A Primer on Memory Consistency and Cache Coherence

Daniel J. Sorin, Mark D. Hill, and David A. Wood

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE #16

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

Many modern computer systems and most multicore chips (chip multiprocessors) support shared memory in hardware. In a shared memory system, each of the processor cores may read and write to a single shared address space. For a shared memory machine, the memory consistency model defines the architecturally visible behavior of its memory system. Consistency definitions provide rules about loads and stores (or memory reads and writes) and how they act upon memory. As part of supporting a memory consistency model, many machines also provide cache coherence protocols that ensure that multiple cached copies of data are kept up-to-date. The goal of this primer is to provide readers with a basic understanding of consistency and coherence. This understanding includes both the issues that must be solved as well as a variety of solutions. We present both high-level concepts as well as specific, concrete examples from real-world systems.

KEYWORDS

computer architecture, memory consistency, cache coherence, shared memory, memory systems, multicore processor, multiprocessor

A Primer on Memory Consistency and Cache Coherence
Daniel J. Sorin, Mark D. Hill, and David A. Wood
www.morganclaypool.com

ISBN: 9781608455645 paperback

ISBN: 9781608455652 ebook

DOI: 10.2200/S00346ED1V01Y201104CAC016

A Publication in the Morgan & Claypool Publishers series

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE #16

Lecture #16

Series Editor: Mark D. Hill, University of Wisconsin

Series ISSN
ISSN 1935-3235 print
ISSN 1935-3243 electronic

Contents

In contrast, a directory protocol orders transactions at the directory to ensure that conflicting requests are processed by all nodes in per-block order. However, the lack of a total order means that a requestor in a directory protocol needs another strategy to determine when its request has been serialized and thus when its coherence epoch may safely begin. Because (most) directory protocols do not use totally ordered broadcast, there is no global notion of serialization. Rather, a request must be individually serialized with respect to all the caches that (may) have a copy of the block. Explicit messages are needed to notify the requestor that its request has been serialized by each relevant cache. In particular, on a GetM request, each cache controller with a shared (S) copy must send an explicit acknowledgment (Ack) message once it has serialized the invalidation message.

This comparison between directory and snooping protocols highlights the fundamental tradeoff between them. A directory protocol achieves greater scalability (i.e., because it requires less bandwidth) at the cost of a level of indirection (i.e., having three steps, instead of two steps, for some transactions). This additional level of indirection increases the latency of some coherence transactions.

8.2 BASELINE DIRECTORY SYSTEM

In this section, we present a baseline system with a straightforward, modestly optimized directory protocol. This system provides insight into the key features of directory protocols while revealing inefficiencies that motivate the features and optimizations presented in subsequent sections of this chapter.

8.2.1 Directory System Model

We illustrate our directory system model in Figure 8.1. Unlike for snooping protocols, the topology of the interconnection network is intentionally vague. It could be a mesh, torus, or any other topology that the architect wishes to use. One restriction on the interconnection network that we assume in this chapter is that it enforces point-to-point ordering. That is, if controller A sends two messages to controller B, then the messages arrive at controller B in the same order in which they were sent.[1] Having point-to-point ordering reduces the complexity of the protocol, and we defer a discussion of networks without ordering until Section 8.7.3.

The only differences between this directory system model and the baseline system model in Figure 2.1 is that we have added a directory and we have renamed the memory controller to be the

[1] Strictly speaking, we require point-to-point order for only certain types of messages, but this is a detail that we defer until Section 8.7.3.

