LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

An Incessantly Coherent Cache Scheme for Shared Memory Multithreaded Systems

Comutation Structure Group Memo 356
September 15,1994

S. K. Nandy
Ranjani Narayan

To appear in *Proceedings of The First International Workshop on Parallel Processing*, **Bangalore, India, December 26–31 1994**

This report describes research done by the authors at the Laboratory for Computer Science, Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310. Financial support for the first author was provided by Indo-U.S. Science and Technology Fellowship Program.

An Incessantly Coherent Cache Scheme for Shared Memory Multithreaded Systems *

S. K. Nandy and Ranjani Narayan
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts, MA 02139
e-mail: nandy@theory.lcs.mit.edu

Abstract

An incessantly coherent cache consistency protocol is proposed in this paper. The protocol supports limitless sharing and obviates the need to invalidate shared cache lines by automatically self invalidating cache lines after the expiry of its lifetime. *The protocol mandates that all writes be performed at the home locations only. The worst case performance of this protocol bears the potential to perform as good as any other directory based protocol. It relieves the network of the additional traffic due to invalidations that are in vogue in any of the existing directory based cache coherence protocols.*

1 Introduction

Supporting global shared references in massively parallel architectures is a major concern. The cost of accessing remote shared memory locations has a direct impact on the performance of an application on such architectures. Caching global shared memory reduces the overall latency of global references. The choice of a shared memory cache coherence protocol is to a large extent determined by the messaging protocol and the processor-memory bus bandwidth which in turn determines the performance of the architecture.

Alewife[1] supports shared memory by integrating message passing and a directory based cache coherence protocol called the LimitLESS scheme[2]. The DASH[3] shared memory multiprocessor also implements a directory based cache coherence scheme which is an invalidation-based ownership protocol. The model of consistency provided in DASH is called *release consistency*[4] and is an extension of the weak consistency model proposed in[5]. The Stanford FLASH Multiprocessor[6] is yet another multiprocessor that supports cache coherent shared memory using a directory based cache protocol. The protocol is based on two components, *viz.* a scalable directory data structure and a set of handlers.

However, in all the three cited multiprocessor architectures the cost of sending cache invalidation messages cannot be ignored. We conjecture that a significant gain in multiprocessor performance can be achieved by totally avoiding invalidations. We propose such a scheme in this paper. The *Incessantly Coherent* cache coherence scheme presented here maintains incessantly coherent shared memory by automatically invalidating cache lines after a specified period called the *lifetime*. The protocol supports limitless sharing and permits writes in home locations only. The worst case performance of the proposed scheme is guaranteed to be at least as good as any directory based scheme. Additionally the proposed scheme relieves the network traffic due to invalidations which is unavoidable in directory based schemes.

The rest of the paper is organised as follows. In section 2, we describe the incessantly coherent cache coherence protocol. This is followed by section 3 that elaborates on the proposed scheme using an walk-through example. We comment on the performance of the proposed scheme in section 4, and finally in section 5 conclude with a summary of the contributions.

2 Incessantly Coherent: A Cache Coherence Protocol

We propose a cache coherence scheme for a system comprising multiple processor sites in a network. A site is a set of tightly coupled processors. Each proces-

*The authors are from Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore, India.

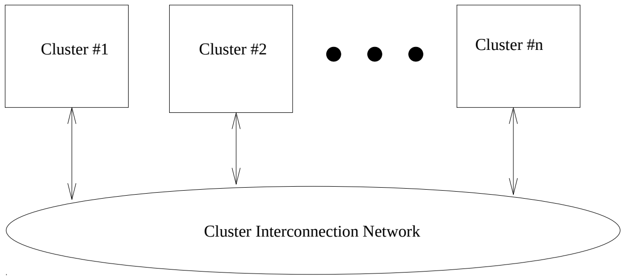


Figure 1: System Model

sor in this system supports a multi-level cache and executes multi-threaded code. The programming model assumes a single global address space. Every global address has an owner site called **home** Global shared memory support for the system is realized as shared memory modules physically distributed amongst the sites. Each site has a shared memory unit(SMU) and a controller to handle shared references. An example architecture of this model is *T-NG [7]. Other distributed memory multicomputer systems are also based on a similar model. Figures 1 and 2 give the conceptual model of such a system.

