CACHE MEMORY PROTOCOLS

The task of a processor in a computer system is to repeatedly execute a cycle consisting of fetching the instruction from memory, decoding it, and executing it. The processor must constantly perform memory read operations on its external bus in order to fetch (A) a piece of data or the next instruction from the memory location pointed to by the program counter. Since the processor typically runs at a much faster speed than the external bus and memory, it stalls when performing a read operation from memory. The cache memory has a much faster access time than the traditional memory. Virtually all advanced processors include a high-speed cache on the chip of the processor itself. The cache memory is located between the central processing unit (CPU) and the main memory. An important use of the cache memory is to hold the data that are most likely to be accessed again. The processor always attempts to access data in the cache memory before it accesses the main memory. When the processor requests an instruction from a particular memory address, the cache performs a very fast lookup in its directory to determine if the requested instruction is already in the cache. If the requested instruction is found in the cache memory, it is passed to the processor directly. Otherwise, the processor fetches a data block containing the requested instruction from the main memory. In the meantime, the fetched block is also put in the cache. Since the bus and memory are very slow, the data block fetched from memory is relatively larger than the requested instruction itself, under the assumption that the information in the data block will be accessed again. This block of information is referred to as a cache block, or line. The cache memory is very effective in reducing memory access latency in the computer.

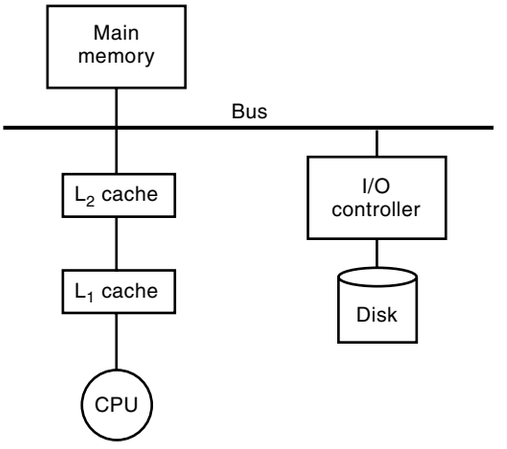

Figure 1. Typical memory hierarchy of CPU, L1 cache, L2 cache, main memory, and secondary memory.

It is possible to include another cache outside the processor (off-chip cache) that intercepts all memory requests and performs a lookup. If the request can be serviced from the level 2 (L_2) cache, the requested information is then returned to the processor. The L_2 cache can be fast enough to match the processor's core speed. Thus, the delivery of information from the L_2 cache is substantially faster than an access to main memory. Figure 1 illustrates a bus-based system with a two-level cache structure. The level 1 (L_1) cache, the one closest to the processor, can be small enough to match the clock cycle of the processor, while the higher-level caches (e.g., the L_2 cache) can be large enough to hold most of the data accessed by the processor. Under this multilevel cache memory architecture, the processor will first try to get the data from the L_1 cache. If the requested data are not present in the L_1 cache, the request is passed on to the L_2 cache. If the data are present in the L_2 cache, the data will be supplied to the L_1 cache, saved with a copy, and then passed on to the processor. Note that all data in the level 1 cache are always in the level 2 cache. The level 2 cache is said to have the multilevel inclusion property. The higher-level cache (e.g., level 2) usually has significantly more space than the lower level cache (e.g., level 1).

There are two different types of the L_2 cache: look-aside and look-through. The look-aside cache (B) sits off to the side of the processor bus and watches for memory accesses from the processor. When it sees one, it performs a lookup in its tag to determine if the requested information is currently resident in the cache. If it is, the cache instructs the main memory not to respond and supplies the requested information to the processor. Since both the cache and main memory respond to memory accesses concurrently, system bus is less available for the memory accesses from other processors (in a multiprocessor environment) or intelligent bus masters. Unlike the look-aside cache, the look-through cache (C) sits in between the processor and the system bus. The look-through cache attempts to fulfill all memory accesses from its cache. If the lookup results in a hit, the system bus isn't needed at all. This leaves the system bus more available for other processors or intelligent bus masters. If the lookup is a miss, the L_2 cache then uses the system bus to fetch the requested block from main memory. Note that the look-through cache introduces a cache lookup latency in the memory accesses for the missed data.

For the memory write operations, there are two different ways to handle them: write-through or write-back policy. These two policies are also referred to as the store-through and store-in policies. For the write-through cache, if the data to be written to the memory are resident in the cache, the data are updated in the cache. In addition, the data are also updated in the main memory by performing a memory write bus transation. In other words, the write-through policy always ensures that the data in the cache and memory are consistent. For the write-back cache, the written data are updated in the cache immediately. The update of data in the memory is delayed until a cache flush instruction is performed or the data in the corresponding cache block are being chosen as a replacement victim.

Research and development of systems with multiple processors have shown them to be able to deliver high computing power to today's typical applications. These systems consist of two basic types: distributed-memory multicomputer and shared-memory multiprocessor. In distributed-memory systems, multiple independent processing nodes with local memory are connected by a general interconnection network. The communications between processes on different processing nodes involve explicit send/receive operations. Programmers must take care of data distribution across different nodes and explicitly manage communications for processes in different processing nodes. The synchronization of processes executing in parallel is implicitly embedded in the send/receive operations performed in different nodes. Most programmers find this programming model more difficult than the programming model with single address space used in uniprocessor systems.

