A Study on Memory Consistency Approaches in Distributed Shared Memory Systems

Aftab Hussain

Department of Computer Science,
Donald Bren School of Information and Computer Sciences,
University of California, Irvine

aftabh@uci.edu

Abstract

Following the increasing popularity and need of parallel computing, the idea of a distributed shared memory (DSM) model gained significant attention in the late 1970s, engendering a new area of research. Examples of some early works in this area include those of Kai Li. The actual implementation of the distributed shared memory models however suffered from same problems of shared memory models. One such problem is the *memory coherence problem*. Because of the distributed nature of DSM, this problem became more complicated to address than it was originally in the shared memory models. In this paper, we discuss some approaches that were used to address the memory coherence in the context of systems that have actually implemented them.

Keywords: shared memory model, message passing, serialization

1. Introduction

The idea of a Distributed Shared Memory (DSM) system first came between the late 1970s to the mid 1980s. Some of the earliest works in DSM were presented by [1]. They were seen as the answer to achieving high scalability in performing parallel computations, while giving the programmer a unified view of the memory of the system. In particular, physical dispersed memories are joined together into a single virtual address space [2]. The main selling point of the DSM was that it combined the benefits of the shared memory model and the distributed memory model.

The *shared memory system* was the dominant architecture in the 1980s, particularly for small numbers of processors (16 or 32) [3]. A shared memory system is shown in Figure 1. These systems were also called *bus-based multiprocessors* or *symmetric multiprocessors (SMPs)*. This is because all processors have the same relationship with the centralized main memory. For this reason, the shared memory model is suited to *tightly-coupled microprocessor systems*, where the processors are generally identical and controlled by a single operating system. A modern day example of such a tightly-coupled microprocessor system is the IBM p690 [4].

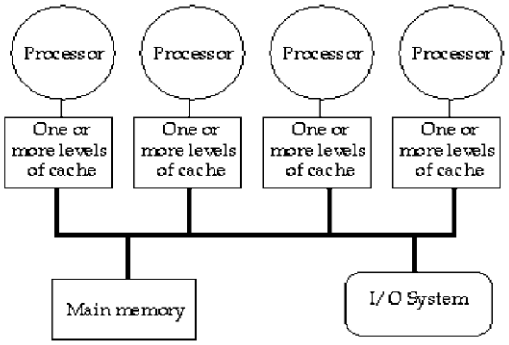

Figure 1: A shared memory multiprocessor. [3]

The main advantage of the shared memory system is that it supports a traditional programming model,

where memory is viewed as a single shared address space. In addition, the shared memory machines had low communication costs because processors were able to interact with each other directly through the bus without the use of a software layer.

However, the shared memory model is difficult to scale as it depends on the use of a central main memory. Also, in tightly-coupled microprocessor system, main memory is accessed by a common bus, which can carry out signals serially, and which thus limits the size of the system to only tens of processors [5].

In order to accommodate more processors the logical approach was to adopt a *distributed memory model*, where each processor is allocated a separate memory module. Also, instead of relying on a bus, in this model the processors rely on a scalable interconnection network to communicate with each other, as shown in Figure 2. The local memory acceses do not consume the bandwidth of the network. In today's distributed memory multiprocessor architecture there exists separate multi-core processor nodes instead of single-core processor nodes. The distributed nature of the of this model allows it to form the foundation of loosely coupled systems.

While the distributed memory model clearly provided better scalability, programming a communication model between a large number of non-identical processors was challenging. In order to overcome this challenge the DSM approach was proposed: it physically distributed memory among multiple processors (achieving scalability) and while implementing a single shared address space (simplifying the programming model). The DSM is, therefore, also known as the shared virtual memory model. An example of a DSM is shown in Figure 3.

Prior to the DSM, the approaches that were used to overcome the programming-model-simplicity and scalability bottleneck relied on message passing architectures. They consist of separate computing nodes with no shared structure, except the interconnection network [3]. Such architectures have many limitations.

Message passing systems operate by the use of user primitives, like *send* and *receive* [6]. These primitives can be used to synchronize parallel programs. However, this requires the programmer to know about the times of data transfers. In addition, these systems have been shown to have difficulties passing complex data structures [6]. For instance, it has been shown that passing a list by sending messages takes significant space and time overhead [7]. The process involves packing and unpacking of the data structures. A *remote-procedure-call* (RPC) based architecture overcomes the problem of having to know when the data structures are to be passed. Nevertheless, it still suffers when passing complex data structures [6].

