DESIGN OF HETEROGENEOUS COHERENCE HIERARCHIES USING MANAGER-CLIENT PAIRING

A Dissertation
Presented to
The Academic Faculty

by

Jesse Garrett Beu

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

May 2013

DESIGN OF HETEROGENEOUS COHERENCE HIERARCHIES USING MANAGER-CLIENT PAIRING

Date Approved: March 29th, 2013

To Regina, for her inspiring patience, love, and support.

ACKNOWLEDGEMENTS

I want to start by thanking my family for their unwavering confidence in my ability to complete this degree, and their understanding and compassion during my extended social absences around publication deadlines over the years. I especially want to thank my mother for instilling such a strong sense of integrity in me, which leaves its mark on every piece of research I do, and my father and stepmother for believing in me with so much enthusiasm that my victories always felt like theirs as well.

I also must thank Gerald Merckel for having such a profound influence on this undecided undergraduate student so many years ago, and for being my point-of-contact to my PhD advisor and life-long friend, Tom Conte. I could easily populate pages attempting to describe the complex dynamic that only a student and advisor know, but I'll keep it simple: Tom, thank you for helping me reach my potential. I also want to thank Eric Rotenberg, whose countless after-lecture discussions and impassioned arguments during my time and NCSU shaped much of my understanding of architecture.

And to the Tinker group, my support web that kept me from sinking countless times, I owe much to you. Saurabh Sharma, whose '*AMAZING*' mentorship got my feet wet on my first publication. Balaji Iyer, the most gullible lab-mate and impressive human-encyclopedia of reference I've known. Paul Bryan, my go-to statistics guru and video/board gaming bud. Jason Poovey, my fellow 'laugh riot comedian' who has an uncanny knack for knocking me off balance with his unexpected perspective. And especially Chad Rosier, my roommate and partner in crime through Swampy, SlaDir, DSDP, CaffeineSim, MCP, and countless other brave adventures in architecture; thank you for helping me through the hardest time of my life.

I also want to thank the 'new guys', the Georgia Tech Tinkers. Rishiraj Bheda, Brian Railing, Phillip Vassenkov and Eric Hein: when things started to become overwhelming, you helped me pull it all back together. This wouldn't have happened without your feedback in all those meetings, all those trips to the 'Zen Garden', the Monday lunches or other million small things that add up to so much. And Brian, we will publish together!

Last, and certainly not least, I want to thank Regina Maniquis, my inspiration to keep going when I thought I had no more to give, and the reason I've been able to push through the countless hours of late night thinking, planning, coding and writing over these past several months. This is dedicated to you.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS iv

LIST OF TABLES ix

LIST OF FIGURES x

SUMMARY xii

CHAPTER 1 - INTRODUCTION 1

1.1 Defining Cache Coherence 2

1.2 Impact of the Trend Towards More Processing Cores 3

1.3 Trend Towards Heterogeneity 6

1.4 Manager-Client Pairing Perspective 7

1.5 The Fallacy of Composition 10

1.6 Contributions 12

1.7 Organization 13

CHAPTER 2 - COHERENCE PROTOCOL STATES AND DESIGN 15

2.1 State Selection 15

2.2 Design Issues For a Simple MSI Protocol 19

2.3 Other Design Considerations 22

CHAPTER 3 - RELATED WORK 24

3.1 Pioneering Work in Coherence Protocol Development 24

3.2 Architectures Employing Coherence Hierarchies 27

3.3 Improvements to Coherence Hardware Scaling 29

3.4 Protocol Design and Integration 31

3.5 Protocol Verification....33

3.6 Other Recent Contributions and Innovations....34

CHAPTER 4 - SIMULATION AND MODELING TOOLS....35

4.1 CaffeineSim Architectural Simulator....35

4.2 Manifold Architectural Simulator....45

4.3 MurPhi State Enumeration Formal Verifier....47

CHAPTER 5 - DESIGN OF THE MANAGER-CLIENT-PAIRING FRAMEWORK....51

5.1 The Problem....52

5.2 Division of Labor for Cache Coherence as a Template....55

5.3 Permission Hierarchy Algorithm....64

5.4 Using MCP to Compare Hierarchy Decisions....69

5.5 Conclusion....78

CHAPTER 6 - VERIFICATION AND ENCAPSULATION SYMMETRY....79

6.1 Introduction....80

6.2 Review of MCP Framework....84

6.3 Reachable State Enumeration Overview....86

6.4 Formal Verification Strategy for MCP....92

6.5 Remote Proxy Client....100

6.6 Results....103

6.7 Conclusion....105

CHAPTER 7 - HETEROGENEOUS HIERARCHY INTERACTIONS....107

7.1 Common Communication Trends....109

7.2 Representative Traces....113

7.3 Heterogeneous Hierarchical Interaction Analysis 119

7.4 Concluding Thoughts on Heterogeneity 133

CHAPTER 8 - CONCLUSION 136

APPENDIX A - MCP Actions 138

REFERENCES 139

LIST OF TABLES

Page

Table 1 - Base functions for standardized communication between processors, clients and managers 58

Table 2 - Verification cost of Directory-MESI and Broadcast-MOSI protocols....89

Table 3 - Comparing verification cost of heterogeneous hierarchical protocols with and without leveraging MCP protocol structural symmetry....105

Table 4 – Parsec benchmark characteristics....110

Table 5 – Summary of Parsec and Splash-2 dynamic communication trends....110

Table 6 - Read-Only local only communication (no evictions)....124

Table 7 - Read-Only coarse-grained local-remote interleaving (no eviction)....124

Table 8 - Read-Only remote-only communication (no eviction)....125

Table 9 - Read-Only fine-grained local-remote interleaving (no eviction)....125

Table 10 - Migratory local-only communication (no eviction)....127

Table 11 - Migratory coarse-grained local-remote interleaving (no evictions)....127

Table 12 - Migratory remote-only communication (no eviction)....128

Table 13 - Migratory fine-grained local-remote interleaving (no evictions)....128

Table 14 - Migratory remote-only communication, employing coarse-grain L3 eviction injection....129

Table 15 - Migratory remote-only communication, employing fine-grain L2 eviction injection....130

Table 16 - Producer-Consumer local-only and remote-only results for single producer-consumer pairs....131

Table 17 - Producer-Consumer fine-grained local-remote interleaving....132

Table 18 - Producer-Consumer coarse-grained local-remote interleaving....133

LIST OF FIGURES

Page

Figure 1 – Progression of coherence hierarchy composition using MCP showing (a) the starting protocols (b) the wrapping of the protocols with MCP interfaces and (c) the interfacing of an upper and lower protocol. 9

Figure 2 – Resultant protocol hierarchy after interfacing three lower-tier protocols with an upper-tier protocol, and attaching processors and memory to the remaining lower and upper interfaces. 10

Figure 3 – (a) A triangle composed of triangles and (b) a non-triangle composed of triangles 11

Figure 4 - Venn diagram of coherence state membership in the spaces of permissions, forwarding capability and whether data is dirty...19

Figure 5 - Basic state machines for an MSI protocol, excluding transient states 20

Figure 6 - Coherence sequence that demonstrates the IMSI state race...........................21

Figure 7 – CaffeineSim Node, containing modules representing architectural components 38

Figure 8 - An 8x8 network of nodes, with 4 independent neighborhood L1/L2 protocols unified by a chip-wide L2/L3 protocol...42

Figure 9 - Virtually flow controlled 6-stage router...43

Figure 10 - CaffeineSim Supported topologies 44

Figure 11 - A 4x4 Torus of Nehalem-like clusters, with 16 independent L2/L3 protocols united through the chip-wide L3/L4 protocol 47

Figure 12 - Coherence hierarchy labeled with MCP terminology..................................53

Figure 13 - Addition of permissions-query capabilities enabling recursive coherence...59

Figure 14 - Manager-Client Pairing and associated interfaces 61

Figure 15 - Coherence hierarchy permission checking algorithm. 65

Figure 16 - (a) Realm-hit Read example and (b) Realm-Miss Write example................67

Figure 17 - Coherence realms (shaded) and local tier miss traffic in (a) 2-Tier 64x4 and (b) 2-Tier 16x16 system 71

Figure 18 - Impact of hierarchy width on L1 miss latency....74

Figure 19 - Histogram of L1 Miss Latency for Water Spatial when varying hierarchy width....74

Figure 20 - Histogram of L1 Miss Latency for Ocean_c when varying hierarchy width 75

Figure 21 - Impact of hierarchy height on L1 miss latency....77

Figure 22 - Histogram of L1 Miss Latency for Water Spatial when varying hierarchy height....77

Figure 23 - Example of a heterogeneous multi-chip system that would benefit from heterogeneous coherence hierarchy support....82

Figure 24 - Manager-Client Pairing coherence hierarchy organization....85

Figure 25 - MCP Interface for (a) lower processor tier and (b) top memory tier....86

Figure 26 - Example of state space explosion when adding 2 additional nodes to a 2-node MSI protocol....88

Figure 27 - State-space of two simple state machines, where each element may transition from 0 to 1....91

Figure 28 - Full state space exploration of state machines operating simultaneously....91

Figure 29 - Permission distribution example for an MCP composition....94

Figure 30 - Graphical representation of coherence hierarchy inductive proof....99

Figure 31 - Local GetReadD Get sequence (Request and Response)....101

Figure 32 - Remote GetReadD Get sequence (Request and Response), using a proxy client to satisfy SupplyDowngradeAck Demand request....101

Figure 33 - Evaluated Heterogeneous Hierarchical Protocol Structure....104

Figure 34 - Fine-grained eviction interleavings targeting the L1 and L2 cache....115

Figure 35 – Read-Only trace sequence....117

Figure 36 – Migratory trace sequences....118

Figure 37 – Producer-Consumer trace sequences....119

Figure 38 - A 4x4 Torus of Nehalem-like clusters, with 16 independent L2/L3 protocols united through the chip-wide L3/L4 protocol....121

SUMMARY

Over the past ten years, the architecture community has witnessed the end of single-threaded performance scaling and a subsequent shift in focus toward multicore and manycore processing. While this is an exciting time for architects, with many new opportunities and design spaces to explore, this brings with it some new challenges. One area that is especially impacted is the memory subsystem. Specifically, the design, verification, and evaluation of cache coherence protocols becomes very challenging as cores become more numerous and more diverse.

This dissertation examines these issues and presents Manager-Client Pairing as a solution to the challenges facing next-generation coherence protocol design. By defining a standardized coherence communication interface and permissions checking algorithm, Manager-Client Pairing enables coherence hierarchies to be constructed and evaluated quickly without the high design-cost previously associated with hierarchical composition. Further, Manager-Client Pairing also allows for verification composition, even in the presence of protocol heterogeneity. As a result, this rapid development of diverse protocols is ensured to be bug-free, enabling architects to focus on performance optimization, rather than debugging and correctness concerns, while comparing diverse coherence configurations for use in future heterogeneous systems.

CHAPTER 1 - INTRODUCTION

Over the past ten years, the architecture community has witnessed the end of single-threaded performance scaling and a subsequent shift in focus toward multicore and future manycore processors [1]. It is well established that physical limitations in power consumption have caused this paradigm shift, due to the lower power cost of thread-level parallelism compared to the single-threaded parallelism techniques prevalent in the late 1990s. Even so, we cannot afford to abandon single-threaded performance gains entirely, which has pushed modern design towards sophisticated heterogeneous systems that enable general-purpose flexibility for both single-threaded and multi-threaded applications in a transistor-abundant, power constrained future [2].

While this is an exciting time for architects, with many new opportunities and design spaces to explore, this brings with it some new challenges. One area that is especially impacted is the memory subsystem. Specifically, the design, verification, and evaluation of cache coherence protocols becomes very challenging as cores become more numerous and more diverse.

This dissertation examines these issues and puts forth the following thesis:

The challenges in designing, verifying and evaluating next-generation coherence hierarchies can be overcome through the use of a standardized coherence communication interface that provides protocol encapsulation for rapid verification and supports protocol modularity for rapid, reconfigurable composition.

1.1 Defining Cache Coherence

Cache coherence is responsible for the management and distribution of data in a shared memory environment, where private processor caches can retain temporary copies of data that may or may not be dirty with respect to main memory [3]. Coherence is an important aspect of correct parallel operation when caching is enabled because without coherence, the agreement between the programmer and the hardware regarding when other processors observe memory loads and stores can be violated. This agreement is known as the consistency model of the architecture, and is a necessary part of any shared memory parallel programming specification [4]. If there is no coherence protocol enforcing consistency, the private processor caches can get out of sync with one another by committing stores to local caches that are not being observed in other processors' local caches. This inconsistency in the multiple versions of data associated with a given address causes a breakdown in the fundamental way processors communicate with one another through shared memory. The reads and writes of every processor have to be observable by other processors for correctness, and cache coherence ensures this behavior takes place.

In general, hardware managed cache coherence is accomplished through the association of state with cached copies of data to represent whether or not the cached copy is currently valid, and if so whether it is exclusive within the entire system, or just one of many possible copies that exist. This provides the basic mechanisms required to support write and read permissions, disallowing writes when a cache does not hold the only exclusive copy of data, and disallowing reads if the caches copy is in the invalid state. In order to enable writes when in a non-exclusive state, a mechanism must also be available to support the invalidation of all other copies through out the system. The two leading classes of coherence protocols accomplish this by either 1) broadcasting read and write intentions to all processors caches, where each cache is self managed, invalidating their local copies as they observe external traffic and 2) issuing requests to a global ordering point which maintains a directory of what caches are currently holding copies and is responsible for taking any actions necessary to ensure proper permission distribution. These are commonly referred to as broadcast (or snoopy) protocols and directory protocols, respectively.

Some cache coherence protocols bend these rules by enabling writes when not holding the only copy of data. To provide a consistent view of memory, the written value is distributed to all sharers, where each sharer then updates their locally shared copies. As a result, protocols employing this strategy, as opposed to invalidating other sharers, are collectively referred to as update-based protocols. The remainder of this dissertation will only concern itself with invalidation-based protocols, but a comprehensive comparison of update-based and invalidation-based protocols can be found in Glasco's dissertation [5].

1.2 Impact of the Trend Towards More Processing Cores

There is no denying that there has been a fundamental shift in computer architecture towards multithreaded technologies, now widely accepted as the most viable means of sustaining the performance gains we've grown accustomed to over the last four decades due to Moore's law [2]. Excerpts like the following are now commonplace, expressing a sentiment being felt across all disciplines within computer engineering:

"The eventual introduction of processors with CMP technology is the best (and perhaps only) chance to avoid the dire future…. if thread-level parallelism is available, using the transistor and energy budget for additional cores is more likely to yield higher performance than any other techniques we are aware of" – Luiz Andre Barroso, Google [1]

While it is true that CMPs do provide better efficiency, when thread level parallelism is available, CMP and multi-threaded execution does so while applying more pressure on the communication infrastructure. Although network on chip (NOC) technology is evolving to meet this demand [6], as we run headlong towards the safe haven of multithreaded execution there will be an unprecedented increase in NOC communication needs. In addition to this increase in pressure, as tiled architectures are more frequently employed as a solution to VLSI design concerns [7], power dissipation, energy, and wire delays concerns in the NOC will come to the forefront as problems in next generation systems as they rapidly scale out [8-10]. Sanchez et al. demonstrate that NOC latency, not bandwidth, is already beginning to play a much more prominent role in the performance of modern CMP systems [11]. This implies that if there are asymmetries in the communication traffic with respect to near traffic and far traffic, it should be exploited. All this discussion ultimately converges on a single important conclusion regarding cache coherence: solutions to future problems will involve improving the intelligence and efficiency of system-wide communication.

Not only do we need to consider communication, but storage for coherence is also an issue that cannot be ignored as core count increases, specifically size increases in directory-type structures which track coherence state across nodes. Fortunately there has been a wealth of research in this area in recent years [12-17], but not all research has focused solely on the storage structure changes alone. Some involve intelligently modifying the design of the coherence enforcement and/or protocols in conjunction with the storage system to enable storage optimizations [18, 19]. As mentioned briefly in [20] and discussed later in this dissertation, coherence hierarchies also contributes towards a reduction of storage needs by creating a kind of course grain tracking at the high-level protocols that manage lower protocols.

There is an additional challenge that threatens our future as we scale out beyond 8 core CMPs as well, the issue of verification. Formal verification of coherence protocols is a strategy employed to guarantee the correctness of a coherence implementation. This guarantee is an especially important consideration because coherence protocol failures do not simply result in graceful performance degradation like many other sub-systems, and cannot simply be shut-off or easily patched postproduction. Coherence design errors can produce difficult to detect consistency violations with unpredictable effects, including incorrect execution and data corruption, and livelock and deadlocking communication sequences have the potential to cause outright system-wide failures. In [21], Arvind et al. further motivate why verification is an important concern in modern architecture and suggest ways to bring verification to the forefront as part of design.

The challenge for coherence verification is that state enumeration requires exploration of the entire global state space of the protocol being verified. For formal checking tools, such as MurPhi [22], this is accomplished by applying every coherence rule to every state recursively in a breadth-first or depth-first search manner, checking invariant violations for each newly encountered state. This global state space grows exponentially with the number of nodes being managed and quickly becomes intractable [23]. Experimentation found that the global state spaces for a MESI directory protocol and MOSI broadcast protocol grew at a rate of approximately 15^n and 30^n distinct global states, respectively, where n is the number of nodes (See Chapter 6 for more details). It is clear that as n approaches even a conservative projection of only 32 cores this is not sustainable, even when applying global state space reduction techniques like state symmetry [24].

As if this were not already enough of a problem, the issue of verification becomes substantially worse when taking protocol heterogeneity into consideration. As soon as two distinct protocols are integrated, the complexity of the new global state space approaches something close to the cross product of the original spaces (see Figure 28 in Section 6.3.2 for a simplified example of this phenomena). While this has obvious ramifications for verification cost, in terms of state enumeration, it has the additional penalty of reducing the efficacy of state symmetry reduction techniques since the procedure for mapping the new states/interactions into the old symmetry framework is currently undefined; what can be defined as symmetric in a heterogeneous, hierarchical environment has never been considered before.

1.3 Trend Towards Heterogeneity

The issues raised regarding verification seem to imply that perhaps coherence heterogeneity should just be avoided as we continue to scale. This would be a grave error, however, considering the opportunities heterogeneity provides. In fact, heterogeneity has already been employed in many CMP architectures [25-30], and to good effect. One of the biggest benefits of heterogeneity is the ability to co-design parts of the communication infrastructure of a large-scale system with different coherence protocols while still providing a unified coherent memory space. In a distributed architecture, the ability to tailor the intra and inter cluster communication to match the constituent parts of the system means you can employ local broadcasts within clusters, for example, while leveraging the scalability benefits of directory operation across clusters. Different bandwidth expectations between near intra-cluster traffic and far inter-cluster communication may advocate using heterogeneity to match this asymmetry. Differences in latency tolerances regarding the criticality of near vs. far accesses could mean accepting different design/complexity tradeoffs or enabling/disabling certain states (see Section 7.4).

This especially becomes apparent when we begin considering integrated accelerators and the challenges that designing an effective homogenous protocol would entail. The authors of [31] point out that many of the common practices in CPU coherence can severely impede the performance of a GPU using the same hardware managed coherence methodologies, and propose an overhaul to the coherence strategy for use in GPUs. The motivation to support coherence heterogeneity is overwhelming, but as pointed out earlier, verification is a real concern, and the question of how to even integrate fundamentally different protocols that employ different states, or worse, different basic design principles, is uncharted territory.

1.4 Manager-Client Pairing Perspective

To resolve the issues of heterogeneous integration of coherence protocols, this dissertation presents Manager-Client Pairing (MCP), a methodology and interface definition for rapidly integrating and constructing coherence hierarchies composed of disparate parts. By defining a generalized interface, and providing a permissions gathering and releasing algorithm, MCP takes many of the complexities involved in coherence integration and distills them to an interface adherence problem. By taking the native functioning of a given component protocol and mapping these to corresponding MCP methods, the details of the component protocol's implementation become abstracted away from the composition of the hierarchy, hidden behind an interface layer.

A correctly ported MCP compliant protocol will result in a single upper interface at the 'manager' and multiple lower interfaces at the 'clients' of the MCP encapsulated protocol. These interfaces are designed such that the upper interface of one MCP compliant protocol can 'plug into' the lower interfaces of any other MCP compliant protocol without any addition modification. This creates manager-client pairs at these junction points and enables the composition of a tree of coherence protocols that distribute coherence permissions throughout the entire system. Finally, the interfaces are also designed such that processor requests map cleanly to the upper interface specification, and main memory maps cleanly to the lower interface specification, allowing processors and memory to be attached to the dangling interfaces of the MCP hierarchy. This development sequence is shown in Figure 1 and Figure 2. A closer look at the interfaces boundary can be found in Figure 25 of Chapter 6.

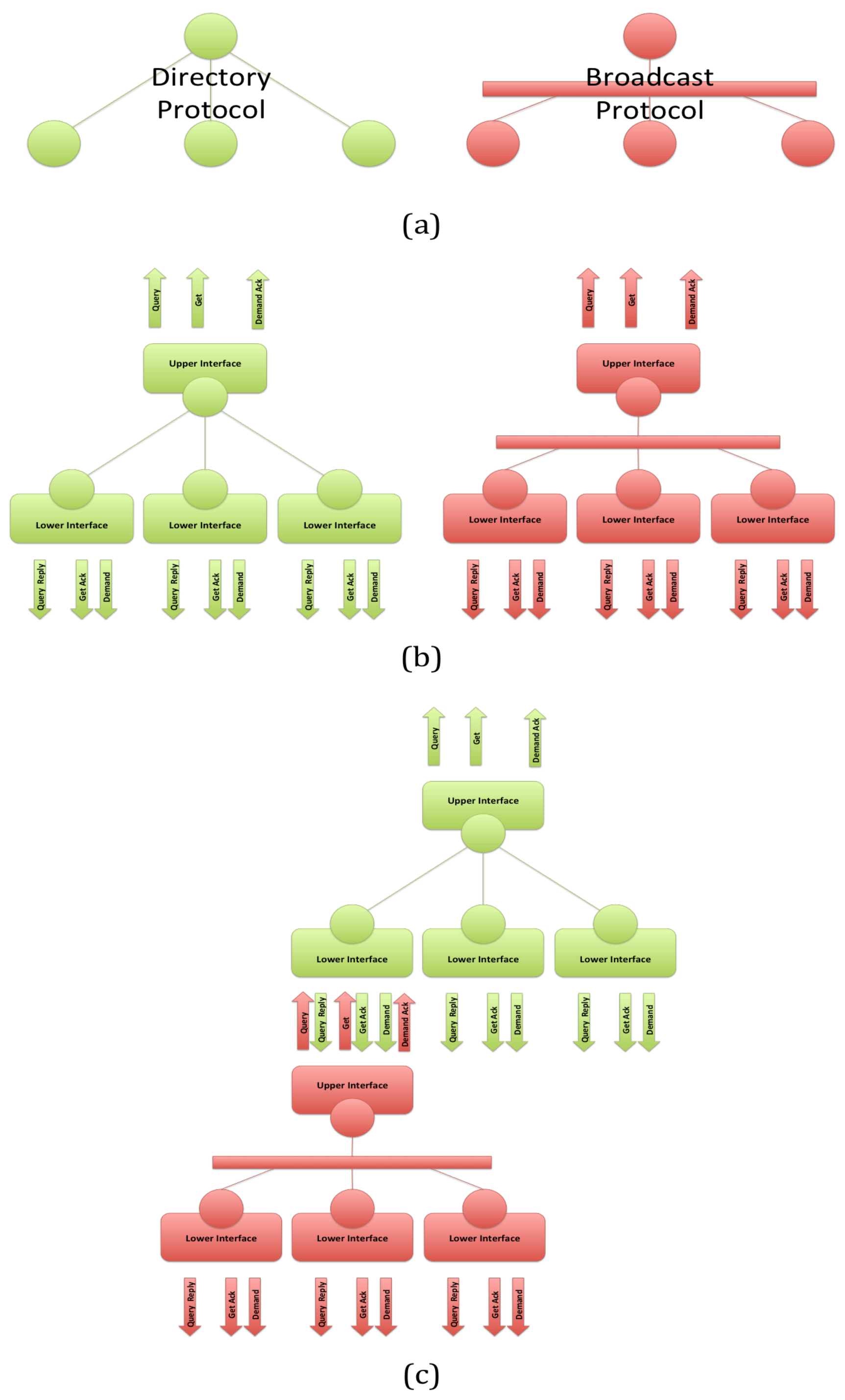


Figure 1 – Progression of coherence hierarchy composition using MCP showing (a) the starting protocols (b) the wrapping of the protocols with MCP interfaces and (c) the interfacing of an upper and lower protocol.

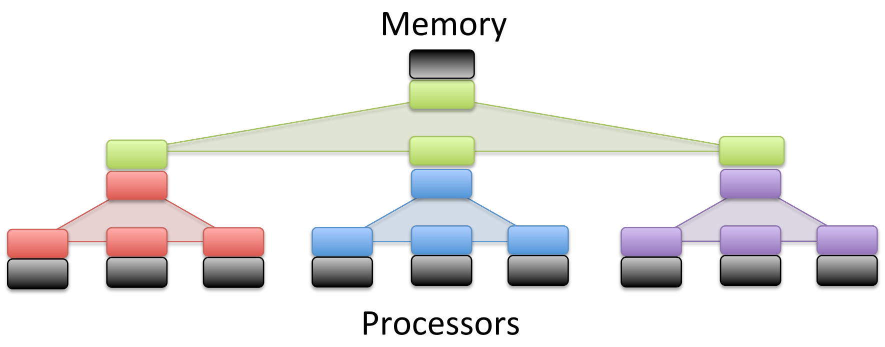


Figure 2 – Resultant protocol hierarchy after interfacing three lower-tier protocols with an upper-tier protocol, and attaching processors and memory to the remaining lower and upper interfaces.

1.5 The Fallacy of Composition

Now that a brief summary of MCP has been presented regarding simplifying integration, we now turn to the issue of verification. When considering verification in the context of a composition framework like MCP, we must consider the *fallacy of composition*: Can it safely be assumed that coherence protocol verifiability is a transferable property? In other words, can we simply assume the whole is verified if all parts are verified, or is a more formal explanation required? This sub-section will quickly explain the fallacy of composition, and explain why coherence verification is susceptible to this fallacy, thereby advocating the need for a more thorough examination of the verification composition properties of MCP.

The fallacy of composition is a logical fallacy whereby a property of the whole is assumed to exist via transference of this property from its constituent parts. A simple example that demonstrates this fallacy is: "Atoms are invisible. Humans are composed of Atoms. Humans, therefore, are invisible." The argument of composition, however, is not always non-validating as in the previous example. Consider the following: "A chair is wooden if it's legs, seat, back and arms are made of wood."[32] Here the property of *being wooden* is transferable from its parts to the whole. This begs the question, when is transferability applicable?

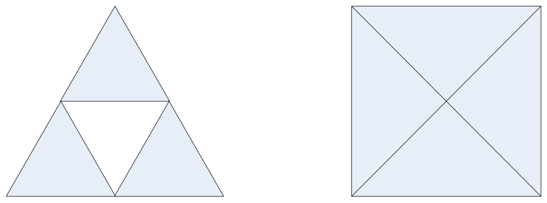


Figure 3 – (a) A triangle composed of triangles and (b) a non-triangle composed of triangles

Differentiating between validating and non-validating forms of the composition argument is reliant on two factors: *is the property in question absolute or relative*; and, *is the property structurally independent*? For brevity, the details of why relative properties are non-transferable, but absolute properties are, will not be discussed here, but can be found in [32]. It is sufficient to say that, since coherence verification is an absolute property (i.e., a protocol is either verified or is not), we have to look to structural independence to expose any logical fallacy. To aid in explaining what structural dependence is and how it introduces fallacious transference, an example from [32] will be borrowed w.r.t triangles. The property of being triangular is absolute; a shape either is or isn't triangular. However, being triangular is *structurally dependent*; the arrangement of the components involved is important in determining the property. Whereas a collection of wooden pieces, whether organized as a chair or just a heap of scraps, is still wooden, a triangle requires specific arrangement in order to exhibit this property.

The lack of structural independence does not mean that a property cannot be transferred to the whole; it simply means it is not *guaranteed* to be transferred. Figure 3 shows two shapes composed of triangles. Clearly, based on the structural arrangement, the property in question may or may not be transferred to the whole. Because of the structurally dependent nature of coherence verification (i.e., the addition of a another client or the introduction of a new state could both potentially invalidate a protocol), assuming compositional transference of verification would be a logical fallacy. Therefore, a stronger proof than a simple composition argument is required. Chapter 6 presents this proof, showing that the encapsulation provided by MCP enables a form of symmetry, which in turn allows each component protocol to be considered in isolation, while providing guarantees regarding livelock and deadlocks. To complete the analogy, much like the composite triangle of Figure 3(a) is only a triangle because the components are arranged in a triangular fashion (there is a triangle superimposed on the collection of triangles), MCP superimposes the verified nature and structure of an upper protocol on the organization/arrangement of the lower protocols.

1.6 Contributions

The contributions of this dissertation is as follows:

- This dissertation presents the Manager-Client Pairing framework as a means to enable rapid construction, and thus evaluation, of coherence protocol hierarchies. This includes defining the primary features of the Manager-Client Pairing interfaces and establishing the MCP algorithm, responsible for enforcing the permission inclusion property of hierarchies, and thus permission distribution. A demonstration of the rapid evaluation capabilities of MCP is also provided herein, exploring hierarchy width and height latency tradeoffs.

- This dissertation presents encapsulation symmetry, a technique for reducing the verification cost of coherence hierarchies. A proof is then provided to support the claim that MCP coherence hierarchies provide encapsulation symmetry, and thus the composite does not need to be re-verified if the component protocols are verified.
- This dissertation provides an analysis of verification cost, in terms of number of states explored and time-to-verify, for a MESI directory protocol, MOSI broadcast protocol, and a composition of the two in order to demonstrate the reduction afforded by encapsulation symmetry, and thus MCP.
- This dissertation also provides an analysis of the impact coherence hierarchy choices have for different common communication patterns found in modern multi-threaded workloads. Namely, Read-Only, Migratory and Producer-Consumer sharing patterns are evaluated as different coherence protocol states are enabled/disabled in different parts of the coherence hierarchy.

1.7 Organization

The remainder of the dissertation is organized as follows: Chapter 2 presents the basics of coherence protocol development and some pertinent design issues. This is followed by Chapter 3, which discusses related work in the areas of coherence protocol design, hierarchy composition, protocol verification and cache organization. Chapter 4 presents the tools used to assist in the development and evaluation of Manager-Client Pairing and MCP hierarchies. Chapter 5 then presents the Manager-Client Pairing framework, including discussion regarding the impact different hierarchy widths and heights has on communication latency. Chapter 6 presents a proof supporting the claim that coherence hierarchies composed of verified, Manager-Client Pairing compliant protocols do not require re-verification. Chapter 7 provides discussion regarding protocol design and the interaction between common application communication behaviors and hierarchical coherence design choices. This is followed by a summary of the conclusions drawn throughout this dissertation in Chapter 8.

CHAPTER 2 - COHERENCE PROTOCOL STATES AND DESIGN

There are many design choices and tradeoffs available when creating a coherence protocol. While there are the obvious parts of the specification regarding what states to enable, there are many other more subtle decisions to be made along the way, with varying tradeoffs concerns. These tradeoffs impact a dimension that is often overlooked or hand-waved away in the familiar dichotomy of architecture between energy/power vs. performance – design complexity. Architects are often willing to accept high degrees of design complexity if it affords a 'free' benefit in terms of performance or energy/power. However, coherence is somewhat special in architectural design in that coherence is widely accepted as one of the most difficult concepts in computer architecture due to the abstract, temporally complex possible interleaving one must consider. Even the simplest designs have many subtle complexities, and sophisticated coherence implementations are very error prone, often requiring several iterations of the design cycle to catch all the state-message pairings that need to be considered. In fact, [33] discusses the development of a research prototype in which 47 functional bugs in the cache controller and protocol were discovered during verification and simulation testing.

2.1 State Selection

Coherence protocol design begins with a selection of the states that will be employed to enforce coherence while providing performance benefit. This section will provide some background by enumerating the most commonly employed coherence states in current state-of-the-art protocols.

Starting with the absolute minimum necessary coherence states, a protocol implementing the I (invalid) and V (valid) states is sufficient to enforce coherence by only allowing one valid client to hold data at any time, thus granting both read and write permissions simultaneously. Typically, however, the V state is coupled with a clean/dirty bit in the client, and is better know by the two resultant states, the M (exclusive-modified) and E (exclusive-clean) states. Thus, a protocol implementing a single exclusive client policy is often referred to as an MEI (or MI) protocol. The most common extension to the MEI protocol is the Shared state (S), which came about as a replacement to the E state in MEI in order to provide the means for multiple shared read-only copies while forgoing write permissions in those clients. The resulting MSI protocol is the root starting point of nearly all modern coherence protocols.