Shared-memory multiprocessors have become very popular mainly because they offer an identical programming model as uniprocessor systems. In shared-memory systems, multiple processors are also connected by an interconnection network. The global physical memory is accessible to all processors. In other words, any processor in a shared-memory system can directly access any location in the global memory address space using read (load) or write (store) operations. As in uniprocessor systems, the memory is decomposed into a number of equally sized blocks. A block is the basic unit of data transferred in the systems. In small-scale shared memory multiprocessors, the global physical memory is made tightly coupled, called uniform memory access (UMA) model shown in Fig. 2, where each node has a uniform memory access latency.

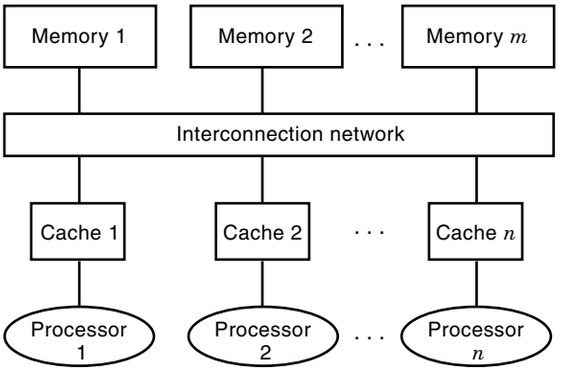

Figure 2. UMA shared memory model.

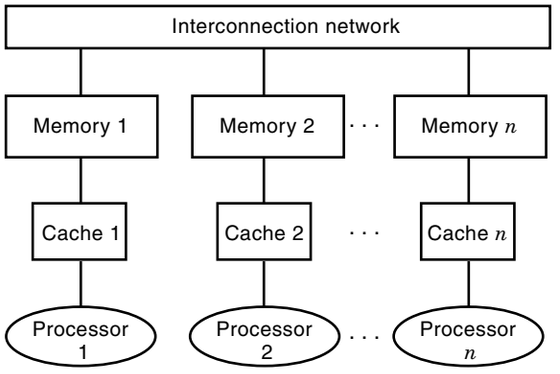

Figure 3. NUMA shared memory model.

The UMA multiprocessor systems are also called symmetric multiprocessor (SMP). Typically, a bus interconnect is used in SMP. Examples of commercial SMPs are four-processor Pentium Pro-based *quad pack,* Sun Ultrasparc-based Enterprise server, SGI Power series, SGI Challenge system, and SGI Power Challenge system. Although the cost of adding a processor to a bus is small, aggregate communication bandwidth on the bus is fixed. Dividing this fixed bandwidth among the large number of processors limits the practical scalability of the bus-based SMP. Alternatively, a scalable interconnect can be used, which provides more bandwidth as more processors are added. The global physical memory can also be physically distributed among the processors, called a nonuniform memory access (NUMA) model as shown in Fig. 3, where the memory access latency is dependent on the physical location of memory.

In both shared memory models, communications between processes in different nodes employ read and write operations. Programmers can readily run most of their programs written for uniprocessor systems. The programming ease and portability reduce the parallel program development costs significantly. However, since the communication speed of the interconnection networks cannot match that of processors, the overall system performance does not improve comparably. Introducing local caches will greatly improve system performance, but cache coherence must be maintained when many copies of data are allowed to coexist in different nodes at the same time (1). The NUMA machines with coherent caches are called CC-NUMA systems.

Cache Coherence

With a multilevel cache structure shown in Fig. 1, the same data may appear in different levels of caches and main memory. This disk controller in the figure can perform read and write operations on the main memory directly without the CPU intervention. The disk controller is said to have the direct memory access (DMA) capability. Assume that a program stored in the hard disk has been loaded in the main memory and that the L_1 and L_2 caches hold a consistent copy of some data blocks of the program. Now, the program is modified by a user and is again loaded into the main memory through a DMA transfer. Since the CPU is unaware of any change made by the disk controller, the data among the caches and main memory will be inconsistent. Similarly, output from the main memory to hard disk through DMA transfer also can cause problems. If the data have been modified at the L_1 cache, and not modified in the L_2 cache or main memory, the disk controller will read from the memory. Thus, special care must be taken to ensure that the CPU and the disk controller will obtain the most recently updated data. In order to keep track of different copies of a cache block at different times, a state is assigned to a block. A cache coherence protocol is designed to check these states of a block at different levels and define an appropriate action to maintain coherence. The most commonly used protocol is MESI, which stands for *modified, exclusive, shared,* and *invalid*—corresponding to the names of the four states of cache lines. Variants of the MESI cache memory protocol is used by PowerPC, Intel Pentium, and MIPS R4400 processors. The meanings of these four states are described as follows:

- *Modified.* The cache line has been modified, may be different from main memory, and is available only in this cache.
- *Exclusive.* The cache line is the same as that in main memory and is not present in any other cache.
- *Shared.* The cache line is the same as that in main memory and may be present in another cache.
- *Invalid.* The cache line does not contain valid data.

The detailed cache coherence operation in a multilevel cache is given in Refs. 2 and 3. Here we describe the events briefly. When the CPU generates a read miss, the requested data will be searched at the L_2 cache. If the requested data are not present, it is retrieved from the main memory and stored first in the L_2 cache and then in the L_1 cache. The state of the lines in the L_1 and L_2 caches becomes shared state. Subsequent reads into the shared line do not result in any cache action. A write to the shared line in the L_1 cache causes its content to be updated and its state to be changed to the exclusive state. Also the write operation is written through to the L_2 cache, and the state of the L_2 line is changed to exclusive. A subsequent write to the exclusive line in the L_1 cache updates the corresponding L_1 and L_2 cache lines and changes their state to modified. For any further writes, the L_1 line remains in the modified state and there is no action on the L_2 cache. This is known as a write-once policy.

