International Journal of Advanced Computer Research (ISSN (print): 2249-7277 ISSN (online): 2277-7970)
Volume-2 Number-4 Issue-6 December-2012

Automated Verification of Memory Consistencies of DSM System on Unified Framework

Pankaj Kumar[1], Durgesh Kumar[2]
[1]Assistant Professor, SRMCEM, Lucknow
[2]Research Scholar, CMJ University, Meghalaya

Abstract

The consistency model of a DSM system specifies the ordering constraints on concurrent memory accesses by multiple processors, and hence has fundamental impact on DSM systems' programming convenience and implementation efficiency. We have proposed the structural model for automated verification of memory consistencies of DSM System. DSM allows processes to assume a globally shared virtual memory even though they execute on nodes that do not physically share memory. The DSM software provide the abstraction of a globally shared memory in which each processor can access any data item without the programmer having to worry about where the data is or how to obtain its value In contrast in the native programming model on networks of workstations message passing the programmer must decide when a processor needs to communicate with whom to communicate and what data to be send. On a DSM system the programmer can focus on algorithmic development rather than on managing partitioned data sets and communicating values. The programming interfaces to DSM systems may differ in a variety of respects. The memory model refers to how updates to distributed shared memory are rejected to the processes in the system. The most intuitive model of distributed shared memory is that a read should always return the last value written unfortunately the notion of the last value written is not well defined in a distributed system.

Keywords

Distributed Shared Memory (DSM) system, automated verification of processor Consistency (PC)

1. Introduction

Despite the advances in processor design, users still demand more and more performance. Eventually, single CPU technologies must give way to multiple processors parallel computers: it is less expensive to run 10 inexpensive processors cooperatively than it is to buy a new computer 10 times as fast. This change is usual, and has been realized to some extent in the specialization of subsystems like bus mastering drive controllers. However, the need for additional computational power has thus far rested solely on advances in CPU technologies [2, 4, 9]. In parallel systems, there are two kinds of fundamental models:

1. Shared memory
2. Distributed Memory.

From a programmer's perspective, shared memory computers (symmetric multiprocessor), while easy to program, are difficult to build and aren't scalable to beyond a few processors. Distributed Memory (Message passing) computers, while easy to build and scale, are difficult to program. In some sense, shared memory model and message passing model are equivalent.

One of the solutions to parallel systems is Distributed Shared Memory (DSM) whose memory is physically distributed but logically shared. DSM appears as shared memory to the applications programmer, but relies on message passing between independent CPUs to access the global virtual address space. Both hardware and software implementations of DSM have been proposed.

2. Related and Previous Works

A lot of work has been done to improve the system performance and effectiveness of distributed shared share memory and still work is under progress some of the details are as given

A. Distributed Shared Memory

DSM is an architectural approach designed to overcome the scaling limitations of symmetric shared memory multiprocessors while retaining a shared memory model for communication and programming. DSM multiprocessors achieve this by using a memory that is physically distributed but logically implements a single shared address space, allowing the processor to communicate through, and share the contents of, the entire memory [1, 3, 11]. DSM multiprocessors have the same basic organization as the machines as shown in the Figure 1. Sharing data is an essential requirement of any distributed system. Distributed system it stands for a multi-computer architecture in which each node is an independent machine connected to each other through a network. Sharing data in a multi-processor architecture is relatively easier, since all of the nodes share the same system bus and hence have a uniform view of the physical memory.

On the other hand a multi-computer system does not enjoy such hardware privileges. So sharing data becomes a problem which has to be tackled in the software (either inside the Operating System or as a user-level application) and not in hardware as in multi-processor systems. Traditional methods of data sharing viz. message passing via sockets are not appealing from a programmer's perspective, in which he or she has to explicitly take care of the networking issues. A DSM provides an abstraction to the programmer of a uniform shared memory located across different machines [5, 6, 8]. Since a DSM system involves moving of data from one node to another which are on typical networks, performance is an important criterion in the design of a DSM system. Just as is the case with multi-processor systems, since same copies of data might reside on different nodes, consistency between these copies is also another major issue.