It was recognized by Papamarcos and Patel [34] that re-enabling the E state in addition to the S state could provide an opportunity for performance benefit. The fundamental mechanism of the MESI protocol is that upon the first read to a data block by anyone, the read requestor is granted more permissions than are necessary in the form of the E state, and it is only lazily downgraded to the S state on the second read to the line. If no second read ever occurs, and a write request is made by the original reading client, that client is able to silently upgrade from E to M without notifying the manager, saving both bandwidth and communication latency.

Another optimization is the Owner state (O), and its related Forwarding (F) state. Beginning with the O state, it was recognized by Sweazy and Smith in [35] that the latencies and bandwidth involved in communicating dirty data back to main-memory when another client reads after a write operation could be reduced substantially if the protocol provided support for dirty shared copies. One challenge with this, however, is evictions, since it is common practice to implement the shared state with silent eviction capabilities so they can downgrade from S to I without incurring any latency or invalidation traffic. The danger then is, if all dirty S blocks evict, the writeback to memory never happens. The O state resolves this by creating a new kind of sharer who not only has to get special permission before evicting (via a writeback), but is also responsible for providing data to any new sharers to avoid off chip accesses that would gather stale data rather than the most recent dirty copy. While these sharers technically hold dirty data provided by the owner, for all intents and purposes they function identically to a 'clean' S state. Though functionally equivalent, for clarity purposes, the subscript O has been added to differentiate between a sharer that contains dirty data provided by an owner block and one whose data is clean. Additionally, to disambiguate between the two S implementation regarding eviction, the subscripts I and W have also been added to the S state to differentiate between an S implementation that can silently evict to the I state (S_I), and the S implementation which requires additional writeback traffic to the manager (S_W), shown in Figure 4.

Coming back to the O state, it was recognized that in certain designs, the forwarding capabilities of the O provided benefit in terms of cache-to-cache transfers, even when writeback avoidance is not an issue. Hence the F, or clean forwarder state, was introduced [36]. The F state is often implemented such that it migrates around the protocol, transitioning to the newest reader as the previous F client downgrades itself back to a conventional S. This has the dual benefit of avoiding hot-spots when a single address sees a burst of request traffic, and also helps in terms of replacement since the

MRU load event across threads holds the block in the F state, reducing the probability of eviction.

Finally, there are two design variations to the already discussed M and E states. While it is frequently the default implementation to provide the M and E states with forwarding capabilities as part of their design, in some instances this is not an option. For example, the Intel Nehalem's implementation of MESI cannot support any client transfers as the L2 caches are connected directly to the L3 cache with no side-band links to one another. Architecturally this makes sense, but it also means that every time a read-after-write is observed across nodes, the L2 client in the M state cannot forward data to the requestor. Rather it is forced to do a writeback into the L3, where the L3 can provide the data to the requestor. In order to help distinguish instances where the E and M state can forward from those when it must invoke writeback traffic, the F and W subscripts have been added to the E and M states as presented in Figure 4.

Figure 4 provides a Venn diagram of all the states mentioned in this section, visually demonstrating the different overlapping properties between them w.r.t having clean/dirty data compared to the manager's copy, having forwarding capability/responsibility, and the various kinds of permissions allotted to each (Read, Write and Eviction). Unfortunately, nothing like the subscripting nomenclature used in this section and Figure 4 has ever been employed before in the coherence literature to help clarify some of these distinctions, and most papers simply do not mention the subtle choices being made. This leaves room for interpretation, which often introduces the minor ambiguities encountered when discussing or implementing coherence protocols.

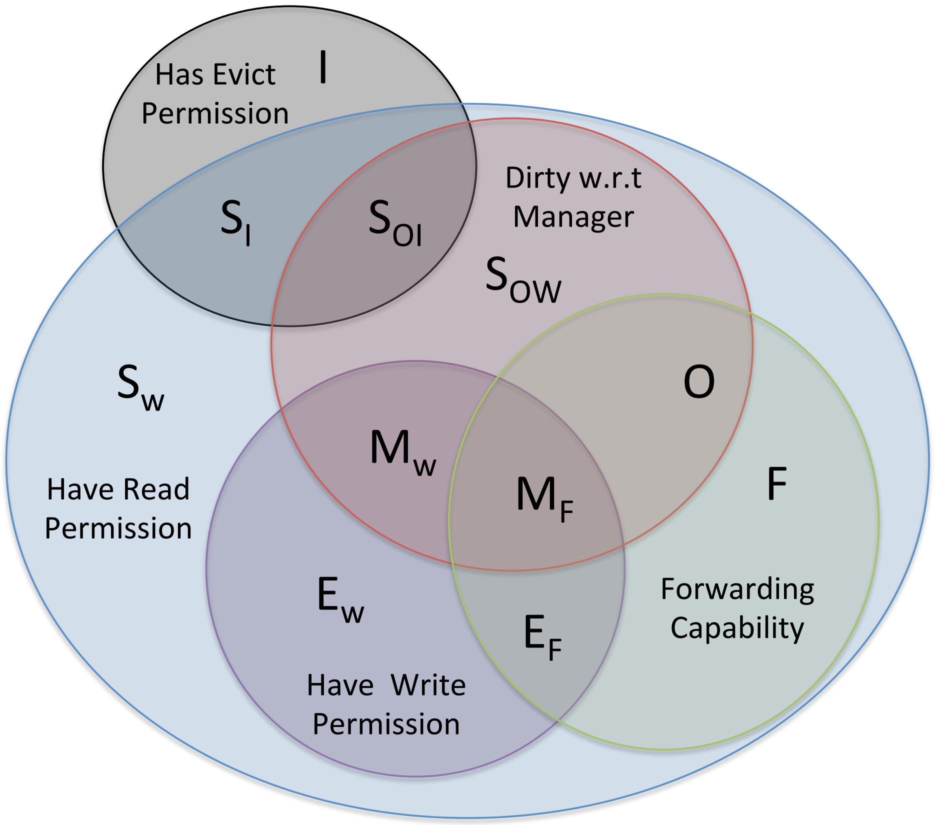


Figure 4 - Venn diagram of coherence state membership in the spaces of permissions, forwarding capability and whether data is dirty, including the subtle differences between common implementation choices for the most frequently employed states.

2.2 Design Issues For a Simple MSI Protocol

To demonstrate the iterative thinking required during design and impart an appreciation for coherence temporal complexity and race potential, we will walk through the first steps of constructing a basic MSI protocol. The natural starting point is to begin with client states M, S, I, and the corresponding states for the manager (Figure 5). Next we consider what is required to transition between these states, introducing our first set of request messages from the client to the manager and first set of transient states: GetI/SI, GetI/MI (for evictions from shared or modified), GetS/IS (for loads), and GetM/IM, GetUpgrade/SM (for stores from invalid and shared, differentiating between when data is required in addition to the permission). We can now consider the reaction of the manager to these requests, introducing the first response messages: GetIAck, GetSAck, GetMAck for a Manager in the Invalid state, Fwd_Invalidate when the client request is a GetM or GetUpgrade and the manager is in the S state (requiring sharers to invalidate and forward their invalidation acknowledgement to the new owner), and FwdS, FwdM for when the manager is in the Modified state and needs to invoke the help of the owner client to satisfy the request. For this exercise, we will ignore the options regarding blocking vs. non-blocking protocols, non-forwarding vs. forwarding, network guarantees, and other choices/constraints. Assuming a non-blocking, forwarding protocol on an ordered network, the manager can update the state, owner and/or sharer fields immediately, and fire-and-forget the response, ready to process more traffic. Upon reception of the GetAck responses, the client can enter the non-transient state they were pursuing, and the sequence is complete.

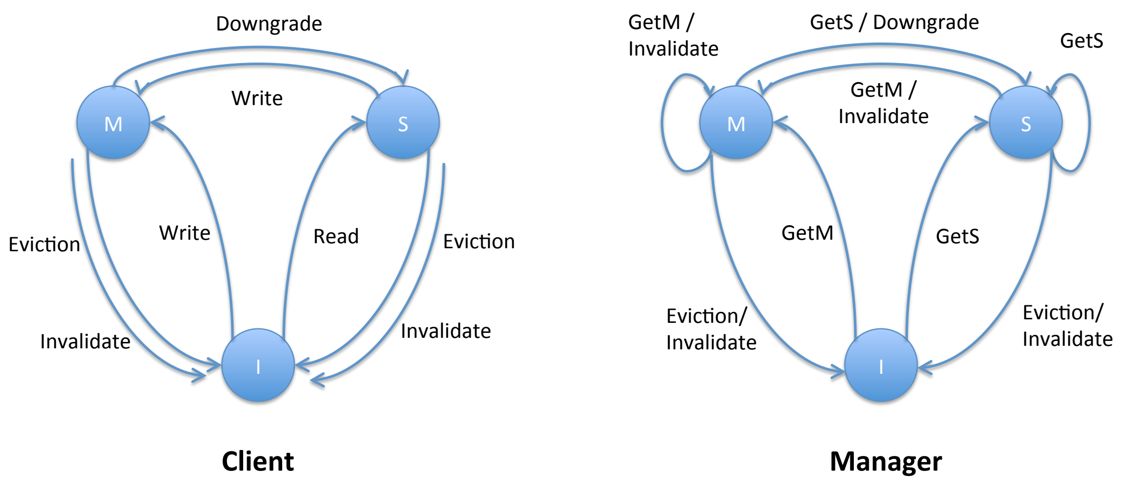


Figure 5 - Basic state machines for an MSI protocol, excluding transient states

This seems simple enough, but now we are left to consider the Fwd-type messages. This is where the first set of complexities emerges. If we reiterate over the states we've established up to this point, we discover that it is possible for a client to receive a Fwd_Invalidate, while in the SM state. The manager effectively is asking the client to invalidate itself while the client simultaneously has a permission upgrade in flight. Other similar race conditions exist as well, such as receiving a Fwd_Invalidate while in the SI state (asked to invalidate while the client is already evicting), or a Fwd_S or Fwd_M being received while in the MI state. Each of these race scenarios require a solution, which in turn create more transient states, messages, and assumptions about how events are handled (e.g. the invalidation while in SM, will that in-flight Get_Upgrade be converted by the manager into a simple GetM, or will it be dropped and the client is expected to re-issue?). These again must iteratively be considered against the current state table possibilities until the steady-state situation of no new messages or states being introduced to the design is reached.

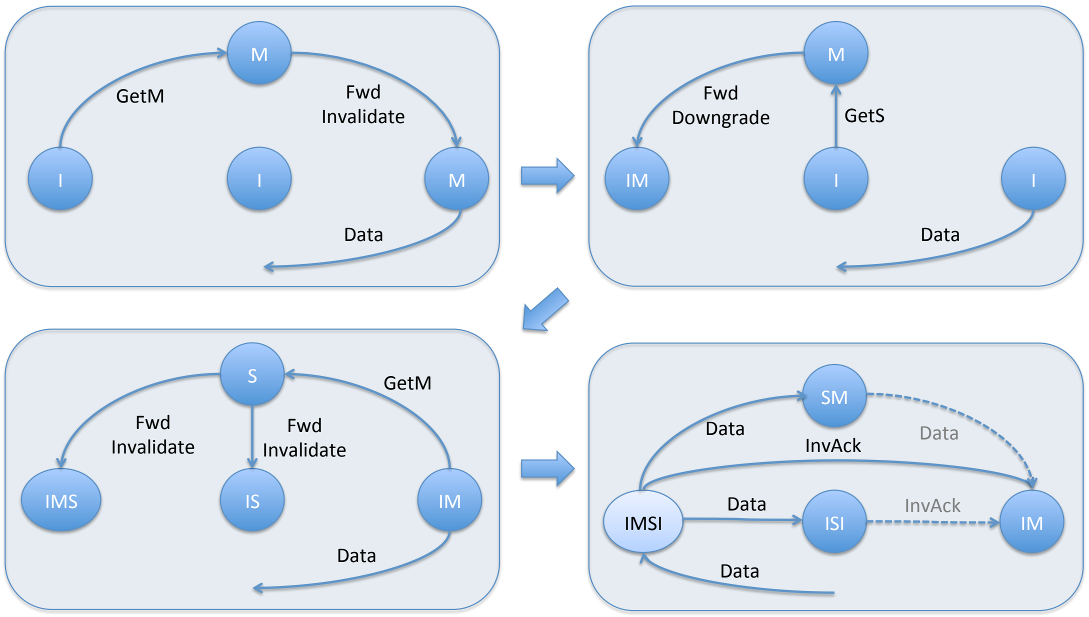


Figure 6 - Coherence sequence that demonstrates the IMSI state race

To see where this eventually leads, fast-forwarding this design sequence a few iterations forward will yield states such as the client IMSI state, indicating an instance where a client requested write permissions from the invalid state, but before receiving the response from the owner client, it receives a Fwd_S from the manager (due to another reader), followed by a Fwd_Invalidate (due to another writer) (Figure 6). The IMSI state is encoding that the client's request has been observed by the manager, a response to it is in flight from the previous owner, and upon reception of that data payload the client can process its own store, immediately forward the data to the Fwd_S initiating client, then self invalidate and send an Invalidation_Ack to the Fwd_Invalidate initiating client. A fairly complex scenario for implementing a non-blocking protocol that doesn't even consider the exclusive (E) or owned (O) states. This is not just a pathological worst-case construed to make a point regarding the design complexity involved in coherence; a state very similar to the described IMSI state actually exists in several of the GEMS implementations of MOSI, the IMOI state [37].

2.3 Other Design Considerations

As alluded to during the design of the MSI protocol, there are choices or constraints beyond just the states involved in protocol selection that have to be considered. Section 2.1.1 already discussed the notion of silent eviction and support for cache-to-cache forwarding. Several other choices are a result of the network, which are either seen as constraints to be designed around, or options in a co-design environment. Ordering guarantees can simplify aspects of coherence, but unordered networks usually have a better performance profile, due to techniques such as dynamic routing [38]. There may also be other design considerations, such as exploiting properties of a tree-based topology that echo back event ordering to the leaf nodes, as leveraged in Fractal Coherence [39].

The decision to employ a blocking vs. non-blocking strategy has implications as well. A blocking strategy effectively locks the manager on request processing, forcing other request traffic to the same address to stall until the current request has completed. This serialization of requests in the manager makes states such as IMSI from the discussion in Section 2.1.2 unnecessary. Blocking protocols make it easier to reason about the design, as well as improve verification properties by reducing the state space, but come at a performance cost. Not only is some coherence parallelism lost due to the removal of concurrency, but blocking protocols also require more buffering at the manager and additional unblock messages to signal to the manager that the sequence is complete to 'unlock' the manager for that address. The final unblock message, however, is not on the critical path, and the concurrency loss is isolated to single address serialization. Previous work [40-42] also show that races are not the common case and should not dominate design choices. Unless otherwise specified, the protocols used in this dissertation employ a blocking strategy.

CHAPTER 3 - RELATED WORK

Coherence became an integral part of computer architecture the moment caching and parallel execution where used simultaneously to improve performance. Over the past several decades many advances have been made to improve several aspects of coherence design, including communication latency, bandwidth consumption, evaluation and design complexity, and verification. This chapter will enumerate some of the biggest contributions that have helped shape modern coherence design principles, as well as discuss the techniques most closely related to the improvements to the state-of-the-art this dissertation presents. To reduce the scope of this chapter, considering the scale and age of this branch of architectural research, priority is given especially to work that has contributed to the enhancement of hierarchical coherence research.

3.1 Pioneering Work in Coherence Protocol Development

In 1978, Censier and Feautrier present the first directory-based protocol, which they dubbed the 'Presence flag technique' [43], proposing the use of presence bits in a centralized structure in order to track coherence state. This was done to reduce the volume of broadcast invalidation traffic and processing, which at the time was sent down to all processor caches on each write, whether or not they held a copy of the data. The authors suggest this was so costly under the 'classical solution' (broadcast) that this parasitic traffic dominated bandwidth, restricting scalability to only two nodes. It is interesting to note that, despite its age, this seminal work also uses heterogeneity as part of its justification, arguing that asymmetries in the communication fabric force all traffic to adhere to the worst-case communication behavior under broadcast, whereas use of a directory restricts the penalty to only the worst case of the processors involved in the coherence event.

In 1984, the first major innovation in new ways of thinking about coherence states is what became known as the Illinois protocol, for being developed at the University of Illinois, and is better known today as the MESI protocol [34]. As explained in Chapter 2, the MESI protocol introduces the exclusive-unmodified state to the conventional three state MSI protocol. This innovation enables the first-reader of a data element to acquire not only a clean copy of the data, but metadata informing the cache that it is exclusive as well. This in turn enables the cache to perform a silent upgrade to the modified state without inducing the conventional traffic associated with a write. In the presence of write-after-read traffic within a single node, this additional state can yield considerable improvement to the bandwidth and latency impact of coherence.

Soon after in 1986, Sweazy and Smith introduced the 5th and last of the canonical coherence states of modern coherence, the Owner state [35]. In their MOESI protocol, they add the notion of block ownership to MESI in order to fully realize the benefits of writeback caches in a multithreaded coherent environment. By introducing ownership to coherence, ownership could be taken away from memory and given to processor caches. Since data supply responsibility is associated with block ownership, this effectively gives a cache block in the O state privileged status over main memory. As a result, fewer writebacks to main memory are incurred because dirty state can reside outside of main memory longer; a transition back into the M state from the O state saves what would have been a writeback under MESI. The transfer of ownership away from memory also meant that the owner cache, which can now maintain dirty contents, is able to supply this most recent dirty copy to other sharers upon read request, giving another benefit of replacing slow main memory accesses with high-speed cache-to-cache transfers, in addition to the originally intended writeback traffic reduction.

The earliest reference to coherence distribution via hierarchies is presented in 1987 by Wilson [44]. Hierarchical coherence scope is restricted to interconnected buses, where directories for each lower bus snoops on the higher bus for relevant traffic (effectively *bus-bus-…-bus* hierarchies). Wilson points out many design concerns, especially those regarding the high bandwidth required at the top-tier protocol. Wallach's master's thesis [45] then describes an extension to Wilson's approach, using a tree-based coherence protocols for *k*-ary *n*-cube topologies, rather than broadcast-based distributed-bus systems. Read requests are satisfied at the lowest common sub-tree, and write invalidations only include the smallest sub-tree that encompasses all sharers, thus reducing traffic.

Finally, while not an innovation with regard to changes to coherence protocols, the high degree of impact Wisconsin's Multifacet GEMS has had in the community and its utility as a modeling tool, specifically the Ruby module, warrants mention in this section, being the first significant open-source coherence protocol simulator. Nearly all modern coherence research from academia provides a reference to the GEMS toolset as a contributing factor in their research, whether directly used as a tool, or indirectly as a source of how to approach the implementation of a specific protocol. GEMS provides several coherent cache models in a custom descriptive language called 'slicc', and separates the coherence operation of each protocol into files associated with each coherence agent's state machine (clients and the manager under the MCP taxonomy).

Slicc describes coherence through a collection of {pre-state, event, post-state} tuples, each of which is associated with a set of actions that execute on state transition upon trigger-event arrival. These actions often entail producing more events, in the form of more coherence messages, driving the simulation forward.

It is worth mentioning that the coherence protocols developed for use in this dissertation, as well as some of the simulation framework software practices of the CaffeineSim simulator's protocol engine (discussed in Section 4.1) were heavily influenced by my experiences with the slicc coherence description format. I personally owe a debt of gratitude to the creators of GEMS for providing the foundation of my understanding of coherence operation, acquired through the use of their tool.

3.2 Architectures Employing Coherence Hierarchies

The Data Diffusion Machine [46] is the earliest instance of a distributed shared memory machine, merging the best of both message passing and shared memory systems of the time, providing a distributed shared view of memory. From a coherence perspective, the most important contribution of the data diffusion machine is that it demonstrated the benefits a hierarchical protocol can provide. The Data Diffusion Machine implements a broadcast bus hierarchy much like that described by Wilson [44], where each bus controller restricts broadcast traffic to only be observable by other relevant nodes within the hierarchy. The DDM, however, also supports several coherence optimizations, most notably a hierarchy-aware optimization that supports 'read combining' to avoid redundant traffic at higher levels in the broadcast hierarchy.

The DASH multiprocessor [30] uses a distributed MOSI directory protocol to manage coherence across clusters, but employs a broadcast bus protocol within clusters for managing intra-cluster communication. The coherence protocol of DASH is perhaps the first instance of differentiating between local, remote and home clusters with regard in the distribution of coherence responsibilities for complex coherence hierarchies. DASH, being the first heterogeneous coherence hierarchy also employs a powerful technique, the pseudo-CPU, which is an agent that can act on behalf of external processors in the local intra-cluster protocol, similar to the proxy client technique discussed in Chapter 6.

The SGI Origin [29] can be viewed as having an improved version of DASH's protocol, implement a coherence hierarchy/protocol similar to that of DASH, but providing several enhancements, such as implementing the E state of MESI, and providing this clean exclusive state with silent eviction support. Wildfire [27] is a distributed shared memory system designed with an emphasis on extracting additional performance by creating more memory locality in the local cluster memory system by creating shadow pages of remote pages in an R-NUMA [47] fashion. Like the SGI Origin, Wildfire also employs a hierarchical strategy similar to DASH, invoking a global coherence layer as necessary, but otherwise performing regular SMP behavior. Wildfire implements a MOSI protocol, rather than MESI or MSI.

The Piranha architecture [25] employs a hierarchical coherence strategy through the use of two programmable coherence engines, a home engine for exporting requests for addresses where the local node is the home node, and a remote engine that is responsible for importing memory. Unlike DASH, however, the protocols of Piranha are more tightly integrated, so intra-cluster communication does not require a pseudo-CPU-like mechanism for making remote requests mimic local traffic. Several additional enhancements are also employed, such as switching between limited pointers [48] and coarse vector [49] sharer tracking in the directories, using NACK-free protocols, and implementing cruise-missile-invalidates, a technique for executing multiple invalidations with a single coherence event.

Most recently the HP Superdome [26] demonstrates a different benefit of hierarchical coherence composition, the ability to create a composite system from parts designed by another group. In the case of Superdome, the primary goal is to provide support for constructing a machine from commodity parts (namely Itanium processors [50]) by providing and interconnection fabric that captures local traffic off the front-side bus, processes it, and transmits appropriate traffic around the system to other cells whenever the relevant commodity parts would require observing traffic to stay coherence. This is accomplished through a crossbar interconnect and directory protocol which is capturing/recording traffic in order to know what nodes need to observe what traffic and when.

3.3 Improvements to Coherence Hardware Scaling

Because coherence hierarchies have natural tracking storage reduction, due to the clustered and summarizing nature of coherence hierarchy permission inclusion, the highest impact research in improving coherence scalability, through size reduction will be highlighted in this subsection.

Two early popular techniques which have become common-place options, proposed by Gupta et al., are coarse vector tracking, which merges nodes when the number of nodes exceeds the sharer bit-vector size (width reduction), and sparse directories which merge directory entries from different addresses into a single entry (height reduction) [49]. Limited directories [48] use a collection of IDs, rather than a bit vector to exploit commonly observed low sharing degree. Taking this the next step, rather than use a static number of IDs per entry, a shared linked-list structure of IDs enables a more dynamic version of this technique [51]. Taking this concept in a different direction by using IDs to have sharers track each other, Scalable Coherence Interface (SCI) became popular as a scalable protocol alternative which uses ID pointers embedded in the sharers to effectively construct a linked-list of sharers [52].

Some more exotic techniques have been proposed in recent years as scalability concerns regarding node count have come to the forefront. Acacio et al. propose a 2-level structure to create a hybrid mechanism to gain the benefits of full tracking for a small subset of recently cached entries, backed by a larger coarse-tracking structure [12]. Waypoint [14] provide a comprehensive description of many previous size reduction techniques, then proposes a tracking mechanism that supports directory entry overflow to avoid the normal entry invalidation associated with an entry eviction due to low directory associativity. The Cuckoo Directory [13] attempts to resolve the same problem, but rather than using an auxiliary overflow structure, disjoint replacement candidates are rotated around the storage structure until a good candidate entry in the replacement chain is found. Tagless Coherence Directory [17], which uses bloom filters to outright remove directory tags from the structure and instead leverage techniques to extract appropriate sharer information. Snoop Probes [53] presents a technique for reducing power for broadcast protocols by recognizing differences in coherence requirements for private vs. non-private data. Finally, by engaging the operating system's page table management, metadata can be added to page entries to track when data is privately accessed by only one node. This can yield a major reduction in directory structure sizing by eliminating the need to maintain any coherence for these sets of privately accessed addresses [54].

3.4 Protocol Design and Integration

This dissertation has already established that the design of coherence protocols, and especially the integration of disjoint protocols, is a difficult challenge. Manager-Client Pairing provides a means to ease protocol integration, but is not the first work to alleviate some of the challenges faced by coherence architects.

Perhaps one of the most famous contributions towards simplifying aspects of coherence design is Token Coherence [40, 41, 55]. Under Token coherence, the design concerns of performance and correctness can be decoupled by designing for the common case and allowing the use of tokens and persistent requests to resolve race conditions and other complex scenarios that must be resolved and designed for, but are infrequent enough that they should not heavily influence design decisions. Marty [56] in his dissertation described a solution to extend coherence across a multi-CMP system through the use of token coherence as an additional decoupled connecting layer, showing the value token coherence can impart for hierarchy composition.

Atomic Coherence [42] advocates the use of an 'instantaneous' nanophotonics substrate to effectively give coherence a set of mutexes that can be leveraged to serialize coherence events. This in turn eliminates many of the transient states required for correctness, which are often a source of design errors due to conflicting race scenarios. In discussion with the authors of this work, they mentioned it is not unlike the software trick of using a global lock to simplify coherence simulation, which was in fact part of the inspiration behind their inventing this technique.

The work by Suh et al. [57-61] is related to Manager-Client Pairing in that it also considers some of the issues involved with heterogeneous integration. Specifically, their work investigates the issues involved with finding congruence between the permission levels of different states in different protocols, and typically at much smaller scales (e.g. integration of 2-4 cores on a bus) due to the embedded system-on-chip nature of the work. Suh's solutions tend to prioritize integration complexity reduction over performance due to the nature of the components involved. For example, [61] discusses disallowing the exclusive state due to the complicated nature of integrating E-type behavior with a protocol that only implements MSI. Some of these issues of state integration will be visited in Chapter 7 of this dissertation.

Another recent integration effort addresses a different kind of coherence heterogeneity than that with which Manager-Client Pairing is concerned. Cohesion [62] proposes a hybrid memory model for supporting both hardware and software managed coherence. Kelm et al. make arguments for the strengths and weakness of both coherence management systems, then present a framework to enable the integration of both by providing the necessary support to transition management responsibilities from one domain to the other.

In the space of hierarchical composition, Ladan-Mozes and Leiserson present Hierarchical Cache Consistency (HCC) [63], a deadlock-free, tree-based coherence protocol which can ensure forward progress in a fat-tree network. While the design presented is reasonably straight forward, this work warrants reference in this section due to the valuable discussion regarding the design issues that are encountered during hierarchy construction. The section enumerating invariant properties adhered to by HCC is especially noteworthy, most of which are also reflected in the Manager-Client Pairing framework. Due to nature of the design specification, HCC also lends itself to providing guarantees with regard to verification as well.

3.5 Protocol Verification

An important work that eases *homogeneous* hierarchical coherence verification is Fractal Coherence [39]. In this work, Zhang et al. propose a tree-based coherence protocol, with the intention of simplifying coherence verification through perfect self-similarity. A fractal based coherence protocol, where children are coherent with their parents, can be verified through the validation of only the kernel coherence protocol. The authors also describe how a bus-based version of the protocol could also be executed through fractal buses. Manager-Client Pairing has several features in common with Fractal Coherence, especially since the recursive nature of the interfaces proposed by MCP can be seen as analogous to the self-similarity of fractal coherence's kernel protocol.

The most important distinction between this prior work and the verification proof presented in this dissertation for Manager-Client Pairing is that fractal coherence was specifically designed with *homogeneity* as a requirement. Fractal coherence does not discuss the benefits of heterogeneous coherence composition nor why it is an important consideration, and is explicitly incompatible with heterogeneity since it relies heavily on homogeneity as part of the proof. It is worth mentioning, however, that because fractal coherence does produce a verified coherence protocol hierarchy, it would be compatible as a component within an MCP coherence hierarchy if made MCP compliant via MCP interfaces.

MurPhi [22] is an enumerative state model checker that can be used to check for invariant violations at every reachable point in the global state space of a coherence protocol. Being a verification tool, the MurPhi verification framework will be discussed in depth in Chapter 4, rather than here. For a survey of other coherence verification techniques and optimizations, we point the reader to the work by Pong and Dubois [64].

3.6 Other Recent Contributions and Innovations

With a resurgence of interest in multicore, and thus coherence and communication, there are a few proposal also that deserve brief mention in this dissertation. Marty and Hill [65] discuss a technique to exploit the coherence decoupling provided by hierarchies to turn *coherence realms* into *coherence domains* to enable virtual-machine hierarchies on a manycore substrate. Enright-Jerger et al. propose virtual tree coherence (VTC) [18] as a mechanism that enforces coherence ordering and provides multicast support through virtual sharer trees overlaid on the network, improving bandwidth and storage overheads while providing fast cache-to-cache transfers. Unified Instruction/Translation/Data (UNITD) coherence [66] proposes a protocol that makes TLBs participate in coherence traffic in addition to instruction and data to remove the need for software TLB shootdowns and their associated poor scalability and high cost. Last, Aisopos and Peh [67] propose a perspective shift and associated methodology for adding minor modifications during coherence protocol design to improve the resilience properties of coherence in the presence of soft errors, such as providing tolerance for dropped packets.

CHAPTER 4 - SIMULATION AND MODELING TOOLS

Several different tools were employed during the course of this research effort. In order to give a comprehensive understanding of the conclusions drawn throughout this work and enumerate any simplifications that may impact the result herein, this chapter discusses these tools, their strengths, their weaknesses, and any shortcomings or intentional simplifications to highlight how these decisions may influence the inferences made.

4.1 CaffeineSim Architectural Simulator

CaffeineSim is a homebrew architectural simulator that began development in the Tinker group at The Georgia Institute of Technology in the Summer of 2009 as an offshoot of a special topics course on manycore architectures taught by Thomas M. Conte and TA-ed by myself and Chad Rosier in the Spring term. The foundation of CaffeineSim is rooted in the final project's design spec, a detailed router pipeline model used to implement an arbitrarily sized mesh topology with fully configurable parameters, populated with coherent cache modules from an earlier course project. While the caches and coherence were eventually replaced, the interconnect network model laid the foundations for all of the core simulation engine features and remains an integral part of CaffeineSim as of this writing.

Due to the exploratory nature of the special topics course and the relatively 'new' nature of manycore design constraints and issues, we quickly discovered that many of the existing simulation tools that were readily available at the time (SESC [68], Pin [69], SimpleScalar [70], PTLSim [71], GEMS [37]) simply did not have sufficient support to explore this new area effectively, most focusing on the core architecture rather than the un-core. At the time, GEMs provided the closest functionality to what we desired for manycore research, due to the Ruby memory module's extensive memory hierarchy and coherence support. Even so, GEMs was seriously hindered by a limitation in the core count due to the guest OS, being most effective for CMP research of 16 threads, and having support for 64 threads only through extensive hacking. This was unacceptable for the manycore research the Tinker group wanted to consider, which begins at the 64-core data point and scaled up from there.

As a result, CaffeineSim was created to bridge this gap in the architecture simulation tools landscape. Interestingly enough, MCP was initially invented during the development of CaffeineSim as a means to an end, rather than a topic in its own right. MCP was originally created as a part of CaffeineSim to enable rapid study of hierarchical coherence organizations by leveraging polymorphism for composable hierarchy construction. It quickly became apparent that the concepts behind MCP had merit beyond the scope of software engineering and simulator design, and were applicable to real-world hardware design concerns.

4.1.1 CaffeineSim Organization and Engine

CaffeineSim is a clock-based simulator with event-driven features. All architectural components within CaffeineSim inherit from 'Module', which provides the naming, identification, event delivery, and time progression support required for communication between components. All modules communicate with one another through a universally defined 'mreq' object, which is embedded with module-traversal history logging capability, for rapid debugging, in addition to the more standard source, destination, timing and request/message content.

