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

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_iX 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_NNB. A limited directory

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.

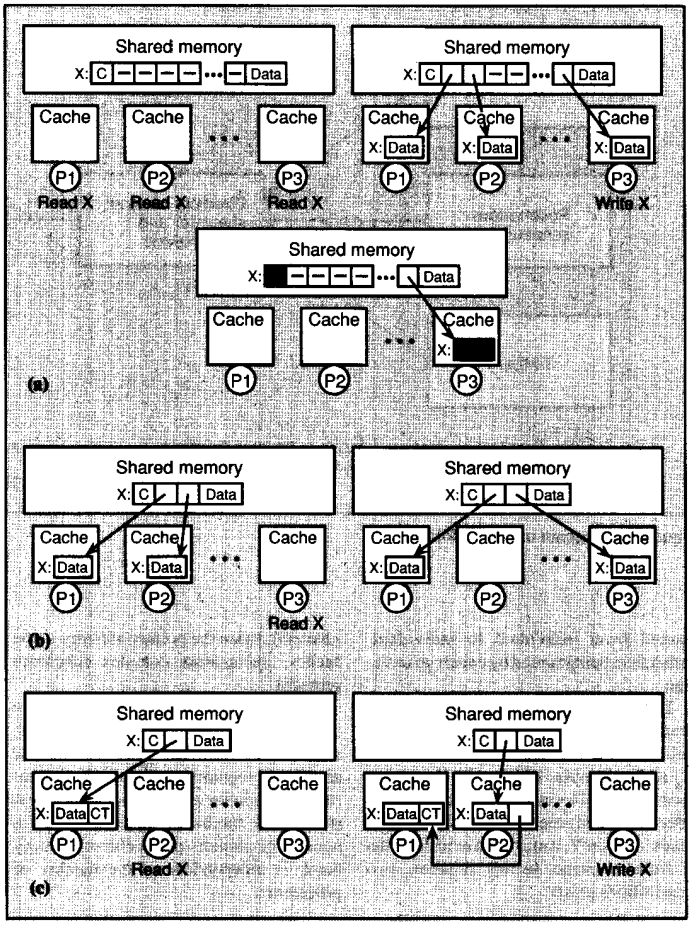

Figure 1. Three types of directory protocols: (a) three states of a full-map directory; (b) eviction in a limited directory; and (c) chained directory.

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

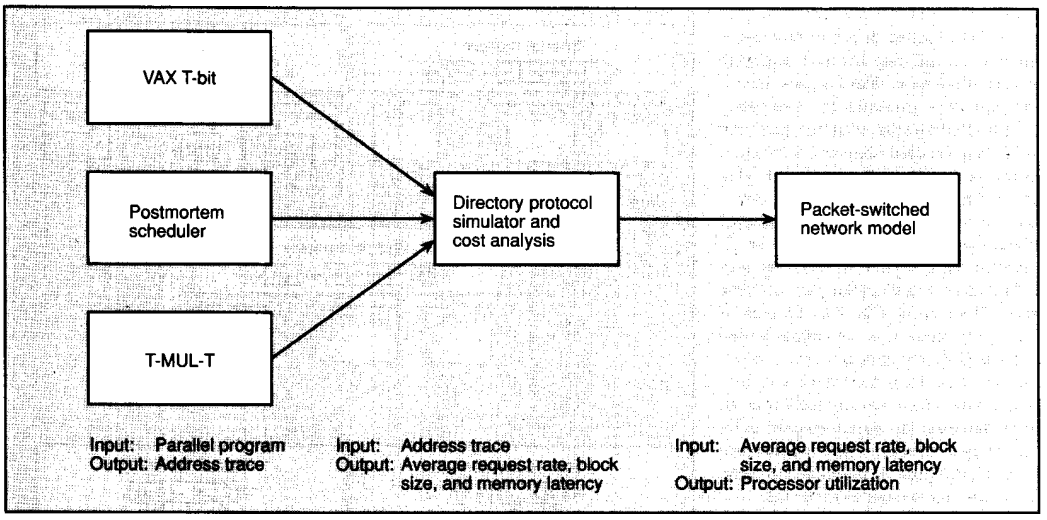

Figure 2. Diagram of methodology.

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

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.

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.

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

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 post-processing 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 non-trivial 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,

Table 2. Simulation parameter defaults for the cache, directory, and network.