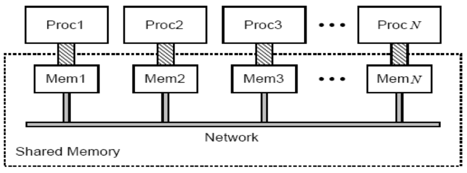

Fig. 1 Distributed Shared Memory

DSM systems can be classified into three broad categories.

- Page-based DSM – in which the unit of data sharing is a memory page.
- Shared variable based DSM – in which the unit of data sharing is a variable.
- An object based DSM – in which the unit of data sharing is an object.

The choice of objects as units of granularity over a page or shared variables is because of the modularity and flexibility offered by objects. Moreover, objects eliminate false sharing. Another reason is that integration of object based DSM with the object oriented languages is easy to achieve. The design also provides flexibility in terms of consistency models used by allowing the user applications to specify under what consistency scheme they want a particular object to be shared.

B. Issues in DSM Design

Any DSM system has to rely on the message passing technique across the network for data exchange between two computers. The DSM has to present a uniform global view of the entire address space (consisting of physical memories of all the machines in the network) to a program executing on any machine. A DSM manager on a particular machine would capture all the remote data accesses made by any process running on that machine[7, 10]. A design of a DSM would involve making following choices

- Where, with respect to the Virtual Memory Manager does the DSM operate?
- What kind of consistency model should the system provide?
- What should be the granularity of the shared data?
- What kind of naming scheme has to be used to access remote data?

3. Unified Framework of Memory Consistency

A. *DSM Consistency Model*

The consistency model of a DSM system specifies the ordering constraints on concurrent memory accesses by multiple processors, and hence has fundamental impact on DSM systems' programming convenience and implementation efficiency. DSM allows processes to assume a globally shared virtual memory even though they execute on nodes that do not physically share memory. The DSM software provide the abstraction of a globally shared memory in which each processor can access any data item without the programmer having to worry about where the data is or how to obtain its value In contrast in the native programming model on networks of workstations message passing the programmer must decide when a processor needs to communicate with whom to communicate and what data to be send. For programs with complex data structures and sophisticated parallelization strategies this can become a daunting task. On a DSM system the programmer can focus on algorithmic development rather than on managing partitioned data sets and communicating values. The programming interfaces to DSM systems may differ in a variety of respects. The memory model refers to how updates to distributed shared memory are rejected to the processes in the system. The most intuitive model of distributed shared memory is that a read should always return the last value written unfortunately the notion of the last value written is not well defined in a distributed system.

The memory consistency model can be categorized into parts one which is based on read and write memory operation called as uniform model and the other which is based on synchronization operation also called hybrid model. The synchronization operations are mapped to corresponding operations provided by concurrency control. Parallel computers can be classified by various aspects of their architecture. Mainly parallel computers are distinguished by the way the processors are connected with the memory. Parallel computer architecture may be designed using shared memory or distributed memory. *Shared memory architecture* is very much authentic in parallel computers due to ease of its programming. Programming a shared memory computer is very convenient due to the fact that all data are accessible by all processors, such that there is no need to copy data. Furthermore the programmer does not have to care for synchronization, since this is carried out by the system automatically. However, it is very difficult to obtain high levels of parallelism with shared memory machines; most systems do not have more than 64 processors. This limitation stems from the fact, that a centralized memory and the interconnection network are both difficult to scale once built. This can be improving by using *distributed memory architecture* in which each processor has its own memory. There is no common address space, i.e. the processors can access only their own memories. Communication and synchronization between the processors is done by exchanging messages over the interconnection network .To combine the advantages of the architectures described above, ease of programming on the one hand, and high scalability on the other hand, a third kind of architecture has been established *virtual shared memory* (VSM). Here, each processor has its own local memory, but, contrary to the distributed memory architecture, all memory modules form one common address space, i.e. each memory cell has a system-wide unique address. In order to avoid the disadvantage of shared memory computers, namely the low scalability, each processor uses a cache, which keeps the number of memory access conflicts and the network contention low. Besides being referred to as Virtual Shared Memory, such architectures have also been referred to as Distributed Shared Memory (DSM), Shared Virtual Memory (SVM), and Distributed Virtual Shared Memory (DVSM) and so on. Arbitrary use of such terms can cause confusion, so an attempt will be made to define the commonly used terms more precisely to characterize different types of architectures by using *shared memory abstraction*. Shared memory machines are convenient for programming but do not scale beyond tens of processors.