When a read request to an L_2 cache line in the modified state is issued from a system bus, the L_2 cache causes the system bus to back off and passes the request to the CPU. The CPU performs a write-back to update main memory and the L_2 cache line. The state of the corresponding lines in the L_1 and L_2 caches is changed to shared. The L_2 cache then releases the bus to perform its read operation. If the system bus makes a write request to the L_2 cache line in the modified state, then again the L_2 cache temporarily blocks the action. The L_2 cache passes the write request to the L_1 cache. The L_1 cache evicts the modified line to main memory and declares the affected line as invalid. Then the L_2 releases the bus, allowing it to complete the write operation.

The situation becomes more complicated in a multiprocessor system, where each processor contains its own private cache, possibly multilevel. A copy of data may exist simultaneously in caches of several processors. Thus, various processors can all execute on the same data concurrently. However, we must make sure that an update on data in one cache

is immediately reflected in all other caches containing the same data. The consistency of the copies of the same data must be maintained in order to make programs execute correctly.

For example, assume that a shared variable X in the main memory is set to 12 initially. At time t_1, processor P_0 reads variable X. P_0 fetches X from the main memory and places it in its cache. At time t_2, processor P_1 reads variable X. P_1 also fetches X from the main memory and places it in its cache. At time t_3, P_0 writes X with a new value 16. The variable X will have values 12, 16, and 16 in P_0's cache, P_1's cache, and main memory, respectively, after time t_3 if there is not any cache coherence mechanism enforced. Various protocols for solving the cache coherence problem have been suggested and are addressed in subsequent sections.

Broadcast-Based Protocols

Several cache coherence schemes have been proposed in the literature (4) to solve the cache consistency problem in multiprocessor systems. The most popular cache protocols used in the bus-based systems are the snoopy protocols, mainly because of their simplicity and low cost of implementation. The bus is a single set of wires connecting several devices, each of which can observe every bus transaction simultaneously. This broadcasting nature of the bus allows each cache controller to observe or "snoop" all memory references on the shared bus. An appropriate action is taken to maintain the cache coherence by each cache controller when a bus transaction involves a memory block, of which it has a copy in its cache. Extra state information of a shared data block is added to the associated cache and memory blocks for facilitating the efficient cache coherence operations. Read hits can always be performed locally in the cache and do not result in any transaction on the shared bus. When a processor encounters a read miss, the requested data will be provided by the main memory or by the another processor that owns the data exclusively. If a cache observes a write to the data that it has a copy of, the cache coherence protocol will either invalidate or update the cached copy, depending on the type of protocol used. When a dirty piece of data in the cache is replaced, it has to be written back to the main memory. Otherwise, no action is taken.

Write Invalidate Protocol. In this protocol, a processor is allowed to change its local copy only after all copies in other caches are invalidated. The invalidation process starts with an invalidation signal sent over the bus by the writing processor. All caches obtain the invalidation message by snooping on the bus and check to see if they have a copy of the data block containing the referenced word; if so, the cache block must be invalidated. This scheme allows multiple readers but only a single writer. The examples of invalidation-based protocols are Goodman's write-once (5), the Write-Through (6), the Berkeley (7), the Illinois (8), the Synapse system (9), and the ones used in Sequent Symmetry and Alliant FX multiprocessors.

Write Update Protocol. Instead of invalidating all copies of the shared blocks, the writing processor broadcasts the new value of the data over the bus. All copies are then updated with the new value. The memory copy is also updated if write-through caches are used. For write-back caches, the memory copy is updated later when the cache block is being replaced.

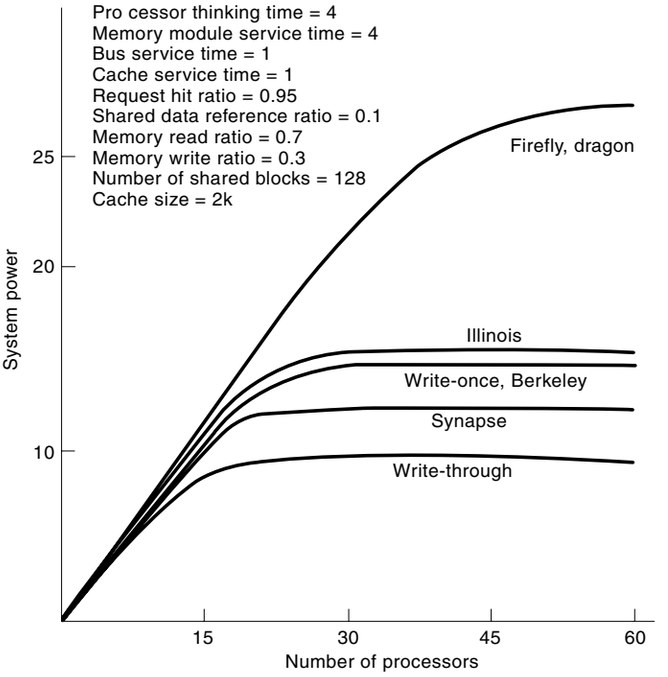

Figure 4. Performance comparison of various cache coherency protocols assuming the probability that a selected private block for replacement is modified is 0.4 and that probability of a write hit on a clean private block is 0.1.

