Manager-Client Pairing: A Framework for Implementing Coherence Hierarchies

Jesse G. Beu
Georgia Institute of Technology
Atlanta, GA USA
Jesse.Beu@gmail.com

Michael C. Rosier
Apple Inc.
Cupertino, CA USA
Chad.Rosier@gmail.com

Thomas M. Conte
Georgia Institute of Technology
Atlanta, GA USA
Tom@conte.us

Abstract

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.

The difficulty with hierarchical coherence protocol design is 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 would affect the system. In this work, we present 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. Using MCP, we then implement a variety of hierarchical coherence protocol configurations for a 256-core system comprised of 4 64-core manycores, and provide insights into what impact different hierarchy depth and width choices can have on system performance.

Categories and Subject Descriptors

B.3.2 [Hardware]: Memory Structures—Cache Memories, Shared Memory; C.1.4 [Computer Systems Organization]: Processor Architectures—Parallel Architectures

Keywords

Cache Coherence, Coherence Hierarchies, Manycore, Memory Hierarchies, Multicore

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]. 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 with 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 [2-7]. 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 [8-12]. 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 providing 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 [14, 15]. 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.

The combined result of current practices is an abundance of large, complex, inflexible and highly-specialized coherence protocols, especially where hierarchies are employed [8-15]. In this work 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-

MICRO 44, December 3–7, 2011, Porto Alegre, Brazil.

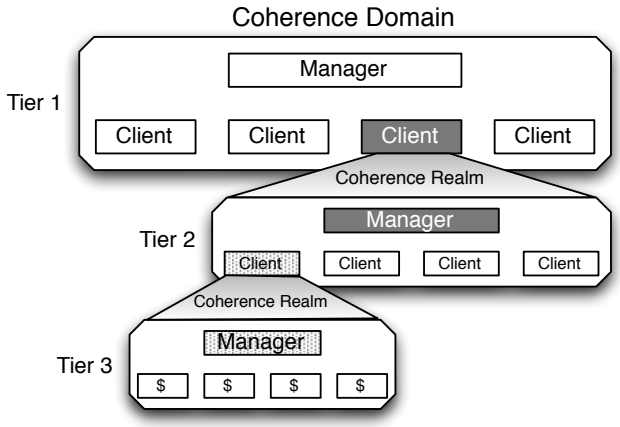


Figure 1 – Coherence hierarchy labeled with MCP terminology

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 problem of 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 1) 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 paper 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 paper is structured as follows. First, Section 2 will discuss previous work in coherence hierarchy design and highlight how MCP differs from these. In Section 3, the base functions of cache coherence are presented based on a definition of client and manager responsibilities. Section 4 describes base function use in the MCP hierarchical permissions algorithm through examples. Section 5 then demonstrates the power and flexibility of MCP by providing new insights through experiments that evaluate the impact of different hierarchy design choices. The paper ends with concluding remarks.

2. Related Work

MCP eases hierarchical coherence protocol design through composition, distributing the coherence responsibility throughout the hierarchy's tiers via encapsulation. The earliest reference to coherence distribution hierarchies is by Wilson [27]. 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. Wallach's master's thesis [28] 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 subtree, and write invalidations only include the smallest subtree that encompasses all sharers, thus reducing traffic.

With the recent resurgence of interest in multicore, Marty, et al. [14] described a solution to extend coherence across a multi-CMP through the use of token coherence as an additional decoupled connecting layer. Marty and Hill [20] later exploit the coherence decoupling provided by hierarchies to turn *coherence realms* into *coherence domains* to enable virtual-machine hierarchies on a manycore substrate.

Recently Ladan-Mozes and Leiserson [18] propose Hierarchical Cache Consistency (HCC), a deadlock-free, tree-based coherence protocol that ensures forward progress in a fat-tree network. They do so by enumerating invariant properties that ensure all children in the tree are coherent with parents, forming the basis of their distributed coherence mechanism. The most closely related work, due to the recursive nature of the interfaces we proposed in MCP, is Fractal Coherence [16]. In Fractal Coherence, Zhang et al. propose a tree-based coherence protocol, but with the intention of simplifying coherence verification through perfect self-similarity. By designing a fractal based hierarchical coherence protocol, where children are coherent with their parents, the entire coherence hierarchy can be validated through the validation of only the kernel coherence protocol. MCP is more general than Fractal Coherence, and our focus in this work is on easing integration of layers, design space evaluation and design reasoning (compared to ad-hoc approaches), as opposed to Fractal's emphasis on verification.