A global ‘Sim’ object manages all the modules within a simulation instance by instantiating an array of node objects (Figure 7). Each node is populated according to the specification provided via a settings/configuration infrastructure. This process includes assigning the coherence protocols to the L1-L2 and L2-L3 caches, realm membership for construction of neighborhoods, as well as assigning whether or not the L3 is a cache that has a backing store or is strictly a tag-only directory structure.

In addition to node and module instantiation, the ‘Sim’ object is responsible for interfacing the SESC MIPS emulator front end with the processor models, attaching the appropriate SESC thread contexts to processors, as well as providing basic operating system modeling functionality, such as scheduling and virtual-to-physical address translation. During execution, the ‘Sim’ object is responsible for advancing the clock and monitoring execution to ensure forward progress is being made in all active cores. If progress on an active core unexpectedly halts due to a very-long outstanding request, the ‘Sim’ object will report when and where this failure has occurred as well as many details regarding the offending request. This includes the associated mreq’s traversal history, and other information that is often relevant to the failure such as all outstanding in-flight requests to the same address, all cache and directory entries in all levels of the hierarchy that match the offending address at the time of failure detection, and even network state. Finally, the ‘Sim’ object is also responsible for statistics management, merging and printing at the end of simulation execution.

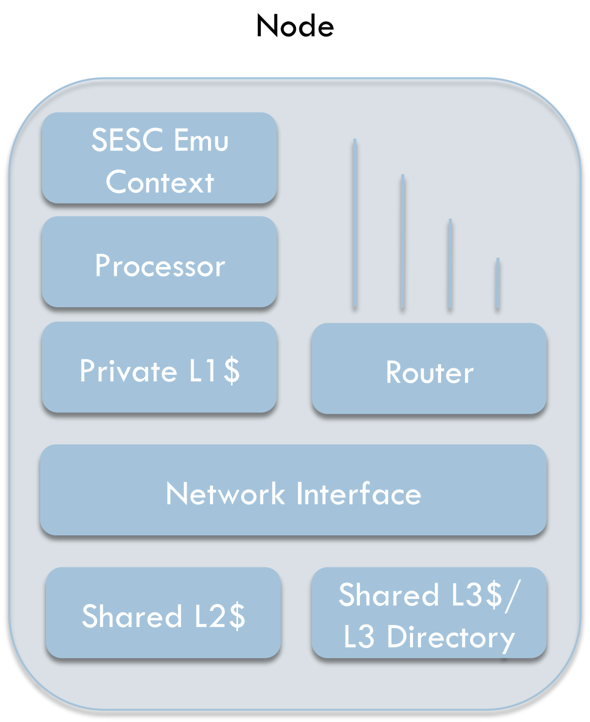


Figure 7 – CaffeineSim Node, containing modules representing architectural components

4.1.2 CaffeineSim Frontend

One of the earliest design decisions for CaffeineSim was regarding what kind of frontend to support, whether trace driven execution, an emulation-directed execute at fetch model or a slow, high fidelity execute at execute model. Ultimately we settled on emulation-directed execute at fetch for two reasons. First, Fellow Tinker member Paul Bryan had been investigating the issues involved with multicore and manycore simulation, specifically how to apply acceleration techniques, such as statistical sampling, to a multithreaded environment. The beginning inklings of what later became known as ‘thread-slip’[1][72] arose in these early discussions, motivating us away from the use of a trace-driven front end and towards a more feedback oriented front-end. Second, for performance and development reasons, execute-at-fetch was chosen over execute-at-execute. Leveraging previous familiarity with SESC, the SESC MIPS emulator frontend was stripped down and combined with a simple scheduler/OS modeling mechanism to feed our processor models with instruction steams.

4.1.3 CaffeineSim Processor Models

CaffeineSim provides support for three different processor models of varying degrees of fidelity: A trace processor, a static IPC memory source, and an in-order pipeline

The trace processor, as the name implies, is a fairly simple trace consuming processor that bypasses the SESC execute-at-fetch front end entirely. This is largely used for development purposes, enabling basic regression testing in the cache/network/memory structures as code expands, as well as providing a means to stress-test coherence and reverse engineer the causes of the many coherence deadlock and livelock scenarios that were encountered during early development.

Another technique employed through the use of the trace processor is the creation of hand-tuned traces for highly controlled experiments that examine in fine detail specific phenomenon. This is a useful technique for evaluating the impact architectural changes

[1] Thread-slip is a simulation artifact that can emerge when processors are allowed to lag/lead one another w.r.t timing compared to a ‘true’ simulation. This happens due to a lack of enforcing timing-model feedback-pressure on the threads/instruction streams. This thread lag/lead ‘slip’ can have considerable

have on these sequences, so much so that it is employed in Chapter 7's evaluation of protocol hierarchy state combinations and interactions. By isolating instruction/memory streams in this way, we remove all external traffic that can introduce varying degrees of noise to the target behavior. While this interference is a very important aspect of architectural research and should not be ignored, it can distort the perception of how or why things work when multiple phenomenon are occurring simultaneously, as is often the case in benchmark simulation result interpretation.

The second processor model, the static-IPC model, is useful when speed is the most important consideration and the details of processor operation can be roughly approximated. Each processor supports basic load-store queue (LSQ) functionality and acts as a kind of filter, consuming instructions from the frontend at a parameterized fixed rate (e.g. 0.5 IPC), discarding all non-memory operations and issuing all memory operations that miss in the LSQ to the cache hierarchy. This model has a variable number of miss status handling registers (MSHRs), which creates backpressure to the frontend, halting execution when all MSHRs are in use. Finally, backpressure can also be observed if the LSQ becomes full, most often due to a single long-latency event stalling at the head of the queue.

The last processor model is the most accurate and slowest of the three. This model faithfully models a simple in-order 5-stage pipeline in a fairly straightforward manner. It is worth noting that this model, as well as the static-IPC model, both support a degree of thread-awareness that enables research dealing with thread swapping and migration concerns. None of the processor models in CaffeineSim simulate instruction fetch memory requests, and therefore CaffeineSim does not support instruction cache modeling.

4.1.4 Coherent Cache Hierarchy

The most sophisticated and complex portion of CaffeineSim is the coherent cache hierarchy. This is due to the foci of Tinker research efforts for manycore centering primarily on the uncore and related memory concerns. The cache responsibility is divided between two major parts: the hash_table and the coherence engine.

The hash_table is responsible for management of cache sizing, indexing, replacement, entry allocation and deallocation, incoming request buffering/stalling, store buffering and MSHR handling. It is effectively where all the heavy lifting of an accurate, event driven cache simulation is done. This is also where the MCP algorithm, described later in detail in Chapter 5, is implemented. Upon mreq delivery, the MCP algorithm is executed to manage proper permission distribution and coherence management across a hierarchy divided into neighborhoods of physically local caches (Figure 8).

The coherence engine implements the state-machine of the coherence protocols being modeled in CaffeineSim. Each hash_entry within the hash_table contains a field for a client protocol and a manager protocol object (where the manager-client pairing takes place), also shown in Figure 8. Object-oriented polymorphism is employed to enable easy swapping of the protocols while also enforcing adherence to the MCP interface definition. This way, the hash_table can manage coherence through the state machines defined by the coherence engines while being unaware of implementation details.

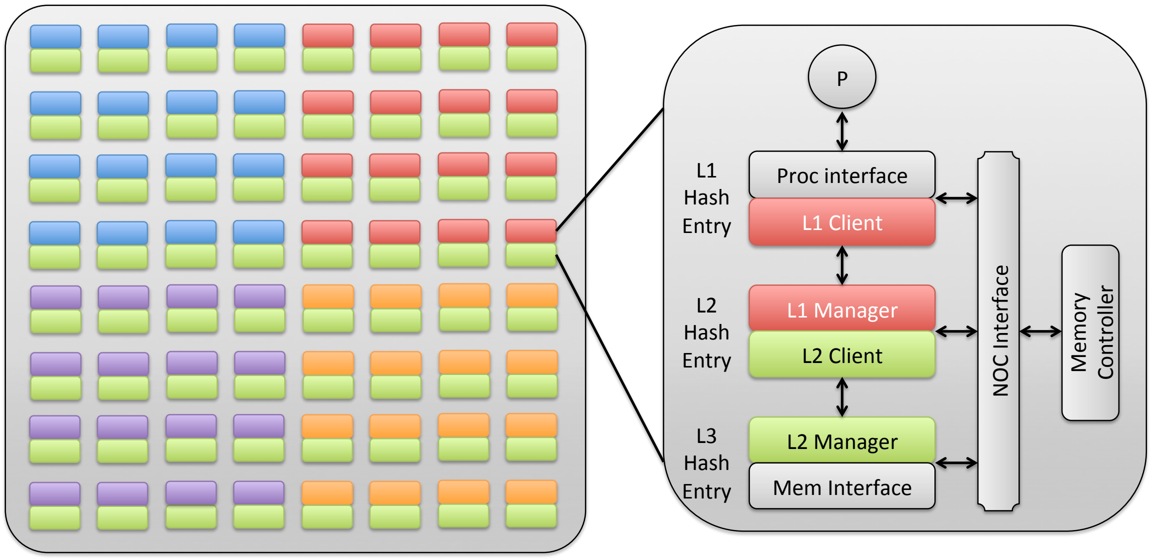
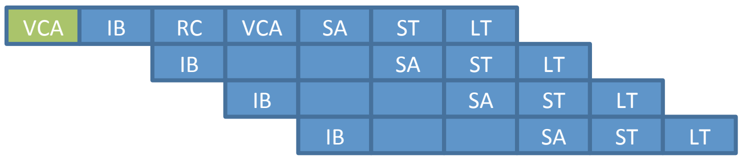


Figure 8 - An 8x8 network of nodes, with 4 independent neighborhood L1/L2 protocols unified by a chip-wide L2/L3 protocol. Coloring shows protocol membership

4.1.5 Interconnection Network

The interconnection network of CaffeineSim is composed of a collection of virtually flow controlled routers [73], each implementing a 6-stage router pipeline matching that described in Section 2.1 of [74], shown in Figure 9. These routers can be composed into the four configurations shown in Figure 10: a simple mesh network, a simple torus, and two flavors of express-channels [75]. These last two options were specifically implemented to enable CaffeineSim to explore network/coherence co-design, where express-channel distance can be configured to match that of the coherence neighborhood sizing. The dimensions of these topologies, as well as the parameters for routers properties, such as the number of virtual channels, the number of buffers per router, flit-width, etc., are completely variable and managed via the interconnect network settings.



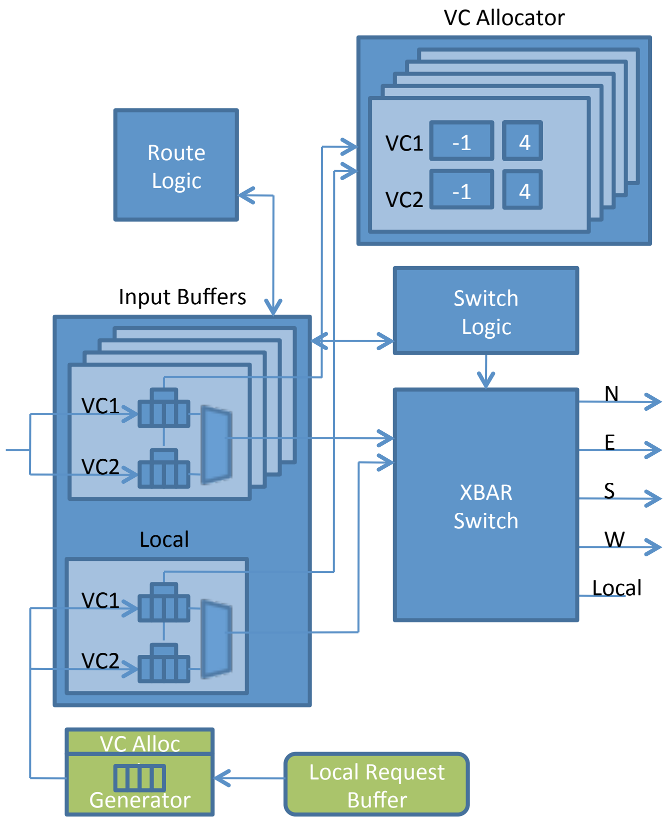


Figure 9 - Virtually flow controlled 6-stage router and associated pipeline stages: input buffer (IB), route calculate (RC), virtual-channel allocate (VCA), switch allocate (SA), switch traversal (ST), and link traversal (LT).

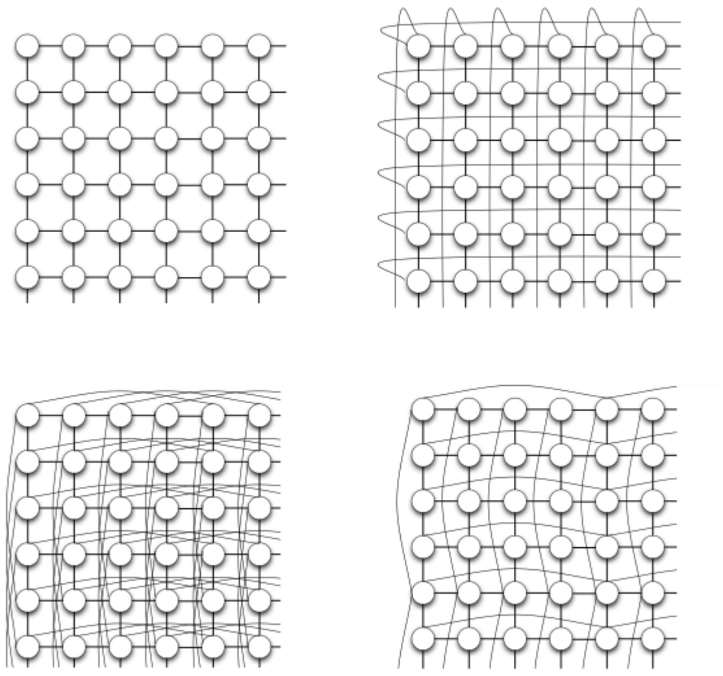


Figure 10 - CaffeineSim Supported topologies: Mesh, Torus, Express-Channel Full (n = 4), and Express-Channel Light (n = 4).

Despite implementing a detailed pipeline model, virtual channel allocation/contention, flit buffering and backpressure, and credit based flow control within the router nodes, this did not extend beyond the interconnect network. As a result, the lack of flow control between terminal modules and the router introduces a modeling deficiency; modules that run out of internal buffer space cannot cause backpressure into the network. This is because all CaffeineSim modules are designed such that they effectively have an unlimited sized temporary input buffer that holds requests delivered by the network until they are processed. This means traffic bursts to hot-spots in the cache hierarchy, which were observed during some simulations due to highly contended synchronization variables, were not interfering with other network functionality as dramatically they should have. Ultimately this phenomenon should have caused a near total collapse of the network’s bandwidth availability until all requests to the offending address’s home-node were properly drained from the system, rather than unlimitedly buffering into the home-node’s terminal, thereby freeing up network resources.

4.1.6 Memory Controller

CaffeineSim provides two memory controller models. The first is a static fixed access latency model, where all requests return from memory after a fixed delay. The second is a higher fidelity model, where timing for rank and bank access conflicts are taken into consideration in computing the access latency, and row buffer modeling is included to model the benefit of open-page row buffer hits. This more detailed memory model was designed to approximate the timing computations of DRAMSim [76]. Neither model provides support for memory controller event scheduling. All requests are processed in the order they are received, and greedily processed as early as possible.

4.2 Manifold Architectural Simulator

Over the course of this research effort, the Tinker group contributed and transitioned to a new environment, carrying some of our simulator design experience with us. Manifold is ‘is an open source software project whose goal is to provide a scalable infrastructure for modeling and simulation of manycore architectures” [77]. Manifold is a joint collaboration currently under development at the Georgia Institute of Technology, being designed with an emphasis on modularity and composability. This enables composition of disparate parts from different research groups into a cohesive whole in order to enable mix-and-matching of models while minimizing development time. Not only will this help grow the library of manifold models, but over time as this library increases, parts from many domain experts can be used when an architectural component is suspected to heavily influence accuracy. When approximation is acceptable to improve simulation performance, simple low fidelity models are also available, as well as several models between these extremes. Many of the details regarding how this is accomplished, from a software engineering perspective, can be found in [78] by Wang, et al. and in the documentation on the website manifold.gatech.edu.

Since Manifold already has a small library of components for each architectural component (several of which are ported versions of the CaffeineSim modules described in Section 4.1), this section will not enumerate all the component options available, but rather only those employed. The portion of this dissertation that leverages Manifold for evaluation (Chapter 7) uses the hand-tuned trace methodology discussed briefly in Section 4.1.3. As a result, no front-end emulation or processors were required, and simple static-delay memory controllers are sufficient since the traces used are intended to model active working sets that primarily stay on-chip after the first cold miss.

These traces are fed to an MCP-algorithm cache hierarchy similar to that described in Section 4.1.4, but organized differently than in CaffeineSim. In an effort to better emulate how future manycore architectures will likely be designed, a clustered approach was taken, compared to the more flat organization of CaffeineSim's 1-to-1 core-to-network terminal mapping. Figure 11 shows the 64-core model, comprised of 16 Intel Nehalem-like [79] quad-cores, where cores within a cluster are locally connected via the cache hierarchy. As shown, L1's and L2's are private to each core and intra-cluster coherence/sharing begins at the L2/L3 boundary. Inter-cluster coherence is maintained via the L3/L4 layer, communicated through the on-chip network.

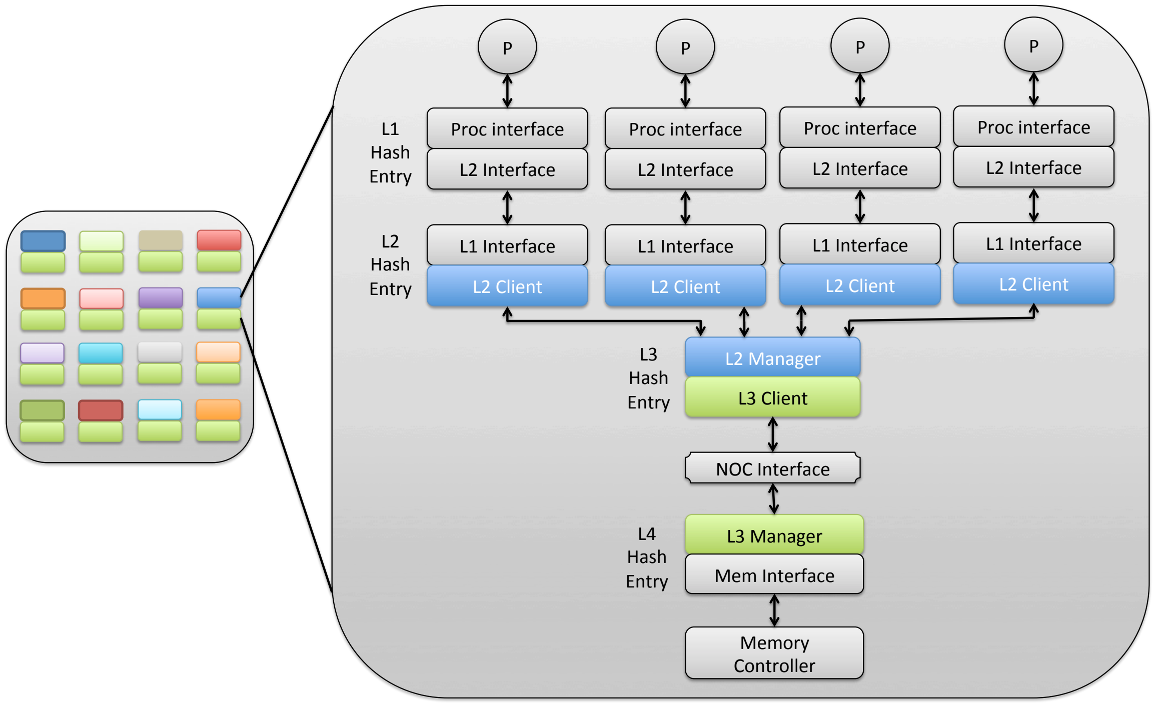


Figure 11 - A 4x4 Torus of Nehalem-like clusters, with 16 independent L2/L3 protocols united through the chip-wide L3/L4 protocol. Coloring shows protocol membership.

The interconnection network employed is a high-fidelity Torus interconnect network, using the Iris network component model found in the manifold library. Unlike CaffeineSim, this interconnect model does provide functionality for flow control out of the network into the terminals, and thus can experience proper back pressure from under provisioned terminal buffers.

4.3 MurPhi State Enumeration Formal Verifier

As mentioned previously, Murφ (or MurPhi) [22] is an enumerative model checker and language (inspired by the Unity modeling language [80]), used in academia and industry for the verification of coherence protocols. This is accomplished by providing the model checker with initial conditions, invariant rules, and a collection of {guard, action} pairs representing the coherence state machines, which are executed inside what can be viewed as an infinite loop. This loop, beginning with the initial state, will attempt to fire all allowable actions based on the satisfied guard conditions. Each guard-action firing changes the global state of the system through modification of either the client protocols' states, the manager protocol's state, the contents of buffers, adding network in-flight messages, or combinations of these. The resulting composite global state is stored in a 'pending-state' queue that represents the collection of all global states that have not had their invariants checked yet. These states also have not been used to seed another generation of next-states yet either.

After an exhaustive firing of all guard-action pairs from the initial state, and a check of the invariant rules for the initial state, the initial state is moved to a 'processed-state' data structure, and the next state is popped off the 'pending-state' queue where the sequence of attempting to fire all guard-action pairs, and checking invariants, is repeated once again. The only difference from this point forward is that when a state is generated from a guard-action pair firing, before being added to the 'pending-state' queue both the 'processed-state' structure and 'pending-state' queue are searched for a match to avoid redundant processing. This process continues iteratively in the main loop either a breadth-first or depth-first search manner, searching the entire global state space for any state that violates the invariants. If every state in the entire state space successfully passes, the protocol has been formally verified.

From this description it should be clear why state space enumeration becomes prohibitively expensive as the state space increases. As execution continues, two different aspects of large state spaces can impede the performance of the model checker as more states are generated and explored. First, the associative search of the 'pending-state' and 'processed-state' structures becomes more expensive with each additionally explored state. Second, the data footprint of the structures grows with each additional state, first having an impact in cache performance during execution, and eventually having an huge impact due to paging when the footprint exceeds the size of main memory.

This can pose a serious problem when the number of states requiring exploration are on the order of tens to hundreds of millions of states for even small scale protocols of only 5 and 6 nodes (Table 2 in Chapter 6). For example, the 5 node MOSI broadcast protocol implementation used in Chapter 6's evaluation required the exploration of 132 million states, at a size of 365 bits per state (~44 GB), explored over the course of 740 million rule firings in a 51 level breadth-first-search, executed over the course of 3½ days of execution. The next increment of verifying 6 nodes reached over 500 million states, at 465 bits per state (~212 GB) before a system failure terminated the execution after 45 days. This is where techniques like state space symmetry reduction can be employed to improve execution, but even at projections of as much as a 90% reduction in the state space [24] this is prohibitively expensive in large scale systems. For an interesting retrospective on murphy and the many optimization that have been applied over the last 20 years, we point the reader to [81].

Finally, while it may not be immediately obvious from this discussion of MurPhi's operation, there is a fundamental limitation to MurPhi with respect to protocol verification. While murphy can do correctness and deadlock checks through the exhaustive enumerative search, there is no way to resolve whether there is potential for livelock, due to the nature of state matching/discarding during the state space graph's traversal. In short, cycles cannot be detected/evaluated effectively and therefore livelock analysis is not possible.

Dill mentions that this is an unfortunate side effect of Murphi's development history, but when given the choice, correctness and deadlock avoidance always held priority over livelock detection from a practical perspective regarding what most often causes verification violation. While there is nothing to be done for it, it is worth keeping in mind the following quote regarding even the correctness guarantees that MurPhi does provide: "I have had several students in class projects develop protocols that they verify successfully in MurPhi only because the protocols livelock before they can reach an error state" [81].

CHAPTER 5 - DESIGN OF THE MANAGER-CLIENT-PAIRING FRAMEWORK

As technology continues to scale, the need for more sophisticated coherence management is becoming a necessity. The likely solution to this problem is the use of coherence hierarchies, analogous to how cache hierarchies have helped address the memory-wall problem in the past. Previous work in the construction of large-scale coherence protocols, however, demonstrates the complexity inherent to this design space [25-27, 30, 46].

The difficulty with hierarchical coherence protocol design is that the complexity increases exponentially with the increase in coherence states, due in turn to interactions between hierarchy tiers. Additionally, because of the large development investment, choices regarding coherence hierarchy are often made statically, with little knowledge of how changes to the organization or protocols involved would affect the system. This results in sub-optimal designs that leave performance on the table due largely to an inability to evaluate alternatives.

This chapter presents Manager-Client Pairing (MCP) as a unifying methodology for designing multi-tier coherence protocols by formally defining and limiting the interactions between levels within a coherence hierarchy to enable composition. This ability to apply composition to coherence hierarchy design enables rapid construction and evaluation of several complex design options while simultaneously restricting the design complexity to a small set of protocol component parts. Using MCP, a variety of hierarchical coherence protocol configurations for a 256-core system, comprised of 4 64-

core manycores, are evaluated to provide insight into what impact different hierarchy depth and width choices can have on system performance.

5.1 The Problem

Over the past ten years, the architecture community has witnessed the end of single-threaded performance scaling and a subsequent shift in focus toward multicore and future manycore processors [1]. Within the realm of manycore, the two leading programming methodologies are message passing and shared memory. While arguments can be made for both sides, it is widely accepted that shared memory systems are easier to program, and that the programs thus developed are more portable. For these reasons, shared memory is more likely to be adopted in the future by programmers *provided that it scales*. The scaling issue is directly due to the burden of coherent data management, which, under shared memory, is shifted away from the programmer and onto the hardware designer.

Data coherence management is a non-trivial concern for hardware design as we move into the manycore era. Efficient coherence protocol design and validation is already a complex task [33, 64, 82-85]. To make matters worse, we hypothesize that just as broadcast-based systems have an upper limit, monolithic directory-based systems too will reach scalability limits, thus requiring *coherence hierarchies* to overcome this performance bottleneck. This notion is reinforced by the prevalence of coherence hierarchies in many existing and prior large-scale, high-performance architectures [25-30, 46]. However, before future advances can be made in hierarchical coherence protocol design, a flexible framework that provides coherence composition is needed that supports variable hierarchy width and depth, as well as insights into how coherence and hierarchy decisions affect system performance.

Compared to their monolithic counterparts, hierarchies are considerably harder to reason about, making composition from “known working parts” attractive. The current general solution is to design an ad-hoc glue layer to tie low-level coherence protocols together. However, this often results in changes to the low-level protocols, specifically the introduction of complex sub-state replication to encode all hierarchy information into protocol state for permission and request handling [41, 56]. This in turn requires management of more states, and thus a more complex state machine. The ad-hoc glue methodology also makes evaluating changes to the hierarchy more difficult, since a new ad-hoc solution must be developed for each change.

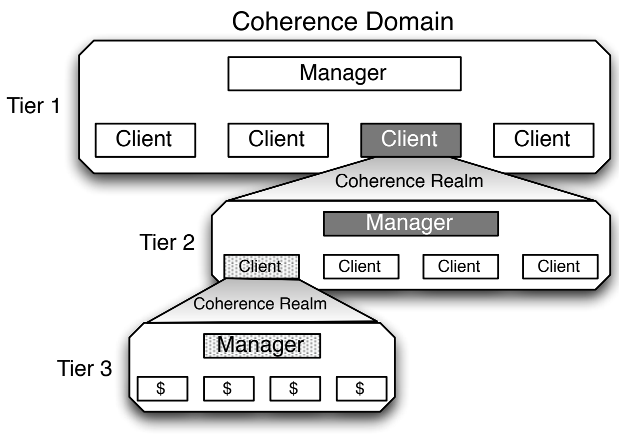


Figure 12 - Coherence hierarchy labeled with MCP terminology

The combined result of current practices is an abundance of large, complex, inflexible and highly specialized coherence protocols, especially where hierarchies are employed. In this dissertation we develop a powerful new way to design coherence hierarchies, *Manager-Client Pairing* (MCP). MCP defines a clear communication interface between users of data (*clients*) and the mechanisms that monitor coherence of these users (*managers*) on the two sides of a coherence protocol interface. This client-manager model is the basis of the interface used by MCP for coherence hierarchy tiers. Application of MCP provides *encapsulation* within each tier of the hierarchical protocol so *each component coherence protocol can be considered in isolation*, mitigating the problem of state space explosion [23] due to ad-hoc solutions, and reducing the design complexity normally inherent to coherence hierarchies. Furthermore, because the interface is *standardized* and tiers are *independent*, MCP can be applied in a divide-and-conquer manner (Figure 12) to partition a manycore processor into arbitrarily deep hierarchies. This enables rapid design of hierarchical coherence protocols using community-validated building blocks that can be readily compared and evaluated.

In short, MCP is a framework for modular and composable hierarchical coherence protocol development. The contributions of MCP as presented in this chapter are as follows:

- It defines and delineates coherence responsibilities between *Client* and *Manager* agents to distill the fundamental requirements of a coherence protocol into a modular and generic set of base functions;
- The definition of a generic protocol Manager-Client interface based on these base functions standardizes coherence protocol communication; this in turn enables rapid development of multi-tier coherence hierarchies by converting stand-alone coherence protocols into coherence-tier building blocks;
- The MCP Hierarchy Permission Checking Algorithm and associated terminology to formalize hierarchical protocol description are also presented; and,

- A qualitative proof of concept, evaluating the impact different hierarchy width and depth choices have on performance for a 256-core system (four connected, 64-core manycores), is shown; this level of analysis is only possible because of the flexibility and rapid design afforded by the MCP framework.

The remainder of this chapter is structured as follows. In Section 5.2, the base functions of cache coherence are presented based on a definition of client and manager responsibilities. Section 5.3 describes base function use in the MCP hierarchical permissions algorithm through examples. Section 5.4 then demonstrates the power and flexibility of MCP by providing new insights through experiments that evaluate the impact of different hierarchy design choices. The chapter then ends with concluding remarks regarding MCP.

5.2 Division of Labor for Cache Coherence as a Template

In a shared memory machine, the cache coherence protocol is responsible for enforcing a consistent view of memory across all caches of all nodes within a coherence domain. This includes defining the mechanisms that control acquisition and holding of read permissions, write permissions, the respective restrictions on each, and how updates to data are propagated through the system. The responsibilities of this effort can be divided between two kinds of agents: *managers* that manage permission propagation and *clients* that hold these permissions.

We begin by formally defining these roles and interactions for a flat, non-hierarchical MOESI directory protocol [37]. These roles are then re-examined to derive interfaces that enable composition of complex coherence hierarchies. Finally, we demonstrate that these interfaces are compatible with broadcast protocols as well, despite having been derived from a directory protocol design.

5.2.1 Defining Base Functions of Coherence via MOESI Example

As a starting point, an EI protocol is perhaps as simple as a protocol can be. Each client can either have the only copy of data in the exclusive (E) state, or not have the data at all (the invalid (I) state). Tracking of this state from the manager's perspective is straightforward since there can only be one exclusive client at any time. Propagation is straight forward as well: if another client requests permission to a block, the manager can take permission away from one client and give it to the new client.

As simple as this example may be, it immediately highlights the most basic responsibilities of the agents involved in coherence. Clients need to be capable of answering whether or not they *have sufficient permission* to satisfy a request. When unable to satisfy a request, clients need a mechanism to *request permission*. Manager agents are responsible for *permission allocation* and *de-allocation*. This includes the capabilities for *accepting* permission requests, *tracking* sufficient client state to satisfy such requests, and mechanisms for *modifying* permissions to carry out these actions.[2]

Let us now consider the question of how a write is handled in an EI protocol. State management for data modification becomes a matter of whom most readily uses this information, or rather, which agent should maintain knowledge of the 'dirty' state associated with a write. Assuming a write-back cache, modified data is a matter of

[2] In a directory-based coherence schemes, the manager agent is synonymous with the directory. Manager agent is used in place of directory, however, to avoid strict association of state management with directory-based coherence protocols.

permissions when a later cache eviction is being made (i.e., can this block be evicted immediately or does some action have to be taken first). Since permission queries have already been defined as a client-side responsibility, the client agent would need an additional state, the Modified (M) state.