The examples of update-based protocols include the DEC Firefly (10) and Dragon (11) protocols.

The performance comparison of various cache coherency protocols of bus-based multiprocessors has been analyzed under various system parameters by queuing network models in Ref. 12. The results are illustrated in Fig. 4, where the system power is defined as the product of the processor utilization and the number of processors in the system. For the assumed parameters, it shows that Dragon and Firefly perform better than the other protocols. The broadcasting of the new data provides a higher hit ratio for the shared blocks. However, the relative performance of the invalidate and update protocols depend heavily on the applicatons. It has been shown that an invalidation protocol usually performs better than an update protocol.

Besides buses, the point-to-point connected rings with high-speed broadcast capability are also used as the broadcast media for snoopy protocols. Each cache connected to the ring can observe all the memory references circulated on the ring and can make appropriate responses. The ring-based systems can connect more processors due to the point-to-point nature of communication. Examples of ring-based systems are Express Ring (13) and KSR1 (14).

The main problem with the broadcast-based protocols is that they rely on a bus or ring that becomes saturated when a large number of requests are generated by the processors attached to the bus. Thus, the physical limitation of a bus or ring is the number of processors that can be connected to it before it becomes saturated. A scalable system is the one whose performance increases linearly with the number of processors in the system. Obviously, the broadcast-based systems do not scale well. The single bus or ring becomes the bottle-

neck of the system. Multiple bus systems can increase the scalibility to a limited extent. Hierarchical multiple-level snoopy buses and rings have been used to design scalable systems. Examples are Encore Multi, KSR1, and recent SunE10000 systems.

Directory-Based Protocols

In order to make shared memory multiprocessors scalable with respect to a large number of processors, point-to-point and switching networks are normally employed. Since the broadcast procedure generates much network traffic on the point-to-point networks, non-broadcast-based protocols must be developed to maintain cache coherence on shared memory multiprocessors. Without broadcast, the protocols must know which caches should be notified for maintaining consistency when a shared piece of data is modified. A logical list of caches holding the same shared piece of data must be maintained so that write notifications are sent only to the caches that have a copy of the data. The list of caches holding a copy of a piece of data is stored in what is known as a "directory." We will refer to the cache coherency information stored in the memory blocks as memory directory and will refer to that stored in the cache blocks as cache tag. The directory may be centralized or distributed, depending on the protocols used. Bit map and linked list are two basic data structures used for implementing the directory. Details of these directory cache coherence protocols are described in the next section, and we give an overview of these protocols below.

For each memory block in the system, the full-map directory scheme maintains a bit map that contains the information about which node has a shared copy of the data in the memory block. A minimal number of state bits are stored in each cache block. When a read or write miss occurs, a request is sent to the home memory module as determined by the address of the requested data. Upon receiving the read-miss request, the home memory module sends a reply along with the data to the requesting node and sets the corresponding bit in the directory. Thus, it takes two messages to serve a read miss request. Upon receiving a write-miss request, the home memory module first sends invalidation messages to all processors caching the data and waits for acknowledgments. The number of invalidation messages is thus proportional to the number of processors that have cached the particular data block. Also, the storage overhead of the bit maps necessary to maintain the directory is large, and becomes prohibitive as the size of the system grows. Several commercial multiprocessors based on a full-map directory cache coherence protocol are currently available in the market. Examples include SGI Origin, HP V2200, and Sequent NUMAQ machines.

To keep the directory size manageable, limited cache coherency protocols have been proposed in the literature that limit the number of pointers in the directory. In such cases, a special action must be taken for the pointer overflow problem when the number of processors needing the data block is more than the number of pointers available in the directory. Typically, a broadcast mechanism or a randomly replacement policy is used to handle the pointer overflow problem. The broadcast mechanism may cause unnecessary invalidation messages. The random replacement policy may lead to serious degradation in performance for some applications which require that the data be read by a large number of processors. Such a high degree of sharing leads to thrashing in the limited directory scheme. To solve the pointer overflow problem, Chaiken et al. (15) proposed a scheme, called the LimitLESS directory scheme, where the directory size is limited by storing a limited number of pointers in hardware, but the exceptional cases that lead to thrashing due to limited directory space are handled by the software. The efficiency of the LimitLESS protocol depends on the rapid trap-handling and context-switching abilities of the processors.

One way to reduce the storage overhead in the directory scheme is to use linked lists instead of a sparsely used bit map to keep track of multiple copies of data. In addition to the state information, some pointers associated to each cache block are also needed to form a linked list for tracking the processors caching the corresponding data. The IEEE Scalable Coherent Interface (SCI) standard project (16) and Stanford's Distributed-Directory protocol (17) apply this approach to implement a scalable cache coherence protocol. In this approach the storage overhead is minimal, but maintaining the linked list is complex and time-consuming. The protocol is oblivious of the underlying interconnection network. Therefore, a request may be forwarded to a distant node, although it could have been satisfied by a neighboring node. The major disadvantage is the sequential nature of the invalidation process for write misses. The scalable tree protocol (STP) (18) and the SCI tree extensions (19,20) were proposed to reduce the latency of write misses. The low latency of read misses is sacrificed in order to construct a balanced tree connecting all the shared copies of a cache block. The large number of messages generated for read misses, however, makes it prohibitive for an application with a smaller degree of data sharing.