The *Data Diffusion Machine* (DDM) overcomes this problem by providing a shared memory abstraction on top of a distributed memory machine. A DDM appears to the user as a conventional shared memory machine but is implemented using distributed memory architecture. In designing a parallel machine to support virtual shared memory, an important consideration is whether the main memory should be conventional or (set-) associative. This is the main distinction between so-called CCNUMA and COMA architectures. A related issue is what additional caches are needed in the memory hierarchy.

Parallel architectures are commonly classified according to their control organization as either MIMD (Multiple Instructions & Multiple Data Stream) or SIMD (Single Instruction & Multiple Data Stream). MIMD machines have a distributed control organization, with every PE having a control unit capable of sequencing an independent computation. In contrast, SIMD machines have a centralized control organization, where the PEs shares a control stream broadcast by a single control unit. The SIMD model has a number of disadvantages that have caused it to be viewed as a special purpose, and even obsolete, model for general purpose parallel computation. Chief among these disadvantages is the inflexible control organization. A model, called *shared control*, overcomes the inefficiency of SIMD machines on control parallel applications by sharing the control organization at a fundamentally different level the instruction (or function) level. Thus, all the PEs executing the same instruction, but not necessarily the same control thread, receives their control from the same control unit concurrently.

Now it is clear that in parallel computer architecture memory plays a great role whether it is SIMD or MIMD parallel computer. Memory architecture in parallel computer can be implemented by shared memory, distributed memory or using the combination of both i.e. distributed shared memory. We have proposed the structural model for automated verification of memory consistencies of DSM System. Two type of memory consistency model are there:

The STRUCTURAL UNIFORM MODEL consider only read & write memory operation to define consistency condition. It is the combination of strong & relaxed memory model. The following memory consistency models have been taken for the proposed structural uniform model:

- Atomic consistency (AC)
- Sequential consistency (SC)
- Causal consistency (CC)
- Processor consistency (PC)
- PRAM
- Cache consistency
- Slow memory

The first two AC & SC is the strong model whereas the other one are relaxed consistency. The PRAM model is evolved by combining the processor consistency as well as causal consistency and slow memory model is the combination of PRAM and Cache Coherence. If we follow the path from Top to Bottom, The sequential consistency is evolved from the atomic consistency so it inherits some property of atomic consistency.

The processor consistency and the causal consistency i.e. defined by the sequential consistency. The PRAM model is evolved by combining the processor consistency as well as causal consistency. The cache consistency is defined by processor consistency and based on cache coherency. Slow memory model is the combination of PRAM and Cache Consistency.