This new M state brings about two important revelations. First, the manager *does not have to be aware of an internal change from the E to M states within it's client* since, as long as the block is exclusive, the client can silently upgrade state to M. As a result, the manager and client states do not have to be perfectly congruent (i.e., client states include M, E and I, while manager states include only E and I). Second, this addition also demonstrates another requirement for client agents: the ability to *downgrade* or forfeit permissions. Before a cache can evict dirty data, it has to be written back to memory and the manager must be notified. This is subtly related to manager permission de-allocation, but with a significant difference: this is *client initiated* instead of *manager initiated*. As a result, manager agents also have the additional need of *permission downgrade* processing when a client wants to voluntarily relinquish permissions.

In order to fully expand the MEI protocol into an MOESI protocol, two additional states require consideration: the shared (S) and the owned (O) states. The *shared* state enables multiple clients to have read permissions simultaneously. This complicates permission handling since these sharers need to be invalidated before write permission can be granted by the manager. In an invalidation-based protocol, downgrade messages are sent to sharers before write permissions can be granted. A common implementation optimization is to have sharers directly send invalidation acknowledgements to the originating requestor rather than back to the directory, advocating the need for client-to-client communication via forwarding. This forwarding is also necessary to take advantage of the *owned* state, which introduces the ability to transfer data between caches by giving a client special status as a data supplier (often dirty data w.r.t main memory). On a new read request, the manager will forward the request to the owner instead of memory, who will then respond to the client with data.

Table 1 summarizes and enumerates a comprehensive list of the base functions required for communication between processors, clients, managers and memory in a flat protocol. These will be used as an aid in the developing a generic protocol interface.

Table 1 - Base functions for standardized communication between processors, clients and managers. The following is an example of how a read sequence would operate on an invalid block. First a Processor issues a (1) ReadP to its client. This client replies with ‘false’, where upon the Processor takes another action, (2) GetReadD. This results in the client executing its GetReadD action, which in turn will cause the Manager to execute its GetReadD action. The Manager GetReadD action is a forward request. Assuming there is no client owner, Memory is regarded as the owner of the data and asked to execute its GrantReadD action. This results in Memory supplying Data to the client, completing the GetReadD. Upon completion, the processor can retry its ReadP action, which the client will respond with ‘true’. The processor can safely execute its DoRead action for which the client will supply data.

Origin Agent	Action Type	Action	Description	Destination Agent(s)	Response Action(s)	
Processor	Permission Query	ReadP	Have read permission?	Client	Reply with True/False	(1), (6)
		WriteP	Have write permission?	Client	Reply with True/False	
		EvictP	Have eviction permission?	Client	Reply with True/False	
	Permission and/or Data Acquire	GetReadD	Get read permission and Data	Client	GetReadD	(2)
		GetWriteD	Get write permission and Data	Client	GetWriteD	
		GetWrite	Get write permission	Client	GetWrite	
		GetEvict	Get eviction permission	Client	GetEvict	
		DoRead	Supply Data to Processor	Client	DoRead	(7)
		DoWrite	Issue Dirty Data from Processor	Client	DoWrite	
Client	Data Supply/Consume	DoRead	Supply Data to Processor	Processor	Complete DoRead	
		DoWrite	Issue Dirty Data from Processor	Processor	Complete DoWrite	
	Permission and/or Data Acquire	GetReadD	Get read permission with Data	Manager	GetReadD	(3)
		GetWriteD	Get write permission with Data	Manager	GetWriteD	
		GetWrite	Get write permission	Manager	GetWrite	
		GetEvict	Get eviction permission	Manager	GetEvict	
	Permission and/or Data Supply	GrantReadD	Forward Data, Downgrade Self	Client	Complete GetReadD	
		GrantWriteD	Forward Data; Invalidate Self	Client	Collect all Acks to Complete GetWriteD	
		GrantWrite	Forward Ack; Invalidate Self	Client	Collect all Acks to complete GetWrite	
Manager	Permission and/or Data Acquire	GetReadD	Grant read permission with Data	Memory/Owner	GrantReadD	(4)
		GetWriteD	Grant write permission with Data	Memory/Owner; Sharers	GrantWriteD; GrantWrite	
		GetWrite	Grant write permission	Owner; Sharers	GrantWrite; GrantWrite	
		GetEvict	Grant eviction permission	Memory; Client	Consume Dirty Data; Complete GetEvict	
Memory	Data Supply/Consume	GrantReadD	Forward Data	Client	Complete GetReadD	(5)
		GrantWriteD	Forward Data	Client	Collect all Acks to Complete GetWriteD	

5.2.2 MCP Interface for Coherence Hierarchy Construction

We now turn our attention to coherence hierarchies. Reviewing the base functions outlined in Table 1, it is evident that there are considerable similarities between the agents involved in coherence. Specifically, the relationship between processor and client agents has similarities to that between manager agents and memory: in both cases, data suppliers are asked to supply data from a mechanism closer to main memory in the memory hierarchy. This is a critically important insight into how to develop an interface that allows for recursion and thus hierarchies. Examining Figure 13, if manager agents were given the ability to issue *permissions-query upwards* like processors do towards their client, then replacing 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.

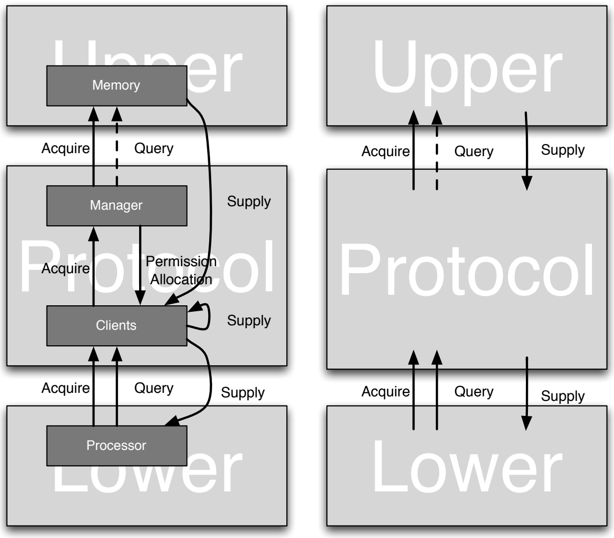


Figure 13 - Addition of permissions-query capabilities enabling recursive coherence.

From this we can see that there are at least three necessary components to the MCP interface: *upward permission querying*, *lower-to-upper permission/data* *acquisition*, and *upper-to-lower data supply*. Introducing permission querying capabilities to manager agents is the origin of Manager-Client Pairing's namesake. Manager queries are accomplished by pairing the manager agent of each coherence realm in a tier with a client in the next higher-up tier in the hierarchy (or an all-permission client if there is no higher tier, e.g., memory). Since there is one logical manager agent per coherence realm, this allows the client to represent the permissions of the entire realm and all tiers beneath this realm. This is explained in more detail when describing the permissions algorithm in Section 5.3.

Permission/Data acquisition and supply are also possible due to the pairing of managers with clients in the next tier. The manager requires no details regarding the operation of the higher coherence protocol provided; it can defer that responsibility to it's paired client. By systematically asking the paired client for either read or write permission, the client can take part in its native coherence scheme until it has completed the request. This is much like how a processor is unaware of how coherence in the caches are implemented; it simply asks if it has permissions and receives data, as shown in the example described with Table 1.

One important detail that must be accounted for is the propagation of a *paired-client downgrade*. Since a client agent effectively represents the coherence state of all its paired manager's lower tiers, the paired client cannot give up its permission rights and transition into another lower permission state until the entire coherence realm below it has been made to match these new permissions. This is addressed in MCP by allowing the paired-client to issue downgrade requests to its paired manager. When the manager executes this downgrade action, it executes its 'Permission and/or Data acquire' action from Table 1, where the forward destination is its paired client. Thus the local sharers will send their invalidation, downgrade acknowledgements and/or data to the manager's paired client. This provides us with our fourth and final interface component: upper-to-lower downgrades. Figure 14shows how the MCP interface enables coherence tier communication while respecting the encapsulation of the component protocols.

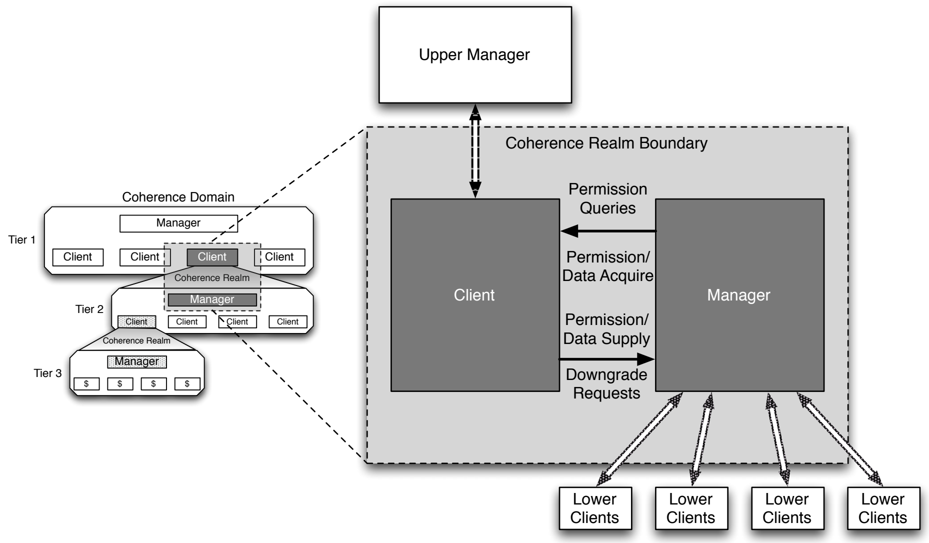


Figure 14 - Manager-Client Pairing and associated interfaces to preserve encapsulation

5.2.3 Broadcast Compatibility

The argument can be made that the base functions, and thus the MCP interface, may be insufficient to encompass broadcast coherence schemes since it was developed specifically for a directory-based implementation. In this subsection we demonstrate that popular broadcast coherence protocols, such as Snoopy-MOSI [37] and TokenB [40], are MCP compliant. Additionally we point out the restrictions of these protocols that need to be accounted for by any architecture employing these protocols as building blocks in an MCP coherence hierarchy.

5.2.3.1 Snoopy Coherence

In a snoopy protocol, all agents are connected together via a shared medium (i.e. a bus) and residents on the shared medium observe coherence traffic through snooping agents. This need for a shared medium represents a limitation specific to broadcast protocols; either broadcast or multicast functionality is required in the network to ensure correctness. However, a benefit of this is that there is no single manager agent responsible for permission allocation and deallocation, as opposed to in a directory scheme; rather, this is a distributed responsibility. In this sense the *manager mechanism is spread across all the snooping mechanisms*; the functionality of GetReadD, GetWrite and GetWriteD, downgrades and invalidation are preserved by the bus-initiated state-machine of the snoopers. For example, a BusReadMiss placed on the shared medium as the result of a client GetReadD action causes the snooping mechanism of the cache with the block in *modified* state to execute a client GrantReadD, providing data on the bus and causing a self-downgrade transition into the *owned* State.

The only manager responsibility of MCP not immediately obvious in broadcast-based protocols is GetEvict, used during writebacks of dirty data. However, in a non-hierarchical broadcast protocol, the shared memory controller plays a special role when data needs to either enter or exit the shared environment. In this sense the memory controller acts as a *gateway* beyond the boundaries of the broadcast protocol's coherence realm, much how the MCP interface is the gateway for a coherence realm. On a GetReadD from a client where no other caches can respond (e.g., because data is not present locally), it is the memory controller's responsibility to acquire data from outside the coherence realm and respond as if it owned the block by in-turn executing an upward GetReadD. Because of this extra responsibility, it is straightforward to assign the memory controller's snooping agent with the responsibility for issuance of a Manager GetEvict on a Client GetEvict request (i.e., by pushing dirty data back out to memory). In essence, it is as if the memory controller client agent represents the coherence state of everything outside the coherence realm, including memory (where memory initially owns all data). While compliant with MCP, this does introduce another limitation: not only does the architecture need broadcast/multicast functionality, but also at least one enhanced snooper for handling these requests. The architects of the *HP Superdome* leverage a similar notion, where a larger, hierarchical broadcast system was constructed using commodity broadcast coherent components tied together by a shared medium for intelligent broadcast distribution. In the HP Superdome, specialized logic at the boundary between the local busses and an intra-cell crossbar behaves like the memory controller described above, converting bus broadcasts that miss locally into system messages that request the data from the rest of the system [26].

5.2.3.2 TokenB Coherence

Since TokenB coherence is based on MOSI broadcast coherence, there are only two additional concerns that need addressing for MCP compliance: *token handling* and *persistent requests*. Token handling is a relatively trivial concern since token message support as well as token accumulation/distribution logic is no more complex than the message extensions and state machine logic required by other component coherence protocols. The largest hurdle for TokenB is the correctness substrate's need for persistent requests. This can be accomplished by adding an extension to the protocol actions to incorporate a Boolean signifying whether the request is persistent or not. In TokenB, persistent requests are activated by the memory controller, which is congruent with the previous notion of the memory controller being a special client agent (i.e., responsible for the extra, non-distributed manager responsibilities). At this point it becomes the responsibility of the underlying implementation to handle persistent requests commands as a special version of the same actions presented in Table 1. Furthermore, because the protocols are encapsulated, this concern does not extend beyond the scope of the coherence realm. Token management and persistent requests are restricted to only the relevant coherence realm.

5.3 Permission Hierarchy Algorithm

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. In order to enforce the permission-inclusion property described by Ladan-Mozes and Leiserson [63], the client agent must behave as a gateway for the manager of the coherence realm, restricting what permissions can be awarded, and taking action when permissions must be upgraded in the coherence realm before the manager can begin request resolution. The manager agents now must consult the gateway client before allocating permission, which in turn may recursively send another permission request to another manager-client pair. The flowchart in Figure 4 demonstrates the Manager-Client Pairing algorithm for processing permission acquires.

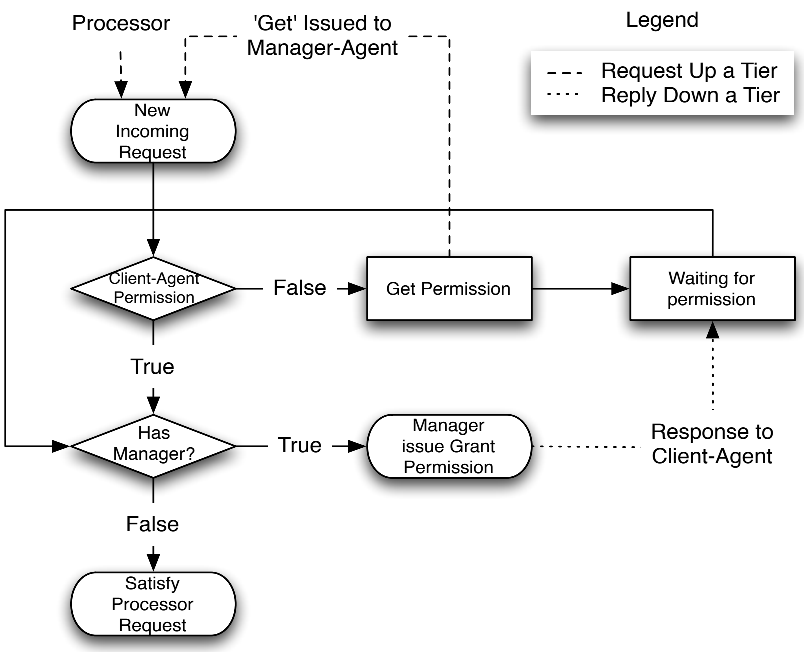
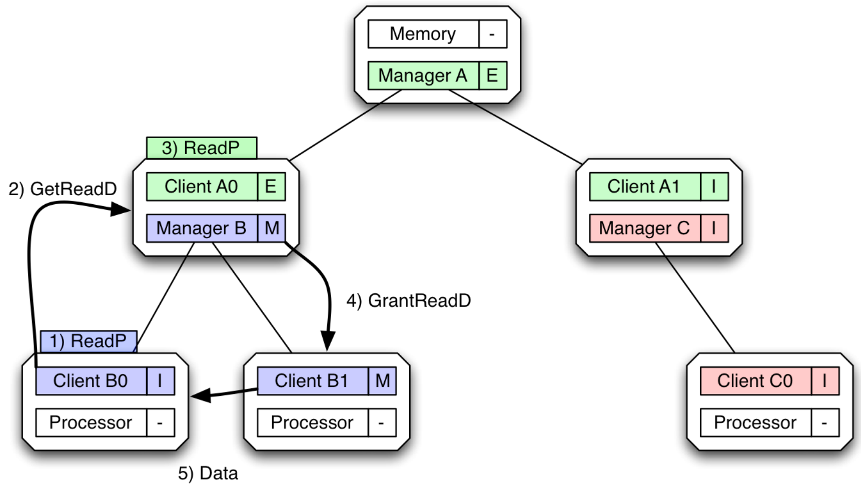


Figure 15 - Coherence hierarchy permission checking algorithm.

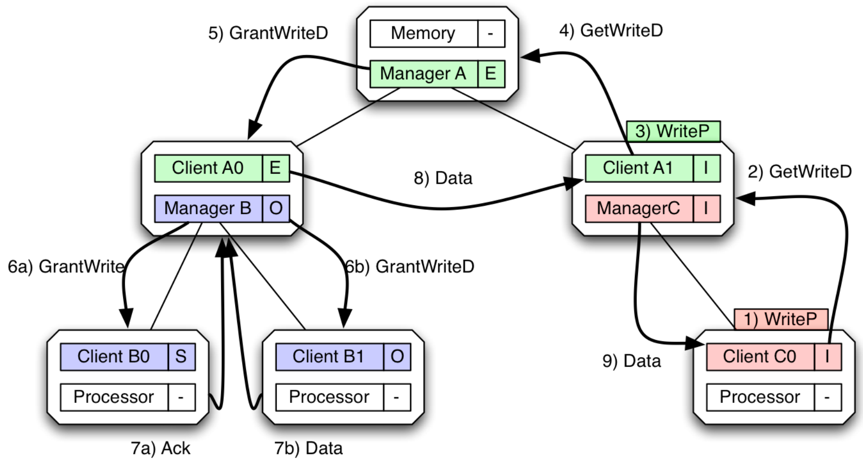
To aid in understanding and to highlight some important details of MCP, two examples are presented. In both examples we have a top-tier coherence realm, *A*, that implements a low-overhead MEI protocol to manage two lower coherence realms, *B* and *C*, both implementing MOESI. Manager *A* resides at memory and therefore has no need for a gateway client— being the highest manager agent in the system it always has permission to satisfy queries. Similarly, clients *B0, B1* and *C1* 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 16(a), an example of a realm-hit from a read request is shown. The processor below client B0 initiates the sequence with a read request, resulting in a ReadP permissions query. Since the *I* state has insufficient permission to satisfy the read request, ReadP yields *false*, causing the request to propagate up to the manager agent via a GetReadD (Figure 4's *'Get issued to Manager-Agent'* arc). At that level, the gateway client state is checked in the next-higher tier where in turn a ReadP yields *true* due to Client *A0* being in the *M* state (e.g., it has sufficient permissions for a read). Client *A0* is a gateway to a manager agent, thus manager *B* receives the request and responds with its native GrantReadD action. For the MOSI protocol implemented, this involves issuing a FwdRead to the current modified owner, client *B1*. Upon receipt, *B1* will downgrade to the *O* state and execute a GrantReadD, providing data and permissions to the originating client *B0*. Now *B0* can supply data to the core.

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 permissions 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 issue a later write request, the coherence realm still has enough permissions to allow a silent upgrade back into the *M* state without having to forward the query up to manager *A*, much like an *E*-to-*M* transition in MESI.



(a)



(b)

Figure 16 - (a) Realm-hit Read example and (b) Realm-Miss Write example

Requests can however cross coherence realm boundaries, referred to as a *realm-miss*, when more permission is needed than is available as shown in Figure 16(b). 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 has sufficient permissions to make forward progress; there is no gateway client at the top level. Upon receipt at manager *A*, a GrantWriteD request is issued to client *A0*. Just as in a flat protocol, where a cache would invalidate the block locally before forwarding an invalidation acknowledgement and data, so too does client *A0* need to invalidate its manager *B* before forwarding. This results in invalidations being issued down to *B*'s clients, which can continue recursively down multiple tiers in a larger coherence hierarchy. Once manager-client pair *B* has collected all the acknowledgements and the invalidation is complete, the modified data that once resided in *B1* can be forwarded to client *A1*. Now that manager *C* has sufficient permissions and data, it can issue data to the originating requestor, completing the transaction with client *C0* in the *M* state.

Although more complex, this second example further serves to demonstrate the decoupling of the protocol coherence realms from one another. When a gateway client's permissions are not high enough, 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 *B* was being invalidated, 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. Extending this to enable full-scale validation is presented in Chapter 6.

5.4 Using MCP to Compare Hierarchy Decisions

There are both hardware and performance issues associated with coherence hierarchies that need to be considered when designing a coherence mechanism for a given architecture. These will be discussed, followed by experiments that provide insights into how these design tradeoffs influence execution.

5.4.1 Hardware Cost

From a hardware perspective, each coherence tier in the hierarchy has an associated structural cost, most specifically regarding the Manager Agents Tag Structures (MATS) for tracking owner state and sharers. There have however been several proposals in the literature to address MATS-related sizing concerns for directory protocols, as discussed in Chapter 3. While there is generally a set of MATS per coherence realm, employing hierarchies creates natural width reduction within the tracking structure, since each realm's manager is designed for the realm degree (number of clients in the realm), not the number of system-wide nodes. For example, in a 256 node, 2-tier hierarchy with realm degrees of 16-16 (one top tier (T1) protocol managing 16 2nd tier (T2) protocols, each managing 16 clients), utilizing a simple directory bit-vector, M/O bit and owner field the MATS entries have a hardware cost of 16 + 1 + 4 = 21 bits in addition to the tag. In comparison, in a flat protocol using full-bit vectors, the overhead would be 128 + 1 + 7 = 136 bits per entry assuming no height or width reduction techniques. Albeit, there can be several T2 entries per T1, so the system-wide cost can range dynamically depending on the degree of replication. The costs are in favor of hierarchies, however, if we assume less than 5 realms are sharing on average and that data-cache tag reuse is an option.

There is also a double-edged memory latency impact when hierarchies are employed. Since the local manager has to be consulted while traversing up the hierarchy, there is an additional indirection cost added by either compulsory misses or misses that only hit in the upper/remote tiers. However, successful realm hits result in better physical locality since the manager and data responder are both closer than the home location of a flat protocol. There is also a similar effect regarding local on-chip network bandwidth. These are considerations MCP allows for that should be acknowledged during hierarchy design yet have not previously been evaluated to our knowledge.

5.4.2 Evaluated Hierarchies

In order to evaluate the impact that coherence hierarchies design decisions have, we use MCP to implement a variety of hierarchical configurations on a 256-core system composed of four interconnected 64-core manycore, where each 64-core manycore uses an on-chip torus network. In all instances, the hierarchy is a composition of only MOESI protocols to reduce the scope of analysis by removing any biases heterogeneity in protocol choice may introduce. We feel this is important analysis, however, and will investigate it in future work.

There are several options regarding how to partition 256 cores into a coherence hierarchy. The two most obvious choices are to use a flat, single tier protocol or a simple 2-tier protocol where coherence realms are restricted to each chip and inter-chip coherence is maintained in the top-tier (these configuration will be referred to as a '1-Tier 256' and '2-Tier 64x4', respectively). There are, however, other viable partitioning choices without introducing an additional tier and its associated hardware tag structures.

Both a 2-Tier 16x16 and 2-Tier 4x64 organization exploit different trade-offs with regard to locality and indirection delay by varying the width of the hierarchy. Figure 17 demonstrates this difference in behavior for a compulsory miss between 2-Tier 64x4 and 2-Tier 16x16. It is worth noting that local home node selection is the equivalent position of the tier 1 home within its realm to provide a deterministic local-home look-up policy that can vary as realm sizes vary.

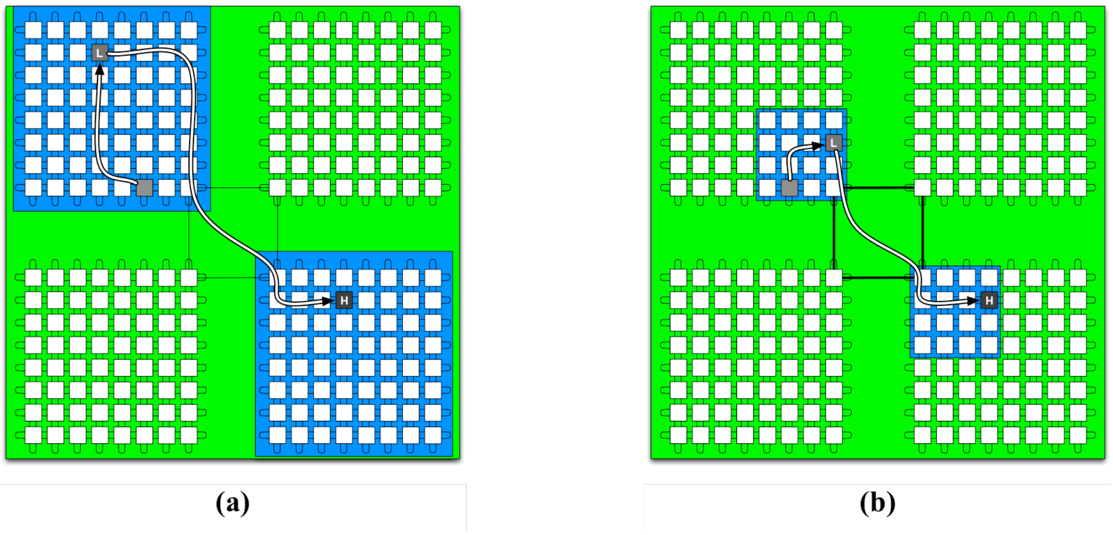


Figure 17 - Coherence realms (shaded) and local tier miss traffic in (a) 2-Tier 64x4 and (b) 2-Tier 16x16 system. In each instance the originating node must first access the local home (indicated by L) where it misses and traverses to the Tier 1 home (indicated by H). While 2-Tier 64x4 encompasses more nodes, increasing the likelihood of local realm hits, in the event of a miss (b) shows that 2-Tier 16x16 has the advantages of faster miss acquisition and lower network bandwidth consumption.

Finally, allowing for the required hardware tag structures, additional tiers can rapidly be added to the hierarchy through MCP's composability feature. To this end, both a 3-Tier 16x4x4 and a 4-Tier 4x4x4x4 configuration are implemented to demonstrate tradeoff evaluation at more extreme hierarchical design points is possible without the complexity of implementing ad-hoc glue layers.

5.4.3 Empirical Examples of Applied MCP

The execution-driven CaffeineSim simulator described in Chapter 4 is used in this work to model a manycore system with a detailed network infrastructure. The MIPS based emulator front-end from SESC [68] is used as the front-end to a simple execution model and detail memory hierarchy that supplies back-end timing information. Synchronization primitives (i.e., load link and store conditional) and fences are modeled as execute-at-execute to enforce consistency, while all other instructions are execute-at-fetch. The execute-at-execute model "peeks" at the state of the emulator without modifying state. This allows lock contention to be faithfully modeled based on simulated timing and not emulation. Each node contains a simple 2-issue in-order processor, a 32k private L1 cache and a 128k slice of the shared L2 cache. For the non-hierarchical run (1 Tier-256) the manager state reuses the L2 caches tags. For the hierarchical configurations, an additional distributed hardware tag structure is added per tier (MATS from Section 5.1), with entry volume equal to that of the L2 cache slice (2048 entries). Assuming approximately 64 bits per entry for tag, owner and sharer state (exact values vary with configuration), this would introduce an additional overhead of about 16KB worth of cache space per node per MATS. While we are aware that dedicating this additional overhead to increasing the cache size could improve performance, this is not taken into consideration for this evaluation since so many tag structure reduction techniques exist and inclusion makes reasoning about the collected hierarchy results more difficult (e.g. is a change in performance due to a change in the cache size, the hierarchy configuration, or a combination).

A small subset of the SPLASH-2 benchmark suite [86] is used to aid in demonstrating MCP's flexibility. To remove cold start effects and to ensure execution of parallel code, hooks were added to each benchmark to indicate the starting point for sampling and results are collected at barrier exits. For evaluation purposes, an unlimited version of the network topology is used in evaluation, where only delay due to the three-stage router pipeline is modeled; virtual channel allocation, switch allocation, and link traversal are contention free. This choice is made to remove network parameter decision bias from the presented results. Considering the large design space involved, evaluation of network and hierarchy co-design is left for future work.

5.4.3.1 Comparison of Hierarchy Width

In this sub-section, using MCP, comparisons are drawn between several two-tier hierarchies of varying widths to demonstrate the different behaviors benchmarks can exhibit as lower-tier scope changes. To begin discussion, we first present L1 miss latency comparisons in Figure 18.

These results demonstrate that hierarchy width selection is not a one-size fits all design choice; benchmark behavior diversity can influence what width is ideal. For both Ocean benchmarks, variation in the width has little impact on performance, and these variations are overshadowed by the performance difference when moving from a flat protocol to a two-tier hierarchy. Water Spatial, however, benefits most from a 16x16 hierarchy. To gain better insight into these differences, we can inspect the histograms of L1 miss latency behavior, shown below in Figure 19 and Figure 20.

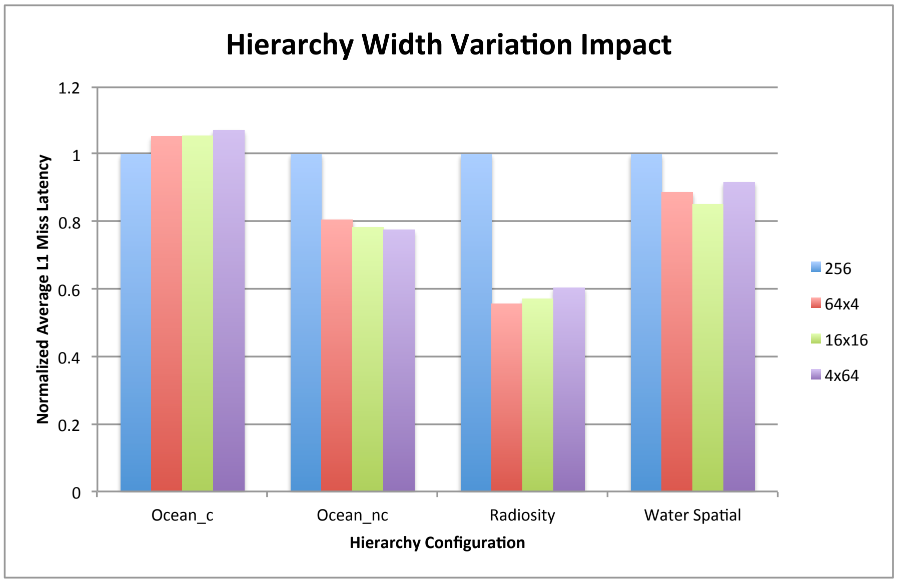


Figure 18 - Impact of hierarchy width on L1 miss latency

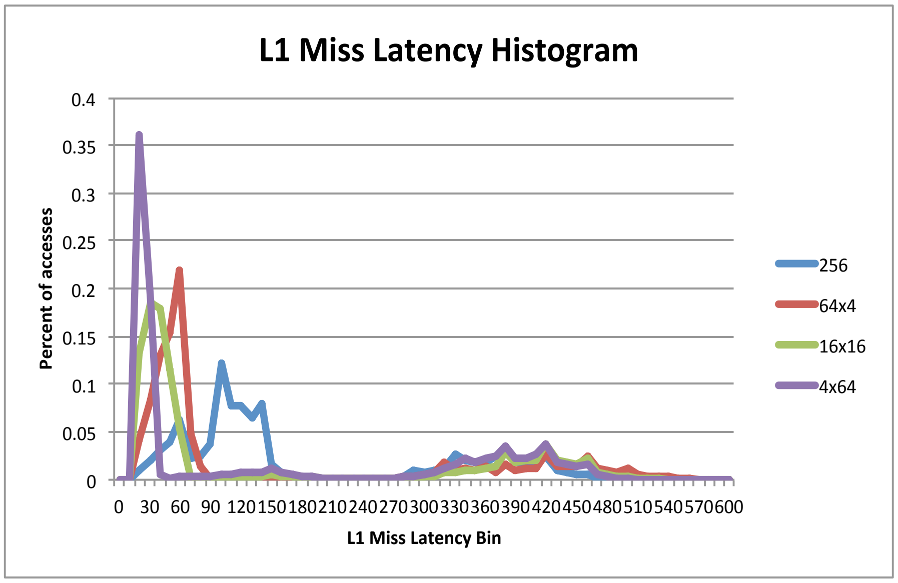


Figure 19 - Histogram of L1 Miss Latency for Water Spatial when varying hierarchy width

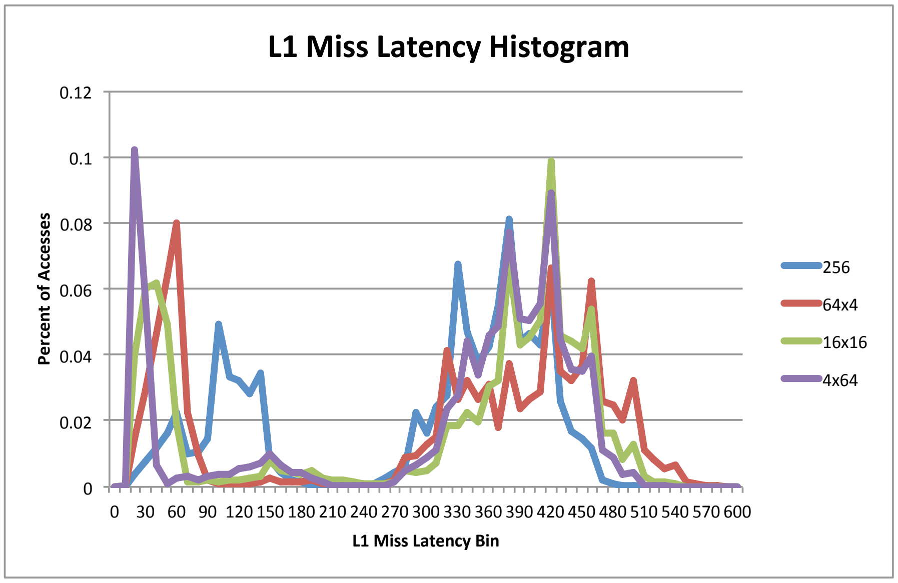


Figure 20 - Histogram of L1 Miss Latency for Ocean_c when varying hierarchy width

The first thing worth noting in both figures is the difference in the first set of humps (0-150 cycle latency). These represent accesses that miss in the L1, but succeed in getting data from the distributed L2 and thus do not suffer an off-chip memory access penalty. It is also clear in both histograms that 4x64 has the fastest response time of the configurations, which matches our intuition that success hits in the smaller realms will result in accelerated L2 hit times. The plot for 1-Tier 256, however, shows a wider, shallower response, demonstrating that access time varies based on home-node distance from the requestor. Nearby nodes satisfy some requests, while many require access to nodes that are on the other side of the chip or even reside in another chip's L2 cache.