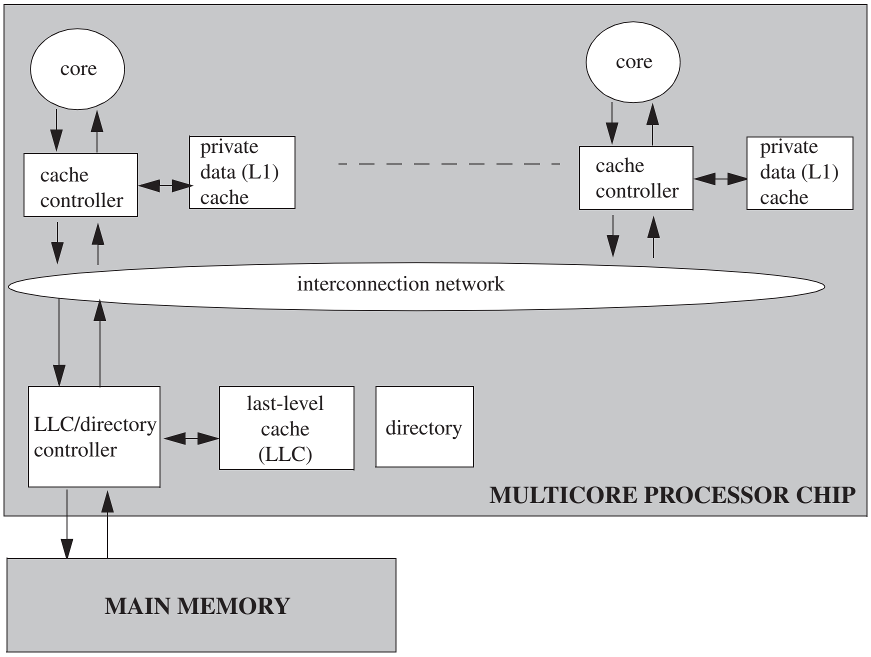


FIGURE 8.1: Directory system model.

directory controller. There are many ways of sizing and organizing the directory, and for now we assume the simplest model: for each block in memory, there is a corresponding directory entry. In Section 8.6, we examine and compare more practical directory organization options. We also assume a monolithic LLC with a single directory controller; in Section 8.7.1, we explain how to distribute this functionality across multiple banks of an LLC and multiple directory controllers.

8.2.2 High-Level Protocol Specification

The baseline directory protocol has only three stable states: MSI. A block is owned by the directory controller unless the block is in a cache in state M. The directory state for each block includes the stable coherence state, the identity of the owner (if the block is in state M), and the identities of the

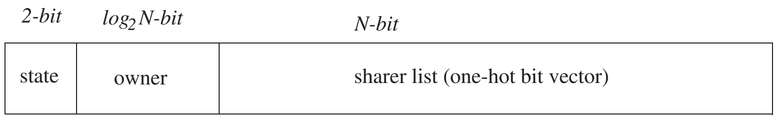
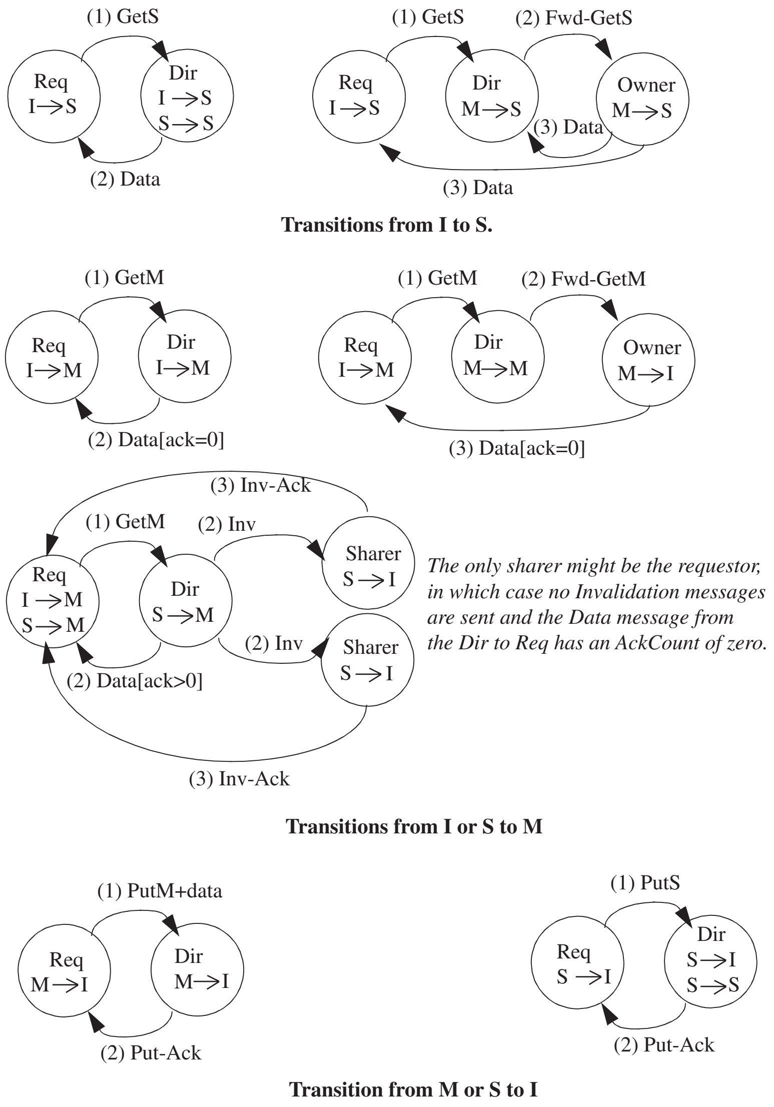


