware design with multiple cores. It presents a performance bottleneck as there are limits in terms of scalability presented by both directory based and broadcast based coherence protocols, thus requiring hierarchical coherence protocol design to overcome this. To achieve this, a

flexible framework is required, which supports variable width and depth of hierarchy. Figure 2.1 shows a model of coherence hierarchy.

The current approach to build such a framework is to design a glue layer to tie low-level

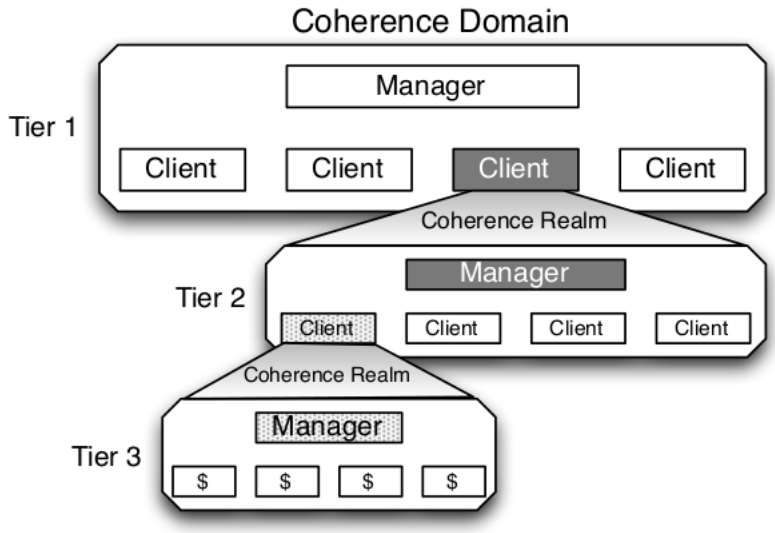

Figure 2.1: Coherence Hierarchy

coherence protocols together. This often results in changes to the low-level protocols. A complex sub-state replication is required which encodes all hierarchy information into protocol state. Therefore, more states need to be managed which leads to a complex state machine. Also, making changes to the hierarchy is difficult as a new solution has to be developed for each change. As a result, there is an abundance of large, complex, inflexible and highly specialized coherence protocols.

2.2 Manager-Client Pairing

Manager-Client pairing(MCP) is a powerful new approach to design coherence hierarchies. MCP employs a client-manager model, which gives a simple coherence protocol interface for clear communication between the entities that use data(clients) on one side and the mechanisms that maintain the coherence of the users(managers) on the other side of the interface. This encapsulates each tier of the coherence hierarchy so that each component can be dealt with in isolation. Application of MCP solves the state-space explosion problem and reduces the complexity in designing hierarchical coherence architecture. As the interface is standardized with independent tiers, a multi-core pro-

cessor design can be partitioned in to arbitrarily deep hierarchies to apply MCP, which ensures rapid design of coherence hierarchies.

It precisely defines the functions to be carried out by the Client and Manager to maintain coherence and reduces the fundamental requirements of a coherence protocol into a modular and generic set of base functions. Based on these base functions, coherence protocol communication is standardized by the generic Manager-Client interface. Thus, by converting stand-alone coherence protocols into coherence-tier building blocks, MCP aids in the rapid development of multi-tier coherence hierarchies.

Cache coherence protocols are responsible for maintaining a consistent view of memory across all the caches of a coherence domain, in a shared memory environment. The responsibilities of the coherence protocol include upgradation or degradation of coherence state of data and deciding how updates to data are propagated through the system. These functions can be divided between two kinds of agents:managers that manage propagation of updates to data and clients which hold the coherence state of data and performs data acquisition when necessary.

These roles are first formally defined for non-hierarchical MOESI protocol and then re-examined to derive interfaces that enable composition of complex coherence hierarchies.

2.3 Basic Functions of MCP

The coherence protocol used here is MOESI protocol. It has five states: Modified(M), Owner(O), Exclusive(E), Shared(S) and Invalid(I). The client which has the data in the Exclusive state or the Modified state has the only copy of that data. Although the client with data in Owner state does not have a unique copy of the data, it is the only client among the sharers, which can respond to a request. Also, at any point of time, only one of these states(M,O,E) can be present among the clients of the same level. Therefore, data propagation can be easily handled by the manager as it can directly grant data from the only client in M/O/E state to the client which requested that data block.

Let us consider the handling of write operation in MOESI protocol. After write, the copy of the data block which is written, becomes dirty with respect to the sharers. In order to track this information, the Modified(M) state is used by the client. In the event of a write hit in the processor, the client changes the state of that data to Modified(M) and sends an invalidate request if the previous state of the data written was Owner(O) or Shared(S) state. During write-back, if the data block is in M/E state, it is directly written in the main memory. If in Owner(O) state, in addition to sending the data to main memory, all the other shared copies are invalidated. If the data is in Shared(S) or Invalid(I) state, the write-back operation is not performed. Table 2.1 summarizes and enumerates a comprehensive list of the base functions required for communication between processors, clients, managers and memory in MOESI protocol. These will be used as an aid in the developing a generic protocol interface.

The basic requirements that have to be satisfied by the agents involved in coherence

Table 2.1: Comprehensive list of the base functions required for communication between processors, clients, managers and memory in MOESI protocol

Origin Agent	Action Type	Action	Description	Destination Agent	Response Action
Processor	Data Acquisition	GetData	R/W miss: Get data to complete CPU request	Client	GetData
	Invalidation	FwdInval	Write hit: Invalidate shared copies	Client	FwdInval
	Data Supply	DoWrite	Block replacement; writeback	Client	DoWrite
		GrantData	Supply Data	Client	GrantData
Client	Data Acquisition	GetData	Request from processor	Manager	GetData
		GetData	Request from manager; change state to O	Processor	GrantData
	Invalidation	DoInval	Invalidate copy in local cache if hit in O/S state	Processor	CompleteInvalidate
		FwdInval	Change state to M; forward request to manager if O/S state	Manager	DoInval
	Data Supply	DoWrite	Only when in M/O/E. Invalidate shared copies when O	Manager	DoWrite; DoInval
		GrantData	Got data from processor	Manager	GrantData
		GrantData	Got data from manager; State – E if from mem, S otherwise; send data to processor	Processor	Complete R/W; when write, send
Manager	Data Acquisition	GetData	Gets data from client if it is in M/O/E state	Client	GetData
		GetData	If no hit in any client	Memory	GrantData
	Invalidation	DoInval	Invalidate shared copies	Client	DoInval
	Data Supply	GrantData	Data supply from M/O/E clients or from main memory; indicate if data obtained from main memory	Client	GrantData
		DoWrite	Forward request to higher level	Memory	GrantWrite
Memory	Data Supply	GrantData	Supply Data	Manager	GrantData
		GrantWrite	Finish write	N/A	N/A

management are as follows. Clients should be able to respond whether or not they have the requested data block. If the cache associated with a client encounters a read or write miss, the client should be able to place a request to the manager. Managers should accept data requests from clients, and provide data from the appropriate location. The client should also be able to send out invalidate request when necessary and the manager should forward it to all the other clients in the same level. In case of write back, the clients and managers should forward the data till the main memory.

As the management of coherency is completely handled by the client, the manager need not be aware of any internal change of coherence states. The manager only has to keep track of whether the data is obtained from the clients of the same level or from higher level. This information is used by the clients to appropriately update the coherence states.

2.4 MCP Interface for Coherence Hierarchy Construction

Now, we need to construct coherence hierarchy. From the basic functions performed by the agents of coherence realm, as inferred from Table 2.1, considerable similarities in the interactions between processor and client and the interactions between manager and memory can be observed. In both cases, the data is provided by the data supplier when requested. The mechanism of data transfer is same as that in main memory when the data is requested from the lower levels of the memory hierarchy. This information allows us to create interfaces that allow recursion, which can be used to develop hierarchies.