The main reason behind the above mentioned drawbacks of message passing and RPC based approaches is that they both manage multiple address spaces. The DSM or shared virtual memory unifies the address space, allowing the processors to work on a single address space. This obviates the need to pack and unpack data structures as done in the above approaches.

Also, DSM simplifies process migration [6]. A *process migration* involves the parallel transfer of all operating system resources (e.g. code and stack) allocated by the process, which is expensive [8]. In a multiprocessor environment that relies on a multi-address space, translating the contents of different address spaces on the fly easily and efficiently becomes challenging. In a DSM system, process migration can be achieved by just transferring a process from the queue of one processor to the queue of the destination processor. This is possible because of the shared address space provided in a DSM.

The implementation of the DSM, however, is not straight-forward. The main difficulty is caused by the problem of memory coherence, in other words, ensuring that all processors have a consistent view of the data in the entire system. If the shared data in the system is not replicated, enforcing memory coherence becomes trivial [5]. In such a scenario, the network can order requests to data, in the order in which they were made. If a node consists of data that is shared, it merely needs to perform each request on the data one at a time. As mentioned in [5], this is the strictest form of memory consistency.

However, in order to perform parallel computations in a DSM system, it becomes necessary to replicate data, and thus such protocols would not work. In this survey, we explore the various approaches that have been adopted to address memory coherence in

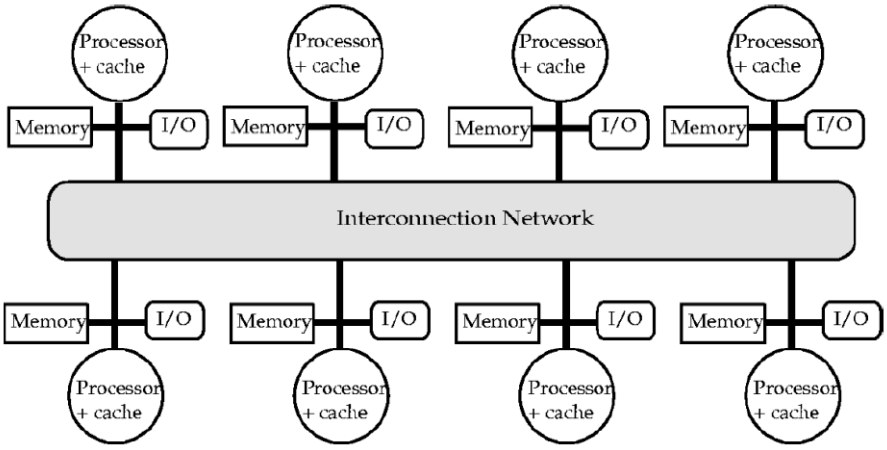

Figure 2: A distributed memory multiprocessor [3].

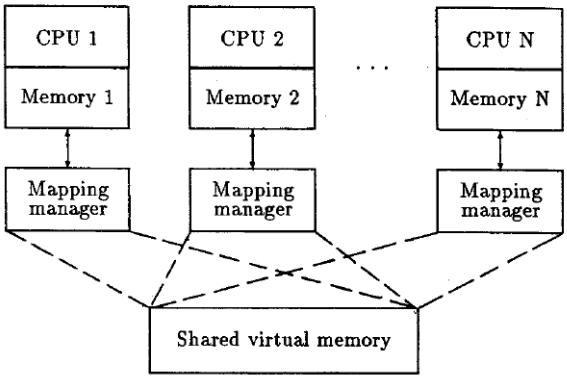

Figure 3: Memory mapping in a distributed shared memory multiprocessor. [6]

distributed shared memory systems.

The rest of this paper is organized as follows: Section 2 gives further details of the DSM Model and describes the memory coherence problem. Sections 3 to 5 discusses various memory coherence schemes in the context of different operating systems that have actually implemented the schemes. We conclude our paper in Section 6.