The insights discussed in Figure 17 regarding variation in realm size are confirmed by these figures as well. In both Figure 19 and Figure 20 we see that, by examining the second hump (>300 cycle latency), 2-Tier 64x4 incurs the highest miss penalty due to the indirection of going to the local home prior to the global home; it's pattern is skewed to the right compared to the other configurations histograms. While 2-Tier 16x16 and 2-Tier 4x64 have to pay this indirection cost as well, the distance to the local home is shorter so less indirection penalty is incurred. For Water Spatial, however, 2-Tier 16x16 strikes the best balance between fast hit access latency, local hit rate (overall <150 cycle hit count is higher than 2-Tier 4x64's narrow spike), and low indirection penalty.

As for explaining Ocean_c, the histogram of Figure 20 gives us some additional important information; compared to Water spatial the ratio of off-chip accesses to L2 hits is much higher. This in turn emphasizes the negative effects of increased indirection as well as reducing the positive effect of local hits.

5.4.3.2 Comparison of Hierarchy Height

In addition to width design considerations, the choice to introduce additional tiers must be considered as well. This can be done quickly, however, using a composition of the MCP compliant MOESI protocol tiers as building blocks for these larger hierarchies. In the previous subsection it was demonstrated that performance in general favors two-tier hierarchies over a single tiered, non-hierarchical coherence protocol. Despite the cost of indirection, it is relatively low compared to the cost of global-home traversal (in figures 8 and 9 the left skew of 1-Tier 256, which has no indirection penalty, compared to the others at >300 cycles is noticeable but not dramatic). Further, this small performance penalty can easily be offset by the frequent, closer realm hits in the presence of high L2 cache hit rates. However, when increasing hierarchy height too much, aggregation of indirection penalty can become a concern as demonstrated in Figure 21.

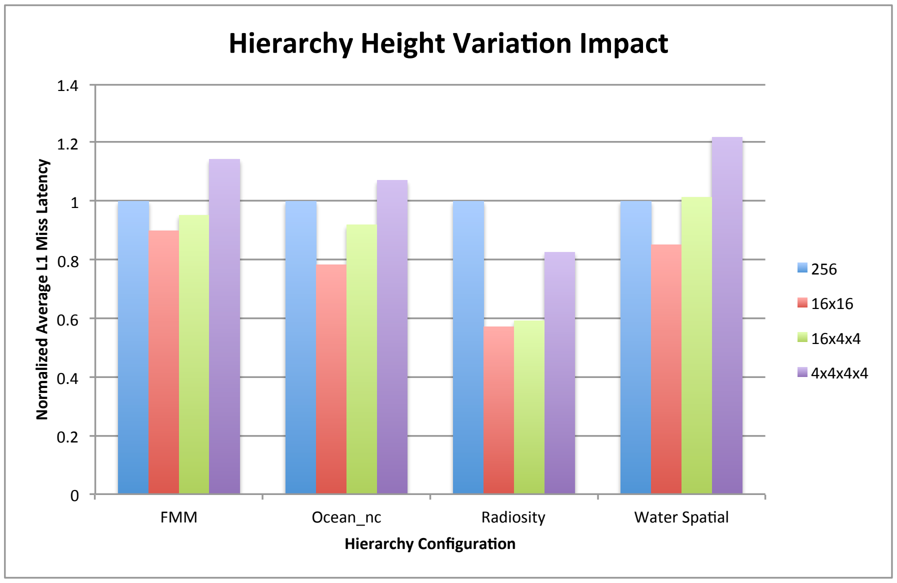


Figure 21 - Impact of hierarchy height on L1 miss latency

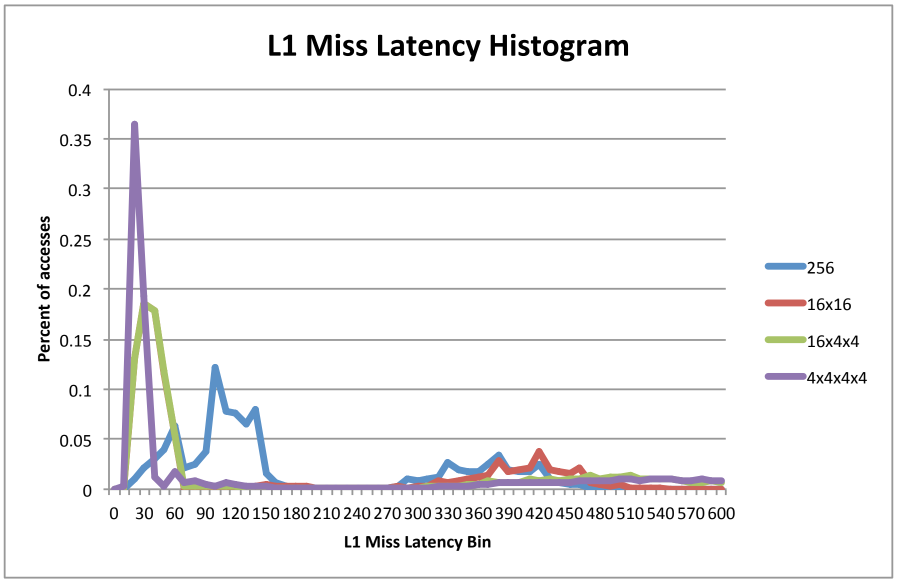


Figure 22 - Histogram of L1 Miss Latency for Water Spatial when varying hierarchy height

To verify this, we again examine a histogram of Water Spatial's L1 miss latency behavior (Figure 22). It is clear that the curve for the off-chip accesses flattens out and shifts to the right as the hierarchy height increases. This makes sense, however, since indirection penalty is not just from distance, but also includes router entry/exit at each tier's realm home node and MATS lookup/access time; each additional hop incurs a penalty that accumulates.

5.5 Conclusion

The primary goal of this qualitative study is to define the Manager-Client Pairing interface in order to create a generic hierarchical coherence implementation framework to support the continued scaling of massively coherent systems. This work demonstrates the impact coherence hierarchies can have on large-scale machines and shows how MCP's rapid design process enables effective reasoning about design decision trade-offs. Further, while hierarchies beyond two-tiers may seem superfluous now, MCP enables the design of arbitrarily deep, diverse coherence hierarchies for future, 1024 and greater core systems. By making different protocols adhere to this unifying interface, more intelligent design decisions regarding coherence solutions can be made. Additionally, the inarguable benefit of protocol modularity provided by MCP will enable architects to compare and communicate their designs decisions more effectively in the future.

CHAPTER 6 - VERIFICATION AND ENCAPSULATION SYMMETRY

As more heterogeneous architecture solutions continue to emerge, coherence solutions tailored for these architectures will become mandatory. Coherence hierarchies will likely continue to be prevalent in future large-scale shared memory architectures. However, past experience has shown that hierarchical coherence protocol design is a non-trivial problem, especially when considering the verification effort required to guarantee correctness.

While some strategies do exist for verification of homogenous coherence hierarchies, support for reasonable verification of heterogeneous coherence hierarchies is currently unavailable. Ideally, hierarchical coherence protocols composed of 'building block' protocols should be able to take advantage of incremental verification to side step the state-space explosion problem which hampers any large-scale verification effort. In this chapter, it is proven this can be accomplished through the use of the Manager-Client Pairing (MCP) framework, which provides encapsulation and permission checking support that enables a form of state-space symmetry. When combined with an inductive proof, this ensures the validation properties of proper permission distribution and livelock/deadlock freedom are enforced by any hierarchical composition of MCP compliant protocols. Demonstration of this methodology through the MurPhi formal verifier shows several orders of magnitude improvement in verification cost compared to full hierarchy verification.

6.1 Introduction

It is well established that power constraints have caused a major paradigm shift in computer architecture towards parallel processing for performance scaling. With it have come new opportunities and design spaces for architects to explore. Among these are the heterogeneous architectures discussed in Chapter 3, where on-chip network and processor diversity can be exploited for performance benefit or power/energy savings. Such systems benefit from the design of diverse interacting coherence protocols, where each protocol is optimized to take advantage of properties of a homogeneous region within the overall heterogeneous architecture. This comes at a cost however, in that the design and verification complexity of such systems is substantially higher than that of their homogenous coherence counterparts.

Despite this cost, the benefit of heterogeneous coherence has resulted in real-world applications of coherence heterogeneity. The Wildfire architecture, for example, was built using the existing first level protocol of the Sun E6500 in a larger hierarchy that enabled *Coherent Memory Replication* for improved node locality[27]. The Piranha architecture [25] had an *intra*-chip coherence management mechanism that was integrated with an independent *inter*-chip coherence protocol engine. This allowed for efficient use of on-chip caches and fast intra-chip data transfers while another DRAM directory-based protocol could be leveraged to enable scalability and performance at the inter-chip granularity. The HP Superdome [26] also employed a similar strategy as Wildfire, but with a different goal in mind. An inter-chip communication layer interfaced the native intra-chip protocol to a higher-level directory protocol. The resulting system was able to restrict message broadcast scope to the local protocol in many cases, enabling the use of commodity parts (i.e., those with “glueless” multiprocessor buses) in a large-scale system while maintaining performance. These examples suggest that heterogeneous coherence hierarchies will become more attractive in the present era as current technology trends continue.

Another factor motivating heterogeneous coherence support is the emergence of *Partitioned Global Address Space* (PGAS) languages, such as X10 [87], which explicitly express physical locality of memory through *places* and processor/thread affinity. Depending on the relationship between the size of the address spaces assigned to a place, the number of active threads operating within a place, and the available architectural resources, localized coherence protocols can be beneficial. Localized protocols can be optimized for a particular *place*'s partition of the address space and architectural real estate, while still maintaining global address space coherence with respect to other localized protocols.

The designers of future architectures can also benefit from coherence heterogeneity. Consider, for example, a production heterogeneous chip that is partitioned across several different development teams. Each team wants to design its own highly optimized and specialized coherence protocol, tailored and verified for one architectural region. Each design group could work independently if a well-defined heterogeneous coherence composition framework were available to integrate the protocols into a final, *verified* hierarchical protocol, as shown in Figure 23. This concept of distributed coherence protocol design does not have to be limited to a single chip. With a composition framework, multi-chip systems comprised of diverse chips (GPUs and CPUs), from different vendors, could be combined and verified into a global coherence protocol.

Before implementing a coherence protocol in hardware, it is important that the protocol be *verified*. Given the extreme rate of processor requests that a protocol handles per second, even the smallest flaw will inevitably lead to a system failure. An incorrectly designed coherence protocol could cause the chip to deadlock or corrupt data by allowing multiple processors to modify the same block simultaneously. One approach to formally verifying a protocol involves modeling the protocol components and examining every possible reachable state for invalid behavior. The total number of global states to be explored increases exponentially with every new node, message type, or state that is added to the protocol.

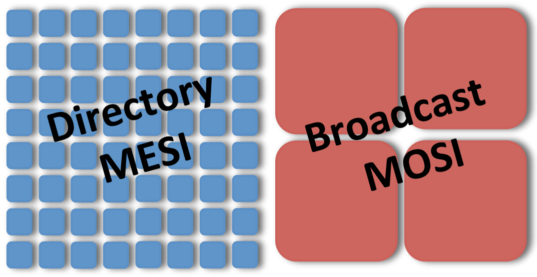


Figure 23 - Example of a heterogeneous multi-chip system that would benefit from heterogeneous coherence hierarchy support.

Intractable verification complexity has the potential to dissuade architects from using hierarchical coherence approaches, despite their many benefits. While many strategies and tools already exist to assist in the verification effort of flat protocols [22, 33, 64, 84, 88-90], hierarchical coherence break these tools by exacerbating many of the problems associated with verification, such as the state space explosion problem [23, 24]. Recent publications [39, 63] have demonstrated very powerful techniques to accelerate verification for hierarchical coherence protocols, but they are limited by a fundamental assumption: that the hierarchy being verified is composed of *homogenous* and *self-*

similar protocols. Such an assumption severely limits the utility and scope of hierarchical coherence for heterogeneous designs or PGAS models. Extending verification to hierarchies of distinct coherence protocols is a hard problem. However, as discussed earlier, there will be a strong desire for flexible, *heterogeneous* coherence hierarchies in the near future. A solution to the verification problem must be found. We will show that the Manger-Client Pairing composition framework holds the key to heterogeneous hierarchy verification.

In this chapter, we extend MCP by proving that using *MCP compliant* protocols in an MCP hierarchy enables rapid verification through a form of *protocol symmetry* [24]. This avoids the need for full state space exploration, reducing verification cost from an intractably large combinatorial space down to verifying each component protocol independently. The contributions of this work are as follows:

- Introduce a new form of protocol structural symmetry called *encapsulation symmetry*, and show how it can reduce verification cost.
- Prove that MCP supports *encapsulation symmetry* and thus can be leveraged as a *verification composition framework* for heterogeneous hierarchies when the hierarchy is composed of formally verified *MCP compliant* protocols.
- Present *remote proxy client* as a technique for porting pre-existing, verified protocols to *MCP compliance* with little design and verification overhead. As a motivating example, this technique is applied to the Broadcast-MOSI protocol from GEMS [37] to enable its integration with a Directory-MESI protocol to form a MCP hierarchy.

- Show through the MurPhi formal checker [22] that this new MCP hierarchy is verified. Further, we use this result to compare the cost of full state-space exploration with that of independent component verification via *encapsulation symmetry*.

The remainder of the paper is organized as follows: Section 2 outlines the related work. Section 3 presents an overview of the MCP framework. Section 4 explains the state enumeration verification strategy in preparation for Section 5, which presents a proof for verification through MCP composition. Section 6 outlines how to adapt existing protocols to be *MCP compliant* via a remote proxy client. Section 7 presents MurPhi verification results followed by a conclusion in Section 8.

6.2 Review of MCP Framework

Manager-Client Pairing (MCP) eases hierarchical coherence protocol design by distinguishing manager agents, those that manage permissions (e.g. directory), from client agents, those that hold permissions (e.g. private caches). By pairing the client agent of a higher protocol with the manager agent of the lower protocol, the client agent behaves as a permissions gateway for the paired manager's protocol. This is possible because MCP defines a permission-checking algorithm that enables component protocols to communicate with each other through a generic query-and-acquire interface, eliminating the need to expose internal operation details outside the protocol's scope. By linking protocols together, coherence hierarchy composition can distribute the coherence responsibility throughout the hierarchy's *coherence realms*. The top-tier *coherence realm* encompasses all users of data within the coherent memory system being monitored by the hierarchical protocol. Each lower-tiered *coherence realm* monitors successively smaller subsets of node coherence. Figure 2 shows an example MCP hierarchy, labeled with MCP terminology.

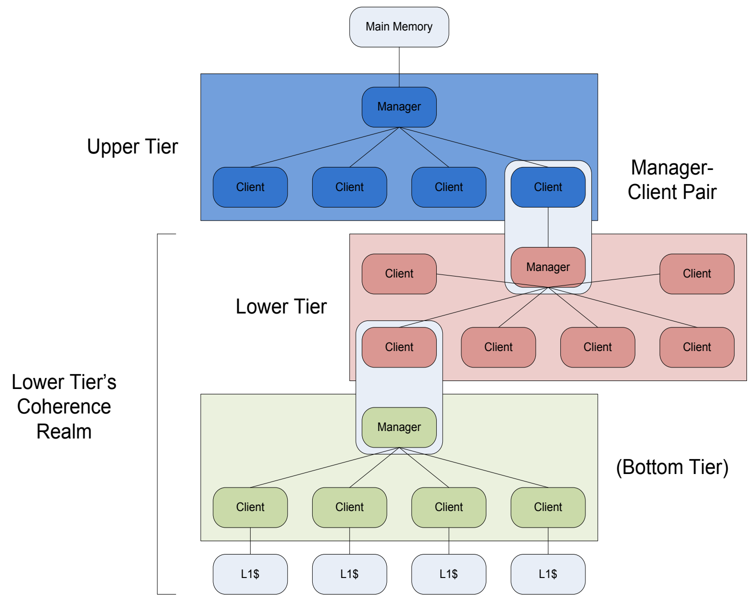


Figure 24 - Manager-Client Pairing coherence hierarchy organization with parts labeled: Manager, Client, Tier, and Realm for the Coherence Domain.

Due to the general interface definition and resultant low level of integration required between realms, Chapter 5 showed that component coherence encapsulation is well preserved, meaning the design details of the protocols used to comprise the system are largely opaque with respect to one another. Furthermore, because this interface's functionality is very similar to the processor and memory interfaces in a conventional flat coherence protocol, the majority of the effort required to adhere to *MCP compliance* is a straightforward one-to-one mapping between MCP actions and already present coherence actions. We define a protocol to be *MCP compliant* if it is a verified invalidation-based coherence protocol that only communicates with the external world through upper (memory) and lower (processor) MCP interfaces as shown in Figure 25.

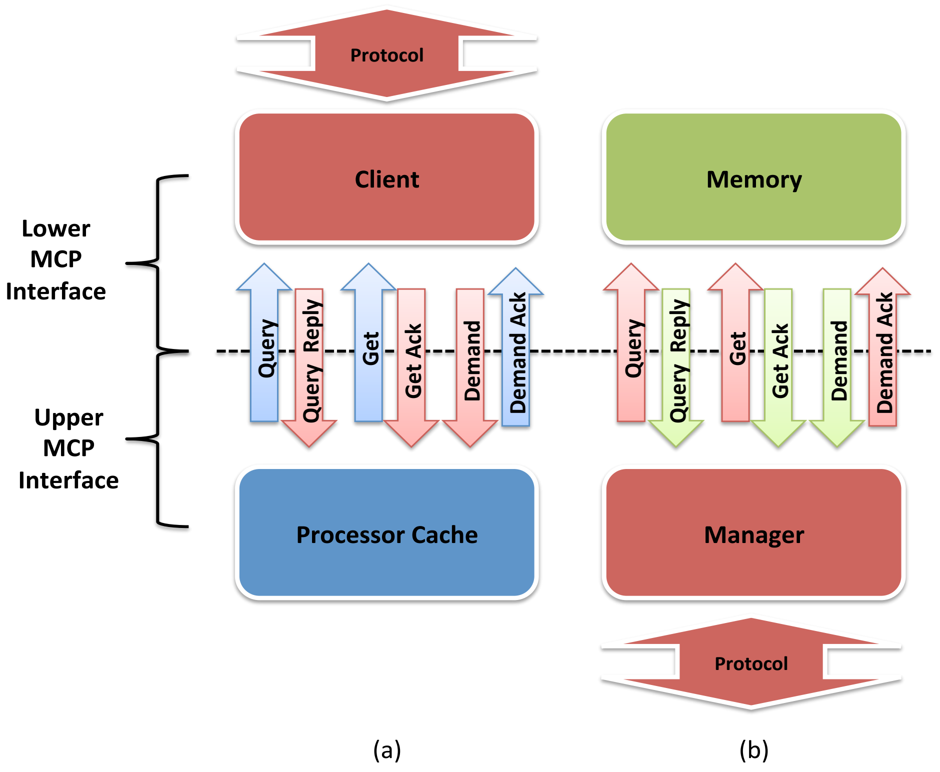


Figure 25 - MCP Interface for (a) lower processor tier and (b) top memory tier

6.3 Reachable State Enumeration Overview

Before constructing the complete proof for MCP-hierarchy validation, an understanding of the underlying verification principles is required. In this section we introduce the problem of *verification through reachable state enumeration.* We discuss verification through enumeration, review the state-space explosion problem, and explain how past research has mitigated this problem through the use of *protocol symmetry*. This leads to our key observation, that the state-space explosion due to hierarchical protocol interactions can also be mitigated if viewed as a form of symmetry.

6.3.1 State Enumeration

Reachable state enumeration is a common strategy employed in coherence protocol verification that automates the process. First, the protocol state machines and surrounding communication medium are described in a protocol description language, such as *MurPhi[22]*. A set of invariants is then defined to establish what conditions must be met for the system to be valid (e.g., *only one modifiable copy of a cache block exists at any time*). Relevant parameters regarding the system configuration (*number of clients, manager organization, network properties*, etc.) are provided, as well as an initial system state from which the verification process can begin. All possible states are then exhaustively generated and invariants checked, following the actions provided in the description. This can be done by either applying a depth-first or breadth-first search, where next-states are generated by applying all possible valid rules to the current state (e.g., *new request generation, request/response event delivery*, etc.). Each new state checks the invariants and, if no violation occurs, marks the current state of the system as *reached* (this is often implemented through the use of a hash table populated with a compressed state notation). If a future-state sequence encounters a state that has already been reached, that branch of the search can be terminated since it has previously been verified. Eventually, all branches will terminate, and, if no violation has been encountered, the protocol can be labeled as verified.

6.3.2 State-Space Explosion and Symmetry

For even reasonably simple coherence protocols, the state space that needs to be exhaustively searched can become intractable quickly. This is due to all the possible state interactions between the clients state machines, manager state machine, and various states of message delivery and ordering, which is aggravated rapidly by how many nodes (i.e. cores) are being modeled. While prior research has proven that modeling of a single cache block address is sufficient to verify a coherence protocol [24], there is no proof that a large-scale system can be fully verified from a similar, scaled-down system. As each additional node is added to the system, the number of possible global states increases exponentially due to all possible interactions between the newly-added client's state machine (and messages) with the previous system's state-space, as well as the additional possible manager states from extending the tracking mechanism to encompass the new node's tracking. For an example of the latter, consider moving from 8 bits to 9 bits in a sharer bit-vector: this results in an increase from 2^8 to 2^9 possible vector states *for each manager state* that requires bit-vector information. Because of the combinatorial nature of the state space problem, we see in Table 2 a dramatic increase in the number of reachable states as the client count increases. These results were collected from a full state space exploration using MurPhi. Figure 26 presents a visual representation of what happens during state-space explosion. This example only shows the reachable states after the first two possible rules are applied to an overly simplified MSI protocol consisting of 2 nodes vs. 4 nodes.

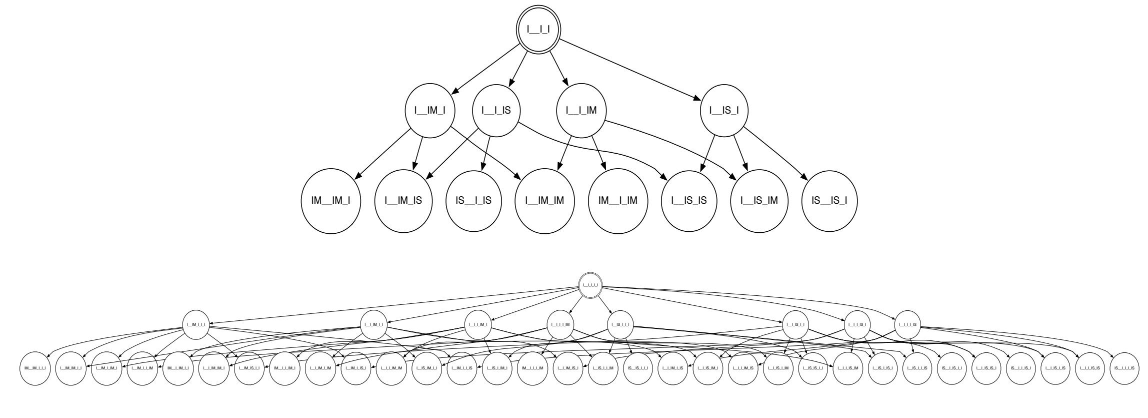


Figure 26 - Example of state space explosion when adding 2 additional nodes to a 2-node MSI protocol

Table 2 - Verification cost of Directory-MESI and Broadcast-MOSI protocols via MurPhi

Protocol	# of States	Time to Verify [s]
2-client Directory-MESI	599	0.10
3-client Directory-MESI	7,077	0.13
4-client Directory-MESI	108,203	3.33
5-client Directory-MESI	1,345,019	91.76
6-client Directory-MESI	26,361,918	15,980.70
2-client Broadcast-MOSI	3,117	0.10
3-client Broadcast-MOSI	166,562	4.79
4-client Broadcast-MOSI	4,307,049	331.82
5-client Broadcast-MOSI	132,871,278	303,244.00
6-client Broadcast-MOSI	500,000,000+	4,000,000+

Due to the often-homogenous nature of client state machines in a coherence protocol, *state symmetry* has been shown to be a powerful way to combat the state-space explosion problem, and can reduce the state-space search scope by as much as 90% [24]. In this approach, several distinct states can be shown to overlap with one another through the exploitation of structural symmetries in the protocol's design, such as abstracting sharer client ID information to a sharer client count. For example, the 4-node composite states {S,S,I,O}, {I,S,S,O} and {O,S,S,I} are symmetric with one another because a simple substitution can show that applying the same sequence of rules that lead from the initial state to each of these states will yield identical results if node ids are rotated/mixed (e.g. {I,S,S,O} becomes {O,S,S,I} if node 0 and node 3 are switched). Again, because of the homogeneity of the client's state machines, there is no behavioral difference at the higher-level description of the protocol behavior; specific node identity information is unimportant. In this way, global state can be viewed as a combination rather than a permutation. In short, *if two system-wide states are symmetric with one another, only one has to be verified to automatically verify the other*. The authors of [24] demonstrate that the notion of structural symmetries extends beyond just node ID abstraction to encompass many other parts of coherence protocol design, including "addresses, data values, memory module-ids and message-ids." In this work we extend this to encompass the *encapsulation symmetries* present in hierarchies composed of independent, well-encapsulated protocols.

6.3.3 Encapsulation Symmetry

Encapsulation symmetry is different from state symmetry in that it does not manifest as a result of protocol homogeneity. Rather, encapsulation symmetry happens when portions of the global state representation can be proven to be independent from other parts of the global state. The simplest example of this phenomenon would be the state-space exploration of two completely isolated state machines, n and m, operating simultaneously. If the size of each state machine's state-space could be expressed as $size_n$ and $size_m$, the state space of both operating simultaneously is ($size_n$ * $size_m$). This is evident because a simple scan could explore the entire space by repeatedly applying a single rule to n, followed by full exploration of state machine m's space.

To express this another way, if the overall state of a system is represented as a string, the state space of each independent state machine can be expressed as strings $string_m$ and $string_n$. The entire state space of these operating simultaneously could then be expressed as the combination of all valid $string_m$ strings concatenated with all valid $string_n$ strings. Figure 27 and Figure 28 show the symmetry in the state space visually for a pair of simple state machines.

Leveraging this kind of symmetry for coherence hierarchy verification would be extremely powerful in combating the state space explosion problem, allowing each component protocol to be verified independently and then merged. However, this symmetry requires proving that the integrated protocols are sufficiently isolated from one another through some form of encapsulation. Additionally, valid merging would require all possible concatenation combinations of these state spaces to guarantee invariant violation freedom. Section 6.4 will develop this further and demonstrate that the interfaced and permission summarizing nature of *MCP compliance* will produce *encapsulation symmetry* in the state space that can safely be leveraged for rapid verification.

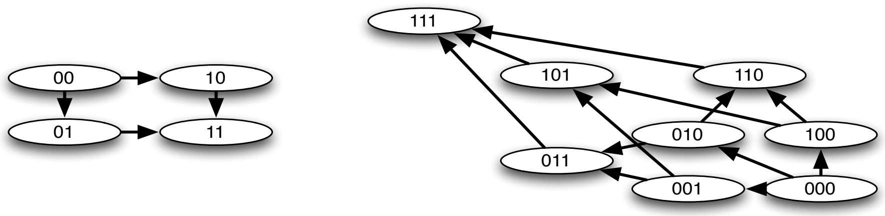


Figure 27 - State-space of two simple state machines, where each element may transition from 0 to 1

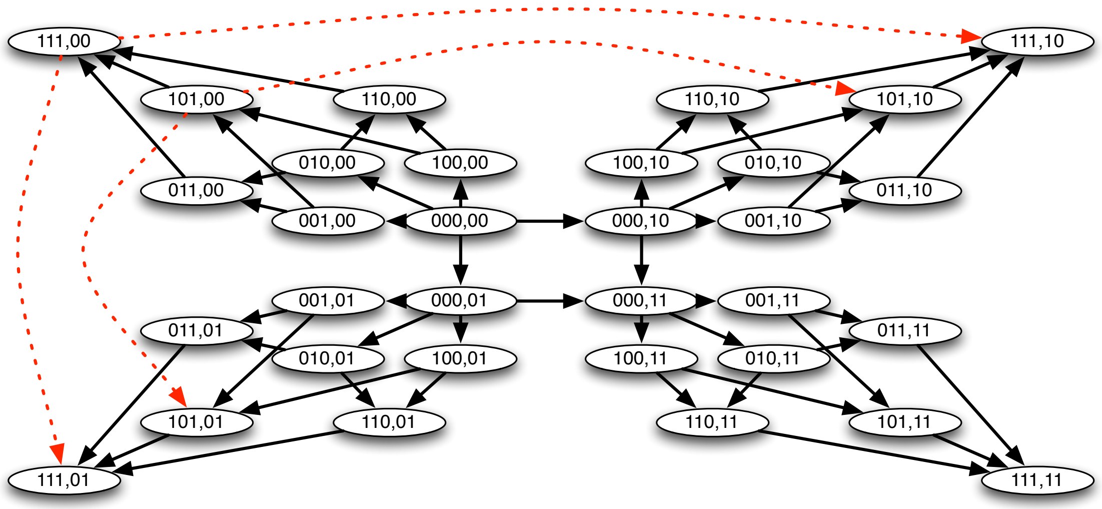


Figure 28 - Full state space exploration of both state machines operating simultaneously, where black arcs represent transitions using the 'execute a single n rule, followed by full m exploration' methodology, and red dotted lines show a few of the alternative paths that would encounter redundant states in the space.

6.4 Formal Verification Strategy for MCP

We propose the use of MCP as a framework for high-speed formal verification of large-scale hierarchical, heterogeneous protocols. In this section we will prove that when formally verified MCP-compliant protocols are assembled into a hierarchy and connected through MCP-interfaces, the hierarchy is also verified. We define '*verified*' to mean a protocol can guarantee the following properties: (1) There can be at most one lowest-tier client with write permission to a block of data; (2) There can be one or more lowest-tier clients with read permission to a block of data if no other lowest-tier client has write permissions; (3) reads are guaranteed to supply the requestor with the most recently written data value at the time the read was inserted into the global order; (4) The system is deadlock and livelock free. These provide a guarantee of coherence protocol design correctness.