FIGURE 8.2: Directory entry for a block in a system with N nodes.



The only sharer might be the requestor, in which case no Invalidation messages are sent and the Data message from the Dir to Req has an AckCount of zero.

FIGURE 8.3: High-Level Description of MSI Directory Protocol. In each transition, the cache controller that requests the transaction is denoted “Req”.

sharers encoded as a one-hot bit vector (if the block is in state S). We illustrate a directory entry in Figure 8.2. In Section 8.5, we will discuss other encodings of directory entries.

Before presenting the detailed specification, we first illustrate a higher level abstraction of the protocol in order to understand its fundamental behaviors. In Figure 8.3, we show the transactions in which a cache controller issues coherence requests to change permissions from I to S, I or S to

M, M to I, and S to I. As with the snooping protocols in the last chapter, we specify the directory state of a block using a cache-centric notation (e.g., a directory state of M denotes that there exists a cache with the block in state M). Note that a cache controller may not silently evict a Shared block; that is, there is an explicit PutS request. We defer a discussion of protocols with silent evictions of shared blocks, as well as a comparison of silent versus explicit PutS requests, until Section 8.7.4.

Most of the transactions are fairly straightforward, but two transactions merit further discussion here. The first is the transaction that occurs when a cache is trying to upgrade permissions from I or S to M and the directory state is S. The cache controller sends a GetM to the directory, and the directory takes two actions. First, it responds to the requestor with a message that includes the data and the "AckCount"; the AckCount is the number of current sharers of the block. The directory sends the AckCount to the requestor to inform the requestor of how many sharers must acknowledge having invalidated their block in response to the GetM. Second, the directory sends an Invalidation (Inv) message to all of the current sharers. Each sharer, upon receiving the Invalidation, sends an Invalidation-Ack (Inv-Ack) to the requestor. Once the requestor receives the message from the directory and *all* of the Inv-Ack messages, it completes the transaction. The requestor, having received all of the Inv-Ack messages, knows that there are no longer any readers of the block and thus it may write to the block without violating coherence.

The second transaction that merits further discussion occurs when a cache is trying to evict a block in state M. In this protocol, we have the cache controller send a PutM message that includes the data to the directory. The directory responds with a Put-Ack. If the PutM did not carry the data with it, then the protocol would require a third message—a data message from the cache controller to the directory with the evicted block that had been in state M—to be sent in a PutM transaction. The PutM transaction in this directory protocol differs from what occurred in the snooping protocol, in which a PutM did not carry data.