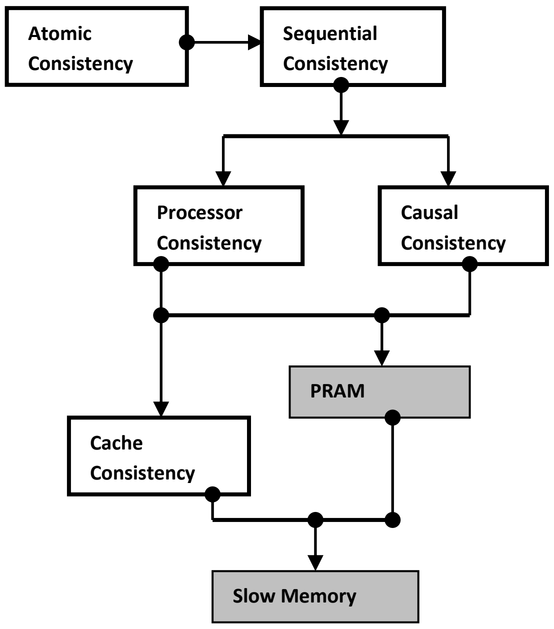

Fig. 2: Structural Uniform Model

4. Verifying DSM Consistency

We frequently read of incidents where some failure occurred due to error in a hardware or software system. For reliable systems, it is very important to develop methods for correctness of such systems. The principal validation methods for complex systems are simulation, testing, deductive verification, and model checking .Simulation is performed on an abstraction or a model of the system, testing is performed on the actual product. In both cases, we will give certain inputs and observe corresponding outputs. Deductive verification consists of axioms and proof rules to prove the correctness of systems. The importance of deductive verification is widely recognized by computer scientists. Deductive verification is a time consuming process that can be performed only by experts who are educated in logical reasoning and have considerable experience. Consequently, use of deductive verification is rare. An advantage of deductive verification is that it can be used for reasoning about infinite state systems. Model checking is a technique for verifying finite state concurrent systems. One benefit of this restriction is that verification can be performed automatically.

The procedure normally uses an exhaustive search of the state space of the system to determine if some specification is true or not. The procedure will always terminate with yes/no answer. Model checking consists of modeling, specification and verification steps. An exciting new research direction attempts to integrate deductive verification and model checking, so that the finite states of complex systems can be verified automatically. As the need for more computing power demanded by new applications constantly increases, systems with multiple processors are becoming a necessity. The gap between processor and memory speed is apparently widening, and that is why the memory system organization becomes one of the most critical design decisions to be made by computer architects. According to the memory system organization, systems with multiple processors can be classified into two large groups: *shared memory systems* and *distributed memory systems*. In a shared memory system (SMS) (often called a tightly-coupled multiprocessor), a single global physical memory is equally accessible to all processors. The advantage of SMS is very simple and easy to program. However, they typically suffer from increased contention in accessing the shared memory, especially in single bus topology, which limits their scalability.

In addition to that, the design of the memory system tends to be more complex. A distributed memory system (often called a multicomputer) consists of a collection of autonomous processing nodes, having an independent flow of control and local memory modules. Communication between processes residing on different nodes is achieved through a message passing model, via a general interconnection network. Such a programming model imposes significant burden on the programmer, and induces considerable software overhead. On the other hand, these systems are claimed to have better scalability and cost-effectiveness. A distributed shared memory (DSM) tries to combine the better of these two approaches.

A DSM system logically implements shared memory model on a physically distributed memory system. This approach hides the mechanism of communication between remote sites from the application writer, so the ease of programming and the portability typical for shared memory systems, as well as the scalability and cost effectiveness of distributed memory systems, can be achieved with less engineering effort. In this work, we formally verified some of the weak consistency properties of distributed shared memory model.

5. Automated Dynamic Verification

Based on the definitions there is need to devise a framework that breaks the verification process into three invariants that correspond to the three steps necessary for processing a memory operation. First, memory operations are read from the instruction stream in program order and executed by the processor. At this point, operations impact micro architectural state but not committed architectural state. Second, operations access the (highest level) cache in a possibly different order.

Consistency models that permit reordering of cache accesses enable hardware optimizations such as write buffers. Sometime after accessing the cache, operations perform and become visible in the globally shared memory. This occurs when the affected data is written back to memory or accessed by another processor. At the global memory, cache orders from all processors are combined into one global memory order. The basic idea of the presented framework is to automatically verify an invariant for every step to guarantee it is done correctly and thus verify that the processing of the operation as a whole is error-free. The three invariants *(Uniprocessor Ordering, Allowable Reordering*, and *Cache Coherence)* described below are sufficient to guarantee memory consistency.

•for X and Y of type OPx and OPy, it is true that if X <p Y and there exists an ordering constraint between OPx and OPy, then X <m Y, and
•a load Y receives the value from the most recent of all stores that precede Y in either the global order <m or the program order <p.

Uniprocessor Ordering: On a single-threaded system, a program expects that the value returned by a load equals the value of the most recent store in program order to the same memory location. In a multithreaded system, obeying Uniprocessor Ordering means that every processor should behave like a uniprocessor system unless a shared memory location is accessed by another processor.

Allowable Reordering: To improve performance, microprocessors often do not perform memory opera operations in program order. The consistency model specifies which reordering between program order and global order are legal. For example, SPARC’s Total Store Order allows a load to be performed before a store to a different address that precedes it in program order, while this reordering would violate SC. In our framework, legal reordering is specified in the ordering tab.

