Directory-Based Cache Coherence in Large-Scale Multiprocessors

David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal
Massachusetts Institute of Technology

This article addresses the usefulness of shared-data caches in large-scale multiprocessors, the relative merits of different coherence schemes, and system-level methods for improving directory efficiency.

In a shared-memory multiprocessor, the memory system provides access to the data to be processed and mechanisms for interprocess communication. The bandwidth of the memory system limits the speed of computation in current high-performance multiprocessors due to the uneven growth of processor and memory speeds. Caches are fast local memories that moderate a multiprocessor's memory-bandwidth demands by holding copies of recently used data, and provide a low-latency access path to the processor. Because of locality in the memory access patterns of multiprocessors, the cache satisfies a large fraction of the processor accesses, thereby reducing both the average memory latency and the communication bandwidth requirements imposed on the system's interconnection network.

Caches in a multiprocessing environment introduce the *cache-coherence problem*. When multiple processors maintain locally cached copies of a unique shared memory location, any local modification of the location can result in a globally inconsistent view of memory. Cache-coherence schemes prevent this problem by maintaining a uniform state for each cached block of data.

Several of today's commercially available multiprocessors use bus-based memory systems. A bus is a convenient device for ensuring cache coherence because it allows all processors in the system to observe ongoing memory transactions. If a bus transaction threatens the consistent state of a locally cached object, the cache controller can take such appropriate action as invalidating the local copy. Protocols that use this mechanism to ensure coherence are called *snoopy* protocols because each cache snoops on the transactions of other caches.[1]

Unfortunately, buses simply don't have the bandwidth to support a large number of processors. Bus cycle times are restricted by signal transmission times in multidrop environments and must be long enough to allow the bus to "ring out," typically a few signal propagation delays over the length of the bus. As processor speeds increase, the relative disparity between bus and processor clocks will simply become more evident.

Consequently, scalable multiprocessor systems interconnect processors using short point-to-point wires in direct or multistage networks. Communication along impedance-matched transmission line channels can occur at high speeds, providing communication bandwidth that

0018-9162/90/0600-0049$01.00©1990IEEE

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

scales with the number of processors. Unlike buses, the bandwidth of these networks increases as more processors are added to the system. Unfortunately, such networks don't have a convenient snooping mechanism and don't provide an efficient broadcast capability.

In the absence of a systemwide broadcast mechanism, the cache-coherence problem can be solved with interconnection networks using some variant of directory schemes.[2] This article reviews and analyzes this class of cache-coherence protocols. We use a hybrid of trace-driven simulation and analytical methods to evaluate the performance of these schemes for several parallel applications.

The research presented in this article is part of our effort to build a high-performance large-scale multiprocessor. To that end, we are studying entire multiprocessor systems, including parallel algorithms, compilers, runtime systems, processors, caches, shared memory, and interconnection networks. We find that the best solutions to the cache-coherence problem result from a synergy between a multiprocessor's software and hardware components.

Classification of directory schemes

A cache-coherence protocol consists of the set of possible states in the local caches, the states in the shared memory, and the state transitions caused by the messages transported through the interconnection network to keep memory coherent. To simplify the protocol and the analysis, our data block size is the same for coherence and cache fetch.

A cache-coherence protocol that does not use broadcasts must store the locations of all cached copies of each block of shared data. This list of cached locations, whether centralized or distributed, is called a *directory*. A directory entry for each block of data contains a number of *pointers* to specify the locations of copies of the block. Each directory entry also contains a dirty bit to specify whether or not a unique cache has permission to write the associated block of data.

The different flavors of directory protocols fall under three primary categories: *full-map directories, limited directories*, and *chained directories*. Full-map directories[2] store enough state associated with each block in global memory so that every cache in the system can simultaneously store a copy of any block of data. That is, each directory entry contains N pointers, where N is the number of processors in the system. Such directories can be optimized to use a single bit pointer. Limited directories[3] differ from full-map directories in that they have a fixed number of pointers per entry, regardless of the number of processors in the system. Chained directories[4] emulate the full-map schemes by distributing the directory among the caches.

To analyze these directory schemes, we chose at least one protocol from each category. In each case, we tried to pick the protocol that was the least complex to implement in terms of the required hardware overhead. Our method for simplifying a protocol was to minimize the number of cache states, memory states, and types of protocol messages. All of our protocols guarantee *sequential consistency*, which Lamport[5] defined to ensure the correct execution of multiprocess programs.