Directory based cache coherence protocols [9] and its variants must invalidate all shared cache lines before a cache line is written. The cost of invalidates in a relatively low bandwidth interconnection network can far offset the performance gains obtained by exploiting parallelism. Invalidations add to the network traffic that lead to network congestion. On the other hand, if only a site is allowed to own a cache line then for programs that exhibit spatial locality, traffic on the interconnection network will increase enormously since single ownership of cache lines will lead to repeated invalidations followed by *wandering* of cache lines.

We propose an *Incessantly Coherent Cache Scheme* that supports limitless sharing, alleviates wandering and eliminates invalidations. All READs are performed at the granularity of a cache line. WRITES are permitted at the home location only. The problem of wandering is alleviated by enforcing a finite period of validity time t_c (called lifetime) for each cache line. A cache line is automatically invalidated after its lifetime. Limitless sharing is supported by honoring multiple READ requests simultaneously. Note that the cache line could be "live" in multiple sites.

A WRITE operation on a cache location is honoured when the lifetime of the latest READ has expired. Multiple WRITES to the same cache line must

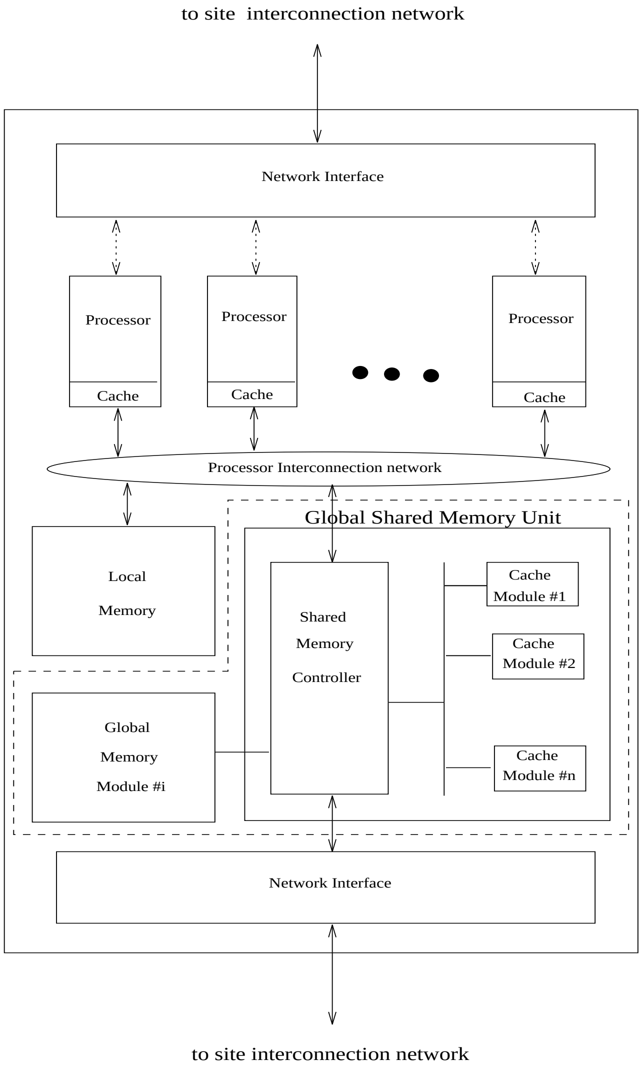


Figure 2: Details of a Site

therefore be queued. It is important to note that a WRITE operation is delayed only if it arrives within t_c of the last READ operation.

In systems that implement global memory cache at a lower level in the overall cache hierarchy, it is possible that cache coherence in higher level caches (in processors within a site) is maintained using an entirely different protocol. It is therefore necessary that on the expiry of lifetime of cache lines in the global cache, copies of this cache line in all higher level caches be invalidated. In such cases, SMU sends appropriate invalidate messages on the site's processor-memory interconnection network.

2.1 Limitless sharing, No wandering, No invalidations

Read requests for locations of a particular cache line originating from different sites may result in the cache line to "*wander*" from one site to another.