Cache Coherence: Coherence defines the behavior of reads and writes to the same memory location. In DSM systems however there are two or more processors working at the same time, so there is the possibility that the processors will all want to process the same value at the same time. Provided none of the processors updates the value then they can share it indefinitely, but as soon as one updates the value, the others will be working on an out-of-date copy. Some scheme is required to notify all processors of changes to shared values; such a scheme is known as a "memory coherence protocol. There are two type coherence protocols one is write-invalidate and other is write-update. Write-invalidate protocol invalidates all write operation if any processor wants to writes. Write-update protocol updates all writes to all processor if any processor wants to write.

6. Uniprocessor Ordering Checker

Uniprocessor Ordering is trivially satisfied when all operations execute sequentially in program order. Thus, *Uniprocessor Ordering* can be dynamically verified by comparing all load results obtained during the original out-of-order execution to the load results obtained during a subsequent sequential execution of the same program. Because instructions commit in program order, results of sequential execution can be obtained by replaying all memory operations when they commit. Replay of memory accesses occurs during the *verification stage*, which we add to the pipeline before the retirement stage. During replay, stores are still speculative and thus must not modify architectural state. Instead they write to a dedicated *verification cache* (VC). Replayed loads first access the VC and, on a miss, access the highest level of the cache hierarchy (bypassing the write buffer). The load value from the original execution resides in a separate structure, but could also reside in the register file. In case of a mismatch between the replayed load value and the original load value, a *Uniprocessor Ordering* violation is signaled. Such a violation can be resolved by a simple pipeline flush, because all operations are still speculative prior to verification. Multiple operations can be replayed in parallel, independent of register dependencies, as long as they do not access the same address.

In consistency models that require loads to be ordered (i.e., loads appear to have executed only after all older loads performed), the system speculatively reorders loads and detects load-order mis-speculation by tracking writes to speculatively loaded addresses. This mechanism allows stores from other processors to change any load value until the load passes the verification stage, and thus loads are considered to perform only after passing verification. To prevent stalls in the verification stage, the VC must be big enough to hold all stores that have been verified but not yet performed.

In a model that allows loads to be reordered, such as RMO, no speculation occurs and the value of a load cannot be affected by any store after it passes the execution stage. Therefore a load is considered to perform after the execution stage in these models, and replay strictly serves the purpose of verifying *Uniprocessor Ordering*. Since load ordering does not have to be enforced, load values can reside in the VC after execution and be used during replay as long as they are correctly updated by local stores. This optimization, which has been used in automated verification of single-threaded execution, prevents cache misses during verification and reduces the pressure on the cache.

7. Allowable Reordering Checker

DVMC verifies *Allowable Reordering* by checking all reordering between program order and cache access order against the restrictions defined by the ordering table. The position in program order is obtained by labeling every instruction X with a sequence number, seqX, that is stored in the ROB during decode. seqX equals X's rank in program order. The rank in perform order is implicitly known, because we verify *Allowable Reordering* when an operation performs. The *Allowable Reordering* checker uses the sequence numbers to find reordering and check them against the ordering table. For this purpose, the checker maintains a counter register for every operation type OPx (e.g., load or store) in the ordering table. This counter, max {OPx}, contains the greatest sequence number of an operation of type OPx that has already performed. When operation X of type OPx performs, the checker verifies that seqX > max {OPy} for all operation types OPy that have an ordering relation OPx<cOPy according to the ordering table. If all checks pass, the checker updates max {OPx}. Otherwise an error has been detected.

It is crucial for the checker that all committed operations perform eventually. The checker can detect lost operations by checking outstanding operations of all operation types OPx, with an ordering requirement OPx<cOPy, when an operation Y of type OPy performs. If an operation of type OPx older than Y is still outstanding, it was lost and an error is detected. In our implementation, we check outstanding operations before Member instructions by comparing counters of committed and performed memory accesses. To prevent long error detection latencies, artificial Members are injected periodically. Member injection does not affect correctness and has negligible performance impact. The implementation of an *Allowable Reordering* checker requires three small additions to support architecture specific features: dynamic switching of consistency models, a FIFO queue to maintain the perform order of loads until verification, and computation of member ordering requirements from a bitmask.