8.2.3 Avoiding Deadlock

In this protocol, the reception of a message can cause a coherence controller to send another message. In general, if event A (e.g., message reception) can cause event B (e.g., message sending) and both these events require resource allocation (e.g., network links and buffers), then we must be careful to avoid deadlock that could occur if circular resource dependences arise. For example, a GetS request can cause the directory controller to issue a Fwd-GetS message; if these messages use the same resources (e.g., network links and buffers), then the system can potentially deadlock. In Figure 8.4, we illustrate a deadlock in which two coherence controllers C1 and C2 are responding to each other's requests, but the incoming queues are already full of other coherence requests. If the queues are FIFO, then the responses cannot pass the requests. Because the queues are full, each con-

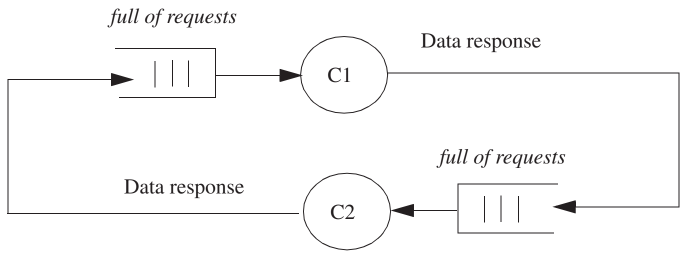


FIGURE 8.4: Deadlock example.

troller stalls trying to send a response. Because the queues are FIFO, the controller cannot switch to work on a subsequent request (or get to the response). Thus, the system deadlocks.

A well-known solution for avoiding deadlock in coherence protocols is to use separate networks for each class of message. The networks can be physically separate or logically separate (called *virtual networks*), but the key is avoiding dependences between classes of messages. Figure 8.5 illustrates a system in which request and response messages travel on separate physical networks. Because a response cannot be blocked by another request, it will eventually be consumed by its destination node, breaking the cyclic dependence.

The directory protocol in this section uses three networks to avoid deadlock. Because a request can cause a forwarded request and a forwarded request can cause a response, there are three message classes that each require their own network. Request messages are GetS, GetM, and PutM. Forwarded request messages are Fwd-GetS, Fwd-GetM, Inv(alidation), and Put-Ack. Response messages are Data and Inv-Ack. The protocols in this chapter require that the Forwarded Request network provides point-to-point ordering; other networks have no ordering constraints nor are there any ordering constraints between messages traveling on different networks.

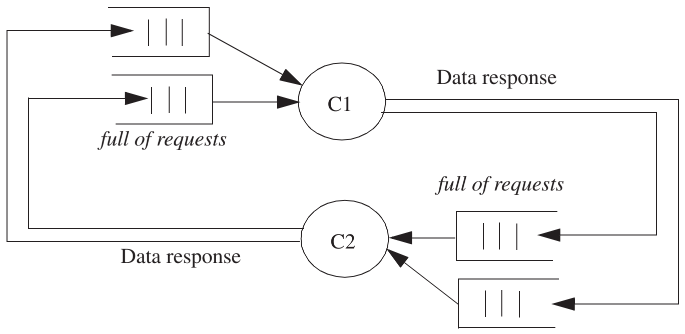


FIGURE 8.5: Avoiding deadlock with separate networks.

We defer a more thorough discussion of deadlock avoidance, including more explanation of virtual networks and the exact requirements for avoiding deadlock, until Section 9.3.

8.2.4 Detailed Protocol Specification

We present the detailed protocol specification, including all transient states, in Tables 8.1 and 8.2. Compared to the high-level description in Section 8.2.2, the most significant difference is the transient states. The coherence controllers must manage the states of blocks that are in the midst of coherence transactions, including situations in which a cache controller receives a forwarded request from another controller in between sending its coherence request to the directory and receiving all of its necessary response messages, including Data and possible Inv-Acks. The cache controllers can maintain this state in the miss status handling registers (MSHRs) that cores use to keep track of outstanding coherence requests. Notationally, we represent these transient states in the form XY^{AD}, where the superscript A denotes waiting for acknowledgments and the superscript D denotes waiting for data. (This notation differs from the snooping protocols, in which the superscript A denoted waiting for a request to appear on the bus.)