Despite having different goals, Fractal Coherence, HCC and MCP all share a similarity with respect to the permission inclusion property discussed by the authors of HCC in [18]. Parent/upper-tier state in many respects can be viewed as a summary of the child/lower-tiers, having to have permissions at least as strong as the lower children/tiers'. However, there is a very important distinguishing feature that sets MCP apart; Both Fractal Coherence and HCC use protocol implementation-specific details to ensure this property is enforced between parents and children, whereas tiers in MCP need not have any understanding of each

other's states as long as each protocol support the permission queries interface discussed in Section 3. Therefore, despite differing goals, both Fractal Coherence and HCC are compatible with MCP through the addition of permission query support.

It is worth noting that while not a necessary requirement, both HCC and Fractal Coherence only implement uniform MSI protocols. More importantly, each assumes a network topology (tree-based) that matches the organization of the coherence mechanism, that in turn can limit portability and imposes limitations (e.g., requiring strict network ordering). Each also only evaluates binary tree implementations, which have the disadvantage of increasing the height of the hierarchy logarithmically with node count. This height increase results in more hardware overhead for tag structures as well as indirection delays as compared to flatter hierarchies (as we demonstrate in Section 5). Additionally, neither can provide support for coherence hierarchy heterogeneity (i.e., the use of different protocol components to create a globally coherent system), whereas MCP does.

3. 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 [13]. 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.

3.1 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.[1]

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 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 with respect to 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.

[1] 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.

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	

3.2 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 2, 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.

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

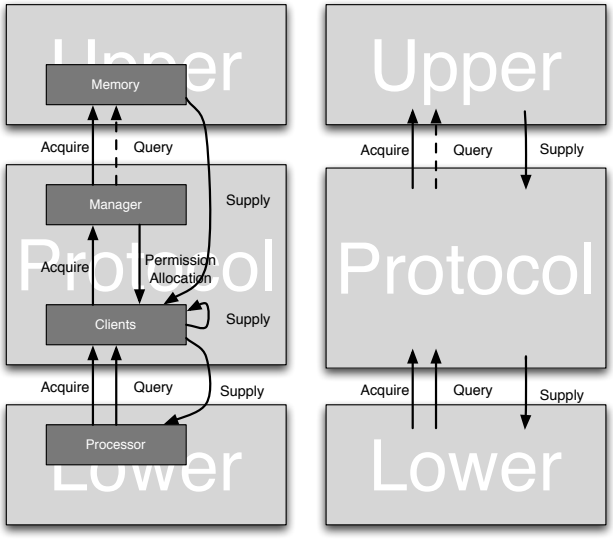


Figure 2 - The addition of permissions-query capabilities enable recursive coherence.

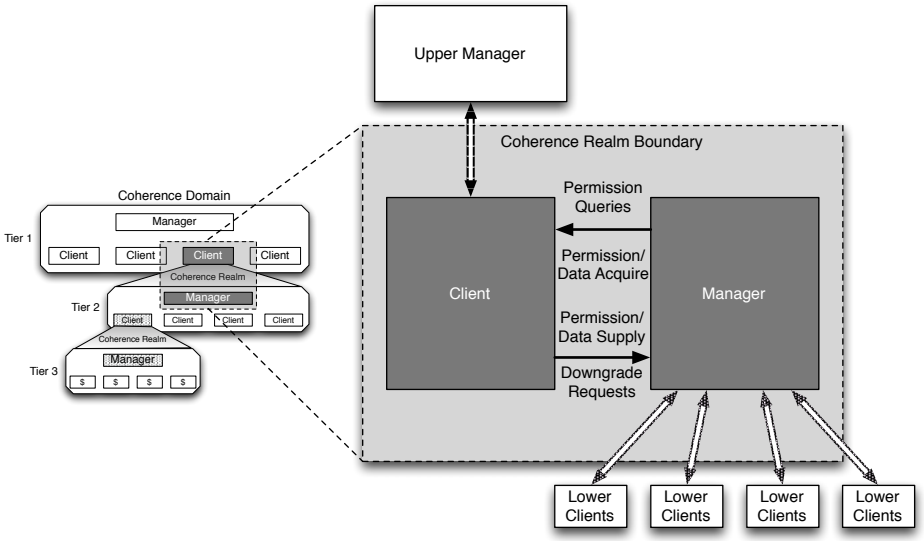


Figure 3 - Manager-Client Pairing and associated interfaces to preserve encapsulation.

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 4.

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 3 shows how the MCP interface enables coherence tier communication while respecting the encapsulation of the component protocols.

3.3 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 [13] and TokenB [19], 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.