Type of Parameter	Name	Default Value
Cache/Directory	Cache size	256 Kbytes
	Cache-block size	16 bytes
	Cache associativity	Direct mapped
	Cache-update policy	Write back
	Directory pointer replace policy	Random
Network	Network message header size	16 bits
	Network switch size	4 × 4
	Network channel width	16 bits
	Processor cycle time	2 × network switch cycle time
	Memory address size	32 bits
	Base memory access time	6 × network switch cycle time

it does produce accurate counts of each type of protocol transaction over the length of a correct execution of a parallel program.

However, since we assume that all side effects of any transaction occur simultaneously, we do not model the difference between sequential and concurrent operations. This inaccuracy particularly affects the analysis of chained directory schemes. Specifically, when a shared write is performed in a system that uses a chained directory scheme, the copies of the written location must be invalidated in sequence, while a centralized directory scheme may send the invalidations in parallel and keep track of the number of outstanding acknowledgments. Thus, the minimum latency for shared writes to clean cache blocks is greater for the distributed schemes than for the centralized schemes.

Analyzing the trade-offs between centralized and distributed schemes requires a much more detailed simulation. While it is possible to accurately model the memory system on a cycle-by-cycle basis, such a simulation requires much higher overhead than our simulations in terms of both programming time and simulation runtime. Our MIT research group is running experiments on a simulator for an entire multiprocessor system. Simulations of the entire system run approximately 100 times slower than the trace-driven simulations used for this article. Variants of coherence schemes are harder to implement in the detailed simulator than in the trace-driven environment. To investigate a wide range of applications and cache-coherence protocols, we avoided the high overhead of such detailed simulations by performing trace-driven simulations.

In a trace-driven simulation, a memory transaction consists of a processor-to-memory reference and its effect on the state of the memory system. Any transaction that causes a message to be sent out over the network contributes to the average request rate, average message size, and average memory latency. Each type of transaction is assigned a cost in terms of the number of messages that must be sent over the network (including both the requests and the responses), the latency encountered at the memory modules, and the total number of words (including routing information) transported through the network. Given a trace and a particular cache-coherence protocol, the cache and directory simulator determines the percentage of each transaction type in the trace. The percentage of the transaction type, multiplied by its cost, gives the contribution of the transaction to each of the three parameters listed above.

In addition to the cache-coherence strategy, other parameters affect the performance of the memory system. We chose values for these parameters (listed in Table 2) based on the technology used for contemporary multiprocessors. Although we chose a 256-kilobyte cache, the results of our analysis do not differ substantially for cache sizes from 256 kilobytes down to 16 kilobytes because the working sets for the applications are small when partitioned over a large number of processors. The effect of other parameters, including the cache-block size, has been explored in several studies (see Eggers and Katz[9] and references therein).

The interconnection network model. The directory schemes that we analyze transmit messages over an interconnection network to maintain cache coherence. They distribute shared memory and associated directories over the processing nodes. Our analysis uses a packet-switched, buffered, multistage interconnection network that belongs to the general class of Omega networks. The network switches are pipelined so that a message header can leave a switch even while the rest of the message is still being serviced. A protocol message travels through n network switch stages to the destination node and takes M cycles for the memory access. The network is buffered and guarantees sequenced delivery of messages between any two nodes on the network.

Computation of the processor utilization is based on the analysis method that Patel[10] used. The network model yields the average latency T of a protocol message through the network with n stages, $k \times k$ size switches, and average memory delay M. We derive processor utilization U from a set of three equations:

$$U = \frac{1}{1 + mT}$$

$$\rho = UmB$$

$$T = n + B + M - 1 + \left(\frac{\rho B(1 - \frac{1}{k})}{2(1 - \rho)}\right) n$$

where m is the probability a message is generated on a given processor cycle, with corresponding network latency T. The channel utilization (ρ) is the product of the effective network request rate (Um) and the average message size B. The latency equation uses the packet-switched network model by Kruskal and Snir.[11] The first term in the equation ($n + B + M - 1$) gives the latency through an unloaded network. The second term gives the increase in latency due to network contention, which is the product of the contention delay through one switch and the number of stages. We verified the model in the context of our research by comparing its predictions to the performance of a packet-switched network simulator that transmitted messages generated by a Poisson process.

