Finding Deadlocks in Cache-Coherent Distributed Shared Memory Machines*

Boon S. Ang and Derek Chiou

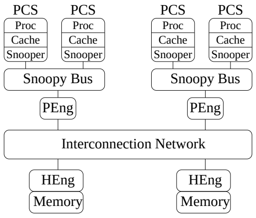

Figure 1: An abstract view of a typical CCDSM system.

Abstract

The design of a cache-coherent distributed shared memory (CCDSM) system is complex and prone to errors due to its *concurrent, distributed* nature. We examine the problem of deadlocks due to resource sharing between different memory requests that are occuring concurrrently, an issue which has not been dealt with extensively in the literature. We discuss a tool for illuminating all possible deadlock scenarios due to resource sharing.

1 Introduction

A CCDSM system supports cache-coherent global shared memory on distributed-memory multiprocessor hardware. Coherence is maintained in a non-broadcast manner across sites, typically using a directory-based approach. Many research[6, 5, 1, 4] and a few commerical[3, 2] CCDSM systems have been built or are being built. Figure 1 shows a typical CCDSM implemention, consisting of processor sites and memory sites connected by a scalable interconnection network.

Each processor *site* has one or more cache-equiped processors connected by a snoopy bus. The PCS entity encompasses the processor core, its cache and its snoopy-bus interface which modern microprocessors integrate onto a single chip. The PEng (Processor Engine) connects the snoopy bus to the interconnection network, offering temporary buffering and translation between snoopy bus operations and network protocol messages. At each memory site, a HEng (Home Engine) maintains directory information, services memory requests, and initiates and coordinates coherence actions to remote sites.

A key component of a CCDSM system is an abstract coherence protocol which is free from logical deadlocks. This coherence protocol is then implemented at the lower, hardware and firmware level, a mapping which must be free from deadlocks due to hardware resource sharing. In the presence of multiple concurrent requests, undisciplined allocation of resources can lead to classical deadlocks where each request needs a resource held by another to complete.

*This work is supervised by Prof. Arvind and is supported in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310.

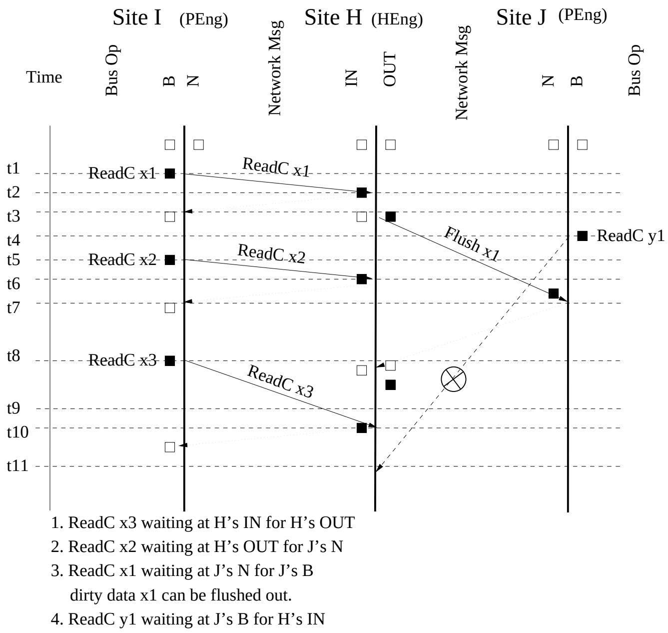

Figure 2: An example of resource sharing deadlock.

This paper focuses on the lower level implementation issues. We illustate the deadlock problem with a simple example, then discuss a tool which will find all deadlocks when used correctly. To simplify discussion, we assume a MESI-based directory protocol with home sites for cache-lines for the rest of this paper.

2 An Example Deadlock

The following example illustrates a deadlock scenario due to resource sharing. Consider an implementation where the PEng has two pools of buffers: (i) **B** buffers storing requests from the site snoopy bus, and (ii) **N** buffers storing messages coming from the network. The HEng also has two pools of buffers: (i) **IN** buffers for requests coming from the network, and (ii) **OUT** buffers for messages going out to the network. Buffers are only deallocated when their data is moved to the next buffer.

We will assume a non-FIFO network which consists of switches only, with no capability for buffering; all buffering of messages are done at the end points with the buffers just described. Such a network is a non-blocking switch and messages are delivered as long as the type of buffer it requires at the destination is available.