To take advantages of both the limited bit-map and tree-based linked list protocols, a hybrid cache coherence scheme was proposed in Ref. 21. The hybrid scheme aims at reducing the latency of both read and write misses. The main idea is to utilize the sharing information available from the limited number of pointers in the directory and construct an appropriate number of trees that may not be balanced.

The rest of this article is organized as follows. In the section entitled "Existing Directory-Based Schemes," existing directory schemes are discussed. Performance comparisons between different protocols are given in the section entitled "Performance Evaluation," by using an execution-driven simulation. Finally, concluding remarks are presented in the section entitled "Conclusion."

EXISTING DIRECTORY-BASED SCHEMES

In our discussion of the coherence protocols, we use the following naming conventions for various processors and memories involved in any given transaction. A *requesting processor,* or simply *requester,* is the processor that originates a given request for a given data stored in a memory block, while a *home processor / memory module,* or simply *home,* is the processor/memory module that contains the requested data and its associated directory. The *owner* of a memory block is nominally the home processor. However, if the data of a memory block are present in the dirty state in a remote processor's cache, then the remote processor is the owner. The owner processor has an exclusive right to read a data from and write a data to any location in the memory block.

Existing directory schemes fall into two categories, namely bit-map and linked list protocols. A nomenclature, $\mathrm{Dir}_i X$, was introduced in Ref. 22 for bit-map coherence protocols. The index i in $\mathrm{Dir}_i X$ represents the number of pointers in the directory of a memory block for recording which processors have cached the data in the memory block, and X is either B or NB depending on whether or not a broadcast is issued when the pointers overflow and a write occurs. In Ref. 23, a generalized notation $\mathrm{Dir}_i \mathrm{H}_X \mathrm{S}_{Y,A}$ was introduced for clearly articulating the differences between various implementations using hardware and software extensions. The subscript X of $\mathrm{Dir}_i \mathrm{H}_X \mathrm{S}_{Y,A}$ denotes the number of pointers recorded in hardware when software extension exists, otherwise, it is B or NB like the X in notation $\mathrm{Dir}_i X$. If a software-extension exists, Y represents B or NB and A represents how software handles the acknowledgments.

Both the notations given above are only suitable for bit-map protocols. We introduce a new notation $\mathbf{Dir}_i\mathbf{Tree}_k$ for the linked list protocols that will cover all the existing linked list protocols. The subscript i in Dir_i represents the number of pointers in the directory, and subscript k in Tree_k is the fan-out of the tree. For example, Stanford's singly linked list protocol (17) and SCI (16,24) belong to $\mathrm{Dir}_1\mathrm{Tree}_1$ because they have a single pointer in the directory pointing to the head of a list. Note that $\mathrm{Dir}_i\mathrm{Tree}_k$ does not distinguish the difference between singly linked list protocol (i.e., with only forward pointer) and double linked list protocol (i.e., with both forward and backward pointers). STP (18) belongs to $\mathrm{Dir}_2\mathrm{Tree}_k$ because it maintains a k-ary tree and keeps pointers to the root of the tree and the latest node joining the tree. Similarly, the STEM tree extension to SCI (19) belongs to $\mathrm{Dir}_1\mathrm{Tree}_2$ because it maintains a balanced binary tree and keeps one pointer to the latest node joining the tree. Our tree-based protocol (21) is a $\mathrm{Dir}_i\mathrm{Tree}_k$ scheme with only forward pointers.

Bit-Map Schemes

The schemes based on bit maps employ a centralized directory containing pointers to the caches holding a copy of the shared data. The centralized directory is only kept in the corresponding memory block as a field along with the data. Only a minimal number of state bits are used for maintaining cache consistency in each cache block. All the read and write misses are handled by the corresponding directory controller in a centralized manner.

Full-Map ($\mathrm{Dir}_n\mathrm{NB}$). In the full-map directory protocol proposed by Censier and Feautrier (25), the directory of a memory block contains n *presence* bits and a *state* bit in an n-node system. Every processor is associated with a presence bit. When a presence bit is set, it means that the associated processor has a copy of the data in its cache. The state bit encodes two states of the memory block, namely, *valid* and *dirty*. The memory block is dirty when the state bit is set; otherwise, it is in the valid state. When the memory block is in the dirty state, one and only one validation bit is set and the associated processor has a cache block storing the corresponding data in the dirty state. For each cache block, two state bits are used to encode three states of cache blocks, namely, *invalid, valid,* and *dirty*. If a processor has a cache block in the dirty state, then it is assured of having the only copy and has permission to read and write the block. If the cache block is in the valid state, some other processors may also have a cached copy of the data at the same time.

On a read miss, the requester first sends a read-miss message to the home memory module. If the corresponding memory block is in the valid state, the memory module supplies the data to the requester directly. The presence bit associated with the requester is then set. Figure 5(a) illustrates details of the state transition and messages transferred for serving a read miss initiated from processor P_2. If the dirty bit is set, the memory module sends a message to the owner of the shared block. The owner then writes the data back to the home memory and changes the state of the corresponding cache block to valid. Upon receiving the writeback data, the home memory module supplies the data to the requester and sets the memory block in the valid state. Thus, read misses require two messages if the data are valid in the memory block and four messages if the data are dirty.