By examining the Figure 2.2, we observe that the replacement of the implementation details of the coherence protocol with a black box yields a self-similar upper and lower interface. Not only does this insight enable recursion through a simple interface definition, but also allows encapsulation of the coherence protocols used in the hierarchy, reducing design complexity.

From this we can see that there are at least two necessary components to the MCP

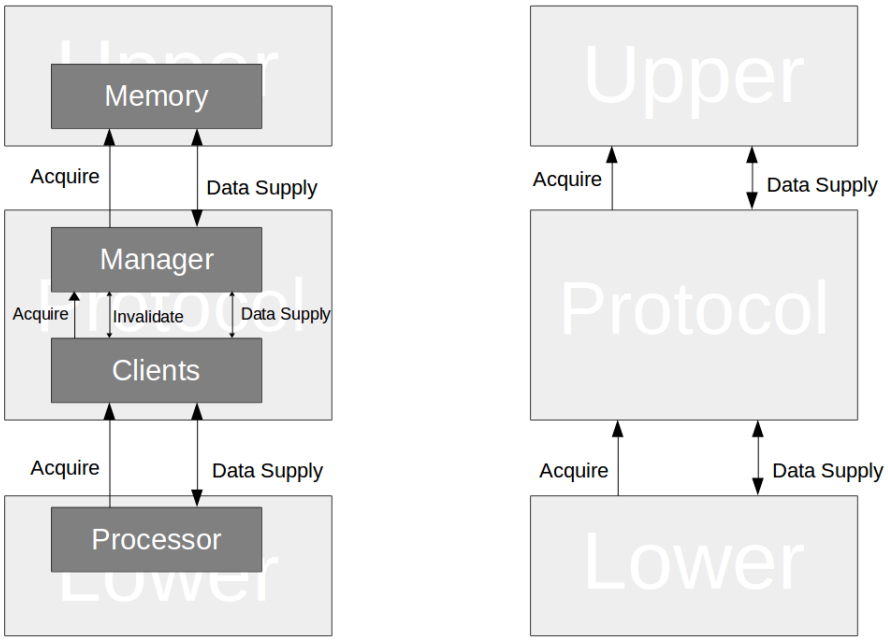

Figure 2.2: Inclusion of Protocol layer in the memory hierarchy

interface: R/W request for data from lower to higher level(write during write-back from processor), and Data supply from higher to lower as well as from lower to higher levels. Manager is paired with a client in the next higher-up client(or to the memory in the highest level). By requesting the paired client when necessary, the manager can perform data transfer without having to worry about the coherence protocol in the higher level. The mechanism of data transfer can be the same as that in main memory, irrespective of the level of memory hierarchy being communicated with. Similarly, the clients of the lower level handle the coherence management silently, and the manager need not be aware of any change in the coherence state within the clients. This is much like how a processor is unaware of the coherence in the caches implemented; it simply requests for data from higher level and receives it.

The other important detail that must be addressed is the propagation of invalidation signal. When a write-hit happens in a processor, the other shared copies of the data have to be invalidated. As the client handles all the coherence management, the processor notifies the client whenever there is a write-hit. The client forwards that request to the manager, if necessary after updating the current coherence state of the data block being written. The manager then has to forward this request to every other client in that

level, so that all the shared copies could be invalidated. Similarly when a client receives invalidate request from the manager, it should invalidate its local copy if it has that data. This provides us the third and final component of the MCP interface: Invalidation. Figure 2.3 outlines a brief description of the working of MCP interface.

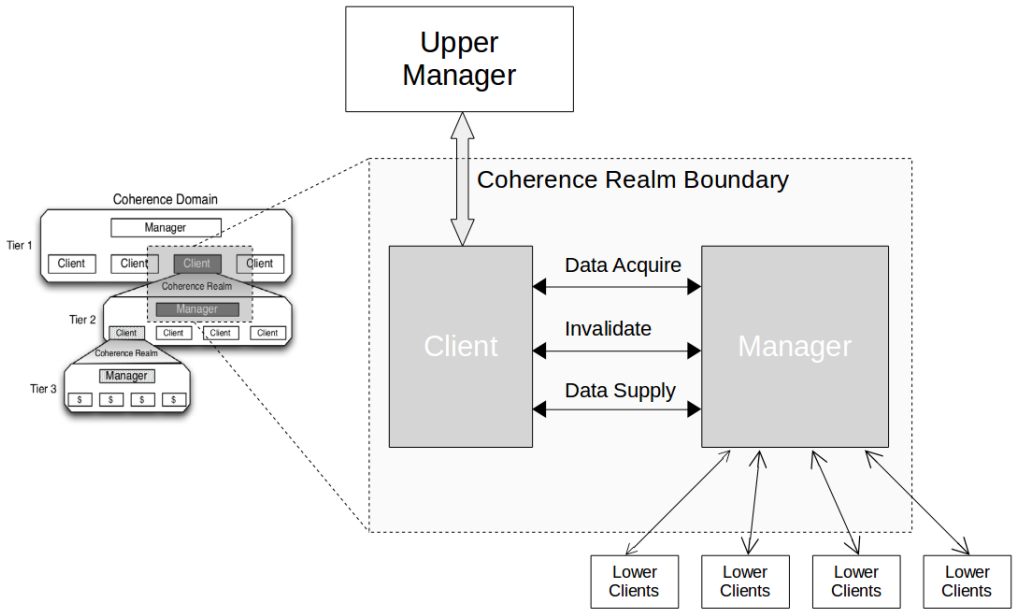

Figure 2.3: Interface between Manager and Client

2.5 Working of MCP

With a common interface defined, we can begin using coherence protocol agents as building blocks in the construction of hierarchical coherence protocols. By expanding the scope of client agents to also monitor coherence realms in addition to processor caches, the coherence effort can be distributed over several protocols by layering the protocols in a tiered fashion. When manager of one coherence realm cannot satisfy the request of one of the clients of lower level, it recursively sends request to the manager of the higher level, through its paired client. In this way, hierarchies of coherence protocols are established. Figure 2.4 explains the algorithm. To aid in understanding and to highlight some important details of MCP, two examples are presented. In both examples we have a top-tier coherence realm A, that implements MEI protocol to manage two lower coherence realms, B and C, both implementing MOESI. Manager A resides

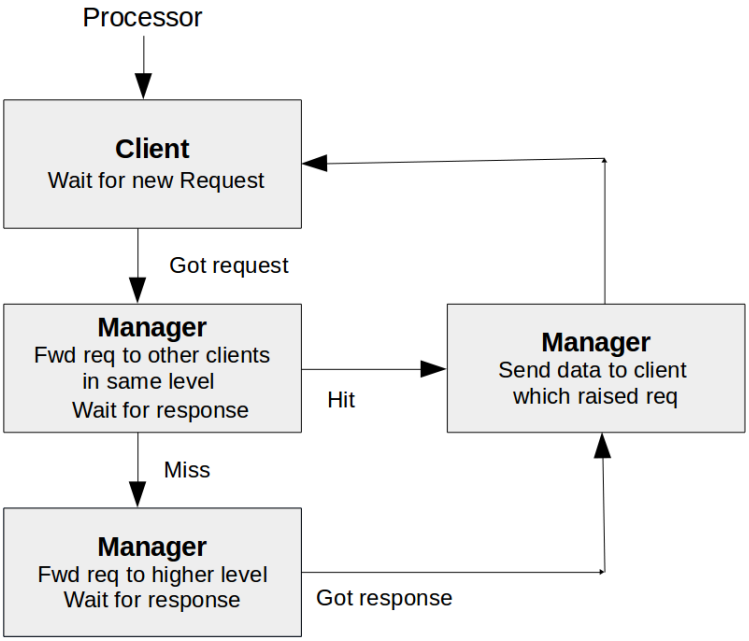

Figure 2.4: MCP State Machine for Data Acquisition

at memory and therefore has no need for a gateway client, being the highest manager agent in the system. Similarly, clients B0, B1 and C0 do not have a matching manager agent because there are no lower tiers to be tracked. They are gateways for processors' private caches, not further coherence realms.