Definition 1 –

A protocol is said to be verified if:
1) ∀ reachable global states in a protocol x, a node may have write permissions to a block iff there are no other nodes with read or write permissions to that block.

2) ∀ reachable global states in a protocol x, one or more nodes may have read permissions to a block iff there are no nodes with write permissions to that block.

3) ∀ reachable global states in a protocol x, read requests to a block obtain the value written by the most recent previous write in the global order, w.r.t the read, to that block.

4) ∀ reachable global states in a protocol x, there are no states without possible exits (deadlock) and no condition where a given data block is locked by one node such that it is permanently prevented from being accessed by other nodes (livelock) [64].

As mentioned previously, we define a protocol to be *MCP compliant* if it is a verified invalidation-based coherence protocol that only communicates with the external world through MCP interfaces.

6.4.1 Theorem 1 – Two-tier MCP Composition and Verification

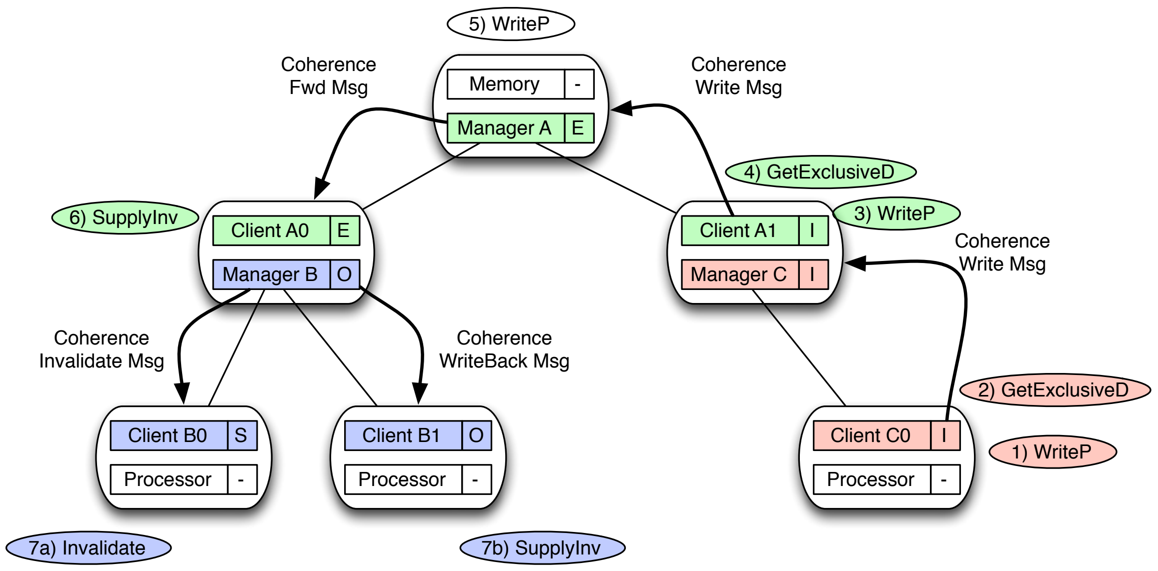
Where R(u,l) := Coherence Realm from interfacing of upper-tier MCP compliant protocol u with lower-tier MCP compliant protocol l through pairing of a u-client with the l-manager.

Lemma 1 – MCP permission distribution ensures R(u, l) will satisfy conditions (1, 2, 3)

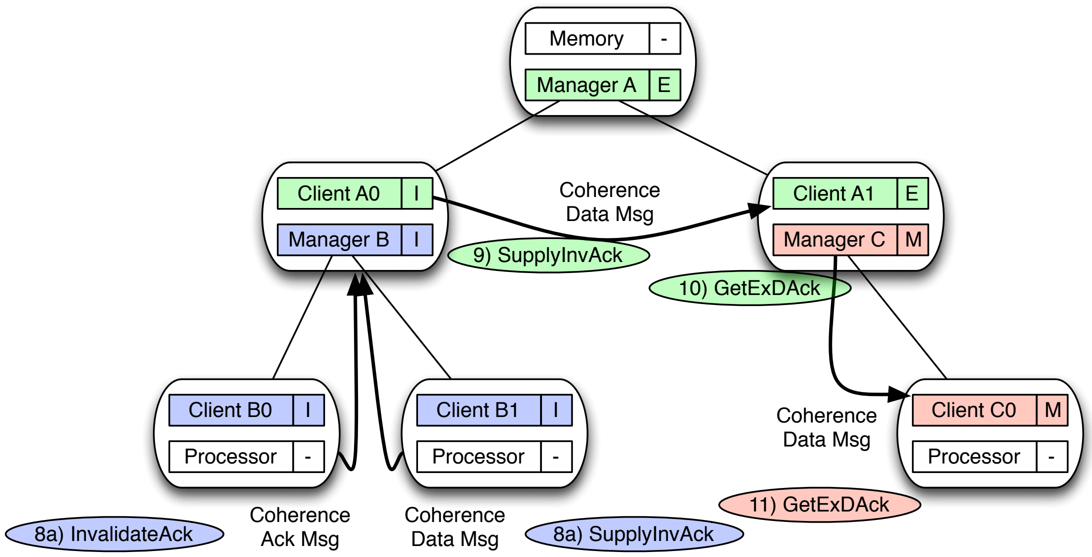
Lemma 2 – For R(u, l), MCP Get/GetAck and Demand/DemandAck pairs do not violate condition (4); all requests are eventually satisfied since both protocols u and l have been previously verified

Theorem 1 – $\therefore \forall u \forall l$, where u and l are MCP compliant protocols, R(u, l) is also verified and MCP compliant.

The supporting lemmas for Theorem 1 have two main themes: Lemma 1 is concerned with proper distribution of permission guarantees to ensure that conditions 1, 2 and 3 of verification are enforced (one writer, multiple readers, read consistency) while Lemma 2 focuses on livelock/deadlock adherence. In Lemma 1, Condition 3 is satisfied because the manager/client pairing is located at the ordering point for its realm, ensuring global ordering of reads and writes is maintained throughout the hierarchy. Conditions 1 and 2 are fundamental properties of MCP composition, and are discussed in depth in Chapter 5 which details the permission allocation algorithm and how the permission inclusion property described by Ladan-Mozes and Leiserson in [63] is implemented by MCP. The realm-miss example from Chapter 5 demonstrating this is reproduced here.



(a)



(b)

Figure 29 - Permission distribution example for an MCP composition

In Figure 29, the sequence of MCP interface events and corresponding coherence actions to acquire data across realm boundaries is shown, starting with (a) the request and demand chain of events and (b) the ack event sequence replying to these requests and demands.

First, the processor paired with Client C0 discovers it has insufficient permission to satisfy a write (1). This results in a GetExclusiveD call to Client C0 (2) which spawns a coherence message to Manager C requesting the data and write permission. Following the MCP algorithm, before responding to the coherence request the paired client A1 is consulted (3). Since A1 does not have sufficient permissions, Manager C temporarily stalls the coherence request from C0 and Manager C issues a GetExclusiveD to its paired client A1 (4). This results in coherence traffic that leads to Client A0 Demanding the lower realm managed by Manager B to supply data and self-invalidate (6). Coherence messages are sent to invalidate all nodes in the realm and request data writeback (7a and 7b).

At this point traffic begins to flow back towards the originating request through reply acknowledgments (8a and 8b). Coherence traffic flows back to Manager B, enabling it to transition to the invalid state and supply data to its paired client A0 (9). The paired client can now proceed by taking its native protocol action, forwarding data to A1 and self invalidating. Upon arrival at A1, the client state transitions to Exclusive and a GetExclusiveDAck is issued across the MCP interface to Manager C. Finally, Manager C can resume processing of the original coherence write message and respond with a coherence data message. Upon reception at Client C0, the write action is complete. This demonstrates that despite having multiple discrete, encapsulated protocols that treat each other as black boxes, permissions are properly enforced across the entire hierarchy because of MCP.

Lemma 2 leverages the fact that all incoming Get actions observed by the lower-tier's lower interfaces (e.g. processor caches) will be satisfied either (a) locally by the lower-tier, (b) remotely by the upper-tier through the lower-tier's upper interface, or (c) by memory via issuance of a Get action from the upper-tier's upper interface with memory. Similarly, all upper-tier lower interfaces not connected to the lower-tier realm will be satisfied either (d) locally by the upper-tier, (e) remotely by the lower-tier through the MCP interface or (f) by memory via the upper-tier's upper interface. In all these instances, eventual completion is guaranteed since memory will always respond and the upper-tier and lower-tier protocols are guaranteed to be livelock and deadlock free prior to composition as a condition of being MCP compliant. All requests are satisfied locally, satisfied by memory, or deferred to another protocol that can guarantee eventual response to any request. Additionally, due to the tree-like organization of an MCP composition and permission distribution, there is no possibility of a cycle in which two MCP compliant protocols are waiting on each other to eventually respond.

Since MCP compliant component protocols are independently verified, we know that no Get action can be delayed indefinitely since Get actions are functionally equivalent to cache actions (read, write, evict). In an MCP composition, Get requests are either satisfied locally, deferred upwards via another Get request which in turn will recursively do the same until satisfied, or deferred downwards via a Demand request (cache-to-cache forwarding behavior, for example). This ensures that all requests will make forward progress as they traverse up or down the tiers until satisfied, proving livelock is not possible. Finally, because MCP does not introduce new states or messages to the component protocols, no new state without exit can arise or be reached, protecting against deadlock.

MCP components meet all the conditions from the definition of verifiability. Rules regarding read permission and write permission distribution for all lowest level clients (i.e., caches) are enforced while guaranteeing livelock and deadlock freedom for all reachable states. Therefore Theorem 1 is proven: a composition of an upper-tier MCP compliant protocol and a lower-tier MCP compliant protocol, connected through an MCP interface will properly distribute permissions and data while retaining livelock and deadlock freedom. Since no verification violation can possibly occur when merging these two protocols, we can safely say the cross product of their respective state spaces into a unified state space will not introduce any new violating states, enabling us to apply encapsulation symmetry from Section 6.3.3 for verification.

6.4.2 Theorem 2 – Arbitrarily Deep MCP Hierarchies

Axiom 1: R(u, l) is both verified and has MCP compliant upper and lower interfaces, being a composition of MCP protocols. ∴ R(u, l) is also a verified MCP compliant protocol.

Theorem 2 – Arbitrarily deep MCP coherence hierarchies are verifiable through induction via the following:

H(2) = R(u, l), where H(2) is a verified MCP compliant protocol hierarchy of two tiers,

and

H(n +1) = R(H(n), l), where H(n+1) is a verified MCP compliant protocol hierarchy of (n +1) tiers

Axiom 1 stems from the structurally recursive nature of MCP composition. From Theorem 1 in the previous sub-section, we know a 2-tier coherence realm composed of independently verified MCP protocols is also verified. Additionally, because each component protocol only has an upper interface and lower interface(s), and the upper interface of protocol 'l' is attached to one of the lower interfaces of protocol 'u', the remaining unconnected interfaces are a single valid upper interface (the 'u' protocol's upper interface), and multiple valid lower interfaces (includes all the lower interfaces of 'l' and all the lower interfaces of 'u' except the most recently connected). Therefore, the whole is a verified protocol with valid upper and lower MCP interfaces, with no other external communication interfaces, meeting all the conditions for MCP compliance.

In a k-tiered MCP hierarchy, the highest coherence realm in the hierarchy (which begins by encompassing only the two top-most tiers of the system) can be proved to be a verifiable MCP compliant protocol through Theorem 2 and Axiom 1. As a result, the two tiers of this realm can logically be replaced by a 'single' MCP compliant protocol, which is the merger of these two tiers (shown in the equations supporting Theorem 2). Through this process, the k-tiered MCP hierarchy has become a (k-1) tiered hierarchy, where the highest protocol in the hierarchy is itself a coherence hierarchy. This can be applied repeatedly until all k-tiers have been merged into the single verified MCP compliant protocol. Figure 30 demonstrates this induction graphically.

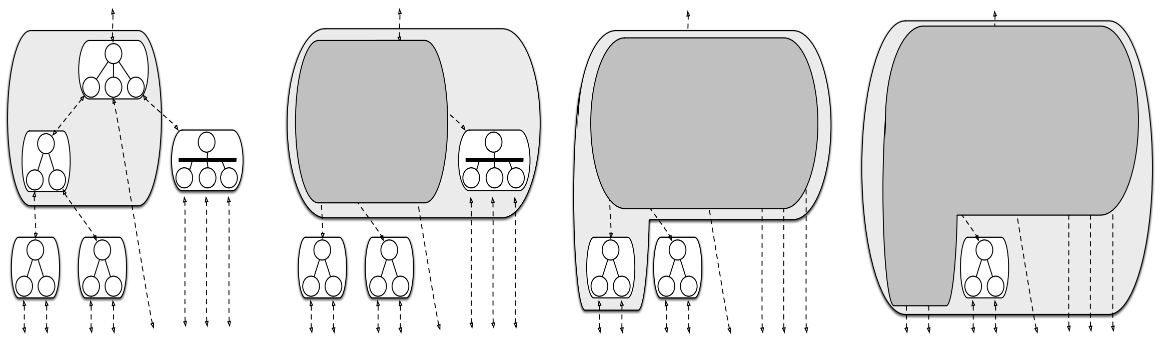


Figure 30 - Graphical representation of coherence hierarchy inductive proof, where the shaded enclosed region represents H(n) for n = 2 → k (k = 4).

6.4.3 Fractal Coherence Viewpoint

Theorem 2 can also be understood through the theorems in the verification process of Fractal Coherence [39]. The two most important properties required for application of Fractal Coherence verification is that (a) the minimum system is formally verified and (b) the hierarchy is observationally equivalent.

Rather than assuming only a single minimum system being replicated, MCP composition assumes multiple systems being integrated that may not be identical. However, if each component is independently formally verified, this is similar to a single kernel protocol being verified and used repeatedly. Additionally, a kind of observational equivalence can be gained through the use of a standardized interface, which MCP provides. From Theorems 1 and 2, we know each component of an MCP hierarchy is formally verified, and connection of these through manager-client pairing of interfaces does not violate verification. As described in Section 6.4.1 regarding Axiom 1 and the interfaces, when treated as black boxes, compositions of MCP component protocols are *nearly* observationally equivalent because each additional tier connects to either upper or lower interfaces while providing new upper or lower interfaces that are functionally equivalent. They are not *strictly* observationally equivalent because the number of interfaces changes depending on the number of clients in the newly attached MCP component protocol. In contrast to Fractal Coherence, which enforces observational equivalence by the 'component protocols' being perfectly self similar, MCP enforces a looser observational equivalence through adherence to a standardized interface definition.

6.5 Remote Proxy Client

6.5.1 Theorem 3 – Verification of Protocols Modified with Remote Proxy Client

Where M(x) := An MCP compliant version of protocol x

Theorem 3 – If protocol x satisfies conditions (1,2,3) for verification, and replaces one client with a remote proxy client, the resulting protocol M(x) does not introduce changes that violate conditions (1,2,3)

∴ ∀x where protocol x is verified, M(x) is also verified

Recall from Section 6.4.1 and Appendix A, there are three major parts to the MCP interface: Queries (permission checking), Gets (permission acquisition) and Demands (permission surrendering). Let us first consider the Query and Get portions of the MCP interface. For Queries, permission status requests cannot modify the state of the protocol as they are simple Boolean checks and therefore have no verification impact. In order to evaluate Theorem 1 with respect to applying the MCP interface specification to a verified non-MCP compliant protocol, the only actions from Appendix A that must be considered are those that can result in state change in the underlying protocol. Each such action much be mapped to an already existing action in the protocol, based on the internal state of the protocol. As mentioned in Section 6.4.1, however, the Get functionality required for supporting MCP corresponds directly to functionality that must already be present for handling and satisfying processor requests in a non-MCP version of the protocol interfaced directly with a processor.

The biggest hurdle when mapping a pre-existing protocol's functionality to the MCP interface is implementing upper-tier initiated Demands. The memory controller interface is typically not able to issue requests for invalidations or downgrades in a conventional flat coherence protocol. However, these actions can be emulated very easily if upper-tier Demands are modeled as requests from a local client, similar to the pseudo-CPU mechanism in DASH [30]. In our framework, this functionality is served by the remote proxy client.

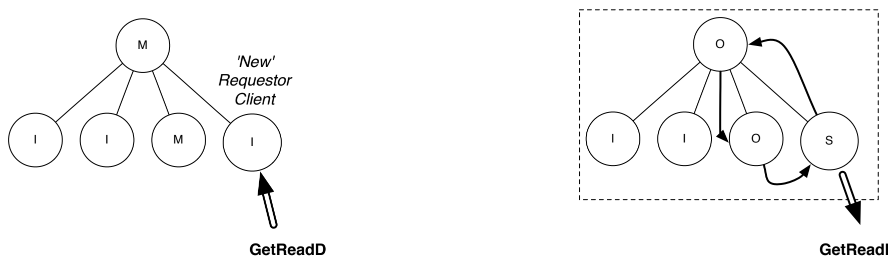


Figure 31 - Local GetReadD Get sequence (Request and Response) in a MOSI protocol that currently holds write permissions.

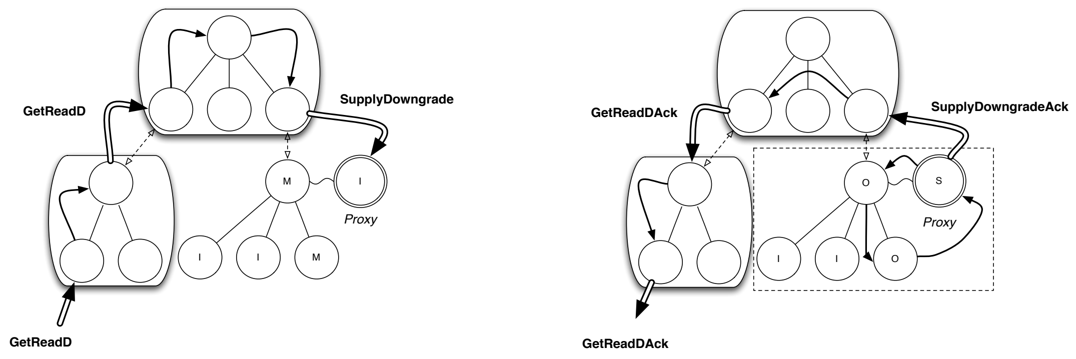


Figure 32 - Remote GetReadD Get sequence (Request and Response), using a proxy client to satisfy SupplyDowngradeAck Demand request in a MOSI tier that currently holds write permissions

The remote proxy client acts on behalf of the upper-tier, issuing local protocol requests to satisfy incoming Demand requests. In addition to this, the remote proxy client is also stateful; it becomes a summary of all the permissions held by nodes external to this coherence realm. As long as the protocol is verified, permission will be assigned to this proxy correctly. This ensures permission exclusion is preserved when necessary, and permission will eventually be passed to the appropriate originator that caused the Demand. The remote proxy client does not communicate directly across tier boundaries; remote traffic is routed through the MCP interface in the manager.

Figure 31 and Figure 32 show an example of how the Demand handling behavior of the remote proxy client is similar to the behavior of a local requestor in a MOESI directory protocol. The actions required by the coherence realm to satisfy a SupplyInvalidate (See Appendix A) are identical to those required for handling a write request from a client in the invalid state. A message is sent to the owner client (the client in either the M, O, or E state) to initiate a cache-to-cache transfer and a self-invalidation. Invalidation messages are sent to all other sharers in the bit vector. When this sequence concludes, the realm manager and other clients will have given write permissions and a copy of the data to this client by removing all readable copies from the realm. In the case of the remote proxy client, the realm can respond with a SupplyInvalidateAck upon completion of the protocol sequence since all conditions are met (i.e., the realm no longer has any copies with read or write permission and the most recent copy of the data is available and ready for forwarding).

Implementing remote proxy client does not actually require adding another client to the protocol. Rather, an existing client can be sacrificed to act as the remote proxy client. So a protocol that is verified for four clients could be made into a 3-client MCP compliant protocol by selecting one of the clients to serve the role of a remote proxy client. Since Demand handling does not introduce new states or messages when a proxy is present, there are no changes to the original state machine, and the protocol remains verified.

In summary, encapsulating a protocol via MCP interfaces can be seen as applying a translation layer that introduces no new additional state transitions, states, or new messages to the protocol. This preserves the verification properties of the original component protocol. Since MCP does not modify the state machine, if the base protocol correctly distributes permissions without deadlocking or livelocking, so does an MCP compliant version of the same protocol.

6.6 Results

In order to demonstrate the usefulness of MCP as a verification technique, a heterogeneous hierarchy was implemented and verified using the MurPhi toolkit. The hierarchy was created from the composition of two protocols: Directory-MESI and Broadcast-MOSI. We designed the Directory-MESI protocol to be natively MCP compliant, without the need for a *remote proxy client.* The Broadcast-MOSI protocol is a MOSI protocol communicating over a shared bus and was ported directly from the GEMS implementation to MurPhi. To make this protocol MCP compliant, a client was scavenged to serve as the *remote proxy client;* no other changes were made to the underlying "off-the-shelf" GEMS protocol.

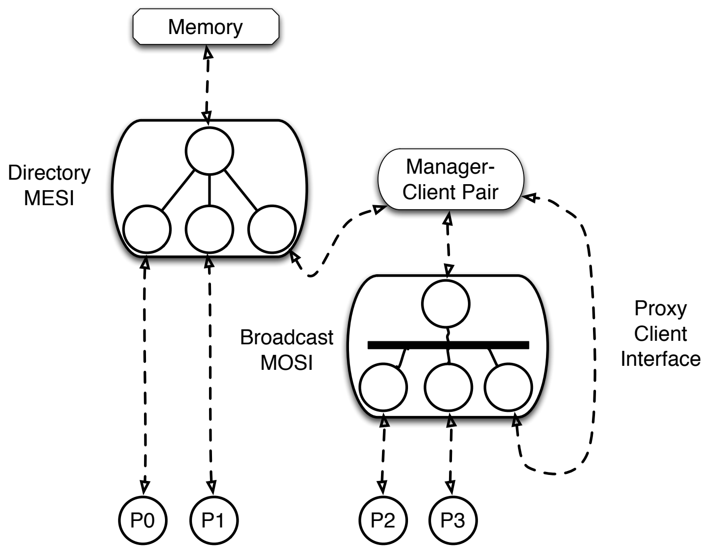


Figure 33 - Evaluated Heterogeneous Hierarchical Protocol Structure

The evaluated heterogeneous hierarchical protocol is shown in Figure 33. The number of clients in each protocol is varied and the configurations are denoted in Table 3. The figure illustrates the Dir3 + B3 configuration, where one client in the Directory-MESI is paired with the Broadcast-MOSI, and one client in the Broadcast-MOSI is dedicated as a *remote proxy client.*

The hierarchical protocol was evaluated via full state exploration using MurPhi, and was also verified by leveraging MCP structural symmetry. The results of this verification effort are denoted in Table 3. The number of states represents the full state space of the combined protocols, and the time to verify without MCP is the total time for MurPhi to complete a full state exploration (similar to Figure 28). The verification costs with MCP is simply the sum of the verification costs of the component protocols, due to encapsulation symmetry. As is evident from these results, MCP greatly reduces the verification cost, especially at the higher client count design points.

Table 3 - Comparing verification cost of heterogeneous hierarchical protocols with and without leveraging MCP protocol structural symmetry[3]

Protocol	# of States	Time to Verify [s]	# of States (w/MCP)	Time to Verify (w/MCP) [s]
Dir2 + B2	11,861	0.34	3,716	0.20
Dir2 + B3	425,990	17.31	167,161	4.89
Dir3 + B2	182,197	8.51	10,194	0.23
Dir3 + B3	5,367,735	542.44	173,639	4.92
Dir4 + B3	71,642,216	84,734.63	274,765	8.12
Dir3 + B4	143,552,706	317,891.00	4,314,126	331.95
Dir4 + B4	500,000,000+	7,000,000+	4,415,252	335.15

6.7 Conclusion

There is a strong interest in multi-core architectures that use flexible, heterogeneous coherence hierarchies, such as CPU+GPU pairings or multi-vendor coherent shared memory ensembles. But without a verification solution, these protocols—and the potentially powerful and energy-efficient systems they enable—cannot be built. It is clear that a solution to the verification problem must be found. Prior solutions were limited to *homogeneous* hierarchies wherein every level of the system must practice the same protocol. This approach leverages the Manager-Client Pairing encapsulation composition framework, which explicitly supports heterogeneity. Using MCP, we proved that any heterogeneous protocol could be verified in no more time than it would take to validate each individual protocol in isolation.

The theoretical nature of this chapter is inescapable. However, we have tried to bring this work to reality by implementing a coherence hierarchy in a formal verification

[3] Dir4+B4 state space was too large for full state space verification to complete, due to the intractable nature of state-space explosion. The numbers presented in the first two columns for these configurations are the minimum bounds collected from the periodic progress report after 80 days of execution.

tool. The intractability of obtaining results for our largest simulations establishes the need for formal verification acceleration. We defined a new form of protocol structural symmetry for coherence hierarchies, based on protocol encapsulation and permission distribution. We proved how MCP can be used as a verification composition framework for heterogeneous hierarchies composed of pre-verified protocols. With the framework presented here, hierarchical, heterogeneous coherence can become an industrial success rather than being limited by practical verification complexities.

CHAPTER 7 - HETEROGENEOUS HIERARCHY INTERACTIONS

One of the biggest strengths MCP brings to the design of coherence hierarchies over previous work is the ability to support heterogeneity across the composing protocols. Hierarchical coherence has already been demonstrated to be a powerful technique several times throughout this dissertation. Within these hierarchies, heterogeneity can be used to exploit the heterogeneous nature of the hierarchically composed architecture if they are less fractally organized than the data diffusion machine's hierarchical buses. Optimizing communication and sharing behaviors to match the structural organization of large composite systems obviously played a roll in the decision to employ heterogeneity in the DASH, Wildfire, Piranha and Superdome architectures [25-30, 46].

Another opportunity for heterogeneity may arise from the applications themselves, specifically from the presence of common patterns of communication. This could also motivate coherence heterogeneity, as was the case in the Flash architecture's decision to employ a programmable protocol through the MAGIC chip [28]. While not a truly hierarchical protocol, Flash's designers did recognize that different sharing and communication behaviors exist across applications, or even within a single application, and providing support to take advantage of the asymmetry in the communication needs of the data transfer patterns could improve performance. While that Stanford team focused primarily on the asymmetry between shared memory and message passing programming models, we know that modern multi-threaded applications have observable asymmetries within and across their shared memory communication, depending on the kind of multi-threaded parallelism being employed.

Recent research suggests that the design patterns [91-93] used when designing and/or implementing an algorithm in software can play a significant role in the observed communication between threads in a multi-threaded application. This implies opportunity for predicting and exploiting very specific communication patterns tailored to those design patterns. While a full investigation of that space is well beyond the scope of this dissertation, providing insights for the most common patterns is not. It is important for architects to understand what impact coherence and their associated state interactions will have on performance, especially for the most common communication patterns of the most popular algorithms. This becomes less intuitive when we begin considering coherence hierarchies and the interaction of asymmetric coherence states across tiers. In providing this information, it is hoped that this chapter will generate thought on what kinds of techniques may prove useful, as specialized design-pattern-aware optimizations become the norm for future multi-threaded hardware design.

It is worth noting that some work has already begun in this general direction of algorithmic-aware coherence design. Barrow-Williams et al. [19] observe that many parallel applications have a disproportionate amount of neighbor-to-neighbor communication as a percent of the overall communication of shared data. The authors thus created a proximity-aware coherence protocol to enable more efficient neighbor sharing and communication to improve performance. Even more specialized, Singh et al. [31] recognized that there exists a class of applications that can benefit from execution on a GPU, but would require coherence for correctness due to some inter-workgroup communication. They demonstrate that they can exploit the properties of these algorithms to accurately predict data usage liveness, and leverage that to construct a coherence protocol that employs a form of timed self-invalidation. The removal of invalidation and invalidation acknowledgements reduced the bandwidth requirements of coherence sufficiently to make it feasible for a GPU environment.

7.1 Common Communication Trends

Before exploring this space, it is important to establish what trends in communication are most commonly employed. From inspection of the SPLASH2 [86] and Parsec [93-95] benchmark suites, we see that while several parallel patterns may exist, the most frequently employed in our benchmarks are data/geometric decomposition and pipeline parallelism. Table 4, borrowed from Bienia's dissertation on the Parsec benchmark suite [96], describes the properties of the Parsec benchmarks w.r.t. the algorithms employed and the data behaviors found within. Table 5 is a reproduction of the information provided in [97], giving a summary of the observed communication patterns of Parsec and Splash-2 as a percent of total dynamic communicating behavior (where values are rounded to the nearest 5% based on the graph data provided).

Table 4 – Parsec benchmark characteristics (from [96])

Program	Application Domain	Parallelization		Working Set	Data Usage	
		Model	Granularity		Sharing	Exchange
blackscholes	Financial Analysis	data-parallel	coarse	small	low	low
bodytrack	Computer Vision	data-parallel	medium	medium	high	medium
canneal	Engineering	unstructured	fine	unbounded	high	high
dedup	Enterprise Storage	pipeline	medium	unbounded	high	high
facesim	Animation	data-parallel	coarse	large	low	medium
ferret	Similarity Search	pipeline	medium	unbounded	high	high
fluidanimate	Animation	data-parallel	fine	large	low	medium
freqmine	Data Mining	data-parallel	medium	unbounded	high	medium
raytrace	Rendering	data-parallel	medium	unbounded	high	low
streamcluster	Data Mining	data-parallel	medium	medium	low	medium
swaptions	Financial Analysis	data-parallel	coarse	medium	low	low
vips	Media Processing	data-parallel	coarse	medium	low	medium
x264	Media Processing	pipeline	coarse	medium	high	high

Table 5 – Summary of Parsec and Splash-2 dynamic communication trends (gathered from [97])

	Read Only			Migratory			Producer-Consumer			
	2	3 - 15	15 - 32	2	3 - 15	15 - 32	1	2	3 - 15	15 - 32
Parsec										
blackscholes	-	-	50%	35%	-	5%	-	-	-	-
bodytrack	-	-	10%	-	30%	-	20%	-	-	-
canneal	5%	30%	60%	10%	15%	-	5%	5%	10%	-
dedup	-	-	-	10%	10%	-	-	-	-	-
facesim	-	-	-	-	20%	-	10%	-	-	-
ferret	-	-	-	-	35%	-	5%	-	-	-
fluidanimate	5%	-	45%	65%	5%	-	60%	-	-	-
freqmine	-	10%	10%	20%	-	-	-	-	-	-
treamcluster	-	50%	40%	-	-	55%	15%	-	-	-
swaptions	-	-	40%	-	-	75%	-	-	-	-
vips	-	-	15%	20%	15%	-	-	-	-	-
x264	-	10%	10%	-	-	-	-	-	-	-
Splash-2										
barnes	-	-	35%	15%	-	15%	20%	10%	10%	-
cholesky	-	-	-	-	-	5%	-	-	-	-
fft	-	-	-	-	-	-	-	-	-	-
fmm	-	10%	45%	20%	15%	5%	20%	10%	10%	-
lu	-	-	-	50%	-	-	-	-	-	-
ocean	5%	10%	40%	-	-	5%	90%	-	-	-
radiosity	-	-	85%	-	15%	45%	-	-	-	-
radix	5%	-	50%	-	-	-	15%	-	-	-
raytrace	-	45%	55%	-	-	90%	-	-	-	-
volrend	-	15%	65%	-	20%	15%	55%	-	-	-
water-nsq	-	-	20%	-	-	90%	-	-	-	5%
water-spatial	-	-	20%	-	-	10%	-	-	-	80%

In [97], the communication patterns are defined as follows:

Read-only – Data that is written by a single thread, and never written to again once read by another thread. Read-only sharing is largely comprised of benchmark input file processing and shared data structure initialization.

Migratory – Data structures that are read-then-modified inside an atomic region by multiple threads throughout a benchmark's execution is considered migratory. A common behavior in parallel workloads, data structures that exhibit migratory sharing tend to not change their behavior throughout the lifetime of the benchmark.

Producer-Consumer – Data that is defined by a persistent relationship between the write and read sets for the data, where data is modified by threads in the write set and may be consumed by threads in the read set before the next write. The authors of [97] discuss a strict producer-consumer (a single writer) and loose producer-consumer (multiple writers in the write set) relationship and ultimately settle on providing analysis under the loose producer-consumer due to changes over time in the write set as data 'ownership' in the algorithm migrates across threads. For example, data decompositions that periodically reevaluate their working sets would exhibit this behavior.