This problem is alleviated by enforcing a lifetime for a cache line. A cache line is "*live*" within a site for t_c, a finite period of time called its lifetime. (t_c starts immediately after the cache line is loaded.) A cache line is automatically invalidated after its lifetime. Multiple READ requests to locations within the cache line are honoured by sending the entire cache line. In this case, note that the cache line could be "*live*" in multiple sites.

A WRITE operation on a cache location is honoured when the lifetime of the latest READ has expired. Multiple WRITES to the same cache line must therefore be queued. It is important to note that a WRITE operation is delayed only if it arrives within t_c of the last READ operation.

3 An Example

Consider the following instruction sequence in a program segment.

```
global shared variable a
.....
Instruction   i> A := A + B
Instruction i+1> P := A + 5
.....
```

A multi-threaded code corresponding to the above program segment will appear as follows. In addition to the shared variables defined in the program, additional global and local counters are defined in the code generated by the compiler for synchronization.

```
global shared variable:  A
global shared counter:  syncA=1
local synchronization counter:  C=3
local variable:  B

Thread T1;generated by instruction i

    Store A from global memory
    into frame
    Decrement C
    If C=0 then start T4
    else stop

Thread T2;generated by instruction i

    Store syncA from global
    memory into frame
    Decrement C
    If C=0 then start T4
    else stop

Thread T3;generated by instruction i

    Store B from local memory
    into frame
    Decrement C
    If C=0 then start T4
    else stop

Thread T4;generated by instruction i

    Frame fetch syncA
    Frame fetch A
    Frame fetch B
    A := A + B
    Decrement syncA
    Perform a guarded write of A
    and syncA in global shared
    memory


Thread T5;generated by instruction
i+1

    Store syncA from global
    memory into frame
    If syncA = 0 then start T6
    else retry


Thread T6;generated by instruction
i+1

    Store A from global memory
    into frame
    Frame fetch A
```

```
Add 5
Store into P in frame
```

Let us assume that threads T1, T2, T3 and T4 are scheduled on processors in site$_i$ and threads T5 and T6 are scheduled on processors in site$_{i+1}$. Further, we assume that `A` and `syncA` are resident in the global memory module of site$_{i+1}$. Clearly the order of execution of threads T1, T2 and T3 are of no consequence. Also thread T4 cannot start before T1, T2, and T3 have completed. This is ensured by maintaining a local counter C in the frame. Threads T1,T2,T3 and T4 together share the same frame in site$_i$. Threads T5 and T6 together share a frame in site$_{i+1}$. Thread T5 cannot start before T4 completes. This is achieved by maintaining a global counter `syncA` that guarantees that thread dependencies are not violated.

The Incessantly Coherent cache coherence protocol can now be verified for the above thread executions. To start with we will impose the following artificial constraints that will be relaxed later.

1. Sites i and i+1 maintain clocks that are globally synchronised
2. Communication latency on the site interconnection network is finite and very close to zero.

The following sequence of actions takes place at global caches.

1. Thread T1 executing in site$_i$ encounters a READ miss on `A`. The global memory controller sends out a request for a cache line to site$_{i+1}$.
2. Site$_{i+1}$ forwards a cache line containing `A` to the global memory controller of site$_i$.
3. Since communication latency is zero, site$_i$ receives the cache line immediately. This cache line is self invalidated after its lifetime t_c in site$_i$.
4. Thread T2 executing in site$_i$ encounters a READ hit on the global memory cache for a reference to `syncA`. (It is assumed that `syncA` and `A` are in the same cache line because of locality of reference.)
5. Thread T3 executing in site$_i$ results in no global memory operations.
6. Thread T4 executing in site$_i$ results in a guarded global WRITE operation. Since all writes are made in the home location, the WRITE operation is delayed until the expiry of the lifetime of the cache line(s) containing `syncA` and `A` in site$_{i+1}$.
7. Thread T5 executing in site$_{i+1}$ results in a READ miss on `syncA`. The global memory controller of site$_{i+1}$ fetches the cache line. If `syncA` is not zero, the cache line is brought in again after a retry interval. The number of retries depends on the lifetime t_c. Each retry is a READ miss.
8. Thread T6 executing in site$_{i+1}$ encounters a READ hit on the global memory cache for `A`. This results in no global memory operation.