3.3.1 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

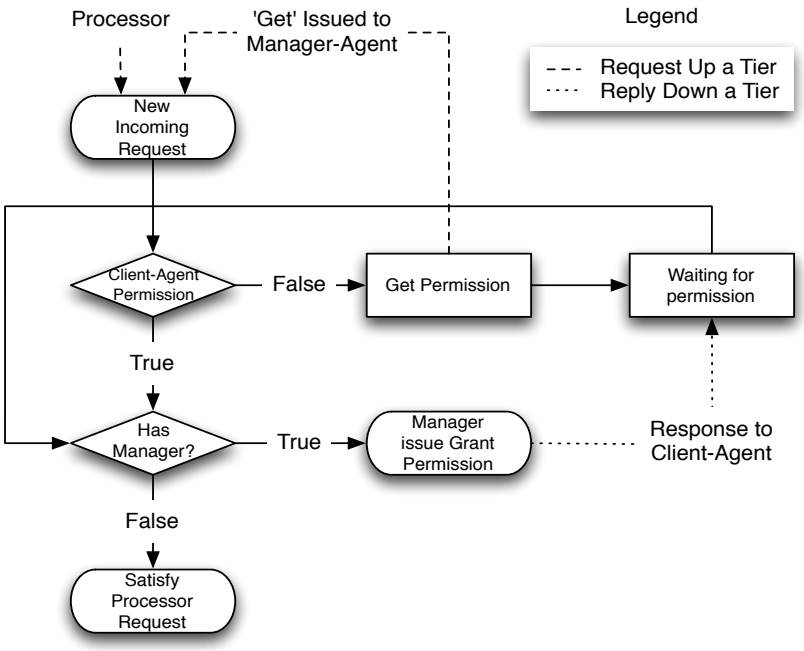


Figure 4 - Coherence hierarchy permission checking algorithm.

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 straight forward 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 [12].

3.3.2 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.

4. 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 [18], 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-

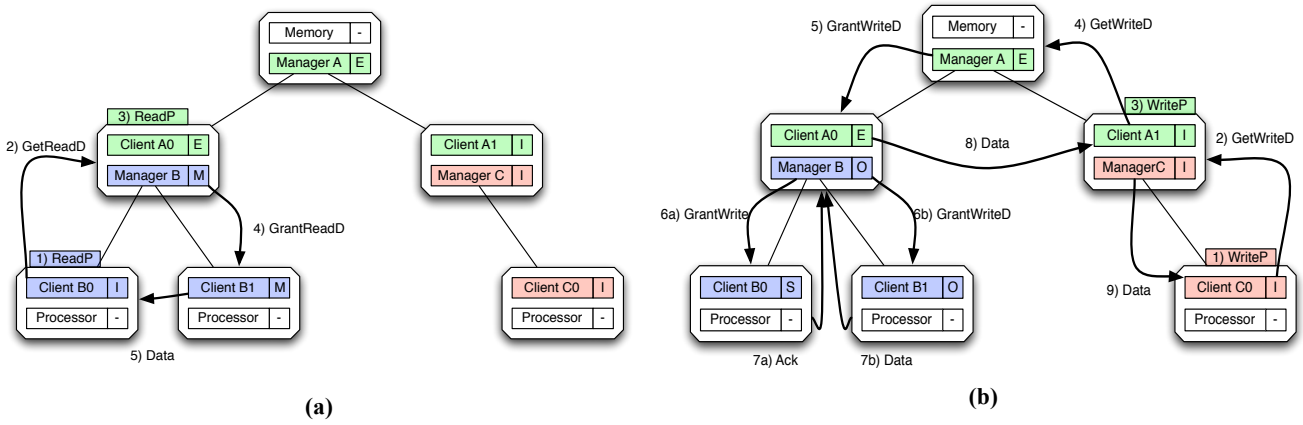


Figure 5 – (a) Realm-hit Read example and (b) Realm-Miss Write example

client pair. The flowchart in Figure 4 demonstrates the Manager-Client Pairing algorithm for processing permission acquires.