Full-map directories. The full-map protocol uses directory entries with one bit per processor and a dirty bit. Each bit represents the status of the block in the corresponding processor's cache (present or absent). If the dirty bit is set, then one and only one processor's bit is set, and that processor has permission to write into the block. A cache maintains two bits of state per block. One bit indicates whether a block is valid; the other bit indicates whether a valid block may be written. The cache-coherence protocol must keep the state bits in the memory directory and those in the caches consistent.

Figure 1a illustrates three different states of a full-map directory. In the first state, location X is missing in all of the caches in the system. The second state results from three caches (C1, C2, and C3) requesting copies of location X. Three pointers (processor bits) are set in the entry to indicate the caches that have copies of the block of data. In the first two states, the dirty bit — on the left side of the directory entry — is set to clean (C), indicating that no processor has permission to write to the block of data. The third state results from cache C3 requesting write permission for the block. In this final state, the dirty bit is set to dirty (D), and there is a single pointer to the block of data in cache C3.

It is worth examining the transition from the second state to the third state in more detail. Once processor P3 issues the write to cache C3, the following events transpire:

(1) Cache C3 detects that the block containing location X is valid but that the processor does not have permission to write to the block, indicated by the block's write-permission bit in the cache.

(2) Cache C3 issues a write request to the memory module containing location X and stalls processor P3.

(3) The memory module issues invalidate requests to caches C1 and C2.

(4) Cache C1 and cache C2 receive the invalidate requests, set the appropriate bit to indicate that the block containing location X is invalid, and send acknowledgments back to the memory module.

(5) The memory module receives the acknowledgments, sets the dirty bit, clears the pointers to caches C1 and C2, and sends write permission to cache C3.

(6) Cache C3 receives the write permission message, updates the state in the cache, and reactivates processor P3.

Note that the memory module waits to receive the acknowledgments before allowing processor P3 to complete its write transaction. By waiting for acknowledgments, the protocol guarantees that the memory system ensures sequential consistency.

The full-map protocol provides a useful upper bound for the performance of centralized directory-based cache coherence. However, it is not scalable with respect to memory overhead. Assume that the amount of distributed shared memory increases linearly with the number of processors N. Because the size of the directory entry associated with each block of memory is proportional to the number of processors, the memory consumed by the directory is proportional to the size of memory ($\Theta(N)$) multiplied by the size of the directory entry ($\Theta(N)$). Thus, the total memory overhead scales as the square of the number of processors ($\Theta(N^2)$).

Limited directories. Limited directory protocols are designed to solve the directory size problem. Restricting the number of simultaneously cached copies of any particular block of data limits the growth of the directory to a constant factor. For our analysis, we selected the limited directory protocol proposed in Agarwal et al.[3]

A directory protocol can be classified as $Dir_i X$ using the notation from Agarwal et al.[3] The symbol i stands for the number of pointers, and X is NB for a scheme with no broadcast and B for one with broadcast. A full-map scheme without broadcast is represented as $Dir_N NB$. A limited directory

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

protocol that uses $i<N$ pointers is denoted Dir_iNB. The limited directory protocol is similar to the full-map directory, except in the case when more than i caches request read copies of a particular block of data.

Figure 1b shows the situation when three caches request read copies in a memory system with a Dir_2NB protocol. In this case, we can view the two-pointer directory as a two-way set-associative cache of pointers to shared copies. When cache C3 requests a copy of location X, the memory module must invalidate the copy in either cache C1 or cache C2. This process of pointer replacement is sometimes called *eviction*. Since the directory acts as a set-associative cache, it must have a pointer replacement policy. Our protocol uses an easily implemented pseudorandom eviction policy that requires no extra memory overhead. In Figure 1b, the pointer to cache C3 replaces the pointer to cache C2.

Why might limited directories succeed? If the multiprocessor exhibits processor locality in the sense that in any given interval of time only a small subset of all the processors access a given memory word, then a limited directory is sufficient to capture this small "worker-set" of processors.

Directory pointers in a Dir_iNB protocol encode binary processor identifiers, so each pointer requires $\log_2 N$ bits of memory, where N is the number of processors in the system. Given the same assumptions as for the full-map protocol, the memory overhead of limited directory schemes grows as $\Theta(N \log N)$. These protocols are considered scalable with respect to memory overhead because the resources required to implement them grow approximately linearly with the number of processors in the system.