In Figure 2.5, an example of a realm-hit from a read request is shown. The processor below client B0 initiates the sequence with a request to acquire data. The client forwards the request to the manager. The manager checks with its paired client to find out if the data requested is available in any of the other clients of the B level. As the clientA0 represents all the agents of the lower level, it can provide the information that the requested data is available in clientB1. Now the manager obtains data from clientB1, after which the state of it changed to O and it is given to the clientB0 to satisfy the request. As the clientA0 follows MEI protocol, it stores the states of data in both the clients in B level in M state.

From this example we see a clear demonstration of the encapsulation of the coherence realm provided by MCP. The request in the example was serviced only within the scope of coherence realm B because the gateway client A0 had sufficient information to allow the request to proceed in a coherent manner. Furthermore, despite a change in

the state of the coherence realm's manager B from M to O, the change does not need to be reflected in client A0 since it is a silent downgrade. Because there is no need to notify manager A of this activity, there is the benefit of reduced traffic while preserving encapsulation. Additionally, if either client B0 or B1 were to modify the state of this data to M later, it can happen without the knowledge of the agents in the higher level.

Requests can however cross coherence realm boundaries, referred to as a realm-miss,

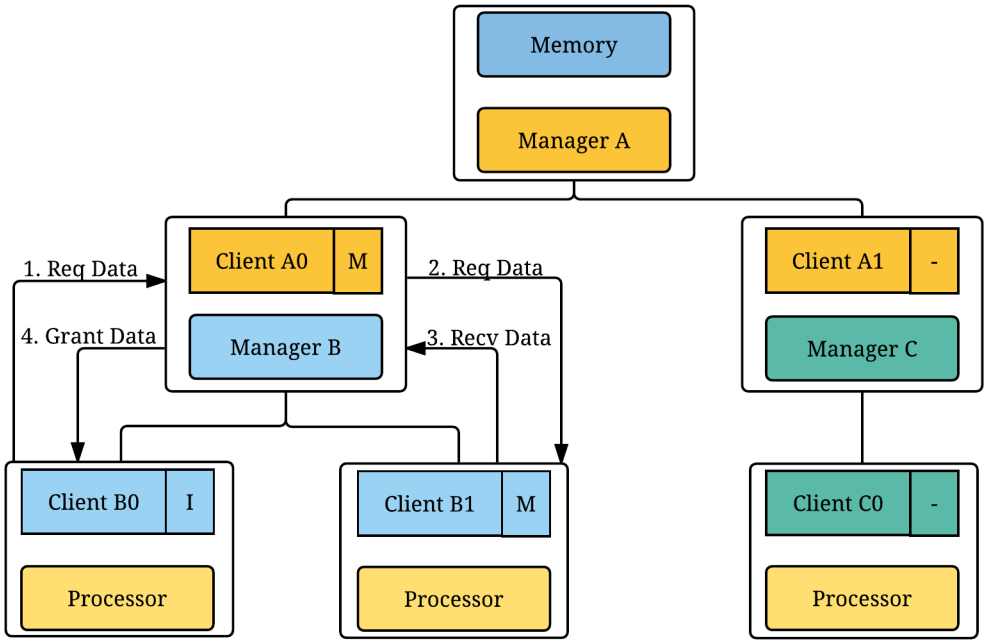

Figure 2.5: Propagation of data acquisition within a coherence realm

when no client agent in that realm has the requested data block, as shown in Figure 2.6. Here the MCP algorithm propagates the request all the way to the top tier where it encounters manager A and memory instead of a client agent. Since there is no higher tier to consult, the top manager always services the request to make forward progress; there is no gateway client at the top level.

In this example, the request is from the ClientC0. Upon requesting the ClientA1, ManagerC gets the information that no client agent in realm C has the requested data block. So, the request is now forwarded to the higher-level, to ManagerA. Now, ManagerA requests ClientA0, and waits for response as there is a hit in realm B. The request propagates to the client which contains the data, through the ManagerB. The data reaches ManagerA through ClientA0 and from ManagerA, the data propagates through ClientA1 and ManagerC to reach ClientC0, which requested the data. Client A1 stores

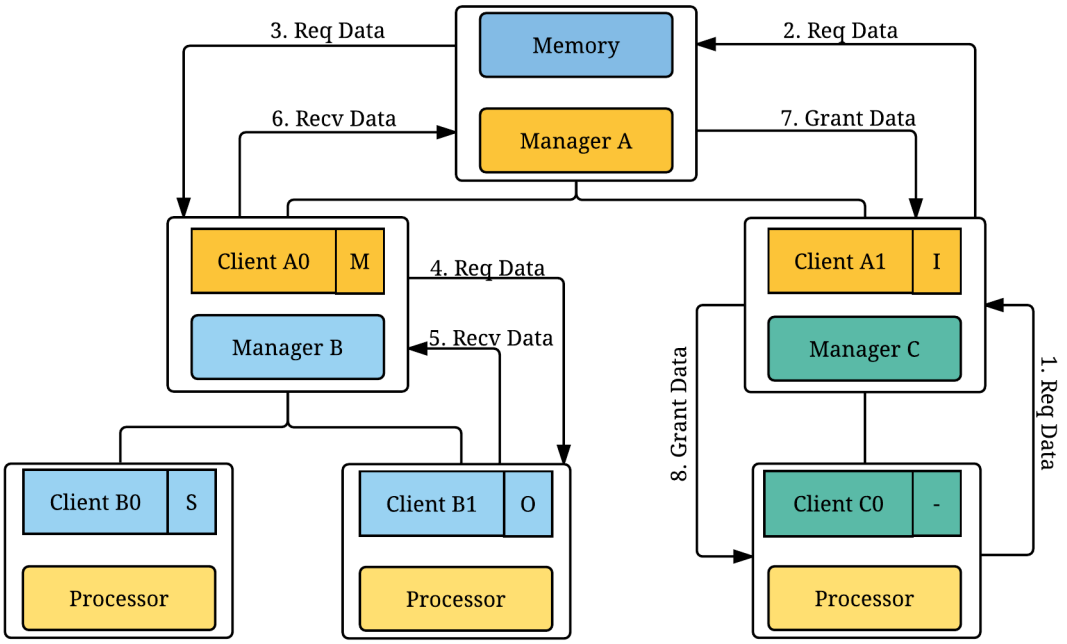

Figure 2.6: Propagation of data acquisition through different coherence realms

the data in the M state as it is not an exclusive copy. ClientC0 stores it in S state as it is a shared copy. The state of data does not change in realm B as the request was serviced by client in O state. Therefore, state of data in ClientA0 also does not change.

Although more complex, this second example further serves to demonstrate the decoupling of the protocol coherence realms from one another. When a gateway client encounters a miss, the entire coherence realm effectively collapses into a single node from the perspective of the manager in the next tier. The next-tier manager does not need to be aware of any details of how the coherence realm guarded by the gateway client operates just as long as it knows how to interact with the gateway client (which obviously it will being the manager). Similarly, when coherence realm C was being updated, this was done opaquely from the perspective of manager A. This coherence realm encapsulation is what enables efficient composition of coherence protocol hierarchies without the need for ad-hoc sub-state replication. Despite the MEI protocol of manager A managing two realms using different protocols (with additional, independent S and O states), the protocol of realm A was never aware of this since it had no need to store information outside its own protocol scope. Furthermore, each component protocol may be validated in isolation.