On a write miss, the requester sends a write-miss message to the home memory module in order to get permission before the write can be performed. In the case that the memory block receiving the write-miss message is in the valid state, the memory module invalidates all the valid copies except the requester in the system and waits for the acknowledgments. After receiving all the acknowledgments, the home memory module supplies the data to the requester and changes the state of the memory block to dirty. Figure 5(b) illustrates details of the state transition and messages transferred for serving a write miss initiated from processor P_3. In the case that the memory block is dirty, the memory module informs the owner of the data to write the dirty data back to the memory and invalidate itself. After receiving the writeback data, the memory module sends the requested data to the requester and keeps the memory block to the dirty state. In both cases, the requester changes the state of the cache block to dirty after it receives the requested data from home memory module. The time taken for invalidation is proportional to the number of valid copies which can be large for applications with a large degree of sharing.

When a cache block is selected to be replaced, no action is taken if the cache block is in the invalid or valid state. However, if the cache block is in the dirty state, it has to write the data back to the home memory. After receiving the writeback data, the memory block is then set to the valid state. In both cases, the state of the cache block is then set to invalid. Notice that it is possible that an invalidation message could be sent to a processor which does not cache the data because of the replacement.

The advantage of this scheme lies in that only the processors that have or had cached the data receive the invalidation messages. Another advantage is that there is no replacement overhead if the replaced cache block is in valid or invalid state. One disadvantage of the full-map protocol is its centralized behavior for memory references. It is possible that the controller of a memory module containing many widely shared data blocks becomes a bottleneck because all the read/write misses go through it. Another disadvantage is that the directory memory requirement is not scalable. The amount of the directory memory in the n-node system is $n[B(n + 1) + 2C]$ bits, where B and C are the numbers of memory blocks and cache blocks in each node, respectively.

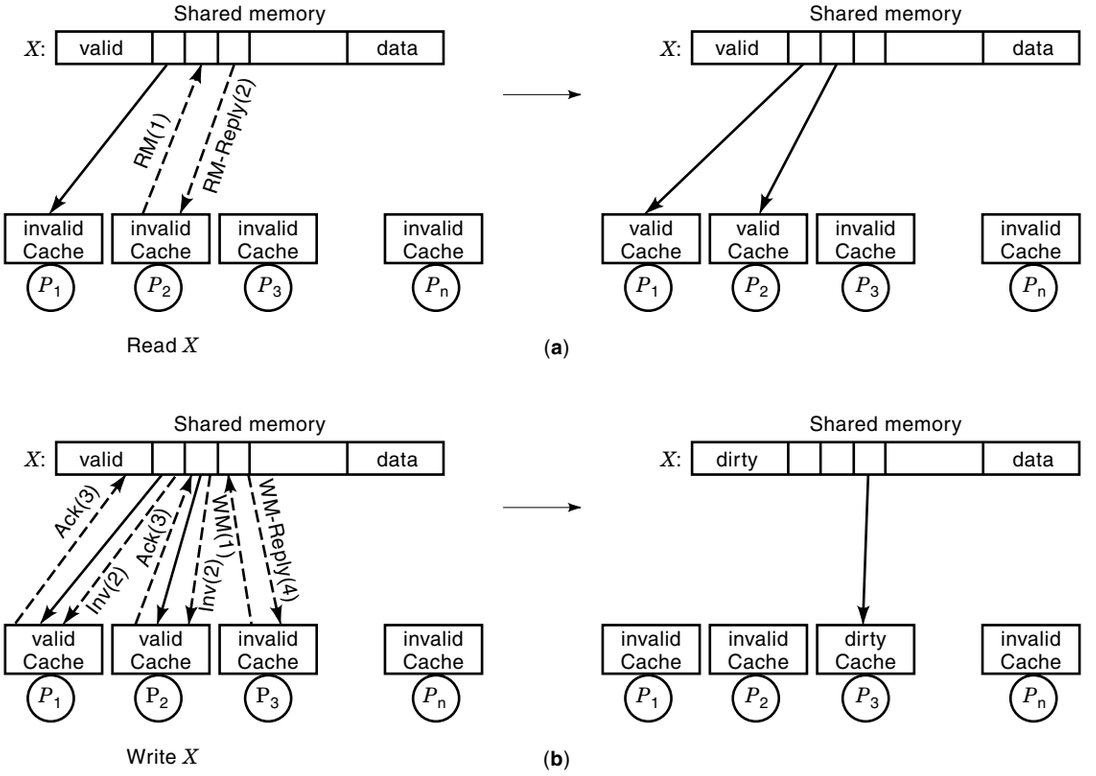

Figure 5. The state transitions and message movements for read and write misses in the full-map protocol. (a) Details of the state transition and messages transferred for serving a read miss initiated from processor P2. (b) Details of the state transition and messages transferred for serving a write miss initiated from processor P3.

Limited Directory Schemes (Dir$_i$B or Dir$_i$NB). Limited directory schemes employ a limited number of pointers to record which processors have a copy of the data. The main idea behind these schemes is based on the empirical results that in most of the applications, only a small number of processors (less than four) share a memory block most of the time. Thus, the limited schemes perform as well as the full-map scheme for most applications. The advantages of having a limited number of pointers are the scalable memory requirement and faster hardware support. If the pointers are not sufficient to record all nodes having the shared copies (i.e., pointer overflow), a mechanism must be employed to deal with the pointer overflow situation.