Table 2 shows the default network parameters we used in our analysis. While this article presents results for a packet-switched multistage network, it is possible to derive results for other types of net-

works by varying the network model used in the final stage of the analysis. In fact, we repeated our analysis for the direct, two-dimensional mesh network that we plan to use in our own machine. With the direct network model, the cache-coherence schemes showed the same relative behavior as they did with the network model described above. The ability to use the results from one set of directory simulations to derive statistics for a range of network or bus types displays the power of this modeling method.

Analysis of directory schemes

The graphs presented below plot various combinations of applications and cache-coherence schemes on the vertical axis and processor utilization on the horizontal axis. Since the data reference characteristics vary significantly between applications and trace gathering methods, we do not average results from the different traces. The results presented here concentrate on the Weather, Speech, and P-Thor applications. We discuss other applications when they exhibit significantly different behavior.

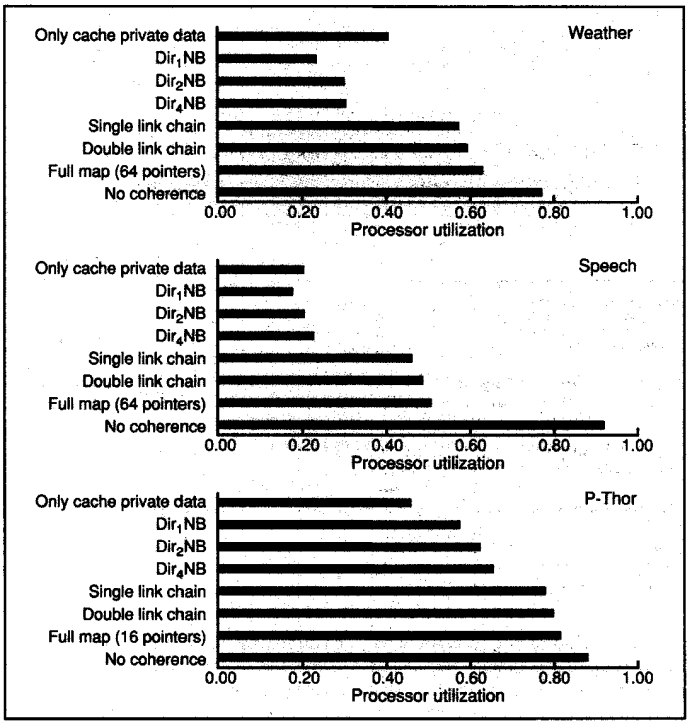

Figure 3. Comparison of coherence schemes.

Are caches useful for shared data? Figure 3 shows the processor utilizations realized for the Weather, Speech, and P-Thor applications using each of the coherence schemes we evaluated. The long bar at the bottom of each graph gives the value for "no cache coherence." This number is derived by considering all addresses in each trace to be not shared. Processor utilization with no cache coherence gives, in a sense, the effect of the native hit/miss rate for the application. The number is artificial because it does not represent the behavior of a correctly operating system. However, the number does give an upper bound on the performance of any coherence scheme and allows us to focus on the component of processor utilization lost due to sharing between processors.

To assess the potential of shared data caching schemes in general, we compare the optimal (full-map) directory scheme to the scheme that caches only private data. For most applications (including the ones shown in Figure 3), the full-map directory yields significantly better processor utilization than the scheme that caches only private data. Generally good performance of the full-map scheme in 16 and 64 processor machines implies that caches are useful for shared data, even when applications are not written or compiled specially for a system with directory-based cache coherence.

However, for two traces (Simple and MP3D), processor utilization for a full-map directory is worse than the utilization for the private data-cache scheme. Examining the network model shows the reason it is possible for private data caches to perform better than full-map directories: Even though the private cache scheme has a higher network message rate, it uses smaller message block sizes. In the model, network latency is proportional to the square of the message block size but is linearly dependent on the message rate.

The fact that for Simple and MP3D the private data-cache scheme performs better than the full-map directory scheme indicates that the average time between writes by different processors to each shared location is low. For these traces, the full-map directory scheme does not perform significantly better than the limited directory schemes.

Limited directory performance. How well do limited directories perform compared to the full-map directory scheme? The answer depends on the amount of shared data, the number of processors that access each shared data location, and the method of synchronization. The P-Thor application was written to minimize communication between processors by reducing the number of synchronization points and the number of processors that read each shared location. It is not surprising that all of the directory schemes perform well for this application.