2. A Shared Virtual Memory or Distributed Shared Memory (DSM) Model

In this section, we first discuss the DSM Model in further detail (Subsection 2.1) and then describe the memory coherence problem (Subsection 2.2).

2.1. Description

A DSM allows the use of a shared programming paradigm in a loosely coupled system which contributes to ease of programming and portability. As was introduced earlier (See Figure 3), a DSM constitutes of a single address space shared by a number of processors, which allows processors to access memory locations directly. The mapping between local memories and the shared virtual memory address space is facilitated by the memory managers. This address space is partitioned into pages, each of which have a status flag that can signal it is *read-only* or *write*. A *read only* page can exists in the memories of many processors. A *write* page can only exist in one processor's physical memory. It is important to note that this shared memory only exists virtually. On the occurrence of a local page fault, a memory manager will try to retrieve the page from the disk or from another processor. In addition, the manager shall guarantee the atomicity if such an operation involves a write.

An important responsibility of the memory managers is to always keep the address space coherent at all times [6], such that any read operation performed on a memory location returns the same value as the most recent write operation on that same location. This is known as memory coherence, which is discussed next.

2.2. The Memory Coherence Problem

"A memory is *coherent* if the value returned by a read operation is always the same as the value writ-

ten by the most recent write operation to the same address." [9]. It follows that if there exists only one memory access path to a memory, there would be no coherence problem, since the memory can be read or written on by only one processor at a time. Of course, this assumption does not hold in today's parallel, multiprocessor architectures.

Before it was encountered in DSM, the memory coherence problem was also faced in uniprocessors and multi-cache processors [9]. A multicache multiprocessor consists of a number of processors that share a physical memory through a private cache. The size of the cache is relatively much smaller than that of the physical memory, and the bus that connects the cache to the physical memory is relatively fast. This allowed the development of coherence protocols whereby conflicts on the same memory location could be solved with very little time delay. These coherence protocols generally involved interrogating all processors in the network (via bias) about the status of a page through a broadcast, and were termed as snoopy protocols [3]. An example of such a protocol is shown in Figure 4.

The memory coherence problem becomes more complicated in DSM. This is because a DSM model works on a loosely coupled system built on top of an interconnection network. This means the communication cost between processors is not negligible. Consequently, any write conflict is not likely to be resolved within a short-time.

2.2.1. Ideas on Ensuring Memory Coherence in the DSM model

Early researchers have addressed the problem from the *granularity* angle. In particular, they proposed minimizing the size of the pages (the unit of data transfer in a DSM model) with the assumption that the cost of communicating large sized pages is not much larger than the cost of smaller ones [9]. The benefit of this approach is that it reduces the probability of memory contention (conflicts) among the processors.

The other design choice researchers focused upon was on *how* the pages were transferred. These strategies are summarized in Figure 5 [9]. The difficulty of the combinations of the strategies were hypothesized by Li and Hudak (For details the reader is encouraged to see [9]). As can be seen, these strategies had two aspects, page synchronization and page ownership.

Page synchronization can be done in 2 ways: the invalidation approach and the write broadcast approach. In the *invalidation approach*, a processor has either *write* or *read* access to a page, where all read-only copies of a page are invalidated before a processor writes to a page [6]. In a *write-broadcast* approach, all copies of a particular page are updated after a processor writes on that page [10].

The *page ownership* aspect of these strategies governed who (which processor or processors) would control the transfers of the pages. These approaches are described in Section 3 in the context of a system that implemented them, IVY.

Over the years, there have been many other strategies proposed, many of which rely on the above concepts, others propose a completely new solution model. We discuss these approaches in the following sections.

3. IVY

3.1. Overview of the System

IVY is among the first implementations of the distributed shared memory system [6]. It was implemented on the Apollo Domain [11, 12]. The Apollo Domain is an integrated system of Apollo workstations and servers connected by a 12M bit/sec baseband. It used a single token ring network. The Apollo Domain uses the Aeges operating system; IVY was implemented on a modified version of Aegis in the Apollo Domain. At the time, IVY achieved significant speedups in performing non-trivial operations like matrix multiply and dot product, and motivated the use of a shared memory model on loosely coupled systems.