8. Conclusion

The main focus of our paper is on the memory architecture of parallel computer. In parallel computer architecture where lots of operations are performed simultaneously, so it become necessary that memory operation must be ordered. All processor having its own local memory like distributed memory architecture that is accessible only to that processor, requires very less memory ordering. But system like distributed Shared Memory system where processors have common memory with single address space, allows all processor to access entire memory.

As DSM system allows multiple processors to access memory location simultaneously, so it requires an abstract model for memory operations that allow using memory location correctly and maintaining memory consistency. The memory consistency plays important role in DSM system because it specifies order of memory operation. The paper has addressed several important issues for distributed shared memory system. But it mainly concentrates on designing of distributed shared memory framework for memory consistency maintenance. The framework is designed with the help of memory consistency model and memory coherence.

References

[1] Paul Krzyzanowski "Distributed Shared Memory and Memory Consistency Models" Rutgers University – CS 417: Distributed Systems ©1998, 2001.

[2] Lisa Higham, Jalal Kawash and Nathaly Verwaal, "Define and Comparing Memory Consistency Model" ©1997 ISCA, Proceeding of PDCS'97.

[3] Abdelfatah Aref Yahya and Rana Mohamad Idrees Bader "Distributed Shared Memory Consistency Object-based Model", Journal of Computer Science 3 (1): 57-61, 2007 ISSN1549-3636© 2007 Science Publications.

[4] Changhun Lee "Distributed Shared Memory" Proceedings on the 15th CISL Winter Workshop Kushu, Japan ¢ February 2002.

[5] Sarita V. Adve, Member, IEEE, Vijay S. Pai, Student Member, IEEE, and Parthasarathy Ranganathan, Student Member, IEEE "Recent Advances in Memory Consistency Models for Hardware Shared Memory Systems" proceedings Of The Ieee, Vol. 87, No. 3, March 1999.

[6] Albert Meixner and Daniel J. Sorin "Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures" Duke University, Department of Electrical and Computer Engineering, Technical Report #2006-1, April 18, 2006.

[7] Alba Cristina Magalhães Alves de Melo "Defining Uniform and Hybrid Memory Consistency Models on a Unified Framework" Proceedings of the 32nd Hawaii International Conference on System Sciences -1999 IEEE.

[8] Jason F. Cantin, Student Member, IEEE, Mikko H. Lipasti, Member, IEEE, and James E. Smith, Member, IEEE "The Complexity of Verifying Memory Coherence and Consistency" IEEE Transactions On Parallel And Distributed Systems, Vol. 16, No. 7, July 2005.

[9] Robert C. Steinke and Gary J. Nutt "A Unified Theory of Shared Memory Consistency" Journal of the ACM, Vol. V, No. N, Month 20YY, Pages 1–47 2002.

[10] Jalal Y. Kawash" Limitations and Capabilities of Weak Memory Consistency Systems" Ph.D. Thesis Calgary, Alberta January, 2000.

[11] Benny Wang-Leung Cheung, Cho-Li Wang and Francis Chimoon Lau, "Migrating-Home Protocol for Software Distributed Shared Memory", Journal of Information Science and Engineering, 2000.

Durgesh kumar is a research scholar in computer science in CMJ University Meghalaya. He has done MCA from UPTU Lucknow. Currently he is working in Aditya Birla Group.

Dr. Pankaj Kumar is currently working as Assistant Professor in Sri Ramswaroop college of engineering & Management Lucknow. He received his PhD. Degree in computer application in 2011 and MCA degree in 2001.His research interests are Parallel Computing, Memory Architecture of Parallel Computer and Distributed Computing. Many of the valuable research papers of Mr. Pankaj Kumar have been published in various national/international journals and IEEE proceeding publication in the area of "Parallel Computing". He is life member of Computer Society of India (CSI) and professional member of International Association of Engineers (IAENG), International Association of Computer Science and Information Technology (IACSIT) and Internet Society (ISOC).