Such a design, though seemly reasonable, can deadlock easily. Figure 2 is a time-line which illustrates a trivially

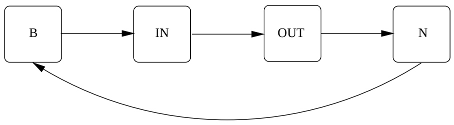

Figure 3: Static Dependence Graph for an invalidation coherence protocol indicating potential deadlocks

simple deadlock case. Suppose there is only one of each of the four types of buffers. Prior to the events indicated in the figure, cache lines x1, x2, and x3, all from home site H, are in MODIFIED state at Site J. The boxes under the **B**, **N**, **IN**, and **OUT** columns indicate whether the respective buffers are full(black) or empty(white). The dotted lines are messages indicating the acceptance of a message, freeing up the buffer at sender's end. The dashed line with an X indicates an unsuccessful attempt to deliver message due to a lack of buffers at the destination.

The deadlock in this example is due to the fact that J's outstanding READC x1 request cannot be completed until the FLUSH x1 is completed. This FLUSH in turn cannot complete until J's READC y1 completes because it depends on a **B** buffer to return the dirty cache-line, which is used by J's READC y1. Unfortunately, the READC y1 cannot complete until H's **IN** buffer is freed by I's READC x3, which requires H's **OUT** buffer to be vacated by I's READC x2, which, in turn, cannot happen until J's **N** buffer is freed by the FLUSH x1. We now have a cycle of dependences.

Surprisingly, adding more buffers to any category does not solve the fundamental problem (though it may solve the problem for a specific implementation.) Even adding a class of buffers at the processor sites to accept FLUSH initiated writebacks of dirty data does not eliminate possible deadlocks.

3 Static Dependence Graphs

Static dependence graphs is a convenient tool for exposing such problems. This technique, utilizing static dependency information, provides a way to illuminate *all* potential dynamic dependences. The static dependence graph conservatively condenses the dynamic information into a static form, ensuring that if dynamic dependence cycles are possible, static dependence cycles must exist in the static dependence graph. Thus, if we can ensure that there is no cycle in the static dependence graph, the system is free from deadlocks due to the resources modeled in the static graph.

Each node of the static dependence graph represents a single set of blocking hardware resources. An arc from node x to y indicates that the coherence protocol may require holding on to resource type x until resource type y is available. Figure 3 shows the static dependence graph of our previous example. The dependences from left to right is the usual flow of requests first getting captured in the **B** buffer, then forwarded to the home site's **IN** buffer, which generates message(s) that go to the **OUT** buffer. These messages are sent to the **N** buffer, either as replies to requests, or as coherence maintenance requests. The problem comes from the arc from the **N** buffer to the **B** buffer. This, due to FLUSH requests to obtain the most up to date data, completes a cycle in the static dependence graph.

The presence of a cycle in the static dependence graph may not lead to actual dynamic cycles if the number of concurrent operations is controlled, and sufficient hardware resources are available. Insuring cycle-free static dependency graphs, however, is a more general and cleaner solution. Of course, the dependency graph must be accurate and complete in order to correctly discover deadlock potentials.

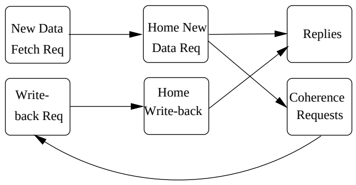

Figure 4: Static Dependence Graph for an invalidation coherence protocol with no deadlock potential

Let us apply this technique to our example. In order to eliminate the cycle in the static dependence graph, the **B** buffers have to be divided into two separate types, for new data fetch requests and write-back requests. The **N** buffers also have to be separated into two types, one for replies coming back with data, and another for coherence requests. Two categories of buffers are needed at home sites, but the categorization should be for new data fetch requests, and write-back requests. This gives rise to the static dependence graph in Figure 4. In a complete analysis of a CCDSM design, the resources within the PCS from the cache outwards have to be taken into account.

4 Conclusions

The static dependence graph technique is not a new one: it has been used in the past by network designers to find deadlocks. Using such techniques allows one to find potential deadlocks in cache-coherent systems easily and accurately.

References

[1] A. Agarwal et al. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocesors.* Kluwer Academic Publishers, 1991.

[2] T. Brewer. A highly scalable system utilizing up to 128 pa-risc processors. Convex Computer Corporation.

[3] H. Burkhardt III et al. Overview of the KSR1 Computer System. Technical Report KSR-TR-9202001, Kendall Square Research, Boston, February 1992.

[4] D. Chiou et al. *T-NG: Delivering Seamless Parallel Computing. Computation Structures Group Memo 371, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, February 1995. (To appear in EURO-PAR '95).

[5] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture, Chicago, Il*, pages 302–313, April 1994.

[6] D. Lenoski et al. The Directory-Based Cache Coherence Protocol for the DASH Multiprocesor. In *Proceedings of The 17th Annual International Symposium on Computer Architecture, Seattle, WA*, pages 148–159, 1990.