The hierarchy of the IVY system is shown in Figure 6. It consists of five modules as shown. Three of the modules act as interfaces to the client (process management, memory allocation, initialization). The process management module implements operations for process control, process migration, and process synchronization. The remote operation module implements RPC mechanisms. The memory allocation module is responsible for allocating memory to the

Processor A Program action	Processor A System Action	Processor B Program Action	Processor B System Action
x = 1	Broadcast cache invalidation for x		
			Receive invalidation and eliminate x from cache
		t = x + 1	Broadcast cache miss for x
	Receive cache miss; recognize that only copy of x is in cache and place value of x on bus		
			Receive value of x (=1) from bus, place into cache, and continue

Figure 4: A demonstration of a snoopy-based cache coherence scheme [3].

Page synchronization method	Page ownership strategy			
		Dynamic		
		Centralized manager	Distributed manager	
	Fixed		Fixed	Dynamic
Invalidation	Not allowed	Okay	Good	Good
Write-broadcast	Very expensive	Very expensive	Very expensive	Very expensive

Figure 5: Fundamental strategies for addressing memory coherence problem in DSM [9].

data (pages) and works in conjunction with the memory mapping module, which implements the mapping between the local memories and the shared virtual memory address space.

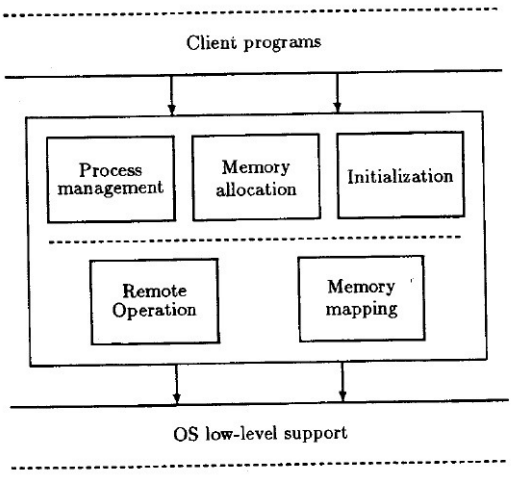

Figure 6: The hierarchy of the IVY system [6].

3.2. Preliminaries on the Memory Coherence Algorithms Used

For the purposes of this study, we focus on the memory coherence handling approaches implemented by the memory allocation module and the memory mapping module in the experiments done by Kai Li [6]. These approaches rely on a DSM architecture where each processor consists of a *page table* data structure, which consists of 3 kinds of information about pages in the processor's local memory [9]: (1) the accessibility of the page *(access)*, (2) the processor number that have copies of that page *(copy-set)*, and (3) a lock for synchronization *(lock)*. For space efficiency, this page table has been compacted by the use of bit vectors to represent the data [1]. They are discussed next in Subsections 3.3, 3.4, and 3.5, according to the descriptions in [9].

3.3. Centralized Manager Algorithm

The memory managers of one of the processors is assigned as the central memory manager. The processor with the central memory manager has two tables: `Info` and `PTable`. The other processors only have `PTable`.

`Info` has a set of 3-tuple data for each page. The constituents of the 3 tuples for any page p are: (1) an *owner* field that stores the name of the processor which had the most recent write access to p, (2)

a *copy set* field that has all processors with copies of p, and *lock* field for synchronizing requests to p. `PTable` has accessibility information about a page in the local processor. It has two fields: *access* and *lock*.

By this design, therefore, only one manager knows the owner of a page. Whenever any processor requests a read copy of page p, the owner of p sends a copy of p to the requesting processor.

In the centralized algorithm the successful writer to a page always has ownership of the page. On finishing a read or write request, a processor sends a confirmation message to the manager to indicate completion of the request. (Later optimization to the protocol, eliminates the need to send confirmations messages.)

As we have mentioned, `Info` table and `PTable` have page-based locks. They are used to synchronize the local and remote page faults. At the local-level, within a processor, if there exists multiple processes waiting for the same page, the locking mechanism prevents the processor from sending more than one request. At the remote-level, if a request for a page arrives and the processor is accessing the page table entry, the locking mechanism enters the request in a queue and holds it until the entry is released. In this manner, the manager is able to synchronize multiple requests from different processors.

3.4. Fixed Distributed Manager Algorithm

The main bottleneck of the previous approach is that all managerial tasks are centralized.