On the other hand, four traces show significantly worse processor utilization for limited directories than for a full-map directory due to naive synchronization techniques (Weather, Simple, and SA-TSP) or widespread sharing of a large read-only data structure (Speech).

Chained directory performance. When applications use data structures that are widely shared and accessed frequently, a limited directory performs significantly

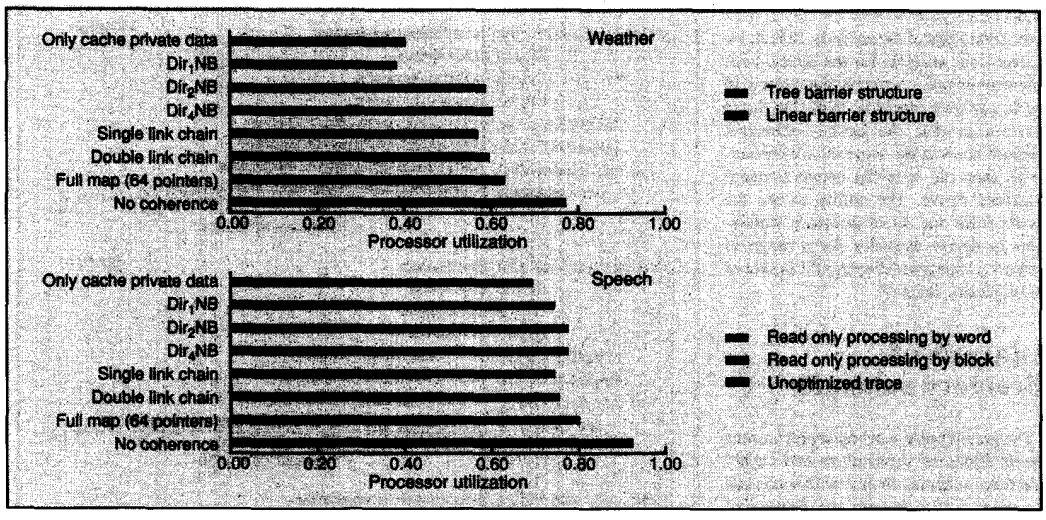

Figure 4. System-level optimizations.

worse than a full-map directory. However, Figure 3 shows that both singly and doubly linked directories perform almost as well as the full-map directory protocols. While the doubly linked scheme always performs slightly better than the singly linked scheme, the small increase in performance may not justify the additional resources needed for the doubly linked scheme. The difference between the schemes is small because the number of replacements as a percentage of total memory accesses is very small, even though we simulated direct-mapped caches.

In general, chained directory schemes yield higher utilization than limited directory protocols. However, chained directory protocols are more complex and have higher write latency than limited directory protocols. We are still investigating the ramifications of this trade-off.

Improving the performance of directories

The results presented above show that limited directory schemes suffer from data types that are both widely shared and frequently referenced. We use the Weather and Speech applications as case studies to demonstrate two methods for ameliorating the effects of this type of data. These methods are examples of *system-level optimizations* because they involve contributions from several components of a multiprocessor system. In addition to improving the performance of limited directory schemes, the methods also enhance the performance of the other coherence schemes.

The Weather application uses barriers as the primary method of synchronization. In the straightforward implementation of barriers, each processor increments a barrier variable and then spin-locks on a barrier flag. The last processor to reach the synchronization point increments the barrier variable to its final value N and writes into the barrier flag, thereby releasing the spinning processors. The memory accesses from many processors spin-locking on a single location cause pointer thrashing (repeated evictions) in the limited directory.

A software solution, called a *combining tree*,[12] can alleviate this problem in directories. Instead of implementing barrier synchronizations with a single barrier variable and barrier flag, a balanced tree structure of nodes can be used for each. To demonstrate the benefits of this barrier implementation, we modified the postmortem scheduler to implement combining tree synchronization. The resulting trace was virtually identical to the original trace, except with respect to the distribution of synchronization address accesses. In the original trace, all of the synchronization addresses were accessed by all of the processors. In the combining-tree trace, almost all of the synchronization addresses were accessed primarily by one processor, with just one access by one other processor.

The top graph in Figure 4 shows that the combining tree dramatically improves the performance of the limited directory schemes. The darker colored bars show the processor utilization of the application with linear barrier synchronization, and the lighter bars show the enhanced utilization when using the combining-tree structure. The two- and four-pointer directories yield nearly the same processor utilization as the full-map scheme. The one pointer directory suffers from sharing of other data between processors. However, this data sharing must exist only between processor pairs, because it does not affect the two-pointer directory. Thus, combining tree structures and limited directory schemes provides an efficient implementation of barrier synchronization.