TABLE 8.1: MSI Directory Protocol—Cache Controller

	load	store	replacement	Fwd-GetS	Fwd-GetM	Inv	Put-Ack	Data from Dir (ack=0)	Data from Dir (ack>0)	Data from Owner	Inv-Ack	Last-Inv-Ack
I	send GetS to Dir/IS^D	send GetM to Dir/IM^{AD}										
IS^D	stall	stall	stall			stall		-/S		-/S		
IM^{AD}	stall	stall	stall	stall	stall			-/M	-/IM^A	-/M	ack--	
IM^A	stall	stall	stall	stall	stall						ack--	-/M
S	hit	send GetM to Dir/SM^{AD}	send PutS to Dir/SI^A			send Inv-Ack to Req/I						
SM^{AD}	hit	stall	stall	stall	stall	send Inv-Ack to Req/IM^{AD}		-/M	-/SM^A	-/M	ack--	
SM^A	hit	stall	stall	stall	stall						ack--	-/M
M	hit	hit	send PutM+data to Dir/MI^A	send data to Req and Dir/S	send data to Req/I							
MI^A	stall	stall	stall	send data to Req and Dir/SI^A	send data to Req/II^A		-/I					
SI^A	stall	stall	stall			send Inv-Ack to Req/II^A	-/I					
II^A	stall	stall	stall				-/I					

TABLE 8.2: MSI Directory Protocol—Directory Controller

	GetS	GetM	PutS-NotLast	PutS-Last	PutM+data from Owner	PutM+data from NonOwner	Data
I	send data to Req, add Req to Sharers/S	send data to Req, set Owner to Req/M	send Put-Ack to Req	send Put-Ack to Req		send Put-Ack to Req	
S	send data to Req, add Req to Sharers	send data to Req, send Inv to Sharers, clear Sharers, set Owner to Req/M	remove Req from Sharers, send Put-Ack to Req	remove Req from Sharers, send Put-Ack to Req/I		remove Req from Sharers, send Put-Ack to Req	
M	Send Fwd-GetS to Owner, add Req and Owner to Sharers, clear Owner/S^D	Send Fwd-GetM to Owner, set Owner to Req	send Put-Ack to Req	send Put-Ack to Req	copy data to memory, clear Owner, send Put-Ack to Req/I	send Put-Ack to Req	
S^D	stall	stall	remove Req from Sharers, send Put-Ack to Req	remove Req from Sharers, send Put-Ack to Req		remove Req from Sharers, send Put-Ack to Req	copy data to memory/S

Because these tables can be somewhat daunting at first glance, the next section walks through some example scenarios.

8.2.5 Protocol Operation

The protocol enables caches to acquire blocks in states S and M and to replace blocks to the directory in either of these states.

I to S (common case #1)

The cache controller sends a GetS request to the directory and changes the block state from I to IS^D. The directory receives this request and, if the directory is the owner (i.e., no cache currently has the block in M), the directory responds with a Data message, changes the block's state to S (if it is not S already), and adds the requestor to the sharer list. When the Data arrives at the requestor, the cache controller changes the block's state to S, completing the transaction.

I to S (common case #2)

The cache controller sends a GetS request to the directory and changes the block state from I to IS^D. If the directory is *not* the owner (i.e., there is a cache that currently has the block in M), the directory forwards the request to the owner and changes the block's state to the transient state S^D. The owner responds to this Fwd-GetS message by sending Data to the requestor and changing the block's state to S. The now-previous owner must also send Data to the directory since it is relinquishing ownership to the directory, which must have an up-to-date copy of the block. When the

Data arrives at the requestor, the cache controller changes the block state to S and considers the transaction complete. When the Data arrives at the directory, the directory copies it to memory, changes the block state to S, and considers the transaction complete.