In this light, the fixed distributed manager algorithm distributes the managerial task among the processors. It does so by assigning each processor a predetermined subset of the pages to manage. This distribution can be done using mapping by a number of ways. One method is to have an even distribution of the pages, using a hashing function, $H(p) = (p \div s) * mod(N)$ where p is the page number, N is the total number of processors, and s is the number of pages per segment. Other methods can also allow clients to provide their own mapping functions.

In this algorithm, when a fault occurs on page p at processor x, x requests processor $H(p)$ (which has the true page), and then continues as in the centralized manager algorithm. Overall, although it is difficult to find a fixed distribution function that will fit all applications, the distributed version has shown to be superior to the centralized version when there is a high page fault rate in the parallel algorithm [9].

3.5. Dynamic Distributed Manager Algorithm

The dynamic distributed manager algorithm introduces the notion of a "probable" owner of a page. It keeps track of all the pages in each processors's page table using an additional field called `probOwner` in each page entry. The value of this field can be "true", which would indicate that this processor is the actual owner of the page, or it can be "probowner", which indicates that the the processor is probably an owner of the page. This approach is at least guaranteed to provide the start of a sequence of processors among which one is the true owner. Whenever a processor receives an invalidation request, disowns a page, or passes a page fault request.

3.6. IVY Speedup

Figure 7, shows the speedups obtained by IVY for some parallel programs [6].

As we can see, the speedups were almost linear with the matrix multiply and the traveling salesman programs. In their implementation, the matrix multiply program assumes that the multiplicand matrices are in the same processor at the start and are paged to the other processors on demand. For the traveling salesman problem, a process is created for each processor, which executes the branch-and-bound algorithm on a branch obtained from the shared virtual memory. The processes continue to run parallelly until a the tour (which constitutes visiting each node in a graph using the minimum cost). The speedups for these algorithms could be attributed to the high degree of localized computations in these programs.

However, for the dot product program, the speedup was low. This can be explained by the fact that the matrix elements are referenced only once in the program, hence communication costs of the program exceeds the computation cost of the program.

4. DASH

4.1. Overview of the System

The Stanford DASH multiprocessor [13] was designed at Stanford University and became operational in 1991. It implements a DSM architecture

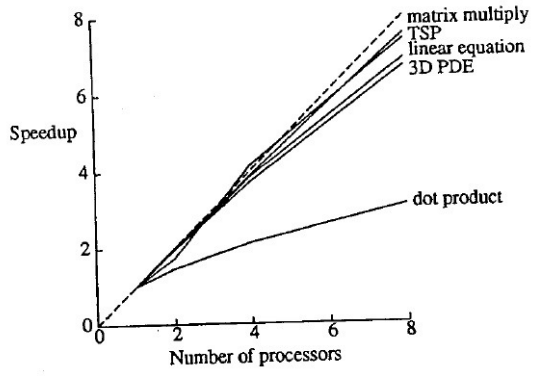

Figure 7: Speedups of the IVY system [6].

that supports cache coherence with distributed directories. In this section, we first look at a memory coherence mechanism (Subsection 4.2) that formed the foundation of the protocols used by DASH. These protocols are subsequently explained in Subsection 4.3.

4.2. Directory Based Coherence

These schemes are similar to the basis of the ones mentioned in the previous section. Here we also have processors relying on a data structure, called a *directory* to find the cached locations (locations of cache blocks) of the copies of a page. It assumes there is a single monolithic directory that consists of all the information. Each cache block operates with 3 states [3]—1) *Invalid*: The cache block cannot be used by the processor, 2) *Shared*: The cache block is readable but has copies in other processors (in which case, the directory entry for this block contains a list of those other processors - the block is readable only), 3) *Exclusive*: the block only exists in the cache of this processor and is writable.

As done in the protocols discussed in the previous section, this protocol ensures cache consistency by invalidating all cache blocks that have a copy of a cache block, before assigning a cache block to the exclusive state. Thus like the protocols in Section 3, these protocols also avoid the pitfall of the snoopy protocols (see 2.2), all processors had to be queried about whether they contained a copy of data via broadcast.