Dir_iB protocols allow more than i copies of each block of data to exist, but they resort to a broadcast mechanism when more than i cached copies of a block need to be invalidated. However, interconnection networks with point-to-point wires do not provide an efficient systemwide broadcast capability. In such networks, it is also difficult to determine the completion of a broadcast to ensure sequential consistency. While it is possible to limit some Dir_iB broadcasts to a subset of the system (see Agarwal et al.[3]), we restrict our evaluation of limited directories to the Dir_iNB protocols.

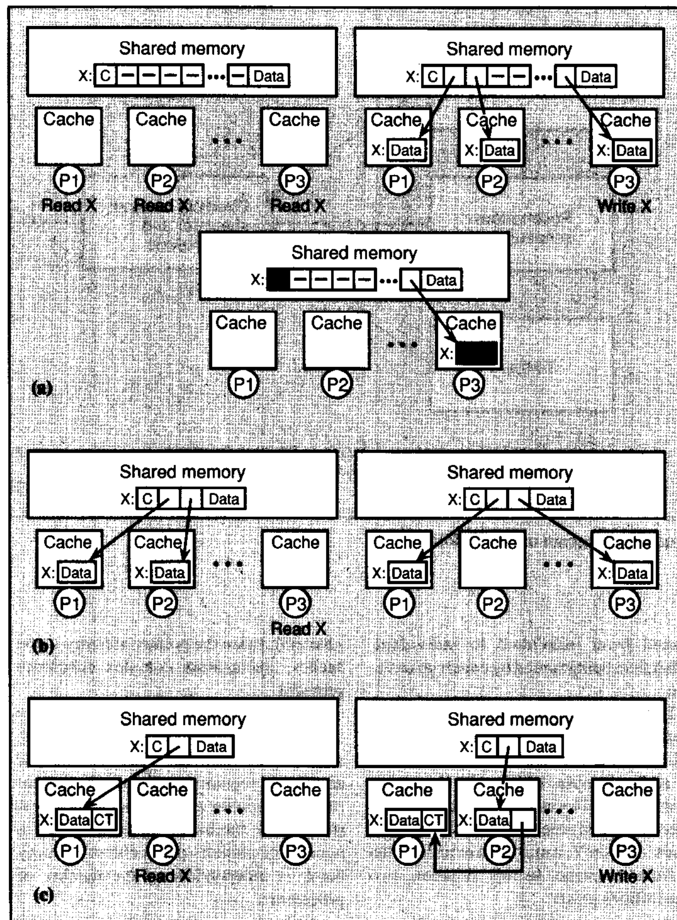

Figure 1. Three types of directory protocols: (a) three states of a full-map directory; (b) eviction in a limited directory; and (c) chained directory.

Chained directories. Chained directories, the third option for cache-coherence schemes that do not utilize a broadcast mechanism, realize the scalability of limited directories without restricting the number of shared copies of data blocks.[4] This type of cache-coherence scheme is called a *chained* scheme because it keeps track of shared copies of data by maintaining a chain of directory pointers. We investigated two chained directory schemes.

The simpler of the two schemes implements a singly linked chain, which is best described by example (see Figure 1c). Suppose there are no shared copies of location X. If processor P1 reads location X, the memory sends a copy to cache C1, along with a chain termination (CT) pointer. The memory also keeps a pointer to cache C1. Subsequently, when processor P2 reads location X, the memory sends a copy to cache C2, along with the pointer to cache C1. The memory then keeps a pointer to cache C2. By repeating this step, all of the caches can cache a copy of location X. If processor P3 writes to location X, it is necessary to send a data invalidation message down the chain. To ensure sequential consistency, the memory module denies processor P3 write permission until the processor with the chain termination pointer acknowledges the invalidation of the chain. Perhaps this scheme should be called a *gossip* protocol (as opposed to a snoopy protocol) because information is