From Table 5, we see that Read-Only data, Migratory data and producer-consumer data provides reasonable coverage for most of the SPLASH and Parsec benchmark suites dynamic communication. This makes a lot of sense when considering the algorithms involved since many of the benchmarks employ a large input source, and/or a leader thread that populate many of the shared data structures (Read-only), many have shared data structures that are guarded by mutex locks (Migratory), and many employ a geometric/data decompositions strategy for extracting parallelism (Producer-Consumer).

It is worth taking a moment to consider why decomposition exhibits producer-consumer sharing. Decomposition often divides the physical object being modeled or the address space of the algorithm in such a way that locality can be leveraged to reduce communication. Locality only works up to a point, however, and neighboring regions within the decomposed space are forced to communicate along their shared edges. Since, decomposition has a notion of data ownership (the thread who manages a given region is responsible for updating state), producer-consumer communication will be observed. As data within a region is updated over time by the owner thread, later computations by neighbors who require those points for their own computation (e.g. gravitational pull in N-body simulations) will be observed as read-after-write traffic, with a degree of predictable persistence in the writer and reader sets.

This is confirmed by the data in Table 5, which shows that for the canonical geometric decomposition benchmarks Ocean and Water-Spatial from SPLASH-2, over 80% of the dynamic communicating traffic is producer-consumer. Most of this is a single consumer for Ocean, due to computation only requiring immediate neighbor information. As a result, sharing only happens on the decomposed region's shared boundaries. Water-Spatial, however, exhibits sharing at the other end of the spectrum, having a relatively high consumer degree comparatively (16-32 consumers for 32 thread simulations). A closer look reveals this is a result of Water-Spatial's molecule interaction 'cut-off sphere' radius being equal to half the box-length in which the simulated water is contained, effectively ensuring most computations will require communication with more than half of the other regions/threads [98]. High amounts of producer-consumer sharing are also observed in Table 5 for other data decomposition benchmarks like Barnes, FluidAnimate, FMM, and Volrend [86].

7.2 Representative Traces

Leveraging the insights provided in Section 7.1, we can construct traces that represent some of the most frequent sharing behaviors seen in our benchmarks. In doing so, we will be able to isolate the impact coherence has on specific communication patterns from noise, and develop an understanding of the benefits different coherence states can have or what kinds of penalties occur for these different memory reference sequences

The interaction between private data and coherence hierarchies will not be considered since [53] and [54] sufficiently motivate that techniques to ignore coherence for private data are quite effective at reducing the pressure on the coherence engine, and can be done safely since truly private data, such as stack-space references and register spill traffic, do not become incoherent since no other threads write to these addresses. Additionally, while there may be implication from the effects of false sharing, they will not be considered during this evaluation either. False sharing [99] is an artifact of architectural choices, such as large cache block size and do not represent true communication, making it a tangential concern to the insights being presented here regarding communication trends.

7.2.1 Modeling Contention and Time

The resource strain in the system from traffic that is independent of the current communication sequence (which may be other communication sequences or thread-private traffic) must be modeled in a consistent way in order to properly contrast the impact of coherence on different inter-thread communication patterns. Care must be taken to avoid introducing other effects that might distort the patterns or conclusions being drawn. System strain is most observable in the degree of cache contention, and thus forced eviction of blocks from the different levels of the cache hierarchy makes an ideal candidate for modeling various levels of contention. For the communication patterns in question, the frequency of access to the address in the sequence and the temporal locality of these accesses versus the volume of 'other' traffic will ultimately determine where in the cache hierarchy the blocks involved in communication are found.

Therefore, we can model the spectrum of behaviors due to cache pressure by simply injecting eviction-causing traffic into the trace, forcing eviction of the communicating data from the different levels of the cache hierarchy. These injections can be controlled such that they target a specific cache within the hierarchy, and can also be employed at various degrees of interleaving with the communication traffic being evaluated. The interleaving options are somewhat pattern specific, but can be generally classified as either coarse-grained, which result in tight memory sequences with few interruptions, or fine-grained, interleaving evictions between all events, resulting in looser memory sequences. Tight sequences can represent either the absence of external traffic or the presence of well-coordinated (or otherwise timely) communication. Loose sequences from fine-grained interleaving of evictions model either higher cache traffic, or communication sequences that are disjoint in time. The producer-consumer pattern has an additional interleaving option referred to as 'split', which splits the producer from the consumer, but not the read and write pair definitive of the producer-consumer's consuming read/producing write sequence.

Read-Only	Thread	Access type	Address
First Write	T0	Write	0x00A0
Read-Only	t3	Read	0x00A0
Read-Only	t6	Read	0x00A0
Read-Only	t25	Read	0x00A0
Read-Only	t4	Read	0x00A0

Read-Only L1	Thread	Access type	Address
First Write	t0	Write	0x00A0
Eviction Access	t0	Read	0x010000A0
Eviction Access	t0	Read	0x020000A0
Eviction Access	t0	Read	0x030000A0
Eviction Access	t0	Read	0x040000A0
Read-Only	t3	Read	0x00A0
Eviction Access	t3	Read	0x110000A0
Eviction Access	t3	Read	0x120000A0
...	...	...	...

Read-Only L2	Thread	Access type	Address
First Write	t0	Write	0x00A0
Eviction Access	t0	Read	0x010000A0
Eviction Access	t0	Read	0x020000A0
Eviction Access	t0	Read	0x030000A0
Eviction Access	t0	Read	0x040000A0
Eviction Access	t0	Read	0x050000A0
Eviction Access	t0	Read	0x060000A0
Eviction Access	t0	Read	0x070000A0
Eviction Access	t0	Read	0x080000A0
Read-Only	t3	Read	0x00A0
Eviction Access	t3	Read	0x110000A0
Eviction Access	t3	Read	0x120000A0
Eviction Access	t3	Read	0x130000A0
Eviction Access	t3	Read	0x140000A0
Eviction Access	t3	Read	0x150000A0
...	...	...	...

Figure 34 - Fine-grained eviction interleavings targeting the L1 and L2 cache in order to incorporate contention and/or temporal effects into the communication sequence for a Read-Only trace. Assumes an L1 with an associativity of 4 and an L2 with an associativity of 8

Figure 34 demonstrates the difference between a baseline Read-Only trace and the same trace with targeted L1 and L2 eviction. Despite this example, the L1 is never flushed because it is private to each core and thus evictions to the L1s have minimal impact on coherence. Similarly, the L4 is also never flushed in these evaluating traces since we are interested in observing on-chip communication behaviors. It is also fairly reasonable to assume a system where the L4 is sufficiently large to capture any temporal locality for recently communicated data.

7.2.2 Modeling Local and Remote Communication

In addition to external traffic, the nature of the communication from a physical thread-mapping perspective must also be considered. For the 2-level architecture being modeled, this results in two classes of access, local (intra-cluster) and remote (inter-cluster). These two classes map to the coherence hierarchy composition as well, where local communication is isolated to a lower tier whereas remote communication will require indirection through the upper-tier protocol.

Since both kinds of accesses are possible and indeed probable, four versions of each trace have been constructed to examine different cluster-locality properties. The first two are simple all-local and all-remote sequences, where either all nodes involved in the trace reside in a single cluster, or all nodes reside in separate clusters. The second two are mixings of these, the first being coarse grain interleaving where all local communication behaviors of the lower-tier protocol are exploited before accesses migrate to another cluster. The second is fine-grained interleaving that effectively causes the active request to ping-pong across clusters. Both of these exhibit the cold and warm effects of cluster locality, and thus are influenced by hierarchical coherence, but in different ways. Figure 35, Figure 36, and Figure 37 present the different traces used in the following evaluation, color-coded by cluster. Due to the prevalence of single producer-consumer pairs in Table 5, special one-consumer versions of producer-consumer were also constructed.

Finally it is worth mentioning that the nature of local and remote in this context does not have any special meaning regarding the location of the home L4 (since the L4 cache is a distributed shared cache) for the trace address. This clarification is to ensure the use of remote in this context is not confused with the homeward notion of 'remote traffic' used in Piranha [25] and other similar architectures.

Read-Only Local Only

[0-0]	GetWrite
[0-1]	GetRead
[0-2]	GetRead
[0-3]	GetRead
[0-0]	GetRead
[0-1]	GetRead
[0-2]	GetRead
[0-3]	GetRead

Read-Only Remote Only

[0-0]	GetWrite
[8-0]	GetRead
[9-0]	GetRead
[10-0]	GetRead
[0-0]	GetRead
[8-0]	GetRead
[9-0]	GetRead
[10-0]	GetRead

Read-Only Coarse Interleaving

[0-0]	GetWrite
[0-1]	GetRead
[0-2]	GetRead
[8-0]	GetRead
[8-1]	GetRead
[8-2]	GetRead
[9-0]	GetRead
[9-1]	GetRead
[9-2]	GetRead
[0-1]	GetRead
[0-2]	GetRead
[8-0]	GetRead
[8-1]	GetRead
[8-2]	GetRead
[9-0]	GetRead
[9-1]	GetRead
[9-2]	GetRead

Read-Only Fine Interleaving

[0-0]	GetWrite
[8-0]	GetRead
[9-0]	GetRead
[0-1]	GetRead
[8-1]	GetRead
[9-1]	GetRead
[0-2]	GetRead
[8-2]	GetRead
[9-2]	GetRead
[8-0]	GetRead
[9-0]	GetRead
[0-1]	GetRead
[8-1]	GetRead
[9-1]	GetRead
[0-2]	GetRead
[8-2]	GetRead
[9-2]	GetRead

Figure 35 – Read-Only trace sequence

Migratory Local Only

[0-0]	GetRead
[0-0]	GetWrite
[0-1]	GetRead
[0-1]	GetWrite
[0-2]	GetRead
[0-2]	GetWrite
[0-3]	GetRead
[0-3]	GetWrite

Migratory Remote Only

[0-0]	GetRead
[0-0]	GetWrite
[8-0]	GetRead
[8-0]	GetWrite
[9-0]	GetRead
[9-0]	GetWrite
[10-0]	GetRead
[10-0]	GetWrite

Migratory Coarse Interleaving

[0-0]	GetRead
[0-0]	GetWrite
[0-1]	GetRead
[0-1]	GetWrite
[0-2]	GetRead
[0-2]	GetWrite
[8-0]	GetRead
[8-0]	GetWrite
[8-1]	GetRead
[8-1]	GetWrite
[8-2]	GetRead
[8-2]	GetWrite
[9-0]	GetRead
[9-0]	GetWrite
[9-1]	GetRead
[9-1]	GetWrite
[9-2]	GetRead
[9-2]	GetWrite

Migratory Fine Interleaving

[0-0]	GetRead
[0-0]	GetWrite
[8-0]	GetRead
[8-0]	GetWrite
[9-0]	GetRead
[9-0]	GetWrite
[0-1]	GetRead
[0-1]	GetWrite
[8-1]	GetRead
[8-1]	GetWrite
[9-1]	GetRead
[9-1]	GetWrite
[0-2]	GetRead
[0-2]	GetWrite
[8-2]	GetRead
[8-2]	GetWrite
[9-2]	GetRead
[9-2]	GetWrite

Figure 36 – Migratory trace sequences

Prod-Cons Single Local

[0-0]	GetWrite
[0-1]	GetRead
[0-0]	GetWrite
[0-1]	GetRead

Prod-Cons Single Remote

[0-0]	GetWrite
[8-0]	GetRead
[0-0]	GetWrite
[8-0]	GetRead

Prod-Cons Local Only

[0-0]	GetWrite
[0-1]	GetRead
[0-2]	GetRead
[0-3]	GetRead
[0-0]	GetWrite
[0-1]	GetRead
[0-2]	GetRead
[0-3]	GetRead

Producer-Consumer Remote Only

[0-0]	GetWrite
[8-0]	GetRead
[9-0]	GetRead
[10-0]	GetRead
[0-0]	GetWrite
[8-0]	GetRead
[9-0]	GetRead
[10-0]	GetRead

Prod-Cons Coarse Interleaving

[0-0]	GetWrite
[0-1]	GetRead
[0-2]	GetRead
[8-0]	GetRead
[8-1]	GetRead
[8-2]	GetRead
[9-0]	GetRead
[9-1]	GetRead
[9-2]	GetRead
[0-0]	GetWrite
[0-1]	GetRead
[0-2]	GetRead
[8-0]	GetRead
[8-1]	GetRead
[8-2]	GetRead
[9-0]	GetRead
[9-1]	GetRead
[9-2]	GetRead

Prod-Cons Fine Interleaving

[0-0]	GetWrite
[8-0]	GetRead
[9-0]	GetRead
[0-1]	GetRead
[8-1]	GetRead
[9-1]	GetRead
[0-2]	GetRead
[8-2]	GetRead
[9-2]	GetRead
[0-0]	GetWrite
[8-0]	GetRead
[9-0]	GetRead
[0-1]	GetRead
[8-1]	GetRead
[9-1]	GetRead
[0-2]	GetRead
[8-2]	GetRead
[9-2]	GetRead

Figure 37 – Producer-Consumer trace sequences

7.3 Heterogeneous Hierarchical interaction Analysis

To evaluate the impact of hierarchical coherence, seven coherence hierarchy composition pairings will be evaluated in this section to demonstrate the impact of the S (shared) and E (exclusive) states at different tiers in the context of protocol hierarchies at different. The O (owner) and F (forwarder) states are not evaluated in this hierarchy composition framework, due to their being enhanced sharers whose primary functionality is to enable cache-to-cache forwarding optimizations, which has little impact on protocol operation outside of potentially reducing data delivery latency and bandwidth. The E state, however, introduces several interesting phenomenon, due to its somewhat hybrid nature as a read-request turned write-permission.

The convention used to describe the hierarchy composition will be {lower-tier protocol, higher-tier protocol}, such that a collection of cluster MESI protocols managed by a top level MI protocol is expressed as {MESI, MI}. The terms intra-cluster (or cluster) protocol, lower-tier (or lower) protocol, and local protocol may be used interchangeably to describe the protocol between the L2 clients and L3 manager that communicate through direct channels. Intra-cluster protocol, top-level (or top) protocol, upper-tier (or upper) protocol and remote protocol all describe the L3 client to L4 manager protocol which communicate through the interconnect torus network. A replica of Figure 11 from Chapter 4, describing the modeled architecture is presented here again in Figure 38

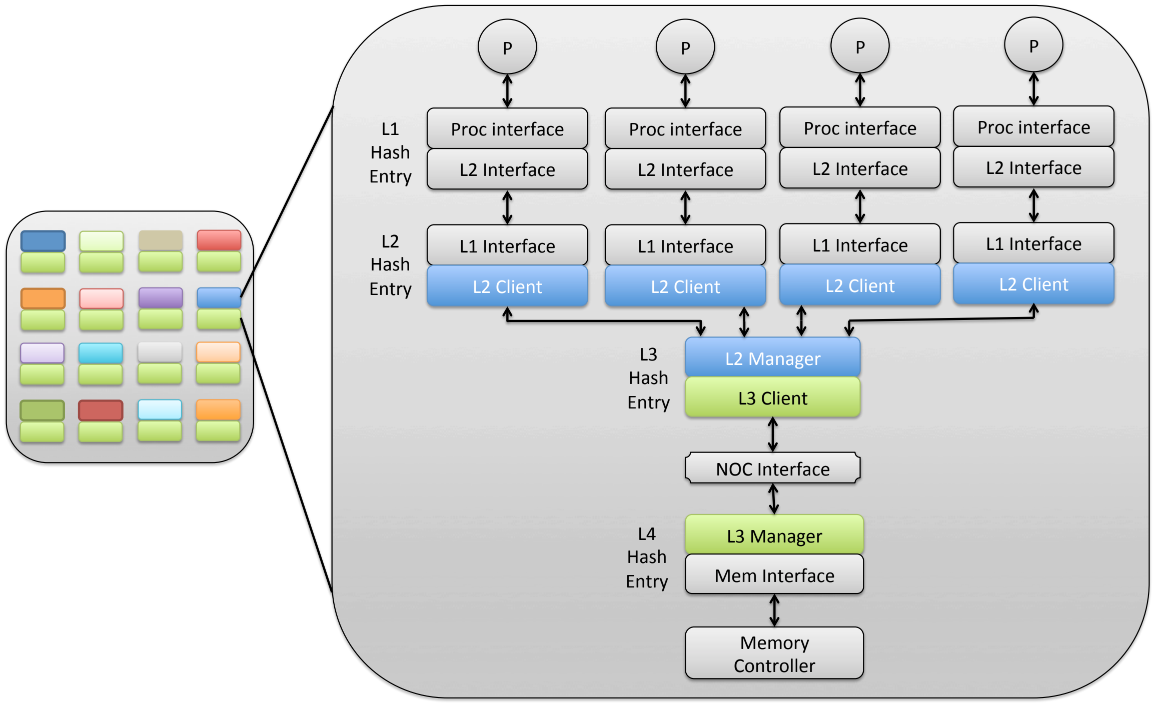


Figure 38 - A 4x4 Torus of Nehalem-like clusters, with 16 independent L2/L3 protocols united through the chip-wide L3/L4 protocol. Coloring shows protocol membership.

As shown, the architecture models a 16-cluster system with 2 levels of coherence. While the trace results presented here could have been computed by hand, due to the highly controlled nature of the evaluation environment (and would potentially make good exam questions), in order to examine the space faster while reducing potential for human error, the traces were evaluated using Manifold, with a latency of 3 cycles for an L1 hit, 10 cycles for an L2 hit, 50 cycles for an L3 hit, and variable >130 cycle latency (dependent on network topology distances and router injection delays) for an L4 hit with sufficient permission. Since only one event is ever active at any time, there is no network contention and therefore L4 hit times are static based on distance from the home L4 cache slice. These numbers for the L1-L3 were chosen based on information in [79] describing the Nehalem architectures cache latency behaviors, and the L4 assumes a

DRAM-cache latency, but these choices are somewhat arbitrary as ‘low, medium and high’ latency would be sufficient descriptors for this kind of evaluation.

The coherence protocols being evaluated are mixings of the MI, MSI and MESI protocols in the lower and upper tier protocols. {MI, MI} is the only instance of a hierarchy using lower-tier MI protocols because the nature of MI operation requires the upgrading of all read requests into write requests. Both the MSI and MESI protocols behave identically to an MI protocol when fed strictly write traffic, which is the case when paired with lower MI protocols, and thus the {MI, MSI} and {MI, MESI} hierarchies are redundant. As a result, these seven protocol pairs represent all nine permutations of MI, MSI and MESI. Additionally, when a lower MESI protocol is interfaced with an upper protocol, in order to support the E state and not default into simple MSI operation, first-time reads (which result in an E instead of an S under MESI) request write permissions through the MCP interface.

The remainder of this sub-section is divided into three parts, each focused on a separate communication pattern, exploring the differences in observed latency while varying the eviction interleavings and eviction cache-depth of those evictions across each hierarchy permutation. In the result tables that follow, cells of interest are highlighted green, yellow and red to indicate significant latency variation, while any other cells of interest are highlighted orange.

7.3.1 Read-Only Communication

Due to the high sharing nature of Read-Only communication, it seems fairly obvious that protocols lacking the ability to support sharers would perform poorly and those that do perform well. The situation, however, is not as straightforward as it seems on the surface, and is reflected in the performance numbers for the traces.

Starting with the local-only communication, we see that all scenarios perform almost equally well. The only note-worthy exception is the last read in the {MI, MI} case, which is a re-read of the previously written data by core 0. For that event, all other lower local-protocols were able to maintain a copy in the L1 cache, but the nature of MI required invalidation to migrate to the other cores. In addition, the MI required additional traffic between the L3 manager and L2 clients before supplying data to make sure it hadn't been modified resulting in an additional 6 cycles of look-up latency, compared to the other protocols which could be supplied clean data directly from the L3 cache.

For the coarse interleaving we see that there still isn't appreciable difference between the protocols until the second round of reads happen. Again, there are small increases or decreases in latency due to the presence/lack of additional confirming invalidations/downgrades in the L1/L2 caches, but one important observation here is that the local MESI protocols (on the right of Table 6) observe the same kinds of minor penalties as {MI, MI} on the first event in a new cluster, and also miss hitting locally in the L1 for the second batch of reads, implying a shared behavior of some kind with {MI, MI}. Otherwise, we also see two of the local MSI protocols gain a bigger benefit on the first read access in the new cluster '9'. Closer observation reveals this is due to the L4 being able to supply clean data without having to incur the additional network latency and L3 lookup penalty the other protocols do.

The similarity between MI and MESI local raises the question, "Why didn't the MESI local protocols get the same benefit as MSI?" Inspection of the coherence traffic to acquire the data reveals that they are behaving much like the MI protocols in the top-tier and with good reason: the E state. From each local protocol's perspective, the first read made within the cluster is a treated as a MESI first-read, which requests write permissions rather than read permission. This means in the upper protocol write traffic is generated, rather than a read sequence, and the S state of the upper protocols is not taken advantage of.

Table 6 - Read-Only local only communication (no evictions)

		Local MI Remote MI	Local MSI Remote MI	Local MSI Remote MSI	Local MSI Remote MESI	Local MESI Remote MI	Local MESI Remote MSI	Local MESI Remote MESI
[0-0]	GetWrite	159	159	159	159	159	159	159
[0-1]	GetRead	56	56	56	56	56	56	56
[0-2]	GetRead	56	50	50	50	50	50	50
[0-3]	GetRead	56	50	50	50	50	50	50
[0-0]	GetRead	56	3	3	3	3	3	3
[0-1]	GetRead	56	3	3	3	3	3	3
[0-2]	GetRead	56	3	3	3	3	3	3
[0-3]	GetRead	56	3	3	3	3	3	3

Table 7 - Read-Only coarse-grained local-remote interleaving (no eviction)

		Local MI Remote MI	Local MSI Remote MI	Local MSI Remote MSI	Local MSI Remote MESI	Local MESI Remote MI	Local MESI Remote MSI	Local MESI Remote MESI
[0-0]	GetWrite	159	159	159	159	159	159	159
[0-1]	GetRead	56	56	56	56	56	56	56
[0-2]	GetRead	56	50	50	50	50	50	50
[8-0]	GetRead	220	220	214	214	220	220	220
[8-1]	GetRead	56	50	50	50	54	54	54
[8-2]	GetRead	56	50	50	50	50	50	50
[9-0]	GetRead	244	244	183	183	244	244	244
[9-1]	GetRead	56	50	50	50	54	54	54
[9-2]	GetRead	56	50	50	50	50	50	50
[0-1]	GetRead	228	228	3	3	228	228	228
[0-2]	GetRead	56	50	3	3	54	54	54
[8-0]	GetRead	220	220	3	3	220	220	220
[8-1]	GetRead	56	50	3	3	54	54	54
[8-2]	GetRead	56	50	3	3	50	50	50
[9-0]	GetRead	244	244	3	3	244	244	244
[9-1]	GetRead	56	50	3	3	54	54	54
[9-2]	GetRead	56	50	3	3	50	50	50

This phenomenon of lower E upgrades really reveals its impact when we look to the remote-only and fine-grain interleaved traces of Table 8 and Table 9. Here we quickly see not only the same benefit from being able to get clean data from the L4 without L3 consultation, but a huge boon to the MSI protocols observing multiple cache hits as each cluster is able to retain a local copy. Meanwhile, the MI-like behavior of the upper-tier protocols, caused by E-state upgrades, are causing only one L3 client to ever hold a valid copy, thus causing a sequence of repeated misses in these inter-cluster cases.

Table 8 - Read-Only remote-only communication (no eviction)

		Local MI Remote MI	Local MSI Remote MI	Local MSI Remote MSI	Local MSI Remote MESI	Local MESI Remote MI	Local MESI Remote MSI	Local MESI Remote MESI
[0-0]	GetWrite	159	159	159	159	159	159	159
[8-0]	GetRead	220	220	220	220	220	220	220
[9-0]	GetRead	244	244	183	183	244	244	244
[10-0]	GetRead	261	261	192	192	261	261	261
[0-0]	GetRead	236	236	3	3	236	236	236
[8-0]	GetRead	220	220	3	3	220	220	220
[9-0]	GetRead	244	244	3	3	244	244	244
[10-0]	GetRead	260	260	3	3	260	260	260

Table 9 - Read-Only fine-grained local-remote interleaving (no eviction)

		Local MI Remote MI	Local MSI Remote MI	Local MSI Remote MSI	Local MSI Remote MESI	Local MESI Remote MI	Local MESI Remote MSI	Local MESI Remote MESI
[0-0]	GetWrite	159	159	159	159	159	159	159
[8-0]	GetRead	220	220	220	220	220	220	220
[9-0]	GetRead	244	244	183	183	244	244	244
[0-1]	GetRead	228	228	50	50	228	228	228
[8-1]	GetRead	220	220	50	50	220	220	220
[9-1]	GetRead	244	244	50	50	244	244	244
[0-2]	GetRead	228	228	50	50	228	228	228
[8-2]	GetRead	220	220	50	50	220	220	220
[9-2]	GetRead	244	244	50	50	244	244	244
[8-0]	GetRead	244	244	3	3	244	244	244
[9-0]	GetRead	244	244	3	3	244	244	244
[0-1]	GetRead	228	228	3	3	228	228	228
[8-1]	GetRead	220	220	3	3	220	220	220
[9-1]	GetRead	244	244	3	3	244	244	244
[0-2]	GetRead	228	228	3	3	228	228	228
[8-2]	GetRead	220	220	3	3	220	220	220
[9-2]	GetRead	244	244	3	3	244	244	244

No appreciable changes in these trends were observed upon activation of L2 eviction traffic injection, outside of L1/L2 latencies increasing to those of an L3 hit. Activating L3 evictions caused all protocols to perform equally poor. It is interesting to note, however, that for the L3 eviction case, though equally poor across all configurations (L4 latency for all traffic), this still represented an improvement in the {MI, MI} and {MESI, *} protocol performance, due to the L4 being able to now provide clean data without L3 indirection, for the remote-only and fine-grained interleaving traces.

7.3.2 Migratory Communication

The performance of migratory communication is more in line with our expectations regarding how well the {MI, MI} protocol performs vs. the other hierarchy combinations. Migratory data follows a read-followed-by-write sequence that rewards more aggressive permission acquisition by allowing the follow-up write to proceed faster. For the local-only case in Table 10, {MI, MI} enjoys the benefit of L2 hits in the follow-up read compared to the other protocols, except the first write in the {MESI, *} cases. This is obviously due to the E state, which is designed to specifically exploit write-after-read behavior. It does, however, lose this benefit for future local events, since the S state is used to satisfy reads when the manager is in the M state. We see the same phenomena in Table 11 as well. This suggests an opportunity for modification to conventional MESI protocol design that could take better advantage of migratory sharing by allowing the manager to make M-to-E transitions on the first read after a write, rather than the current practice of supporting first-time read E promotion only from the manager I state.

Table 10 - Migratory local-only communication (no eviction)

		Local MI Remote MI	Local MSI Remote MI	Local MSI Remote MSI	Local MSI Remote MESI	Local MESI Remote MI	Local MESI Remote MSI	Local MESI Remote MESI
[0-0]	GetRead	159	159	159	159	159	159	159
[0-0]	GetWrite	10	50	159	50	10	10	10
[0-1]	GetRead	56	56	56	56	56	56	56
[0-1]	GetWrite	10	56	56	56	56	56	56
[0-2]	GetRead	56	56	56	56	56	56	56
[0-2]	GetWrite	10	56	56	56	56	56	56
[0-3]	GetRead	56	56	56	56	56	56	56
[0-3]	GetWrite	10	56	56	56	56	56	56

Table 11 - Migratory coarse-grained local-remote interleaving (no evictions)

		Local MI Remote MI	Local MSI Remote MI	Local MSI Remote MSI	Local MSI Remote MESI	Local MESI Remote MI	Local MESI Remote MSI	Local MESI Remote MESI
[0-0]	GetRead	159	159	159	159	159	159	159
[0-0]	GetWrite	10	50	159	50	10	10	10
[0-1]	GetRead	56	56	56	56	56	56	56
[0-1]	GetWrite	10	56	56	56	56	56	56
[0-2]	GetRead	56	56	56	56	56	56	56
[0-2]	GetWrite	10	56	56	56	56	56	56
[8-0]	GetRead	220	220	220	220	220	220	220
[8-0]	GetWrite	10	50	220	220	10	10	10
[8-1]	GetRead	56	56	56	56	56	56	56
[8-1]	GetWrite	10	56	56	56	56	56	56
[8-2]	GetRead	56	56	56	56	56	56	56
[8-2]	GetWrite	10	56	56	56	56	56	56
[9-0]	GetRead	244	244	244	244	244	244	244
[9-0]	GetWrite	10	50	244	244	10	10	10
[9-1]	GetRead	56	56	56	56	56	56	56
[9-1]	GetWrite	10	56	56	56	56	56	56
[9-2]	GetRead	56	56	56	56	56	56	56
[9-2]	GetWrite	10	56	56	56	56	56	56

As expected from the previous discussion, when the sequence is dominated by first-time cluster accesses, as is the case for the sequences of Table 12 and Table 13, both {MI, MI} and all the {MESI, *} protocols perform well by enabling private lower-level caches to immediately satisfy the follow-up write request. Meanwhile, the {MSI, MSI} and {MSI, MESI} variants are pretty consistently left under-privileged, repeatedly having to traverse all the way down to the L4 to acquire the write permission upgrade. The

{MSI, MI} protocol fares slightly better due to the upper-tier MI protocol converting L3-to-L4 read permission requests into write permission requests. This produces L3 hits for the follow-up write in the read-write pairs of migratory communication, since the L3 manager has already been awarded sufficient permission to make forward progress without requiring L4 consultation.

Table 12 - Migratory remote-only communication (no eviction)

		Local MI Remote MI	Local MSI Remote MI	Local MSI Remote MSI	Local MSI Remote MESI	Local MESI Remote MI	Local MESI Remote MSI	Local MESI Remote MESI
[0-0]	GetRead	159	159	159	159	159	159	159
[0-0]	GetWrite	10	50	159	50	10	10	10
[8-0]	GetRead	220	220	220	220	220	220	220
[8-0]	GetWrite	10	50	220	220	10	10	10
[9-0]	GetRead	244	244	244	244	244	244	244
[9-0]	GetWrite	10	50	244	244	10	10	10
[10-0]	GetRead	261	261	261	261	261	261	261
[10-0]	GetWrite	10	50	260	260	10	10	10

Table 13 - Migratory fine-grained local-remote interleaving (no evictions)

		Local MI Remote MI	Local MSI Remote MI	Local MSI Remote MSI	Local MSI Remote MESI	Local MESI Remote MI	Local MESI Remote MSI	Local MESI Remote MESI
[0-0]	GetRead	159	159	159	159	159	159	159
[0-0]	GetWrite	10	50	159	50	10	10	10
[8-0]	GetRead	220	220	220	220	220	220	220
[8-0]	GetWrite	10	50	220	220	10	10	10
[9-0]	GetRead	244	244	244	244	244	244	244
[9-0]	GetWrite	10	50	244	244	10	10	10
[0-1]	GetRead	228	228	228	228	228	228	228
[0-1]	GetWrite	10	50	228	228	10	10	10
[8-1]	GetRead	220	220	220	220	220	220	220
[8-1]	GetWrite	10	50	220	220	10	10	10
[9-1]	GetRead	244	244	244	244	244	244	244
[9-1]	GetWrite	10	50	244	244	10	10	10
[0-2]	GetRead	228	228	228	228	228	228	228
[0-2]	GetWrite	10	50	228	228	10	10	10
[8-2]	GetRead	220	220	220	220	220	220	220
[8-2]	GetWrite	10	50	220	220	10	10	10
[9-2]	GetRead	244	244	244	244	244	244	244
[9-2]	GetWrite	10	50	244	244	10	10	10

While injection of L2 evictions has no bearing on these results for coarse-grained eviction injection, L3 evictions has two less impacts. First, we again see a reduction in indirection costs across all hierarchy configurations, this time visible in the initial read event latencies (~240 to ~180 cycles). Second, the evicting of L3 entries between migratory pairs improves the performance of {MSI, MESI} across all migratory traces due to the upper MESI protocol treating more reads as first-time reads on re-allocation after the evictions. The data in Table 14 present the remote-only traces under L3 eviction injection and can be contrasted against Table 12's results.