The processing of the requests are serialized at the directory, i.e. the directory can be accessed by one processor at a time. This synchronous approach avoids race conditions, when processors write to the same block. The problem with this approach, however, is that it is too conservative and it takes away any opportunity for parallelization. For instance, it may happen that two processors request to read the directory to write to different cache blocks. In such a scenario, their requests can be processed in-parallel.

Another problem with this scheme was having a single directory. This introduced the single-point-of-failure bottleneck. Also, it was not scalable, as it was not feasible to have a directory large enough to accommodate information of cache blocks in a network of many processors. As a consequence, this idea didn't get traction [3] and the next logical step was to distribute the directory among the processors.

4.3. Distributed Directory Based Coherence

A typical cache-coherent DSM multiprocessor architecture with distributed directories is shown in Figure 8. The directory information is the same as those described in Subsection 4.2.

This protocol implements a type of message passing approach, where messages are sent among the requesting processor node (the local node), the node containing the address of the block that the local node desires to read or write (the home node), and the node that contains the cache block in the exclusive state (the remote node). A demonstration of this protocol is shown in Figure 10([3]).

The problem with this protocol is that satisfying a remote request requires at least two messages: from the local to the home node to request a cache block and then from the home to the local node to reply with the data. First, let us consider the case of a remotely cached data item in the exclusive state. Here at least three messages are required (local to home, home to remote, and remote to local). Also, if the request entails the invalidation of a heavily shared object, far greater number of messages will need to be passed. Since the architecture forces all these operations to perform atomically, the likelihood of a deadlock becomes high.

The DASH architects wanted to avoid this deadlock bottleneck by introducing some kind of serialization as was done in the snoopy coherence schemes, keeping in mind that a complete import of the bus based scheme would mandate a sacrifice in

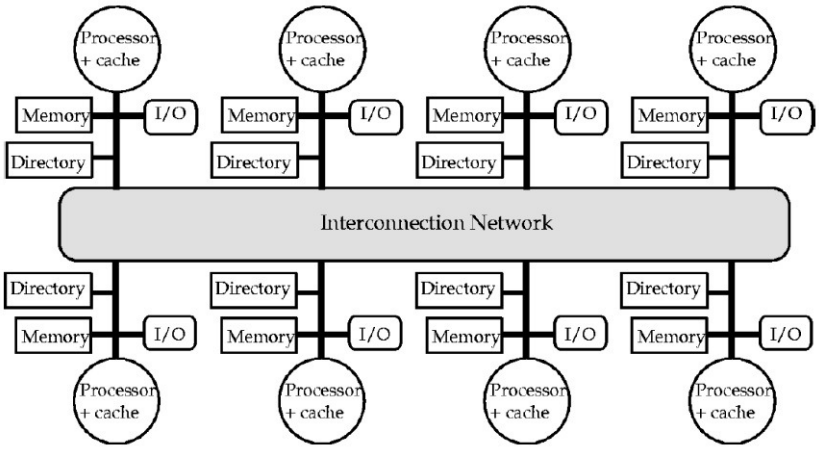

Figure 8: A distributed memory multiprocessor using distributed directories [3].

scalability. They thus designed a scheme where each node of a DSM network consisted of four processors. Although this was a moderate compromise on the performance of the system, it helped to mitigate the deadlock problem. (The performance was later improved using a two-processor node design.) While the nodes were connected to each other by an interconnection network, the processors in each node are connected to each other via bus. This prototype has a total of 16 such nodes, tallying the total count of processors to 64. This is shown in Figure 9.

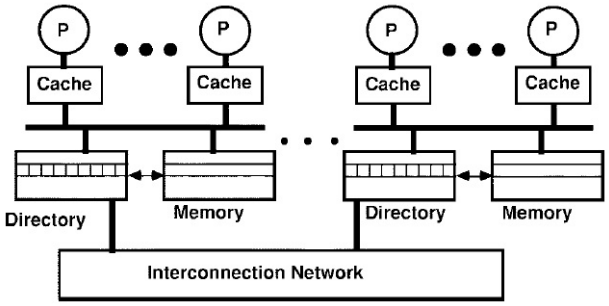

Figure 9: The Stanford DASH architecture [3].