CHAPTER 3

Bluespec System Verilog

Bluespec System Verilog is a Hardware Description Language (HDL), which is used for specification, synthesis, modeling and verification of ASIC and FPGA design. With a radically different approach to high-level synthesis, bluespec offers significantly higher productivity. It allows designers to express intended hardware through high-level constructs, where all behavior is described as a set of guarded atomic actions.

3.1 Limitations of Verilog

Verilog focusses more on simulation than logic synthesis. The source text of verilog often explicitly contains aspects of circuit that could be readily determined by the compiler, such as size of registers, width of busses etc. This makes the design less portable. Handling concurrency in hardware is relatively difficult in verilog as the designer should manage all the aspects of handshaking between combinational circuits. Shared use of register and other memory resources should also be elaborated. The behavioral specification of design in verilog often consumes multiple clock cycles. Attempts to resolve this problem results in a highly unreadable code with possible bugs. In practice, this problem is solved by separating the combinational and sequential parts of the circuit. Due to these shortcomings, the synthesis and verification of hardware in verilog is slowed down. This is a huge problem during the design of SOC.

3.2 Bluespec

Bluespec is based on atomic transactions, which increases the level of concurrency abstraction above SystemC and RTL without compromising the control over hardware design. It enables automatic synthesis of complex control logic, which is the source of many bugs. This results in highly adaptable, reusable and reconfigurable designs.

Control-adaptive parameterization in bluespec provides flexibility, where a significantly different micro-architecture can be generated by changing the parameters in the design with the associated control structures generated automatically. Bluespec allows user-defined data types and static type checking. It provides several features of the modern high level languages and all of them can be synthesized.

In recent times, several attempts have been made to move the hardware design language towards a more software like specification of the circuit behavior. Languages like C, C++ are used to express designs as sequential programs. However, the semantic gap between the software model and the hardware results in suboptimal designs with unpredictable speed and area. Bluespec System Verilog tackles this problem by building upon the traditional hardware semantics. It exploits advanced concepts from software only for static elaboration and static verification. It uses the standard hardware structure model of verilog such as modules, module instances, hierarchy etc. For communication between modules it uses the System verilog model of interfaces and interface instances. These added with the advanced features of the high level languages, makes designing and verification in bluespec much faster.

3.3 Features of Bluespec

3.3.1 Modules and Interfaces

Module is the basic element of the hardware design hierarchy in bluespec. A module can be instantiated multiple times, and also different parameters can be passed during every instantiation. Unlike verilog, bluespec does not have input, output and in-out pins as interface to modules. Methods are used to drive signals and busses in and out of modules. These methods are grouped together into interfaces. Modules contain rules, which use methods in other modules. Figure 3.1 shows how these methods, interfaces and rules fit into a hierarchy of modules.

In BSV, the interface declaration is done separately, outside the module definition. This allows declaration of common interfaces which can be used in multiple modules, without having to declare them repeatedly. All the modules which share the same inter-

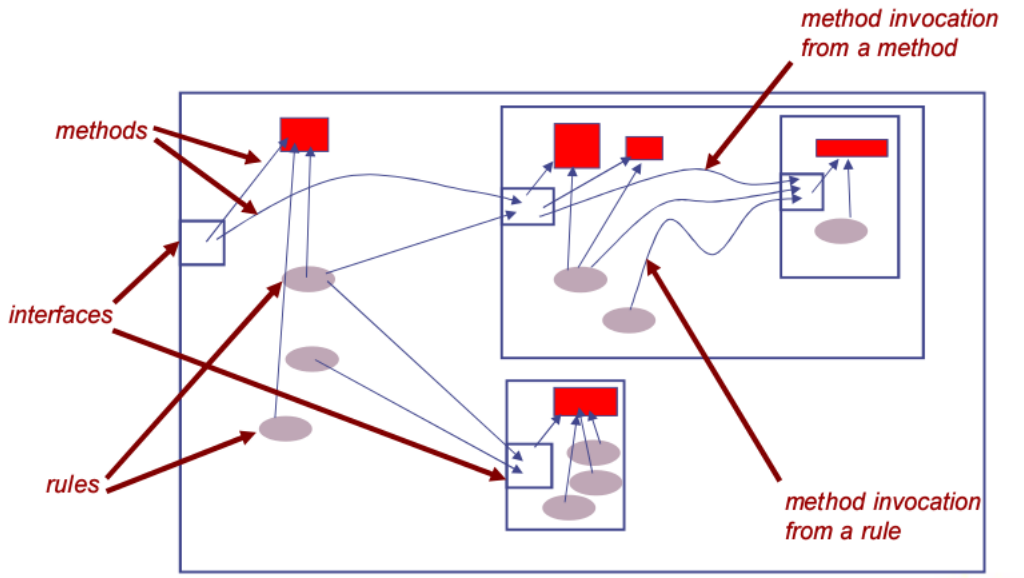

Figure 3.1: Representation of methods, interfaces and rules in a module hierarchy

faces also share same methods and therefore share same number and type of inputs and outputs.

3.3.2 Data Types

In verilog, all the representation is done in bits. Also, ultimately in hardware all computation is done in bits. However, representation in terms of integers, floating point numbers, fixed point numbers etc, makes the process of coding much easier. Different representations may be more appropriate depending on the application environment. By separating out the type abstraction from its bit representation, we can easily change representations without modifying or breaking the rest of the program.

In BSV, every variable has a type and only the values of compatible types can be assigned to a variable. The BSV compiler provides a strong, static type-checking environment. Type checking is done before the program elaboration and it ensures that the object types are compatible and the conversion functions are valid for the context. Bluespec also allows the usage of user-defined types. BSV has a typeclass which can be considered as a set of types. It implements overloading across related data types. Overloading is the ability to use a common name for a collection of types, with the specific type for the variable being chosen by the compiler based on the types on which it is actually used. Functions and operators are shared by all the data types within a

typeclass.

Some common scalar types used in Bits typeclass are Bit#(n), Bool, UInt#(n) and Int#(n). The values stored in registers, FIFOs and other memory elements and also the values passed by wires, must be in the Bits typeclass. Other common data types include Integer, which belongs to the Arith typeclass and String, which belongs to the Literal typeclass etc.

3.3.3 Rules

Rules manage the movement of data from one state to another, within the module. It consists of two parts: rule conditions and rule body. Rule conditions are boolean expressions which decide whether the rule can be fired. Rule body is a set of actions for state transitions. Rules in BSV are atomic. The actions within the rule completely describes the state transition. The process of determining the functional correctness of a design is greatly simplified by one-rule-at-a-time semantics. That is, because of the atomic property of rules, each rule can be looked at in isolation, without considering the actions of the other rules to determine functional correctness. Multiple rules can be executed concurrently in the hardware implementation.

The actions in a rule are executed simultaneously. This can be thought of as similar to the execution of non-blocking statements in always blocks of verilog. Also, as the rule has atomic property, the entire body of rule is executed and there is no partial execution of a rule. When there are several rules within a module, the execution of rules is ordered by the compiler. No two rules can execute simultaneously. The ordering of the rules by the compiler is called scheduling.

3.3.4 Methods

A method is a procedure which takes arguments and returns a value. It could also return a value without taking any arguments. It becomes a bundle of wires when translated into RTL. The method definition is written within the definition of the interface and it can be different in different modules sharing a common interface. A method also contains

implicit conditions which are handshaking signals and logic automatically generated by the compiler.

Methods are of three types: Value Methods, Action Methods and Action Value Methods. Value methods return a value. They do not alter any state within the module. Action methods cause actions to occur. They create state changes within the module. Action value methods are a combination of value methods and action methods. They cause state changes and also return values.