Clearly, the above sequence of operations are hazard free. Now we relax the artificial constraints and review the above set of operations under the following conditions.

1. Processors site$_i$ and sites$_{i+1}$ maintain independent clocks
2. Communication latency on the site interconnection network is finite but not constant over all transactions

Figure 3 (a) gives the sequence of global memory references between site$_i$ and site$_{i+1}$ when the communcation latency is zero. Figure 3(b) shows the same set of operations with finite communication latency. It is clear from the figures that with non-zero communication latency the global memory remains consistent throughout its execution since the same set of transactions are stretched in time. There is however an increase in overall latency of program execution. Thus the Incessantly Coherent protocol guarantees cache coherence throughout the execution of a program.

4 Performance

The performance of any caching depends on the model of multi-threaded computation. In a data driven model of computation, threads of computations can be viewed as representing nodes in a dataflow graph. In such a model, thread schedules are not known *a priori*. In contrast, in data parallel model of computation threads are scheduled statically. In the implicit parallel programming model synchronization of threads is automatically handled by the compiler. Synchronization is normally emulated with software counters in memory locations. In a data driven model synchronization is absolutely necessary to ensure proper execution order of instructions. In the Threaded Abstract Machine (TAM) [8] such synchronization threads are called *inlets* and are executed in response to messages.

The synchronization operation is essentially an atomic *read-modify-write* to a memory location, followed by a branch on the result. Usually these memory locations are shared. In [7] the expected frequency of such syncronization events is stated to occur once every 50 or so instructions. Further it is reasonable to assume that the same synchronization counter is not accessed more than once by the same thread. This behaviour of synchronization threads clearly swings the pendulum in favor of the Incessantly Coherent protocol since a cache line is always read without the additional latency to invalidate previously shared copies in other caches. Further since the syncronization events are compiler generated, the compiler can additionally specify the lifetime of such cache lines. Clearly such cache lines will have shorter lifetimes compared to computation threads. In general, the compiler can choose to specify different lifetimes for different cache lines. This feature is particularly valuable if prefetches are used to bring global data into the cache before they are actually used.

Further, based on the reference pattern, a given site can send messages in advance to the home site to renew its lifetime. The lifetimes of cache lines can thus adapt to the reference pattern in multi-threaded program execution.

Based on these observations, it is expected that on the overall, the average latency of accessing global shared memory through cache (using the Incessantly Coherent protocol) is lower than that compared to any directory based caching scheme. Since the lifetime of cache lines is very sensitive to the overall system architecture and the model of multi-threaded computation, it is a good candidate to experiment on the *T-NG [7] architecture for which coherence protocols can be implemented in software.

5 Conclusions

In this paper we proposed a new shared memory cache coherence protocol. The salient features of the protocol being that it supports limitless sharing and relieves the network of any traffic due to invalidation messages. Since cache lines auto-invalidate themselves on the expiry of their lifetime, and writes are done only at the home location on the expiry of the lifetime, all caches are always coherent.

Acknowledgments

The authors acknowledge the discussions and suggestions received from Prof. Arvind and thank Dr. Horace Thompson, Prof. John Morris, Prof. Guang R. Gao, Derek Chiou, Boon Seong Ang, Xiao Wei, James Hoe,and Any Boughton for their helpful comments for a better understanding of the issues in global cache coherence.

References

[1] John Kubiatowicz and Anant Agarwal, "Anaotomy of a Message in the Alewife Multiprocessor", 7th International Conference on Supercomputing.

[2] David Chaiken, John Kubiatowicz and Anant Agarwal, "LimitLESS Directories: A Scalable Cache Coherence Scheme", Proceedings of the fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV), pp. 224-234, ACM, APril 1991.

[3] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta and John Hennessy, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor", Proceedings of the 17th International Symposium on Computer Architecture, pp. 148-159, Seattle, WA, May 1990.