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

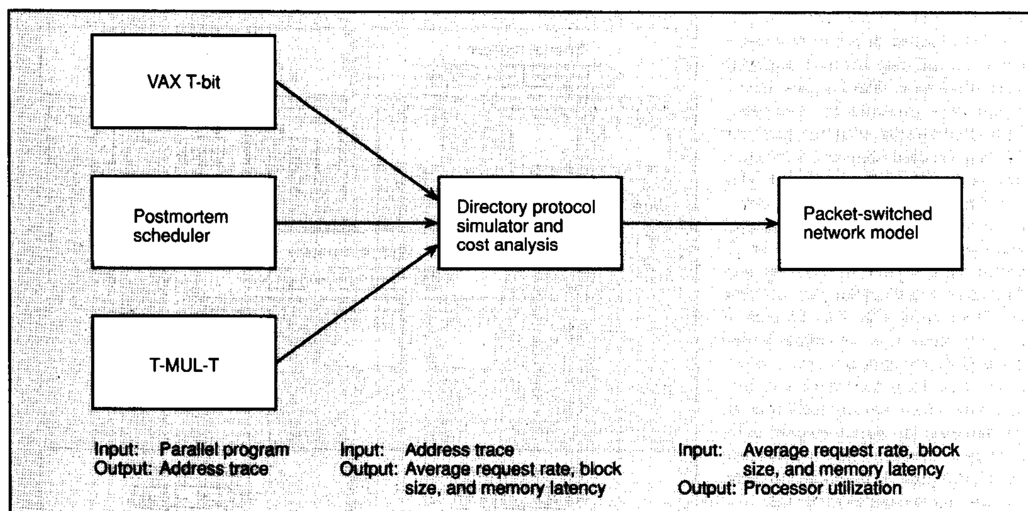

Figure 2. Diagram of methodology.

passed from individual to individual, rather than being spread by covert observation.

The possibility of cache-block replacement complicates chained directory protocols. Suppose that cache C_1 through cache C_N all have copies of location X and that location X and location Y map to the same (direct-mapped) cache line. If processor P_i reads location Y, it must first evict location X from its cache. In this situation, two possibilities exist:

(1) Send a message down the chain to cache C_{i-1} with a pointer to cache C_{i+1} and splice C_i out of the chain, or

(2) Invalidate location X in cache C_{i+1} through cache C_n.

For our evaluation, we chose the second scheme because it can be implemented by a less complex protocol than the first. In either case, sequential consistency is maintained by locking the memory location while invalidations are in progress.

Another solution to the replacement problem is to use a doubly linked chain. This scheme maintains forward and backward chain pointers for each cached copy so that the protocol does not have to traverse the chain when there is a cache replacement. The doubly linked directory optimizes the replacement condition at the cost of a larger average message block size (due to the transmission of extra directory pointers), twice the pointer memory in the caches, and a more complex coherence protocol.

Although the chained protocols are more complex than the limited directory protocols, they are still scalable in terms of the amount of memory used for the directories. The pointer sizes grow as the logarithm of the number of processors, and the number of pointers per cache or memory block is independent of the number of processors.

Caching only private data. Up to this point, we have assumed that caches are allowed to store local copies of shared variables, thus leading to the cache-consistency problem. An alternative shared memory method avoids the cache-coherence problem by disallowing caching of shared data. In our analysis, we designate this scheme by saying it only caches private data. This scheme caches private data, shared data that is read-only, and instructions, while references to modifiable shared data bypass the cache. In practice, shared variables must be statically identified to use this scheme.

Methodology

What is a good performance metric for comparing the various cache-coherence schemes? To evaluate the performance of the memory system, which includes the cache, the memory, and the interconnection network, we determine the contribution of the memory system to the time needed to run a program on the system. Our analysis computes the *processor utilization*, or the fraction of time that each processor does useful work. One minus the utilization yields the fraction of processor cycles wasted due to memory system delays. The actual system speedup equals the number of processors multiplied by the processor utilization. This metric has been used in other studies of multiprocessor cache and network performance.[6]

In a multiprocessor, processor utilization (and therefore system speedup) is affected by the frequency of memory references and the latency of the memory system. The latency (T) of a message through the interconnection network depends on several factors, including the network topology and speed, the number of processors in the system, the frequency and size of the messages, and the memory access latency. The cache-coherence protocol determines the request rate, message size, and memory latency. To compute processor utilization, we need to use detailed models of cache-coherence protocols and interconnection networks.

Figure 2 shows an overview of our analysis process. Multiprocessor address traces generated using three tracing methods at Stanford University, IBM, and MIT

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

are run on a cache and directory simulator that counts the occurrences of different types of protocol transactions. A cost is assigned to each of these transaction types to compute the average processor request rate, the average network message block size, and the average memory latency per transaction. From these parameters, a model of a packet-switched, pipelined, multistage interconnection network calculates the average processor utilization.

Table 1. Summary of trace statistics, with length values in millions of references to memory.