3.4 TLM Library

The TLM package includes definitions of interfaces, data structures, and module constructors which allow users to create and modify bus-based designs in a manner that is independent of any one specific bus protocol. Designs created using the TLM package are thus more portable as it allows the core design to be easily applied to multiple bus protocols.

The TLM interfaces define how TLM blocks interconnect and communicate. The TLM package includes two basic interfaces: The TLMSendIFC interface and the TLMRecvIFC interface. These interfaces use basic Get and Put sub-interfaces as the requests and responses. The TLMSendIFC interface generates (Get) requests and receives (Put) responses. The TLMRecvIFC interface receives (Put) requests and generates (Get) responses. Additional TLM interfaces are built up from these basic blocks. The TLMSendIFC interface transmits the requests and receives the responses. The TLMRecvIFC interface receives the requests and transmits the responses. This is represented in Figure 3.2.

The two basic data structures defined in the TLM package are TLMRequest and TLMResponse. By using these types in a design, the underlying bus protocol can be changed without having to modify the interactions with the TLM objects. A TLM request contains either control information and data, or data alone. A TLMRequest is tagged as either a RequestDescriptor or RequestData. A RequestDescriptor contains control information and data while a RequestData contains only data. Table 3.1 describes the

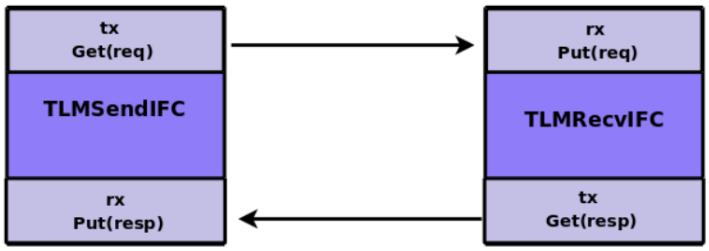

Figure 3.2: Representation of TLMSendIFC and TLMRecvIFC

components of RequestDescriptor and the valid values for each of its members. Table 3.2 presents the components of RequestData and the valid values for its members. Table 3.3 describes the components of a TLMResponse and the valid values for its members.

In the above BSV code definitions the compiler macros 'TLM_TYPES are used

Table 3.1: Components of RequestDescriptor

RequestDescriptor		
Member Name	DataType	Valid Values
command	TLMCommand	READ, WRITE, UNKNOWN
mode	TLMMode	REGULAR, DEBUG, CONTROL
addr	TLMAddr#(`TLM_TYPES)	Bit#(addr_size)
data	TLMData#(`TLM_TYPES)	Bit#(data_size)
burst_length	TLMUint#(`TLM_TYPES)	UInt#(uint_size)
byte_enable	TLMByteEn#(`TLM_TYPES)	Bit#(TDiv#(data_size, 8))
burst_mode	TLMBurstMode	INCR, CNST, WRAP, UNKNOWN
burst_size	TLMBurstSize#(`TLM_TYPES)	Bit#(TLog#(TDiv#(data_size, 8)))
prty	TLMUInt#(`TLM_TYPES)	UInt#(uint_size)
lock	Bool	
thread_id	TLMId#(`TLM_TYPES)	Bit#(id_size)
transaction_id	TLMId#(`TLM_TYPES)	Bit#(id_size)
export_id	TLMId#(`TLM_TYPES)	Bit#(id_size)
custom	TLMCustom#(`TLM_TYPES)	cstm_type

Table 3.2: Components of RequestData

RequestData		
Member Name	DataType	Valid Values
data	TLMData#(`TLM_TYPES)	Bit#(data_size)
transaction_id	TLMId#(`TLM_TYPES)	Bit#(id_size)
custom	TLMCustom#(`TLM_TYPES)	cstm_type

in the typedef statements. A 'define statement is a preprocessor construct used to place prepackaged text values into a file. In this case, the macros contain parameters to be used in the data definitions. Placing the parameters in a separate file allows them to be easily modified for different protocol requirements.

Table 3.3: Components of TLMResponse

TLMResponse		
Member Name	DataType	Valid Values
command	TLMCommand	READ, WRITE, UNKNOWN
data	TLMData#(`TLM_TYPES)	Bit#(data_size)
status	TLMStatus	SUCCESS, ERROR, NO_RESPONSE
prty	TLMUInt#(`TLM_TYPES)	UInt#(uint_size)
thread_id	TLMId#(`TLM_TYPES)	Bit#(id_size)
transaction_id	TLMId#(`TLM_TYPES)	Bit#(id_size)
export_id	TLMId#(`TLM_TYPES)	Bit#(id_size)
custom	TLMCustom#(`TLM_TYPES)	cstm_type

3.5 Tile Link

Tilelink is a protocol designed to be a substrate for cache coherence transactions implementing a particular cache coherence policy within an on-chip memory hierarchy. Its purpose is to orthogonalize the design of the on-chip network and the implementation of the cache controllers from the design of the coherence protocol itself. Any cache coherence protocol that conforms to TileLink's transaction structure can be used interchangeably with the physical networks and cache controllers we provide.

Tilelink is roughly analogous to the data link layer in the IP network protocol stack, but exposes some details of the physical link necessary for efficient controller implementation. It also codifies some transaction types that are common to all protocols, particularly the transactions servicing memory accesses made by agents. The usage of Tile Link is explained in detail in the next chapter.

CHAPTER 4

Implementation of MCP

This dissertation presents the implementation details of a flat MOESI protocol using MCP. The hierarchy consists of two clients in the lowest level which are connected to two processors. The highest level consists of memory and its manager. These two levels are connected by an intermediate level of a manager-client pair. The hierarchy is shown in Figure 4.1.

The clients are connected to processors through TLM interface. The interface between

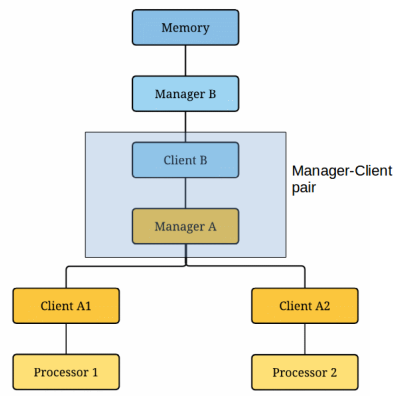

Figure 4.1: MCP hierarchy

memory and manager is also of type TLM. The propagation of data request till memory takes place as follows. When a request is raised by the processor1, it is forwarded to managerA through clientA1. The managerA then forwards this request to the other client on the same level, clientA2. If a hit occurs in clientA2, the requested data is sent to managerA, which is given to clientA1, and the transaction is completed. If there is a miss in clientA2, the request is forwarded to the memory through clientB and managerB. Now the managerA gets the data from memory, again through managerB and clientB. This data is given to clientA1, to finish the transaction.

4.1 Tile Link Architecture

Tilelink is a protocol used for communication between manager and client in MCP. Tilelink defines five independent transaction channels: Acquire, Probe, Grant, Release and Finish. The description of these channels is represented in Table 4.1.

There are two types of transaction that can occur on a cache block managed by Tilelink.

Table 4.1: Channels of Tilelink

Channel	Description
Acquire	Initiates a transaction to acquire access to a cache block. Also used to write data without caching it. This channel is from client to manager.
Probe	Queries a client to determine whether it has a cache block. The query is raised by the manager.
Release	Used by client to send status of hit/miss to manager. In the event of hit, data is also sent by this channel. Also used to voluntarily write back data and to send invalidate signals.
Grant	Manager provides data to the original requester. Also used to acknowledge voluntary Releases.
Finish	Final acknowledgement of transaction completion from requester to the manager.