The Speech application provides an example of both a different programming model and a different type of widely shared data. There are two primary data structures in the Speech application: an utterance (the sentence to be identified) and a dictionary (the algorithm's vocabulary). For the duration of the application, these data structures are only read, but they are shared by all the processors in the system. This type

of data reference pattern causes pointer thrashing in limited directories.

Given the nature of the Speech application, it is fair to assume that all the read-only variables can be identified by the programmer. To assess the potential benefits of marking read-only data, we post-processed the trace to find all the data locations that were only read for the duration of the trace. The read-only locations were then marked as private to prevent the cache and directory simulator from executing coherence transactions for this data. When these locations were identified on a block-by-block basis, the system showed moderate improvement for the limited directory schemes. However, when the post-processor identified the read-only locations on a word-by-word basis and relocated the data to a special segment of memory, the improvement was more pronounced. The bottom graph in Figure 4 demonstrates the increase in processor utilization realized by specially processing read-only data. The darkest bars show the unoptimized performance of the Speech application; the lighter bars show the gains due to processing read-only data.

The boost in processor utilization due to read-only data detection on a word-by-word basis can be explained by the reduction of sharing due to cache blocks that contain unrelated data words accessed by different processors. The Mul-T runtime system ignored the boundary of cache blocks and allocated read-write data words in the same cache blocks as read-only data words. This data allocation policy prevented the block-by-block postprocessor from properly identifying read-only data words and lowered processor utilization by creating unnecessary shared data traffic in the network.

When multiprocessor algorithms and software are optimized for caches, large-scale cache-coherent systems realize their execution potential. In the case of the Weather and Speech applications, system-level optimizations resulted in processor utilizations between 0.6 and 0.8 for scalable cache-coherence protocols. Coordinating multiprocessor hardware and software requires some subset of programmer specifications, new language primitives, special compile-time analysis, support in the runtime system, specialization in the processor-to-cache interface, and additional states in the cache-coherence protocol. The modifications described in this article represent archetypes of systemwide efforts to improve multiprocessor performance.

This article has shown that, by using system-level optimizations, it is possible to build large-scale cache-coherent multiprocessors. Using processor utilization as a metric, we evaluated the performance of several cache-coherence protocols, including limited directories and chained directories. We compared protocols that are scalable in terms of their memory overhead to a protocol that cached only private data and to a nonscalable protocol (full-map). While the scheme that cached only private data performed fairly well, the shared data caching schemes performed better for the majority of the applications that we studied. Limited and chained directory schemes permitted the use of caches to significantly reduce the effective shared memory latency.

There is no hardware panacea for the cache-coherence problem. As with many other problems in computer architecture, good solutions balance hardware and software optimizations that combine to improve system performance. When we applied system-level optimizations to caching, we were able to improve the performance of systems with large numbers of processors.

Our work can be extended in several ways. The most straightforward extension would repeat our trace-driven evaluation using other network models.

Our research group at MIT is currently performing more detailed simulations of directory schemes, coupled with processor and network simulators, to get accurate multiprocessor performance statistics. Such simulations allow us to address the issue of hot spots, the impact of high-latency operations, and the effect of interrupting local cache accesses with external invalidation messages. We are also researching various methods for alleviating the effects of communication latency. These methods include using multithreaded processors with coherent caches, software emulation of directories, and coherence models other than sequential consistency. ■

Acknowledgments

It is impossible to analyze a large range of applications, programming models, and architectures without becoming indebted to a host of collaborators.

Mathews Cherian laid the foundation for our analysis by writing both the postmortem scheduler with Kimming So at IBM and the cache and directory simulator. Pat Teller of New York University provided the Simple and Weather programs, and FFT was written at IBM. Harold Stone and Kimming So helped us obtain the IBM traces. Wolf-Dietrich Weber and Anoop Gupta provided us with the four VAX T-bit traces, which were generated using a system developed by Steve Goldschmidt at Stanford. David Kranz wrote the Mul-T compiler and the T-Mul-T trace generator and helped analyze the results from the Mul-T application. Kirk Johnson, who wrote and traced the Speech application, is responsible for the read-only data processing results. Gino Maa and Sue-Kyoung Lee wrote the packet switched network simulator that validated our network model.