In the limited directory broadcast scheme Dir$_i$B (22), an additional *overflow* bit for each memory block is employed to handle pointer overflow. If there is no pointer available in the directory for a request, copy of the block is supplied and the overflow bit is set in the directory for that block. Not every processor having a shared copy is recorded in the directory. In case of a write, invalidation messages will be broadcast to all the processors in the system to maintain cache coherence. This scheme performs poorly if the number of shared copies is just greater than the number of pointers. Each of the i pointers in each memory block requires $\log n$ bits to store the processor identification in an n-node system. For each pointer, an additional bit is needed to indicate if it contains a valid processor number. With two state bits for each cache block, the memory requirement for limited directory scheme is $n[B(2 + i + i \cdot n \log n) + 2C]$, where B and C are the numbers of memory blocks and cache blocks in each node, respectively.

The nonbroadcast scheme Dir$_i$NB avoids the broadcast designed for solving the pointer overflow problem in Dir$_i$B by invalidating one of the processors having a copy of the block and replacing it with the current requesting processor in the directory. This scheme does not perform well when the number of shared copies is much greater than the number of the pointers and frequent read requests are issued by the processors holding a copy of shared data. The memory requirement for the Dir$_i$NB limited directory scheme is $n[B(1 + i + i \cdot n \log n) + 2C]$, where B and C are the numbers of memory blocks and cache blocks in each node, respectively.

In LimitLESS$_i$ (15,23) and Dir$_1$SW (26,27), the pointer overflow problem is solved by software. MIT Alewife (15) maintains coherence with a LimitLESS directory that has four pointers in hardware. All the pointers that cannot fit into the limited hardware-supported directory space are stored in the traditional memory by trapping to the software handler. The programs with a large degree of sharing will cause traps frequently and thus run slowly. The delay in calling the software handler is their major disadvantage. However, the primary benefit is the simplicity of cache coherence hardware.

Other limited schemes proposed in Refs. 22, 28, and 29 handle the pointer overflow problem in different ways. These schemes change the formats of the bit maps so that the bit

maps present as few processors as possible. Thus, the invalidation messages are broadcast to a subset of processors, instead of all the processors. However, all these schemes generate some unnecessary invalidation traffic in some cases, thereby degrading the system performance. Since these schemes are more complex than full-map scheme, they make the design of hardware-supported directory difficult.

Linked List Schemes

The schemes based on linked lists employ a distributed directory among the main memory and the caches. It is different from bit-map schemes in that there are pointer fields in both the memory blocks and cache blocks. The use of these pointers is to organize the set of caches holding a copy of the shared data in a linked list or tree structure. These schemes reduce the size of the directory and do not require invalidation messages to be sent to all processors.

Singly Linked List Protocol. The singly linked list protocol proposed in Ref. 17 forms the directory by chaining the memory block and the cache blocks having the shared data as a singly linked list. Each cache block only keeps a pointer to a node that also caches a copy of the same data. The home memory block points to a node called *head,* which is the last one joining the linked list. The head in turn uses its pointer to point to another node that also has a valid copy. Continuing the above pointing process, a singly linked list is formed. The last node in the list called *tail* points back to the home memory module.

On a read miss, a request is first sent to the home memory module. If a copy of the shared data does not present in any other cache block, the memory module directly sends the reply to the requester. Otherwise, the memory module informs the head of the list to send the reply consisting of the data and the ID of the head node to the requester. In the meantime, the memory module updates the pointer of the memory block to point to the requester. Upon receiving the reply, the requester sets its pointer to point to the head node, as indicated by the ID of the head node. The requester now becomes the new head of the list. Figure 6(a) illustrates the process of a read miss issued from node P_2. P_2 first sends a read-miss (RM) message to the home memory module. The home memory module then sends a read-miss-forward (RM-F) message to the head, P_1, and sets the pointer to P_2. When P_1 receives the forwarded message, it sends a read-miss-reply (RM-R) message to P_2. After receiving the reply from P_1, P_2 then sets its pointer to point to P_1. Thus a linked list is formed.

When a write miss occurs, a write-miss message is sent to the home memory module. The memory module updates its pointer to point to the requesting node and then sends a write-miss-forward message to invalidate all the shared copies in the system by following the pointers on the linked list. When the node receives the write-miss-forward message, the copy of the data in its cache must be invalidated. In addition, the head is also responsible for supplying the requested data to the requester. The tail of the list must send an acknowledgment to the requester to indicate the completion of the invalidation process. The write operation is considered to be performed after the requester receives both the requested data from the head and the acknowledge from the tail of the list. Figure 6(b) illustrates the process of a write miss issued from node P_3. P_3 sends a write-miss message to the home memory module. The home memory module sends the write-miss-forward (WM-F) message to the old head (P_2), and the pointer is updated to point to the requester (P_3). When P_2 receives the WM-F message, it invalidates its copy and forwards the message to P_1. Since the pointer of P_1 points to the home memory module, it can determine locally that P_1 is the tail of the list. Thus, P_1 invalidates its copy and sends an acknowledgment to the requester. The write is complete after P_3 receives both the requested data from P_2 and the acknowledge from P_1.