The first type enables clients to acquire a cache block: A client sends an Acquire to a manager. The manager sends any necessary Probes to clients. The manager waits to receive a Release for every Probe that was sent. The manager communicates with backing memory if required. Having obtained the required data, the manager responds to the original requester with a Grant. Upon receiving a Grant, the original client responds to the manager with a Finish to complete the transaction. The second type of transaction supports clients voluntarily releasing a cache block: A client sends a Release to a manager, specifying that it is voluntary. The manager communicates with backing memory if required. The manager acknowledges completion of the transaction using a Grant. This voluntary release is used for write-back by processor and also to send invalidate signals.

The channels of tile-link has several fields to aid in the request transfer and response propagation in coherence hierarchies. The table 4.2 contains the fields and their description for each channel.

Table 4.2: Description of fields of Tilelink Protocol

Channel	Field	Description
Acquire	a_type	command: read/write
	client_xact_id	client id
	addr_block	address
	data	data
Probe	p_type	command
	addr_block	address
	data	data
Release	voluntary	Bool value to indicate vol release
	r_type	command
	r_state	coherence state
	status	status
	client_xact_id	client id
	addr_block	address
	data	data
Grant	status	status
	g_type	command
	g_state	coherence state
	client_xact_id	client id
	data	data
Finish	client_id	client id

4.2 Working of MCP

4.2.1 Processor

Each processor has a local cache, in which read and write operations are done by the processor, in response to the CPU requests. The data cache of processor has the following fields: valid, tag and data. The valid bit is used by the processor to check whether the data stored in that particular location of the cache is valid or not. The tag is used to find out if the requested data block is present in it or not. When a read or write miss happens in the local cache, the processor has to get the data from the higher level of the memory hierarchy. In order to do this, it requests the paired client. The processor then obtains the data from this client and stores in its cache. In case of a read request, the data is sent to the CPU. In case of write request, the data block is updated with new data and status is sent to the CPU. Whenever a miss happens, the processor has to make space in cache for the incoming data from higher level. Therefore, before requesting for data, the processor performs a write-back of the least used data block in its cache. After every write(write-hit/ write-miss), the processor sends out an invalidate request to the

client in-order to invalidate all the other shared copies. The processor also responds to requests from the client. When the client requests a data block, the processor provides it. Also, when the processor receives an invalidate request from client, it invalidates the requested block.

4.2.2 Client

As we have already established, one of the important requirements to implement MCP is the ability of the client to respond whether or not it has the requested data block. Also, the client is responsible for coherence management. In order to achieve this, additional memory is allocated to the client. This is similar to the data-cache of the processor, but instead of data field, it has coherence-state. The client performs three major functions: Handling data request from processor, handling write-back and invalidate from processor and handling request from manager.

Request from Processor

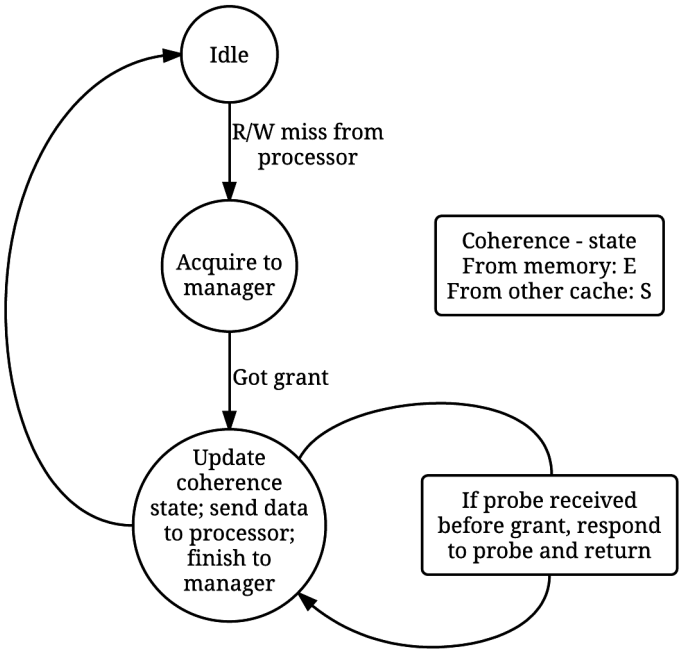

Figure 4.2: Acquire-Grant-Finish in Client

The client receives the request to obtain data from processor through TLM interface. It then sends Acquire to the manager, using Tile-Link protocol. Now, the client has to wait for the manager to respond. When the manager sends a grant, the client obtains the

data and sends it to the processor. The grant from the manager has a state field which indicates whether the data was obtained from main memory or from the other caches of the same level. If obtained from memory, the coherence state of that data is stored in the client as Exclusive(E). If not, it is stored as Shared(S). Also, when the data is obtained from main memory, the valid and tag fields of the memory in the client are updated. While waiting for grant, if the client receives a probe from the manager, it needs to address it first and then get back to the state of waiting for grant. The state-machine of client when it handles request from processor is shown in Figure 4.2.

Write-back and Invalidate

During write-back, the client sends a voluntary release to the manager, if the coherence state of that block is M/O/E. This is because, we need not write back an invalid data and if the data is shared, it will be written back to memory when the processor containing the Owner copy performs a write-back. Also, the coherence state of this data block need not be updated in the client now, as it will be updated when new data is written in its place.

When the processor completes a write operation, it sends out an invalidate request.

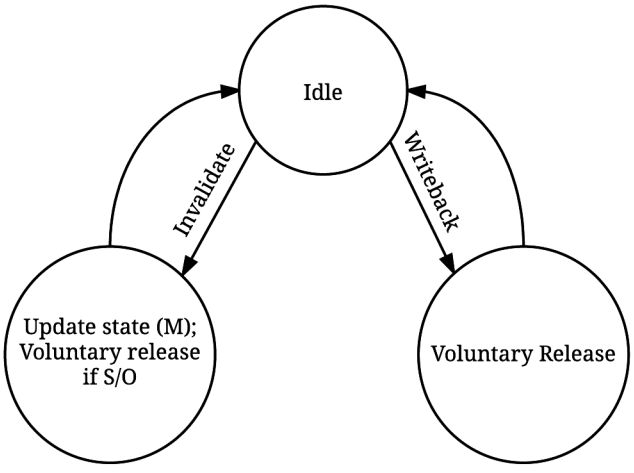

Figure 4.3: Voluntary Release by Client

This request is received by the client and then forwarded to the manager if the state of that data block in the client in S/O. This is because, data in M/E state would mean that they are exclusive copies. Also, at this point the state of the data block which was written is changed to M in the client. The state-machine of the client, in this case is

shown in Figure 4.3.

Request from Manager

When the client receives a probe from the manager, it searches its tags to find out if there is a match. When the request from the manager is invalidate, and there is a match in the client which is not already in Invalid(I) state, the state of that block is updated to I and the invalidate request is sent to the processor.