I to S (race cases)

The above two I-to-S scenarios represent the common cases, in which there is only one transaction for the block in progress. Most of the protocol's complexity derives from having to deal with the less-common cases of multiple in-progress transactions for a block. For example, a reader may find it surprising that a cache controller can receive an Invalidation for a block in state IS^D. Consider core C1 that issues a GetS and goes to IS^D and another core C2 that issues a GetM for the same block that arrives at the directory after C1's GetS. The directory first sends C1 Data in response to its GetS and then an Invalidation in response to C2's GetM. Because the Data and Invalidation travel on separate networks, they can arrive out of order, and thus C1 can receive the Invalidation before the Data.

I or S to M

The cache controller sends a GetM request to the directory and changes the block's state from I to IM^{AD}. In this state, the cache waits for Data and (possibly) Inv-Acks that indicate that other caches have invalidated their copies of the block in state S. The cache controller knows how many Inv-Acks to expect, since the Data message contains the AckCount, which may be zero. Figure 8.3 illustrates the three common-case scenarios of the directory responding to the GetM request. If the directory is in state I, it simply sends Data with an AckCount of zero and goes to state M. If in state M, the directory controller forwards the request to the owner and updates the block's owner; the now-previous owner responds to the Fwd-GetM request by sending Data with an AckCount of zero. The last common case occurs when the directory is in state S. The directory responds with Data and an AckCount equal to the number of sharers, plus it sends Invalidations to each core in the sharer list. Cache controllers that receive Invalidation messages invalidate their shared copies and send Inv-Acks to the requestor. When the requestor receives the last Inv-Ack, it transitions to state M. Note the special Last-Inv-Ack event in Table 8.1, which simplifies the protocol specification.

These common cases neglect some possible races that highlight the concurrency of directory protocols. For example, core C1 has the cache block in state IM^A and receives a Fwd-GetS from C2's cache controller. This situation is possible because the directory has already sent Data to C1, sent Invalidation messages to the sharers, and changed its state to M. When C2's GetS arrives at the directory, the directory simply forwards it to the owner, C1. This Fwd-GetS may arrive at C1 before all of the Inv-Acks arrive at C1. In this situation, our protocol simply stalls and the cache controller waits for the Inv-Acks. Because Inv-Acks travel on a separate network, they are guaranteed not to block behind the unprocessed Fwd-GetS.

M to I

To evict a block in state M, the cache controller sends a PutM request that includes the data and changes the block state to MI^A. When the directory receives this PutM, it updates the LLC/memory, responds with a Put-Ack, and transitions to state I. Until the requestor receives the Put-Ack, the block's state remains effectively M and the cache controller must respond to forwarded coherence requests for the block. In the case where the cache controller receives a forwarded coherence request (Fwd-GetS or Fwd-GetM) between sending the PutM and receiving the Put-Ack, the cache controller responds to the Fwd-GetS or Fwd-GetM and changes its block state to SI^A or II^A, respectively. These transient states are effectively S and I, respectively, but denote that the cache controller must wait for a Put-Ack to complete the transition to I.

S to I

Unlike the snooping protocols in the previous chapter, our directory protocols do not silently evict blocks in state S. Instead, to replace a block in state S, the cache controller sends a PutS request and changes the block state to SI^A. The directory receives this PutS and responds with a Put-Ack. Until the requestor receives the Put-Ack, the block's state is effectively S. If the cache controller receives an Invalidation request after sending the PutS and before receiving the Put-Ack, it changes the block's state to II^A. This transient state is effectively I, but it denotes that the cache controller must wait for a Put-Ack to complete the transaction from S to I.