4.4. DASH Speedup

Figure 11 [3], shows the speedups obtained by DASH for some parallel programs from the SPLASH [14] Suite. The applications include partial differential equation solvers, like Ocean, to applications using a variety of *n*-body modeling techniques— BarnesHut, fast multipole method (FMM), and radiosity. As can be seen from the figure, the speedups were significant.

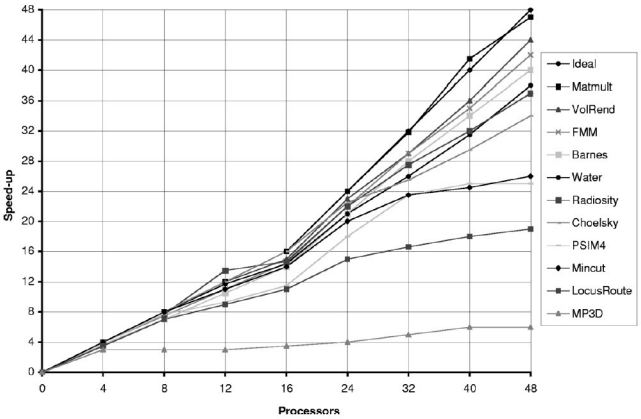

Figure 11: Speedups for DASH [3].

5. TCC

5.1. Design Motivations

A Transactional memory Coherence and Consistency (TCC) model, a novel shared memory model, was presented in [15]. TCC aims to take advantage of the interprocessor bandwidth and thereby simplify the protocols used to manage communications and synchronization between the processors. In particular, TCC takes advantages of both message passing systems and shared-memory model systems, while avoiding the pitfalls of both—

1) Message passing systems provide a challenging programming model, making the programmers explicitly decide how to distribute the data items of

Processor A Program action	Processor A System Action	Processor B Program Action	Processor B System Action
x = 1	Send ownership request to home node. Home node sends invalidations to all processors that cache x.		
			Receive invalidation, eliminate x from cache, and acknowledge invalidation.
	A has an exclusive copy of x; store1 into x.		
		t = x +1	Send message to home node, which forwards request to A for value.
	Receives request from home node; changes cache state to shared; sends value to both B (for its use) and to the home node (for writing back into memory).		
			Obtain the value of x from A, and change state of x to shared.
		x = t	Send message to home node requesting ownership of x. Home node sends invalidation to A.
	Receive invalidation, eliminate x from cache, and acknowledge invalidation.		
			B has an exclusive copy of x; store t into x.

Figure 10: A demonstration of the distributed directory based cache coherence protocol [3].

the parallel program over the system, although it performs the synchronization and communications implicitly.

2) Shared memory model systems simplifies the programming model by providing a common coherent view of the system, but adds additional hardware to the underlying system, which can make programming the hardware complicated [16]. As such hardware are required to track the program data anywhere in the system, performance of these systems have been an issue. As was discussed earlier, the serialization of various communication events to the granularity of individual read and write requests further slows down the operation of such systems.

TCC reduces the need for hardware to support frequent, latency-sensitive coherence requests for individual cache lines. It incorporates implicit message synchronization as message passing protocols.

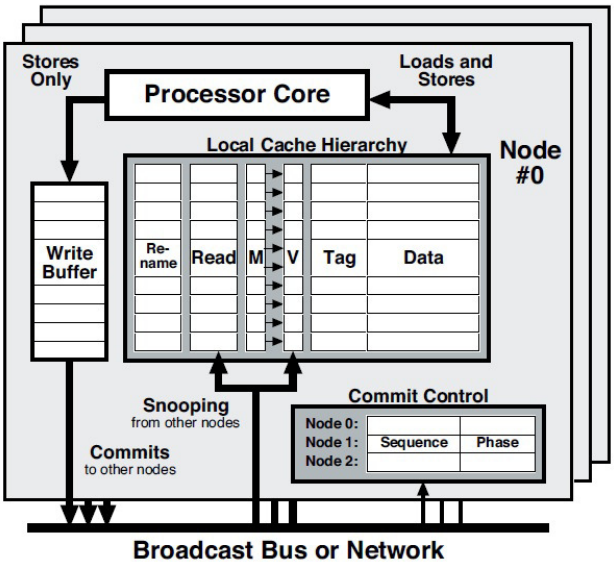

Figure 12: A 3-node TCC system [15].