When a match is found in the client and the manager requests data, the request is

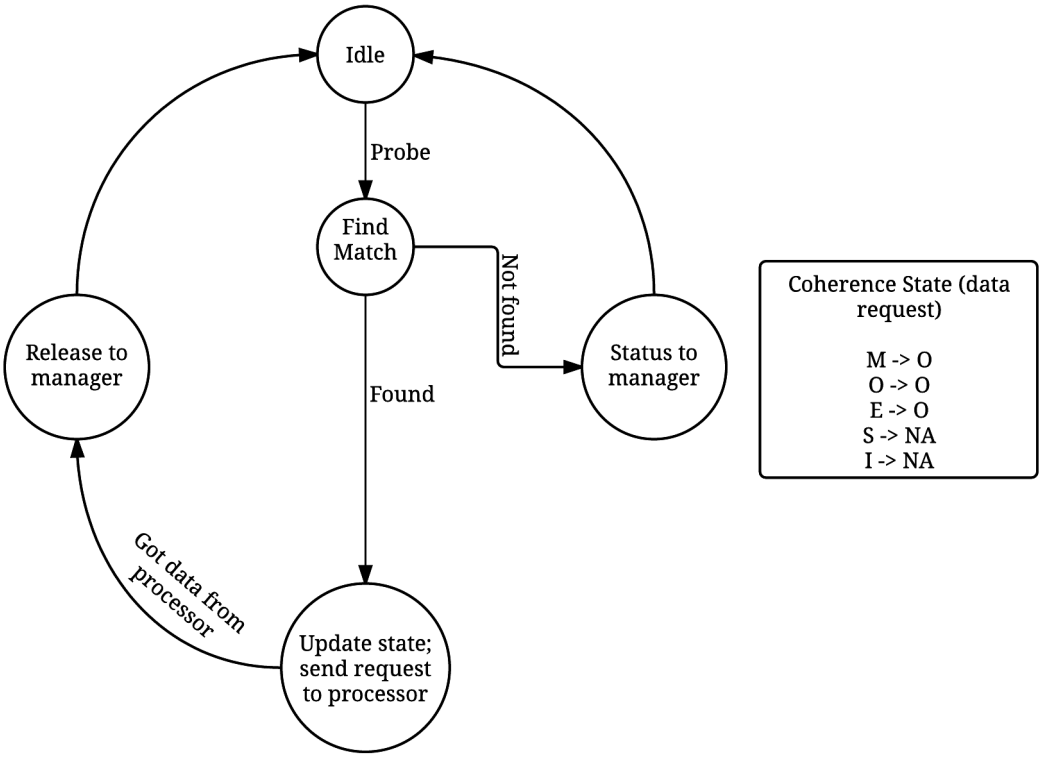

Figure 4.4: Probe-Release in Client

sent to the processor only if the state of the data requested is M/O/E. This is because, only the Owner has to respond when a shared data is requested. The state of this data is updated to Owner(O) in the client, before sending the request to the processor. After receiving the data block from the processor, the client sends a release to the manager with the data and status true. If no match is found, release is sent to manager with status false. Figure4.4 shows the state-machine.

4.2.3 Manager

The two major responsibilities of manager is to handle data requests (Acquire) and to handle write-back and invalidate (voluntary release) from clients.

Data request

On receiving Acquire from a client, the manager sends probe to all the other clients in the same level. It has to wait till it gets release from all these clients. After receiving all the releases, the manager examines if atleast one of these releases has status true(It is to be noted that only one of these releases will have status true). If a release with status true is found, the manager responds to the acquire with a grant. Then it waits for finish from the client. The transaction ends when finish is received from the client which raised the request.

If all the releases received have status false, the request is forwarded to the higher

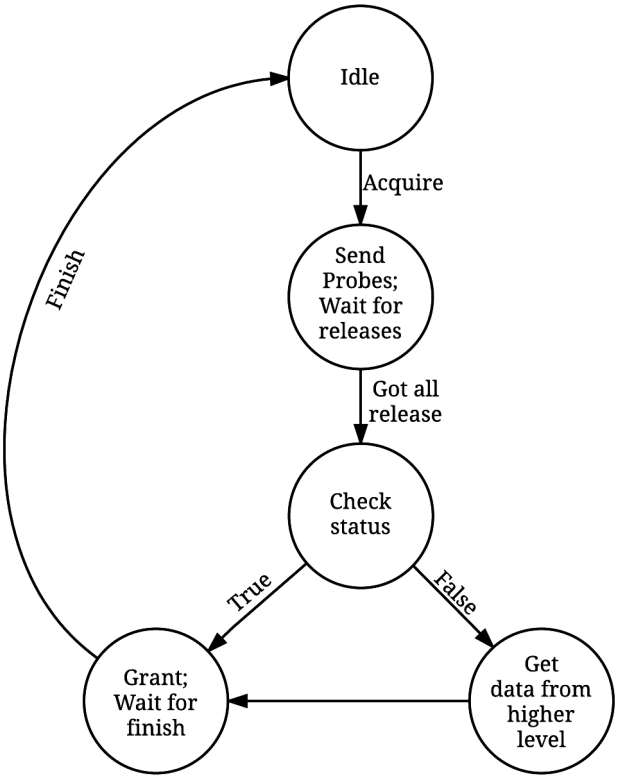

Figure 4.5: Acquire-Grant-Finish in Manager

level of hierarchy through the paired client. This is done by sending a probe to the paired client. On receiving release from the paired client, grant is sent to the client

which requested this data. The state field of the grant is updated to 'Memory', to notify the client that the requested data has been fetched from the memory(or higher level of hierarchy). After sending grant, the manager waits for finish from the origin client. The transaction ends when finish is received by the manager. The state machine of manager handling data request is shown in Figure 4.5.

Voluntary Release

When the manager receives an invalidate request from client, it probes all the other clients. No grant is required in this case.

If the voluntary release received by the manager is for write-back, the manager sends

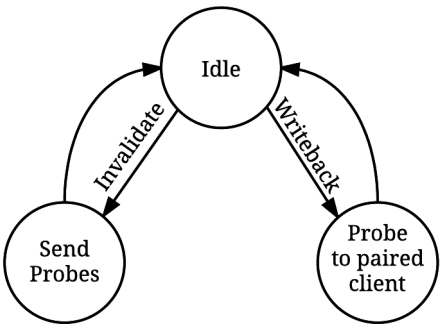

Figure 4.6: Handling Voluntary Release in Manager

a grant to the origin client with status true. This request is forwarded to the higher level of the memory hierarchy through the paired client. The manager sends a probe to the paired client and it does not wait for release in this case. The state-machine for this case is given in Figure 4.6.

4.2.4 Working of Hierarchy

Read miss

When a read miss occurs in a processor, the processor sends a request to the client through TLM interface. The clients receives this request and sends an acquire to the manager. All communication between manager and clients can only happen by Tile-Link protocol. The manager then, sends probe to all the other clients in the same level

as the client which requested the data. Upon receiving the probe, the client checks if the requested data block is present in its cache or not. If it is present, the coherence state of that data is updated in the client and the request is forwarded to the processor. The processor provides the data block requested, which is sent to the manager through release by the client. The release in this case is sent with a status true. If the requested data is not found, the release is sent with a status false. The manager waits till it receives release from all the clients probed. If the status of one of these releases is true, the manager obtains the data and sends a grant to the client which requested this data. If none of the releases have status true, the data has to be fetched from the higher level of the hierarchy. So, the manager probes the paired client. Upon receiving probe, the paired client sends an acquire to the memory manager. Now, the memory manager requests the main memory, obtains the data and responds to the client with a grant. On receiving this grant, the paired client sends finish to the memory manager which ends the transaction with it. Now the client sends this data to the paired manager by sending a release. The manager can now respond to the origin client with a grant as it has obtained the data. On receiving the data, the client forwards it to the processor, updates the coherence state and sends finish to the manager, which completes the transaction. The process is shown in Figure 4.7.

Write miss

During write miss, the processor requires the data block which has to be written. So, fetching the data block happens exactly in the same manner as read miss. After obtaining the data block, the processor completes the write and then sends an invalidate request to the client. The client now checks the coherence state of that data block. If the state is O/S, the client sends a voluntary release to the manager. The state of this block in the client is changed to M(this happens irrespective of the initial state of the block). On receiving the voluntary release, the manager probes all the other clients. The clients which receive the probe, update the coherence state of the data block to I, if it has that data block. The invalidate is then forwarded to the processor and the data is invalidated in the data cache of the processor. The process is shown in Figure 4.8.