Table 14 - Migratory remote-only communication, employing coarse-grain L3 eviction injection

		Local MI Remote MI	Local MSI Remote MI	Local MSI Remote MSI	Local MSI Remote MESI	Local MESI Remote MI	Local MESI Remote MSI	Local MESI Remote MESI
[0-0]	GetRead	164	164	164	164	164	164	164
[0-0]	GetWrite	10	50	159	50	10	10	10
[8-0]	GetRead	175	175	175	175	175	175	175
[8-0]	GetWrite	10	50	175	50	10	10	10
[9-0]	GetRead	183	183	183	183	183	183	183
[9-0]	GetWrite	10	50	183	50	10	10	10
[10-0]	GetRead	191	191	191	191	191	191	191
[10-0]	GetWrite	10	50	191	50	10	10	10

Finally, while less likely to occur in practice, due to the general programming rule-of-thumb to use small critical sections, which in turn creates high temporal locality between the read and write pair of migratory communication, when fine-grained eviction is applied, the performance impact can be felt in the well-performing {MI, MI} and {MESI, *} protocols. Effectively all 10 cycle write permission latencies in the tables of this section, which represent L2 cache hits, increase to 50-cycle latency L3 hits when L2 eviction is applied, shown in Table 15 which can also be contrasted against Table 12's results.

Table 15 - Migratory remote-only communication, employing fine-grain L2 eviction injection

		Local MI Remote MI	Local MSI Remote MI	Local MSI Remote MSI	Local MSI Remote MESI	Local MESI Remote MI	Local MESI Remote MSI	Local MESI Remote MESI
[0-0]	GetRead	159	159	159	159	159	159	159
[0-0]	GetWrite	50	50	159	50	50	50	50
[8-0]	GetRead	214	214	214	214	214	214	214
[8-0]	GetWrite	50	50	214	214	50	50	50
[9-0]	GetRead	238	238	238	238	238	238	238
[9-0]	GetWrite	50	50	238	238	50	50	50
[10-0]	GetRead	254	254	254	254	254	254	254
[10-0]	GetWrite	50	50	254	254	50	50	50

7.3.3 Producer-Consumer Communication

Despite the Read-Only and Migratory communication traces showing clear advantages and disadvantages to certain hierarchy compositions, the producer-consumer pattern does not present the same contrast. For the most commonly observed variant of producer-consumer, that between one pair of nodes, there is no performance variation between any of the hierarchy permutations for all combinations of fine-grained, split and coarse-grained eviction at different eviction depths. In fact, even across the different eviction depths, producer-consumer communication displays a surprising resilience to cache miss effects. This is because producer consumer communication almost never results in a cache hit to any level higher than the L3. This is not unexpected when considering the nature of producer-consumer, since the L3 is the first level of shared caching and each write event, the production portion of the producer-consumer pair, invalidates all shared copies in all private levels.

Table 16 - Producer-Consumer local-only and remote-only results for single producer-consumer pairs

		Local MI Remote MI	Local MSI Remote MI	Local MSI Remote MSI	Local MSI Remote MESI	Local MESI Remote MI	Local MESI Remote MSI	Local MESI Remote MESI
[0-0]	GetWrite	159	159	159	159	159	159	159
[0-1]	GetRead	56	56	56	56	56	56	56
[0-0]	GetWrite	56	56	56	56	56	56	56
[0-1]	GetRead	56	56	56	56	56	56	56

		Local MI Remote MI	Local MSI Remote MI	Local MSI Remote MSI	Local MSI Remote MESI	Local MESI Remote MI	Local MESI Remote MSI	Local MESI Remote MESI
[0-0]	GetWrite	159	159	159	159	159	159	159
[8-0]	GetRead	220	220	220	220	220	220	220
[0-0]	GetWrite	220	220	220	220	220	220	220
[8-0]	GetRead	220	220	220	220	220	220	220

Things for production-consumer only become slightly more interesting when considering large sharing degrees like those observed in water-spatial. While nearly identical performance is observed across the configurations, interleaved access does yield a few note-worthy performance differences. Most obvious in Table 17, we see that the nature of the protocols exhibit performance trends similar to those we already observed for read-only traffic. In fact, the first half of Table 17 is identical to the first half of Table 6, and for the same reasons (MI dominant behavior removing S state usage). This makes sense, however, since Read-Only is similar to a degenerative case of producer-consumer that only produces once. Recall Read-Only, however, will re-consume the same value from the initial write (which could be in the distant past, depending on the application) vs. producer consumer's single-consumption of a more temporally recent write. This difference in communication patterns accounts for the difference in trends in the second half of the tables.

In any case, the latency trend repeats itself for producer-consumer, with the second production for the {MSI, MSI} and {MSI, MESI} compositions having a slightly higher latency compared to the other hierarchies. This is due to the lower tier invalidation of clusters 8 and 9, required from the S state in the upper-tier, compared to only the cluster 9 invalidation of the upper M state for the other MI and MESI hierarchies. These invalidations happen in parallel, however, hence only a 13-cycle difference (which can be attributed to network delay differences). To complete the picture for high-degree producer-consumer results, data for coarse-grained interleaving is shown in Table 18, which also exhibits this same cost difference for the MSI protocols. Table 18 also shows the same few-cycle L1/L2 downgrade/invalidation penalties explained in Section 7.3.1.

Table 17 - Producer-Consumer fine-grained local-remote interleaving

		Local MI Remote MI	Local MSI Remote MI	Local MSI Remote MSI	Local MSI Remote MESI	Local MESI Remote MI	Local MESI Remote MSI	Local MESI Remote MESI
[0-0]	GetWrite	159	159	159	159	159	159	159
[8-0]	GetRead	220	220	220	220	220	220	220
[9-0]	GetRead	244	244	183	183	244	244	244
[0-1]	GetRead	228	228	50	50	228	228	228
[8-1]	GetRead	220	220	50	50	220	220	220
[9-1]	GetRead	244	244	50	50	244	244	244
[0-2]	GetRead	228	228	50	50	228	228	228
[8-2]	GetRead	220	220	50	50	220	220	220
[9-2]	GetRead	244	244	50	50	244	244	244
[0-0]	GetWrite	228	228	241	241	228	228	228
[8-0]	GetRead	220	220	220	220	220	220	220
[9-0]	GetRead	244	244	183	183	244	244	244
[0-1]	GetRead	228	228	50	50	228	228	228
[8-1]	GetRead	220	220	50	50	220	220	220
[9-1]	GetRead	244	244	50	50	244	244	244
[0-2]	GetRead	228	228	50	50	228	228	228
[8-2]	GetRead	220	220	50	50	220	220	220
[9-2]	GetRead	244	244	50	50	244	244	244

Table 18 - Producer-Consumer coarse-grained local-remote interleaving

		Local MI Remote MI	Local MSI Remote MI	Local MSI Remote MSI	Local MSI Remote MESI	Local MESI Remote MI	Local MESI Remote MSI	Local MESI Remote MESI
[0-0]	GetWrite	159	159	159	159	159	159	159
[0-1]	GetRead	56	56	56	56	56	56	56
[0-2]	GetRead	56	50	50	50	50	50	50
[8-0]	GetRead	220	220	214	214	220	220	220
[8-1]	GetRead	56	50	50	50	54	54	54
[8-2]	GetRead	56	50	50	50	50	50	50
[9-0]	GetRead	244	244	183	183	244	244	244
[9-1]	GetRead	56	50	50	50	54	54	54
[9-2]	GetRead	56	50	50	50	50	50	50
[0-0]	GetWrite	228	228	241	241	228	228	228
[0-1]	GetRead	56	56	56	56	56	56	56
[0-2]	GetRead	56	50	50	50	50	50	50
[8-0]	GetRead	220	220	214	214	220	220	220
[8-1]	GetRead	56	50	50	50	54	54	54
[8-2]	GetRead	56	50	50	50	50	50	50
[9-0]	GetRead	244	244	183	183	244	244	244
[9-1]	GetRead	56	50	50	50	54	54	54
[9-2]	GetRead	56	50	50	50	50	50	50

7.4 Concluding Thoughts on Heterogeneity

From these results we can make several inferences about coherence design in heterogeneous hierarchical protocols. First, there are clearly two classes of heterogeneous protocol when dealing with MI, MSI and MESI protocols: those which are M state dominated ({MI, MI} and {MESI, *}), which perform well for migratory communication but poorly for Read-Only sequences, and those which are S state dominated protocols ({MSI, MSI} and {MSI, MESI}) which exhibit the inverse trend. From Section 7.3.3, it is also clear that the design choices regarding heterogeneity have little impact on Producer-Consumer communication. This suggests that techniques, such as use set prediction [100] may be a better avenue for improving performance when an application's communication is known to be producer-consumer dominated. This is especially true for the common single-consumer case, where a more active approach has much higher potential for improvement compared to most of our current coherence enhancements which tend to be lazy-acquisition centric, favoring less timely 1st-time responses over speculative transactions which could benefit 1st-time events, but consume extra bandwidth and power when incorrectly speculated.

At a higher level of abstraction, we also notice that the trends are relatively dominated by the lower-tier protocol choices, only diverging from this when the upper-tier protocol is more restrictive than the lower protocols (as in the case for {MSI, MI}). Related to this, the MESI protocol's E-state requires some additional design consideration since this could just be an artifact of the static translation mechanism used by MESI requests when crossing MCP tier boundaries. An extension to the Manager-Client Pairing interface, for example, which supports functionality like 'GetFirstReadP' for conveying E state-ness could help in making more intelligent choices than always defaulting first-time reads to GetWriteP traffic, finding a middle ground between that implemented in this evaluation and the static method proposed by Suh et al. [61] of shutting down the E state entirely. Taking this idea further, an interesting avenue of research would be to provide support for dynamic upgrades, where higher tiers actually make the decision as to whether the requests from a MESI tier are awarded E or S state based on other contextual information or predictors.

Finally, it is worth recognizing that at several data points in the tables of this section there were penalties associated with demand/invalidation traffic that disappear once eviction traffic was introduced. This supports the claims made in [101], which suggest that eager eviction of data can improve performance. While many of these costs seem small (on the order of 4 – 6 cycles for 56 cycle L3 accesses), an early eviction technique may merit consideration in the top-tier protocol where the cost associated with demand/invalidation, including cache access and the network latency, can account for as much as 61 of the 244 observed cycles.

Through this data analysis, it is clear that in order to make meaningful innovations for future hierarchical architectures, coherence architects need to understand not only coherence design and hierarchical interaction, but also require an in-depth understanding of the communication trends of our applications. Specifically, understanding how these communication sequences will play out in terms of coherence traffic, how the tier protocols will interact with each other under certain conditions, and what modifications matter is necessary to make the right design choices that yield the biggest performance impact.

CHAPTER 8 - CONCLUSION

This dissertation has established that as the trend of increasing core count reaches scales beyond 4 or 8 cores on a die, hierarchical coherence becomes the solution of choice for managing many of the problems associated with large-scale coherence integration. Much previous pioneering work has already shown that the many advantages hierarchical coherence imparts are worth the design complexity involved, but until now there has been no generalized way to manage this design complexity. This is especially true in the case of heterogeneous hierarchical coherence, where integration of disjoint protocols in the past resort to ad-hoc solutions, applying some complex glue logic tightly coupled to the components being connected, coming at high cost in terms of complexity and especially verification.

Manager-Client Pairing leads the way for enabling rapid design and evaluation of hierarchical coherence by providing a generic interface definition for modular merging of component protocols into a coherent whole. Chapter 5 outlines the details of this interface definition and demonstrates its strength for rapid design evaluation. Through defining general interface functions and establishing a permissions-checking algorithm, multiple protocols can be united in the effort of maintaining coherence across a diverse, heterogeneous system while being transparent to one another's existence. This encapsulation of protocols provides the basis of proof presented in Chapter 6, outlining how formal verification efforts can be improved considerably through Manager-Client Pairing via encapsulation symmetry.

The final contribution of this dissertation is Chapter 7, which looks at several popular communication patterns in conjunction with heterogeneous hierarchy design. This in-depth analysis explores how these different communication sequences interact with different hierarchy choices, demonstrating the implications of lower and upper protocol selection and revealing how unexpected interactions among tiers can make the component protocols behave in unexpected ways. Most notably, component MESI protocols exhibit much more M/E dominated behavior than would otherwise have been expected, due to the lack of a global perspective regarding whether reads are actually the first read in the system. The breakdown of this basic principle behind the MESI optimization causes it to become over-active and hurt performance in several cases, much like an allergic response in an otherwise helpful immune system. This kind of insight demonstrates the care that needs to be taken when composing heterogeneous hierarchies, and why a technique like Manager-Client Pairing, which allows rapid redesign without worrying over all the ad-hoc additions and tight coupling of protocols is so powerful.

In summary, Manager-Client Pairing provides enough ease-of-use and flexibility to avoid ever being locked into a single poor design by side-stepping the high sunken-costs normally associated with coherence integration, and delivers this flexibility while providing verification composition guarantees.

Appendix A – MCP Actions

Lower Tier Manager to Upper Paired Client Permission Query

HaveReadP	Return true if paired Client has read permission
HaveWriteP	Return true if paired Client has write permission
HaveEvictP	Return true if paired Client can be safely evicted

Lower Tier Manager to Upper Paired Client Permission *Get*

GetReadD	Paired Client begins data and read permission acquisition sequence within it's native coherence realm. L1/Lower Manager expects GetReadDAck upon completion.
GetExclusiveD	Paired Client begins data and write permission acquisition sequence within it's native coherence realm. L1/Lower Manager expects GetExclusiveDAck upon completion.
GetExclusive	Paired Client begins write permission acquisition sequence within it's native coherence realm. L1/Lower Manager expects GetExclusiveAck or GetExclusiveDAck upon completion.
	Used when data is already available in L1/Lower Manager (HaveData == true) and only a permission upgrade is required.
	May be satisfied by a GetExclusiveDAck if upper tier protocol demands a downgrade while GetExclusive is in flight, causing HaveData to become false.
GetEvict	Paired Client begins eviction sequence within it's coherence realm. L1/Lower Manager expects GetEvictAck upon completion.
	Used when block ownership or most recent dirty version resides in L1/Lower Manager's realm.
	Needs to include data payload when data being evicted is dirty.

Upper Tier Client to Lower Paired Manager Permission Request Reply

GetReadDAck	Response by paired Client to complete previous GetReadD request. Supplies data packet and signifies paired Client (and thus lower Manager's realm) now has read permissions.
GetExclusiveDAck	Response by paired Client to complete previous GetExclusive/GetExclusiveD request. Supplies data packet and signifies paired Client (and thus lower Manager's realm) now has write permissions.
GetExclusiveAck	Response by paired Client to complete previous GetExclusive request. Signifies paired Client (and thus lower Manager's realm) now has write permissions.
GetEvictAck	Response by paired Client to complete previous GetEvict request. Signifies paired Client has become invalid. Therefore, Manager's realm can safely eliminate all local copies of the block.

Upper Tier Client to Lower Paired Manager *Demand*

Supply	Demand data supply from lower tier's paired Manager or L1. No additional actions required by lower tier.
	Used for data forwarding to satisfy remote read when Manager-Client pair permission levels already match.
Invalidate	Demand lower realm to forfeit write permissions and read permissions, invalidating all local copies of data.
	Used to satisfy remote write request which requires exclusive rights when remote realm already has a copy of the data.
SupplyDowngrade	Demand Data from lower realm's paired Manager. Additionally, lower realm must forfeit write permissions but can retain read permissions and data.
	Used for data forwarding to satisfy remote read when upper-tier paired Client state is forfeiting exclusive/write permissions.
SupplyInvalidate	Demand Data from lower realm's paired manager. Additionally, lower realm must forfeit write permissions AND read permissions, invalidating all local copies of data.
	Used for data forwarding to satisfy remote exclusive/write request when remote realm expects data supplied from this realm.

Lower Tier Manager to Upper Paired Client Demand Reply

SupplyAck	Response by paired Manager to complete previous Supply demand. Supplies data packet.
InvalidateAck	Response by paired Manager to complete previous Invalidate demand. Signifies realm invalidation has completed.
SupplyDowngradeAck	Response by paired Manager to complete previous SupplyDowngrade demand. Supplies data packet and signifies realm downgrade has completed.
SupplyInvalidateAck	Response by paired Manager to complete previous SupplyInvalidate demand. Supplies data packet and signifies realm invalidation has completed.

REFERENCES

[1] L. A. Barroso, "The Price of Performance," *Queue,* vol. 3, pp. 48-53, 2005.

[2] *ITRS. International technology roadmap for semiconductors, 2010 update, 2011.* Available: http://www.itrs.net.

[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*: Morgan Kaufmann Publishers Inc., 2006.

[4] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: a tutorial," *Computer,* vol. 29, pp. 66-76, 1996.

[5] D. B. Glasco, "Design and analysis of update-based cache coherence protocols for scabable shared-memory multiprocessors.," Dept. of Electrical Engineering and Computer Science, Stanford University, Stanford, California, 1995.

[6] P. Gratz, K. Changkyu, R. McDonald, S. W. Keckler, and D. Burger, "Implementation and Evaluation of On-Chip Network Architectures," in *Computer Design, 2006. ICCD 2006. International Conference on*, 2006, pp. 477-484.

[7] T. Michael Bedford. (2002) The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. 25-35. Available: http://doi.ieeecomputersociety.org/10.1109/MM.2002.997877

[8] J. Balfour and W. J. Dally, "Design tradeoffs for tiled CMP on-chip networks," presented at the Proceedings of the 20th annual international conference on Supercomputing, Cairns, Queensland, Australia, 2006.

[9] R. Ho, K. W. Mai, and M. A. Horowitz, "The future of wires," *Proceedings of the IEEE,* vol. 89, pp. 490-504, 2001.

[10] R. Ho, M. Ken, and M. Horowitz, "Managing wire scaling: a circuit perspective," in *Interconnect Technology Conference, 2003. Proceedings of the IEEE 2003 International*, 2003, pp. 177-179.

[11] D. Sanchez, G. Michelogiannakis, and C. Kozyrakis, "An analysis of on-chip interconnection networks for large-scale chip multiprocessors," *ACM Trans. Archit. Code Optim.,* vol. 7, pp. 1-28, 2010.

[12] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato, "A Two-Level Directory Architecture for Highly Scalable cc-NUMA Multiprocessors," *IEEE Trans. Parallel Distrib. Syst.,* vol. 16, pp. 67-79, 2005.

[13] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo directory: A scalable directory for many-core systems," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, 2011, pp. 169-180.

[14] J. H. Kelm, M. R. Johnson, S. S. Lumettta, and S. J. Patel, "WAYPOINT: scaling coherence to thousand-core architectures," presented at the Proceedings of the 19th international conference on Parallel architectures and compilation techniques, Vienna, Austria, 2010.

[15] A. Ros, M. E. Acacio, Jose, and M. Garcia, "A scalable organization for distributed directories," *J. Syst. Archit.,* vol. 56, pp. 77-87, 2010.

[16] D. Sanchez and C. Kozyrakis, "SCD: A scalable coherence directory with flexible sharer set encoding," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, 2012, pp. 1-12.

[17] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos, "A tagless coherence directory," presented at the Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, New York, New York, 2009.

[18] N. D. Enright Jerger, L.-S. Peh, and M. H. Lipasti, "Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence," in *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, 2008, pp. 35-46.

[19] N. Barrow-Williams, C. Fensch, and S. Moore, "Proximity coherence for chip multiprocessors," presented at the Proceedings of the 19th international conference on Parallel architectures and compilation techniques, Vienna, Austria, 2010.

[20] M. M. K. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," *Commun. ACM,* vol. 55, pp. 78-89, 2012.

[21] Arvind, N. Dave, and M. Katelman, "Getting Formal Verification into Design Flow," presented at the Proceedings of the 15th international symposium on Formal Methods, Turku, Finland, 2008.

[22] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol verification as a hardware design aid," in *Computer Design: VLSI in Computers and Processors, 1992. ICCD '92. Proceedings., IEEE 1992 International Conference on*, 1992, pp. 522-525.

[23] G. J. Holzmann, "Algorithms for automated protocol validation," *AT&T technical journal,* vol. 69, pp. 32-44, 1990.

[24] C. N. Ip and D. L. Dill, "Better verification through symmetry," *Form. Methods Syst. Des.,* vol. 9, pp. 41-75, 1996.

[25] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano*, et al.*, "Piranha: a scalable architecture based on single-chip multiprocessing," presented at the Proceedings of the 27th annual international symposium on Computer architecture, Vancouver, British Columbia, Canada, 2000.

[26] G. Gostin, J.-F. Collard, and K. Collins, "The architecture of the HP Superdome shared-memory multiprocessor," presented at the Proceedings of the 19th annual international conference on Supercomputing, Cambridge, Massachusetts, 2005.

[27] E. Hagersten and M. Koster, "WildFire: A Scalable Path for SMPs," presented at the Proceedings of the 5th International Symposium on High Performance Computer Architecture, 1999.

[28] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo*, et al.*, "The Stanford FLASH multiprocessor," presented at the Proceedings of the 21st annual international symposium on Computer architecture, Chicago, Illinois, United States, 1994.

[29] J. Laudon and D. Lenoski, "The SGI Origin: a ccNUMA highly scalable server," *SIGARCH Comput. Archit. News,* vol. 25, pp. 241-251, 1997.

[30] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the DASH multiprocessor," *SIGARCH Comput. Archit. News,* vol. 18, pp. 148-159, 1990.

[31] I. Singh, A. Shriraman, W. W. Fung, M. O'Connor, and T. M. Aamodt, "Cache Coherence for GPU Architectures," 2013.

[32] F. H. v. Eemeren, "The fallacies of composition and division," R. Grootendorst, Ed., ed. JFAK. Essays Dedicated to Johan van Benthem on the Occasion of his 50th Birthday: Amsterdam University Press, 1999.

[33] D. A. Wood, G. A. Gibson, and R. H. Katz, "Verifying a Multiprocessor Cache Controller Using Random Test Generation," *IEEE Des. Test,* vol. 7, pp. 13-25, 1990.

[34] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," presented at the Proceedings of the 11th annual international symposium on Computer architecture, 1984.

[35] P. Sweazey and A. J. Smith, "A class of compatible cache consistency protocols and their support by the IEEE futurebus," in *ACM SIGARCH Computer Architecture News*, 1986, pp. 414-423.

[36] H. H. Hum and J. R. Goodman, "Forward state for use in cache coherency in a multiprocessor system," ed: Google Patents, 2005.

[37] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen*, et al.*, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Comput. Archit. News,* vol. 33, pp. 92-99, 2005.

[38] P. Gratz, B. Grot, and S. W. Keckler, "Regional congestion awareness for load balance in networks-on-chip," in *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, 2008, pp. 203-214.

[39] A. L. Meng Zhang, Daniel Sorin, "Fractal Coherence: Scalably Verifiable Cache Coherence," presented at the International Symposium on Microarchitecture, Atlanta, Georgia, 2010.

[40] M. M. K. Martin, M. D. Hill, and D. A. Wood, "Token coherence: decoupling performance and correctness," presented at the Proceedings of the 30th annual international symposium on Computer architecture, San Diego, California, 2003.

[41] M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M. K. Martin, and D. A. Wood, "Improving Multiple-CMP Systems Using Token Coherence," presented at the Proceedings of the 11th International Symposium on High-Performance Computer Architecture, 2005.

[42] D. Vantrease, M. H. Lipasti, and N. Binkert, "Atomic Coherence: Leveraging nanophotonics to build race-free cache coherence protocols," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, 2011, pp. 132-143.

[43] L. M. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *Computers, IEEE Transactions on,* vol. 100, pp. 1112-1118, 1978.

[44] J. A. W. Wilson, "Hierarchical cache/bus architecture for shared memory multiprocessors," presented at the Proceedings of the 14th annual international symposium on Computer architecture, Pittsburgh, Pennsylvania, United States, 1987.

[45] D. A. Wallach, "PHD: A Hierarchical Cache Coherent Protocol," Master of Science, Electrical Engineering and Computer Science, MIT, 1990.

[46] S. Haridi and E. Hagersten, "The Cache Coherence Protocol of the Data Diffusion Machine," presented at the Proceedings of the Parallel Architectures and Languages Europe, Volume I: Parallel Architectures, 1989.

[47] B. Falsafi and D. A. Wood, "Reactive NUMA: a design for unifying S-COMA and CC-NUMA," *ACM SIGARCH Computer Architecture News,* vol. 25, pp. 229-240, 1997.

[48] D. Chaiken, J. Kubiatowicz, and A. Agarwal, *LimitLESS directories: A scalable cache coherence scheme* vol. 26: ACM, 1991.

[49] A. Gupta, W.-D. Weber, and T. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," 1990.

[50] H. Sharangpani and H. Arora, "Itanium processor microarchitecture," *Micro, IEEE,* vol. 20, pp. 24-43, 2000.

[51] R. T. Simoni, "Cache coherence directories for scalable multiprocessors," to the Department of Electrical Engineering.Stanford University, 1992.

[52] D. V. James, A. T. Laundrie, S. Gjessing, and G. S. Sohi, "Distributed-directory scheme: Scalable coherent interface," *Computer,* vol. 23, pp. 74-77, 1990.

[53] C. S. Ballapuram, A. Sharif, and H.-H. S. Lee, "Exploiting access semantics and program behavior to reduce snoop power in chip multiprocessors," in *ACM Sigplan Notices*, 2008, pp. 60-69.

[54] B. Cuesta, A. Ros, M. E. Gomez, A. Robles, and J. Duato, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, 2011, pp. 93-103.

[55] S. Burckhardt, R. Alur, and M. M. K. Martin, "Verifying Safety of a Token Coherence Implementation by Parametric Compositional Refinement," in *VMCAI*, 2005.

[56] M. R. Marty, "Cache Coherence Techniques for Multicore Processors," Doctor of Philosophy, Computer Science, University of Wisconsin, 2008.

[57] T. Suh, D. M. Blough, and H.-H. S. Lee, "Supporting Cache Coherence in Heterogeneous Multiprocessor Systems," presented at the Proceedings of the conference on Design, automation and test in Europe - Volume 2, 2004.

[58] T. Suh, H. H. S. Lee, and D. M. Blough, "Integrating cache coherence protocols for heterogeneous multiprocessor systems. 1," *Micro, IEEE,* vol. 24, pp. 33-41, 2004.

[59] T. Suh, H. H. S. Lee, and D. M. Blough, "Integrating cache coherence protocols for heterogeneous multiprocessor system. Part 2," *Micro, IEEE,* vol. 24, pp. 70-78, 2004.

[60] S. Taeweon, "Integration and Evaluation of Cache Coherence Protocols for Multiprocessor SoCs," 2006.

[61] S. Taeweon, K. Daehyun, and H. H. S. Lee, "Cache coherence support for non-shared bus architecture on heterogeneous MPSoCs," in *Design Automation Conference, 2005. Proceedings. 42nd*, 2005, pp. 553-558.

[62] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel, "Cohesion: a hybrid memory model for accelerators," presented at the Proceedings of the 37th annual international symposium on Computer architecture, Saint-Malo, France, 2010.

[63] E. Ladan-Mozes and C. E. Leiserson, "A consistency architecture for hierarchical shared caches," presented at the Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures, Munich, Germany, 2008.

[64] F. Pong and M. Dubois, "Verification techniques for cache coherence protocols," *ACM Comput. Surv.,* vol. 29, pp. 82-126, 1997.

[65] M. R. Marty and M. D. Hill, "Virtual hierarchies to support server consolidation," presented at the Proceedings of the 34th annual international symposium on Computer architecture, San Diego, California, USA, 2007.

[66] B. F. Romanescu, A. R. Lebeck, D. J. Sorin, and A. Bracy, "UNified Instruction/Translation/Data (UNITD) coherence: One protocol to rule them all," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, 2010, pp. 1-12.

[67] K. Aisopos and L.-S. Peh, "A systematic methodology to develop resilient cache coherence protocols," presented at the Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, Porto Alegre, Brazil, 2011.

[68] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze*, et al.* (Jan. 2005). *SESC Simulator*. Available: http://sesc.sourceforge.net.

[69] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney*, et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM SIGPLAN Notices*, 2005, pp. 190-200.

[70] D. Burger and T. M. Austin, "The SimpleScalar tool set, version 2.0," *ACM SIGARCH Computer Architecture News,* vol. 25, pp. 13-25, 1997.

[71] M. T. Yourst, "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator," in *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, 2007, pp. 23-34.

[72] P. D. Bryan, J. A. Poovey, J. G. Beu, and T. M. Conte, "Accelerating Multi-threaded Application Simulation Through Barrier-Interval Time-Parallelism," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, 2012, pp. 117-126.

[73] W. J. Dally, "Virtual-channel flow control," *Parallel and Distributed Systems, IEEE Transactions on,* vol. 3, pp. 194-205, 1992.

[74] A. Kumar, L.-S. Peh, P. Kundu, and N. K. Jha, "Express virtual channels: towards the ideal interconnection fabric," in *ACM SIGARCH Computer Architecture News*, 2007, pp. 150-161.

[75] W. J. Dally, "Express cubes: improving the performance of k-ary n-cube interconnection networks," *Computers, IEEE Transactions on,* vol. 40, pp. 1016-1023, 1991.

[76] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob, "DRAMsim: a memory system simulator," *ACM SIGARCH Computer Architecture News,* vol. 33, pp. 100-107, 2005.

[77] (2013). *Manifold*. Available: manifold.gatech.edu

[78] J. Wang, J. Beu, S. Yalamanchili, and T. Conte, "Designing Configurable, Modifiable And Reusable Components For Simulation of Multicore Systems."

[79] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System," in *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, 2009, pp. 261-270.

[80] K. M. Chandy and J. Misra, "Parallel program design," 1989.

[81] D. Dill, "A Retrospective on Mur ϕ," *25 Years of Model Checking,* pp. 77-88, 2008.

[82] E. M. Clarke and J. M. Wing, "Formal methods: state of the art and future directions," *ACM Comput. Surv.,* vol. 28, pp. 626-643, 1996.

[83] K. L. McMillan, "Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking," presented at the Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods, 2001.

[84] S. Park and D. L. Dill, "Verification of FLASH cache coherence protocol by aggregation of distributed transactions," presented at the Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures, Padua, Italy, 1996.

[85] U. Stern and D. L. Dill, "Improved probabilistic verification by hash compaction," presented at the Proceedings of the IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods, 1995.

[86] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," *SIGARCH Comput. Archit. News,* vol. 23, pp. 24-36, 1995.

[87] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu*, et al.*, "X10: an object-oriented approach to non-uniform cluster computing," in *ACM SIGPLAN Notices*, 2005, pp. 519-538.

[88] P. Loewenstein and D. L. Dill, "Verification of a Multiprocessor Cache Protocol Using Simulation Relations and Higher-Order Logic," presented at the Proceedings of the 2nd International Workshop on Computer Aided Verification, 1991.

[89] F. Pong and M. Dubois, "A New Approach for the Verification of Cache Coherence Protocols," *IEEE Trans. Parallel Distrib. Syst.,* vol. 6, pp. 773-787, 1995.

[90] U. Stern and D. L. Dill, "A new scheme for memory-efficient probabilistic verification," presented at the IFIP TC6/ 6.1 international conference on formal description techniques IX/protocol specification, testing and verification XVI on Formal description techniques IX : theory, application and tools: theory, application and tools, Kaiserslautern, Germany, 1996.

[91] T. G. Mattson, B. A. Sanders, and B. Massingill, *Patterns for parallel programming*: Addison-Wesley Professional, 2005.

[92] J. A. Poovey, B. P. Railing, and T. M. Conte, "Parallel pattern detection for architectural improvements," in *Proceedings of the 3rd USENIX conference on Hot topic in parallelism*, 2011, pp. 12-12.

[93] C. Bienia and K. Li, "Characteristics of Workloads Using the Pipeline Programming Model," in *Proceedings of the 3rd Workshop on Emerging Applications and Many-core Architecture*, 2010.

[94] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," presented at the Proceedings of the 17th international conference on Parallel architectures and compilation techniques, Toronto, Ontario, Canada, 2008.

[95] M. Bhadauria, V. M. Weaver, and S. A. McKee, "Understanding PARSEC Performance on Contemporary CMPs," in *Proceedings of the 2009 International Symposium on Workload Characterization*, 2009.

[96] C. Bienia, *Benchmarking modern multiprocessors*: Princeton University, 2011.

[97] N. Barrow-Williams, C. Fensch, and S. Moore, "A Communication Characterization of SPLASH-2 and PARSEC," in *Proceedings of the 2009 International Symposium on Workload Characterization*, 2009.

[98] J. P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford parallel applications for shared-memory," *ACM SIGARCH Computer Architecture News,* vol. 20, pp. 5-44, 1992.

[99] W. J. Bolosky and M. L. Scott, "False sharing and its effect on shared memory performance," in *Proc., Fourth Symp. on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, 1993.

[100] M. M. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood, "Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, 2003, pp. 206-217.

[101] H.-H. S. Lee, G. S. Tyson, and M. K. Farrens, "Eager writeback - a technique for improving bandwidth utilization," presented at the Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture, Monterey, California, USA, 2000.