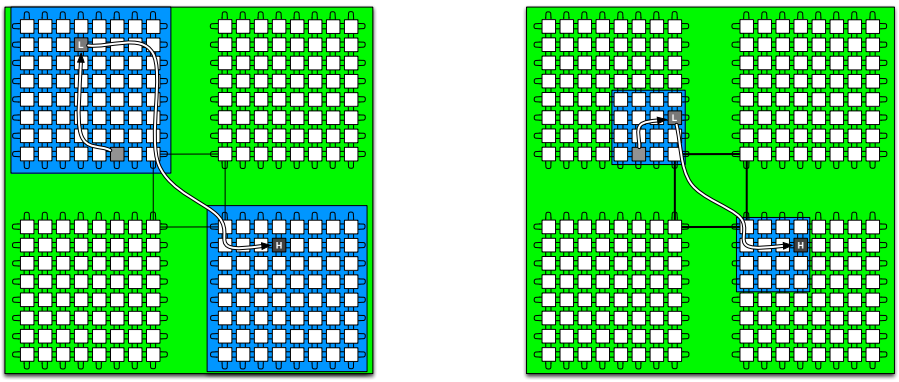
To aid in understanding and to highlight some important details of MCP, two examples are presented. In both examples we have a top-tier coherence realm, *A*, that implements a low-overhead MEI protocol to manage two lower coherence realms, *B* and *C*, both implementing MOESI. Manager *A* resides at 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 5(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 a 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.

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 5(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



(a) (b)

Figure 6 - 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.

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, as demonstrated in [16] except for more general protocols, is reserved future work.

5. 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.

5.1 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 [21-24]. 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.

5.2 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. The implemented MCP configurations were verified through a combination of random test case generation and hand-written sequences to stress potential corner-cases, similar to the approach discussed in [6]. 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 heterogeneity 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

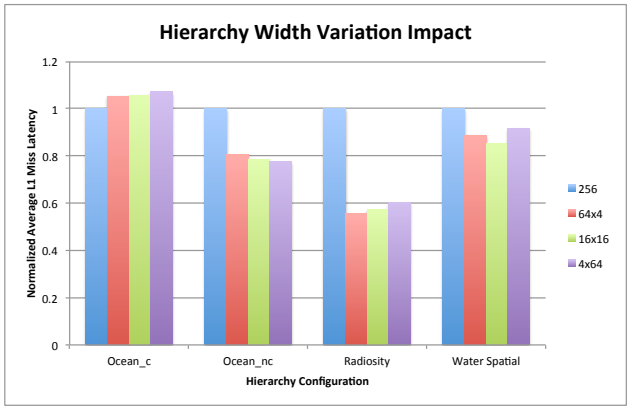


Figure 7 - Impact of hierarchy width on L1 miss latency

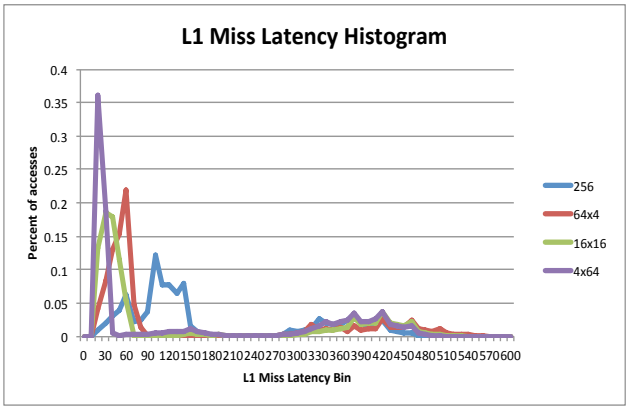


Figure 8 – Histogram of L1 Miss Latency for Water Spatial when varying hierarchy width

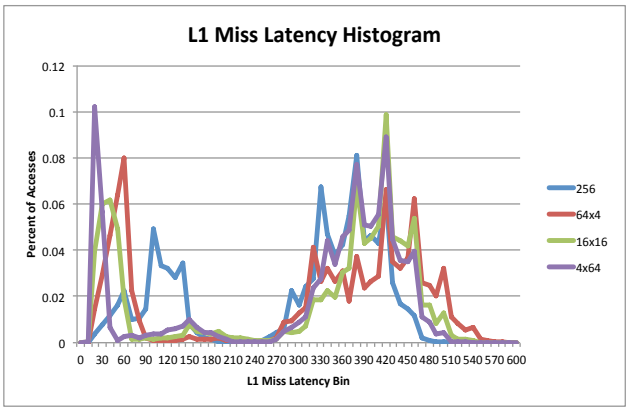


Figure 9 – Histogram of L1 Miss Latency for Ocean_c when varying hierarchy width

exploit different trade-offs with regard to locality and indirection delay by varying the width of the hierarchy. Figure 6 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.

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 adhoc glue layers.

5.3 Empirical Examples of Applied MCP

An internal execution-driven simulator is used in this work that models a manycore system with a detailed network infrastructure. The MIPS based emulator front-end from SESC [25] 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 [26] 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.

5.3.1 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 7.