Encore Computer Corporation provided the Multimax system that runs T-Mul-T. Digital Equipment Corporation and Harris Computer Systems provided the machines we used to manipulate gigabytes of trace data. We would also like to thank the rest of the Alewife group for putting up with our interminable trace-driven simulations.

The research reported in this article is funded by DARPA contract No. N00014-87-K-0825 and by grants from the Sloan Foundation and IBM.

References

1. J.R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," *Proc. 10th Ann. Symp. Computer Architecture*, June 1983, pp. 124-131.

2. L.M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. Computers*, Vol. C-27, No. 12, Dec. 1978, pp. 1,112-1,118.

3. A. Agarwal et al., "An Evaluation of Directory Schemes for Cache Coherence," *Proc. 15th Int'l Symp. Computer Architecture*, CS Press, Los Alamitos, Calif. Order No. 861, June 1988, pp. 280-289.

4. D.V. James et al., "New Directions in Scalable Shared Memory Multiprocessor Architectures: Scalable Coherent Interface," *Computer*, June 1990, Vol. 23, No. 6, pp. 74-77.

5. L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Trans. Computers*, Vol. C-28, No. 9, Sept. 1979, pp. 690-691.

6. J. Archibald and J.-L. Baer, "Cache-Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Trans. Computer Systems*, Vol. 4, No. 4, Nov. 1986, pp. 273-298.

7. D. Kranz, R. Halstead, and E. Mohr, "Mul-T: A High-Performance Parallel Lisp," *Proc. SIGPlan 89, Conf. Programming Languages Design and Implemenation*, June 1989, pp. 81-90.

8. W.-D. Weber and A. Gupta, "Analysis of Cache Invalidation Patterns in Multiprocessors," *Third Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, CS Press, Los Alamitos, Calif. Order No. 1936, Apr. 1989, pp. 243-256.

9. S.J. Eggers and R.H. Katz, "The Effect of Sharing on the Cache and Bus Performance of Parallel Programs," *Third Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, CS Press, Los Alamitos, Calif. Order No. 1936, Apr. 1989, pp. 257-270.

10. J.H. Patel, "Analysis of Multiprocessors with Private Cache Memories," *IEEE Trans. Computers*, Vol. C-31, No. 4, Apr. 1982, pp. 296-304.

11. C.P. Kruskal and M. Snir, "The Performance of Multistage Interconnection Networks for Multiprocessors," *IEEE Trans. Computers*, Vol. C-32, No. 12, Dec. 1983 pp. 1,091-1,098.

12. P.-C. Yew, N.-F. Tzeng, and D.H. Lawrie, "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors," *IEEE Trans. Computers*, Vol. C-36, No. 4, Apr. 1987, pp. 388-395.

David Chaiken is a member of the Laboratory for Computer Science at the Massachusetts Institute of Technology and a graduate student in the MIT Department of Electrical Engineering and Computer Science. His research involves computer architectures and programming models for parallel processing.

Chaiken received his BS in mathematics and chemistry from Brown University in 1986.

The authors can be contacted at the Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.

Craig Fields is an applications programmer for Project Athena at the Massachusetts Institute of Technology. He is also a member of the Alewife group in the institute's Laboratory for Computer Science. His research interests are parallel architectures and systems development.

Fields attended MIT until 1989.

Kiyoshi Kurihara, a systems engineer at IBM Japan, is a postgraduate studying in the Department of Electrical Engineering and Computer Science of the Massachusetts Institute of Technology under an MIT Overseas Study Program scholarship from the company. His research interests are computer architecture and system performance evaluation methodology. He is carrying out research work at MIT's Laboratory for Computer Science.

Kurihara received his bachelor of engineering degree from the University of Tokyo in 1981.

Anant Agarwal has been an assistant professor of electrical engineering and computer science in the Laboratory for Computer Science at the Massachusetts Institute of Technology since January 1988. His research interests include the design of scalable multiprocessor systems, VLSI processors, parallel processing software, and performance evaluation. He initiated MIT's Alewife project, which aims to design and implement a large-scale cache-coherent multiprocessor.

Agarwal received the BTech degree in electrical engineering from the Indian Institute of Technology, Madras, India, in 1982, and the MS and PhD degrees in electrical engineering from Stanford University in 1984 and 1987, respectively. He is a member of the IEEE Computer Society.