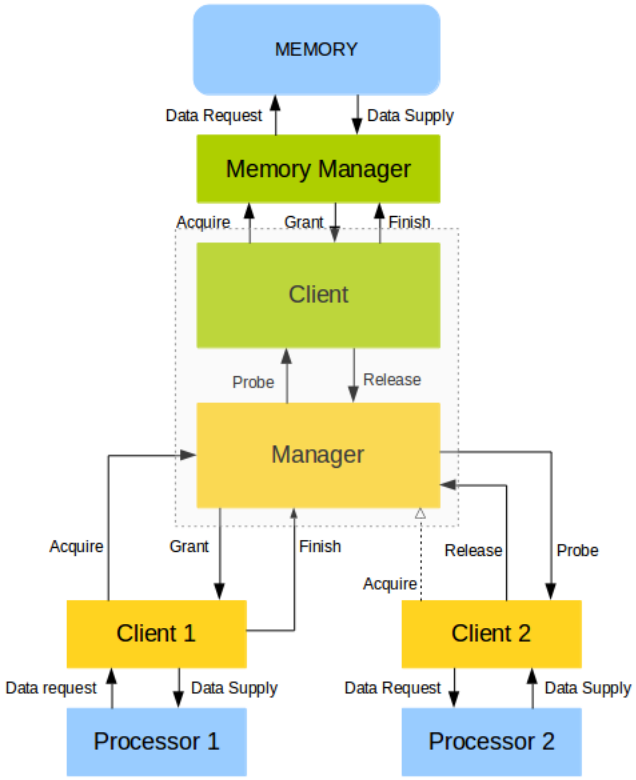

Figure 4.7: Handling Read-Miss

Write hit

After the processor completes the write operation in the event of a write-hit, it sends an invalidate request to the client. The propagation of the invalidate request happens exactly as that during write miss.

Write-back

During write-back the processor sends request to the client. This clients sends a voluntary release to the manager and does not expect a response. After receiving the voluntary release, the manager forwards the data to higher level of memory hierarchy by sending a probe to its paired client. It does not wait for a release. The paired client, then sends this request to the memory manager through a voluntary release. The memory manager forwards the data to the main memory and completes the write-back. No response is sent to the client from the memory manager as it is not required. The process is shown in Figure 4.9

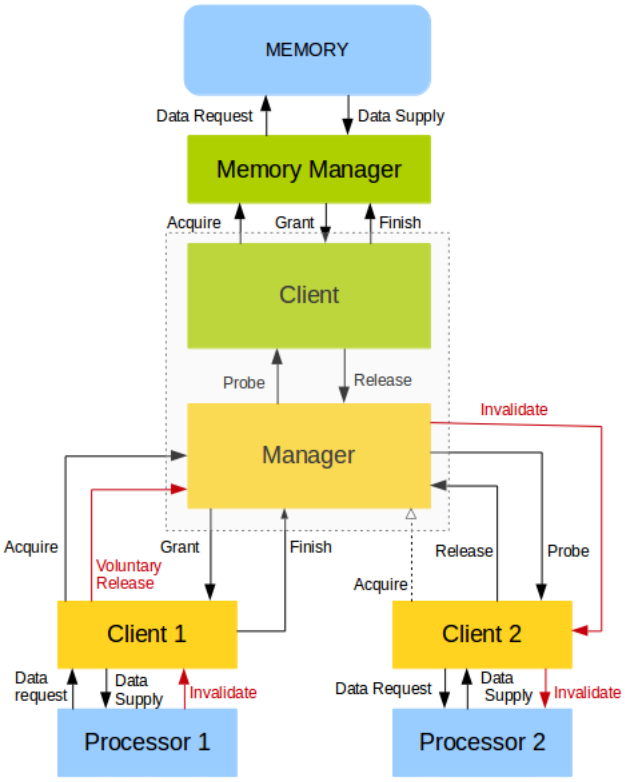

Figure 4.8: Handling Write-Miss

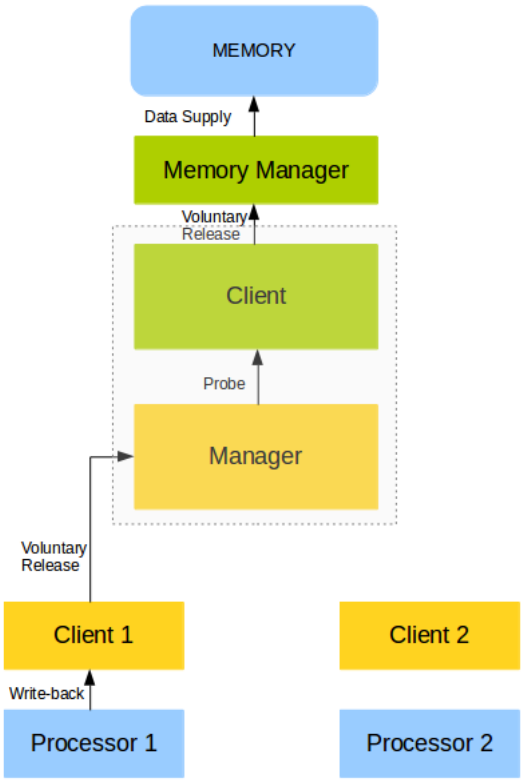

Figure 4.9: Handling Write-Back

CHAPTER 5

Conclusion and Future Work

The memory hierarchy constructed using Manager-Client pairing and Tilelink protocol is integrated with the I-class processor of the Shakti series of processors. The design is simulated and checked for functional correctness. The significant increase in the ease of designing memory subsystem, when the responsibility of coherence management is decoupled from the processor is successfully demonstrated. The use of Tilelink protocol greatly simplified the state machines of manager and client. In the worst-case scenario, the manager-client pairing technique uses 13 clock cycles to fetch data from main-memory. This number can be greatly reduced by using parallelism in manager and client. However, at present, this is 9 clock cycles more than the conventional memory hierarchy design with each cache connected to every other cache. Inspite of this, the design using MCP has much less complicated interfaces because of the tilelink protocol. Also, the implementation of cache coherence is done with greater ease. The major advantage of MCP is that, this design can be replicated to construct hierarchy of coherence protocols. The design of even heterogeneous hierarchies can be done rapidly, with ease.

The Tilelink protocol provides scope for parallelizing the functions of manager and client. This should be exploited in future implementations of MCP as this would result in significantly faster data transfer operations. This dissertation presents the design of memory hierarchy using a flat coherence protocol. However, to fully appreciate the advantages of using MCP, more coherence realms should be introduced. By extending the implementation presented in this dissertation, hierarchy of heterogeneous coherence protocols can to be constructed to demonstrate the ease of designing using MCP. Also, the designs need to be verified and evaluated. The time taken for verification can be compared against the time taken for verifying the designs using conventional methods, to observe the increase in the speed of verification due to MCP.

REFERENCES

[1] Beu Jesse G., Rosier Michael C., Conte Thomas M., "Manager-Client Pairing: A Framework for Implementing Coherence Hierarchies", 44th Annual IEEE/ACM International Symposium on Microarchitecture, Pages 226 - 236, 2011.

[2] Ros Alberto, Davari Mahdad, Kaxiras Stefanos, "Hierarchical private/shared classification: The key to simple and efficient coherence for clustered cache hierarchies", IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), Pages 186 -197, IEEE - 10.1109/HPCA.2015.7056032, Feb 2015.

[3] C. Anderson, "A multi-level hierarchical cache coherence protocol for multiprocessors", Parallel Processing Symposium, pp. 142-148, 1993

[4] Bluespec,Inc.Bluespec System Verilog Reference Guide, Revision 30 July 2014.

[5] Tilelink 0.3.3 specification, 2015. URL: https://github.com/ucb-bar/uncore

[6] Overview of the Rocket chip, 2015. URL: http://www.lowrisc.org/docs/untether-v0.2/overview/