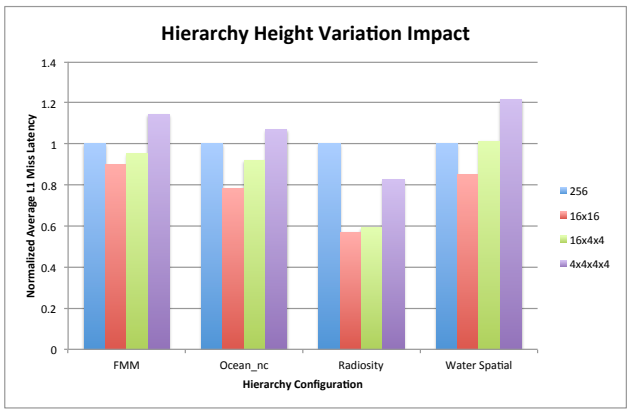


Figure 10 - Impact of hierarchy height on L1 miss latency

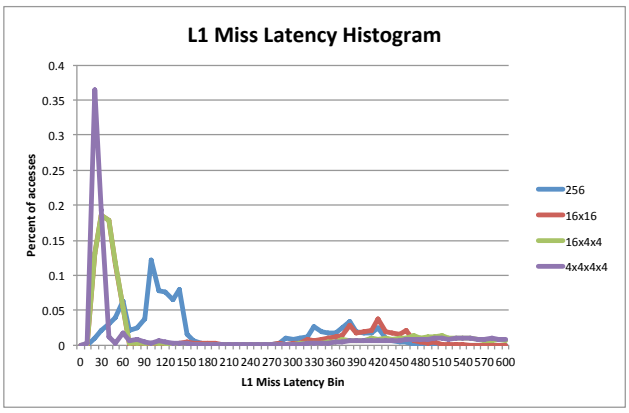


Figure 11 – Histogram of L1 Miss Latency for Water Spatial when varying hierarchy height

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 8 and 9.

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 6 regarding variation in realm size are confirmed by these figures as well. In both Figure 8 and 9 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 9 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.

5.3.2 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 10.

To verify this, we again examine a histogram of Water Spatial's L1 miss latency behavior (Figure 11). 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.

6. 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. As of this writing, MCP is currently the initial

starting point of an IEEE standards working group seeking to create a standard for coherent inter-operability between multi-vendor ensemble systems. In addition to the standards work, the inarguable benefit of protocol modularity provided by MCP will enable architects to compare and communicate their designs decisions more effectively in the future.

7. References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] D. A. Wood, et al., "Verifying a Multiprocessor Cache Controller Using Random Test Generation," IEEE Des. Test, vol. 7, pp. 13-25, 1990.

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

[8] 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.

[9] D. Lenoski, et al., "The directory-based cache coherence protocol for the DASH multiprocessor," SIGARCH Comput. Archit. News, vol. 18, pp. 148-159, 1990.

[10] 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.

[11] L. A. Barroso, 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.

[12] G. Gostin, et al., "The architecture of the HP Superdome shared-memory multiprocessor," presented at the Proceedings of the 19th annual international conference on Supercomputing, Cambridge, Massachusetts, 2005.

[13] M. M. K. Martin, et al., "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," SIGARCH Comput. Archit. News, vol. 33, pp. 92-99, 2005.

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

[15] M. R. Marty, et al., "Improving Multiple-CMP Systems Using Token Coherence," presented at the Proceedings of the 11th International Symposium on High-Performance Computer Architecture, 2005.

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

[17] J. Kuskin, et al., "The Stanford FLASH multiprocessor," presented at the Proceedings of the 21st annual international symposium on Computer architecture, Chicago, Illinois, United States, 1994.

[18] 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.

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

[20] 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.

[21] M. E. Acacio, et al., "A Two-Level Directory Architecture for Highly Scalable cc-NUMA Multiprocessors," IEEE Trans. Parallel Distrib. Syst., vol. 16, pp. 67-79, 2005.

[22] J. Zebchuk, et al., "A tagless coherence directory," presented at the Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, New York, New York, 2009.

[23] A. Ros, et al., "A scalable organization for distributed directories," J. Syst. Archit., vol. 56, pp. 77-87, 2010.

[24] J. H. Kelm, et al., "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.

[25] J. Renau, et al. (Jan. 2005, SESC Simulator. Available: http://sesc.sourceforge.net.

[26] S. C. Woo, et al., "The SPLASH-2 programs: characterization and methodological considerations," SIGARCH Comput. Archit. News, vol. 23, pp. 24-36, 1995.

[27] 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.

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