Source	Language	Processors	Application	Length
VAX T-bit	C	16	P-Thor	7.09
			MP3D	7.38
			LocusRoute	7.05
			SA-TSP	7.11
Postmortem scheduler	Fortran	64	FFT	7.44
			Weather	31.76
			Simple	27.03
T-Mul-T	Mul-T	64	Speech	11.77

Getting multiprocessor address trace data. The address traces represent a wide range of parallel algorithms written in three different programming languages. The programs traced at Stanford were written in C; at IBM, in Fortran; and at MIT, in Mul-T,[7] a variant of Multilisp. The implementation of the trace collector differs for each of the programming environments. Each tracing system can theoretically obtain address traces for an arbitrary number of processors, enabling a study of the behavior of cache-coherent machines much larger than any built to date. Table 1 summarizes general characteristics of the traces. We will compare the relative performance of the various coherence schemes individually for each application.

The SA-TSP, MP3D, P-Thor, and LocusRoute traces were gathered via the Trap-Bit method using 16 processors. SA-TSP uses simulated annealing to solve the traveling salesman problem. MP3D is a 3D particle simulator for rarified flow. P-Thor is a parallel logic simulator. LocusRoute is a global router for VLSI standard cells. Weber and Gupta[8] provide a detailed description of the applications.

Trap-bit (T-bit) tracing for multiprocessors is an extension of single-processor trap-bit tracing. In the single processor implementation, the processor traps after each instruction if the trap bit is set, allowing interpretation of the trapped instruction and emission of the corresponding memory addresses. Multiprocessor T-bit tracing extends this method by scheduling a new process on every trapped instruction. Once a process undergoes a trap, the trace mechanism performs several tasks. It records the corresponding memory addresses, saves the processor state of the trapped process, and schedules another process from its list of processes, typically in a round-robin fashion.

The Weather, Simple, and fast Fourier transform traces were derived using the postmortem scheduling method at IBM. The Weather application partitions the atmosphere around the globe into a three-dimensional grid and uses finite-difference methods to solve a set of partial differential equations describing the state of the system. Simple models the behavior of fluids and employs finite difference methods to solve equations describing hydrodynamic behavior. FFT is a radix-2 fast Fourier transform.

Postmortem scheduling is a technique that generates a parallel trace from a uniprocessor execution trace of a parallel application. The uniprocessor trace is a task trace with embedded synchronization information that can be scheduled, after execution (*postmortem*), into a parallel trace that obeys the synchronization constraints. This type of trace generation uses only one processor to produce the trace and to perform the postmortem scheduling. So, the number of processes is limited only by the application's synchronization constraints and by the number of parallel tasks in the single processor trace.

The Speech trace was generated by a compiler-aided tracing scheme. The application comprises the lexical decoding stage of a phonetically based spoken language understanding system developed by the MIT Spoken Language Systems Group. The Speech application uses a dictionary of about 300 words represented by a 3,500-node directed graph. The input to the lexical decoder is another directed graph representing possible sequences of phonemes in the given utterance. The application uses a modified Viterbi search algorithm to find the best match between paths through the two graphs.

In a compiler-based tracing scheme, code inserted into the instruction stream of a program at compile time records the addresses of memory references as a side effect of normal execution. Our compiler-aided multiprocessor trace implementation is T-Mul-T, a modification of the Mul-T programming environment that can be used to generate memory address traces for programs running on an arbitrary number of processors. Instructions are not currently traced in T-Mul-T. We assume that all instructions hit in the cache and, for processor utilization computation, an instruction reference is associated with each data reference. We make these assumptions only for the Speech application, because the other traces include instructions.

The trace gathering techniques also differ in their treatment of private data locations, which must be identified for the scheme that only caches private data. The private references are identified statically (at compile time) in the Fortran traces and are identified dynamically by postprocessing the other traces. Since static methods must be more conservative than dynamic methods when partitioning private and shared data, the performance that we predict for the private data caching scheme on the C and Mul-T applications is slightly optimistic. In practice, the nontrivial problem of static data partitioning makes it difficult to implement schemes that cache only private data.

Simulating a cache-coherence strategy. For each memory reference in a trace, our cache and directory simulator determines the effects on the state of the corresponding block in the cache and the shared memory. This state consists of the cache tags and directory pointers used to maintain cache coherence. In the simulation, the network provides no feedback to the cache or memory modules. Assume all side effects from each memory transaction (entry in the trace) are stored simultaneously. While this simulation strategy does not accurately model the state of the memory system on a cycle-by-cycle basis,

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase®
Smarter legal research.