8.2.6 Protocol Simplifications

This protocol is relatively straightforward and sacrifices some performance to achieve this simplicity. We now discuss two simplifications:

- The most significant simplification, other than having only three stable states, is that the protocol stalls in certain situations. For example, a cache controller stalls when it receives a forwarded request while in a transient state. A higher performance option, discussed in Section 8.7.2, would be to process the messages and add more transient states.
- A second simplification is that the directory sends Data (and the AckCount) in response to a cache that is changing a block's state from S to M. The cache already has valid data and thus it would be sufficient for the directory to simply send a data-less AckCount. We defer adding this new type of message until we present the MOSI protocol in Section 8.4.

to the home. Assume that C2's snoop response arrives at the home before C1's snoop response. In this case, C1's request is ordered first and the home sends data to C1 and informs C1 that there is a race. C1 then sends an acknowledgment to the home, and the home subsequently sends a message to C1 that both completes C1's transaction and tells C1 to send the block to C2. Handling this race is somewhat more complicated than in a typical directory protocol in which requests are ordered when they arrive at the directory.

Source Snoop mode uses more bandwidth than Home Snoop, due to broadcasting, but Source Snoop's common case (no race) transaction latency is less. Source Snoop is somewhat similar to Coherent HyperTransport, but with one key difference. In Coherent HT, a request is unicasted to the home, and the home broadcasts the request. In Source Snoop, the requestor broadcasts the request. Source Snoop thus introduces more complexity in resolving races because there is no single point at which requests can be ordered; Coherent HT uses the home for this purpose.

8.9 DISCUSSION AND THE FUTURE OF DIRECTORY PROTOCOLS

Directory protocols have come to dominate the market. Even in small-scale systems, directory protocols are more common that snooping protocols, largely because they facilitate the use of point-to-point links in the interconnection network. Furthermore, directory protocols are the *only* option for systems requiring scalable cache coherence. Although there are numerous optimizations and implementation tricks that can mitigate the bottlenecks of snooping, fundamentally none of them can eliminate these bottlenecks. For systems that need to scale to hundreds or even thousands of nodes, a directory protocol is the only viable option for coherence. Because of their scalability, we anticipate that directory protocols will continue their dominance for the foreseeable future.

It is possible, though, that future highly scalable systems will not be coherent or at least not coherent across the entire system. Perhaps such systems will be partitioned into subsystems that are coherent, but coherence is not maintained across the subsystems. Or perhaps such systems will follow the lead of supercomputers, like those from Cray, that have either not provided coherence [14] or have provided coherence but restricted what data can be cached [1].

8.10 REFERENCES

[1] D. Abts, S. Scott, and D. J. Lilja. So Many States, So Little Time: Verifying Memory Coherence in the Cray X1. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003. doi:10.1109/IPDPS.2003.1213087

[2] A. Agarwal, R. Simoni, M. Horowitz, and J. Hennessy. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 280–89, May 1988. doi:10.1109/ISCA.1988.5238

[3] J. K. Archibald and J.-L. Baer. An Economical Solution to the Cache Coherence Problem. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pp. 355–62, June 1984. doi:10.1145/800015.808205

[4] J.-L. Baer and W.-H. Wang. On the Inclusion Properties for Multi-Level Cache Hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 73–80, May 1988. doi:10.1109/ISCA.1988.5212

[5] P. Conway and B. Hughes. The AMD Opteron Northbridge Architecture. *IEEE Micro*, 27(2):10–21, March/April 2007. doi:10.1109/MM.2007.43

[6] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *IEEE Micro*, 30(2):16–29, March/April 2010. doi:10.1109/MM.2010.31

[7] A. Gupta, W.-D. Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *Proceedings of the International Conference on Parallel Processing*, 1990.

[8] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–18, Nov. 1993. doi:10.1145/161541.161544

[9] Intel Corporation. An Introduction to the Intel QuickPath Interconnect. Document Number 320412-001US, Jan. 2009.

[10] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 241–51, June 1997.

[11] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, Mar. 1992. doi:10.1109/2.121510

[12] R. A. Maddox, G. Singh, and R. J. Safranek. *Weaving High Performance Multiprocessor Fabric: Architecture Insights into the Intel QuickPath Interconnect*. Intel Press, 2009.

[13] M. R. Marty and M. D. Hill. Virtual Hierarchies to Support Server Consolidation. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[14] S. L. Scott. Synchronization and Communication in the Cray T3E Multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 26–36, Oct. 1996.

• • • •