5.2. *Operation Mechanism*

A 3-node TCC system is shown in Figure 12. As can be seen the three nodes can be connected with each other using a bus or a network.

TCC's design is based on continually performing speculative *transactions*. A transaction is defined as a sequence of instructions that is guaranteed to execute and complete only as an atomic unit. After the completion of a transaction it yields a block of writes committed to the shared memory (only as an atomic unit). The hardware then gives a system-wide notification for permission to commit its writes. On receiving this permission, the processor broadcasts the all writes for the entire transaction as one large packet to the rest of the system by taking advantage of the interconnect bandwidths provided by the system. The broadcast is unordered. However, the individual stores for the *same* commits are separated and reordered.

Snooping, as done in message passing systems, is also implemented by other processors on the stores. In addition, the combination of all writes of a whole transaction and passing them as a large packet helps

with reducing latency — it requires lesser number of messages to transfer for synchronization of the data. As a consequence, the speed-ups obtained by TCC for several recent benchmark parallel algorithms was shown to be close to the optimal linear.

The benefit of TCC over previous memory consistency and coherence models like sequential and relaxed consistency models [2] is that it just imposes a sequential ordering of between transaction commits, rather than individual loads and stores. This is made possible because in TCC, the stores are buffered and kept within the processor for the entire length of the transaction. This also ensures atomicity of the transaction.

6. Conclusion

The quest for designing efficient coherence protocols for the shared memory model remains ongoing. It is still unclear what combinations of hardware and software solutions would give the best outcomes of such protocols. In this paper, we have discussed some memory coherence approaches that were used in some of the more successful shared memory multiprocessors in terms of their speedups with benchmark parallel program applications.

References

[1] K. Li, Shared Virtual Memory on Loosely Coupled Multiprocessors, Ph.D. thesis, New Haven, CT, USA, 1986.

[2] S. V. Adve, K. Gharachorloo, Shared memory consistency models: A tutorial 29 (1996) 66–76.

[3] J. Hennessy, M. Heinrich, A. Gupta, Cache-coherent distributed shared memory: perspectives on its development and future challenges, Proceedings of the IEEE 87 (1999) 418–429.

[4] Ibm pseries 690, http:// www-03.ibm.com/ systems/power/hardware/ pseries/highend/p690/index.html, 2005.

[5] B. Nitzberg, V. Lo, Distributed shared memory: a survey of issues and algorithms, Computer 24 (1991) 52–60.

[6] K. Li, Ivy: A shared virtual memory system or parallel computing, in: Proceedings of the 1988 International Conference on Parallel Processing, Pennsylvania State University Press, 1988, pp. 94–101.

[7] M. P. Herlihy, B. Liskov, A value transmission method for abstract data types, ACM Trans. Program. Lang. Syst. 4 (1982) 527–551.

[8] M. L. Powell, B. P. Miller, Process migration in (demos/mp), in: Proceedings of the Ninth ACM Symposium on Operating System Principles (SOSP 1983), pp. 110–119.

[9] K. Li, P. Hudak, Memory coherence in shared virtual memory systems, ACM Trans. Comput. Syst. 7 (1989) 321–359.

[10] D. A. Patterson, Lecture notes on snooping vs. directory based coherency, 1996.

[11] Apollo archives, http://jim.rees.org/apollo-archive/, 1998.

[12] P. Leach, P. Levine, B. Douros, J. Hamilton, D. Nelson, B. Stumpf, The architecture of an integrated local network, IEEE Journal on Selected Areas in Communications 1 (1983) 842–857.

[13] D. Lenoski, J. Laudon, K. Gharachorloo, W. D. Weber, A. Gupta, J. Hennessy, M. Horowitz, M. S. Lam, The stanford dash multiprocessor, Computer 25 (1992) 63–79.

[14] J. P. Singh, W. Weber, A. Gupta, SPLASH: Stanford Parallel Applications for Shared-memory, Technical Report, Stanford, CA, USA, 1992.

[15] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, K. Olukotun, Transactional memory coherence and consistency, in: Proceedings of the 31st Annual International Symposium on Computer Architecture, ISCA '04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 102–.

[16] A. Charlesworth, Starfire: Extending the smp envelope, IEEE Micro 18 (1998) 39–49.