[4] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory Consistency and Event ordering in Scalable Shared-Memory Multiprocessor", Proceedings of the 17th Annual International Symposium on Computer Architecture, June 1990.

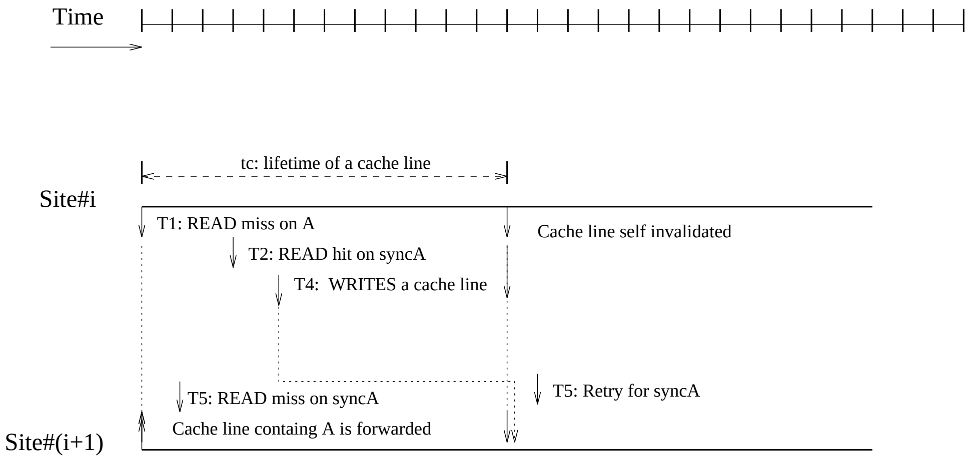
[5] M. Dubois, C. Scheurich, and F. Briggs, "Memory Access buffering in Multiprocessors", Proceedings of the 13th Annual International Symposium on Computer Architecture, pp. 434-442, June 1986.

[6] Jeffrey Kuskin, *et al* "The Stanfoed FLASH Multiprocessor", Proceedings of the 21st Annual International Symposium on Computer Architecture, 1994 (to appear).

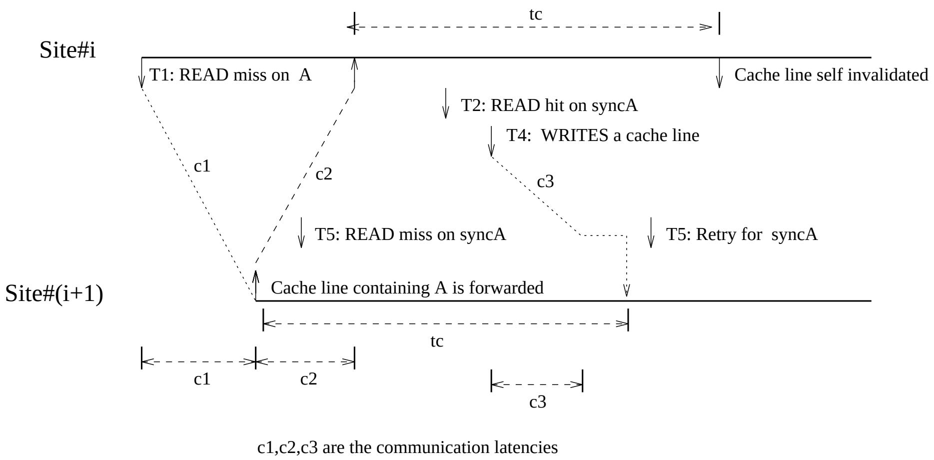
[7] Boon Seong Ang, Arvind, and Derek Chiou, "StarT the Next Generation: Integrating Global Caches and Dataflow Architecture", LCS CSG Memo 354, February 25, 1994.

[8] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken, "TAM – A Compiler Controlled Threaded Abstract Machine", Journal of Parallel and Distributed Computing, 18(3):347-370, 1993.

[9] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence", Proceedings of the 15th Annual International Symposium on Computer Architecture, June 1988, pp. 280–289.



(a) Zero communication latency; Globally synchronized clocks



(b) Realistic communication latency; Independent clocks

Figure 3: Global memory accesses