Replacement of a cache block in a shared list is done by invalidating the lower portion of the list. A race may happen

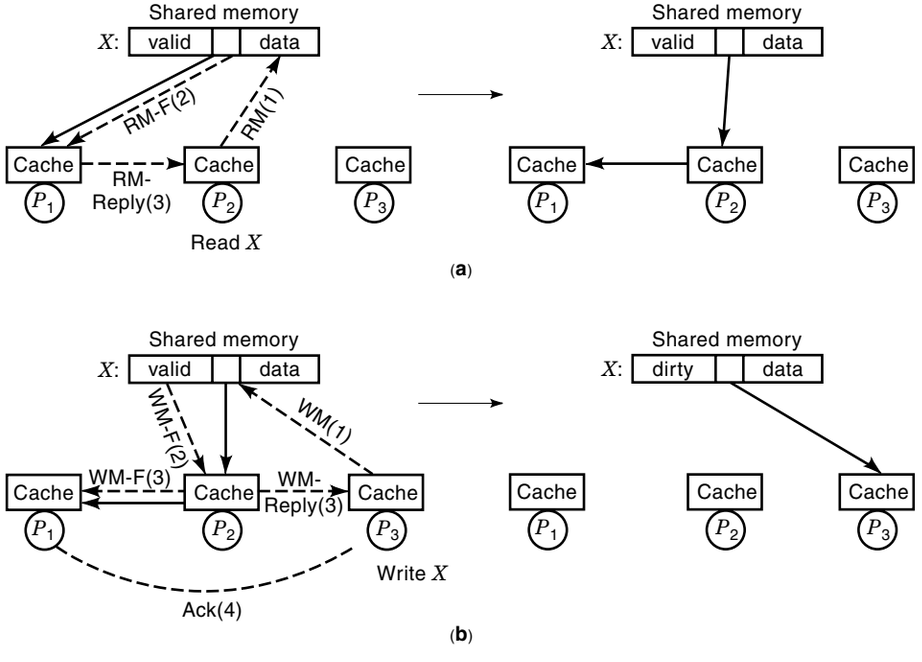

Figure 6. The state transitions and message movements for read and write misses in the singly linked list protocol. (a) Process of a read miss issued from node P2. (b) Process of a write miss issued from node P3.

during this replacement. Thus, a care must be taken since the race condition is complex. The sequential invalidation process results in poor performance for a write on an address with a long shared list. The directory memory requirement for this protocol is $O(n(C + B) \cdot (\log n + 2))$ bits, where B and C are the numbers of memory and cache blocks in each node (30), respectively.

Scalable Coherent Interface. Scalable coherent interface (SCI) is an IEEE standard (P1596) (16,24). The motivation behind SCI is to allow multiple vendors to develop components of a computer system that follow the SCI specifications. A parallel computer can be built by integrating these components. SCI specifies a topology-independent network and a cache coherence protocol. The SCI cache coherence protocol is based on a noncircular, doubly linked list of cache blocks to keep track of cached copies. The directory at the main memory includes 3-bit state field and 16-bit *forward* field. The forward field specifies the first node in the sharing list. Each cache block contains a 7-bit state field and two 16-bit pointer fields, *forward* and *backward,* which usually point to the adjacent shared caches. Each cached entry has an address that is partitioned into a 16-bit memory-controller identifier and a 48-bit address offset. For entries at the head of the list, the backward field is not needed, since the memory-controller field is part of the address. We now summarize the base SCI protocol. The state names of cache and memory blocks are changed in order to keep a consistent terminology for this chapter.

On a read miss, the requesting node sends a request to the home memory module. The memory module updates the pointer to point to the requester after it receives the read-miss message. If the requested data are not shared by any other cache, the memory module directly supplies the data to the requester. Otherwise, the memory module only sends the pointer pointing to the head to the requester. The head of the list is responsible for supplying the requested data. After receiving the reply from memory, the requester sends a new request to the head of the list for attaching itself to the sharing list. The head returns the requested data after it updates its backward pointer to point to the requester. The requester sets its forward pointer to the head and becomes the new head of the list. Unlike the centralized full-map schemes, requests are never blocked at the memory controller; instead, all requests are immediately added to the head of the existing sharing list.

On a write miss, the requester puts itself as the new head of the list as in the read miss situation. Then it sends an invalidation message to its successor pointed to by the forward pointer and waits for an acknowledgment. After the successor is invalidated and taken out of the list, the requester updates its forward pointer to the successor of its old head and continues the same invalidation process until the tail of the sharing list is invalidated. It takes $2P$ messages to invalidate a list of P cached copies. Adding the four messages for inserting itself as a new head, the writing cache requester takes $2P + 4$ messages to get the write permission. The replacement of a cached copy can be done locally by sending the backward pointer to its successor and the forward pointer to the node pointed to by its backward pointer. Multiple cache blocks can be deleted simultaneously. Special precedence rules are applied to avoid corruption of the linked list when adjacent deletions are issued simultaneously. To ensure forward progress, the cache block closer to the tail has higher priority and is deleted first.

As described above, the invalidation process is done sequentially as in the singly linked list protocol. However, the number of network messages required by the SCI protocol for both read and write operations is higher than those required by the singly linked list protocol. The directory memory requirement is similar to the singly linked list protocol.

Scalable Tree Protocol. The Scalable Tree Protocol (STP) (18) uses a top-down approach to construct a single balanced tree from the caches having a copy of data. Take a binary tree as an example. Each memory block contains three pointers, *root, last,* and *writepending.* Root points to the root of the tree. Last pointer points to the cache called C_L that caches the shared data most recently. Writepending pointer points to the cache with a pending write request. Each cache block contains 5 pointers, called *father, son*[0], *son*[1], *backward,* and *forward.* The father pointer of a cache block is used to point to its father node in the tree. The two pointers of a cache block, son[0] and son[1], point to its two children. Pointers backward and forward are used in the same manner as in SCI protocol.

The first node issuing a read request to a specific memory block will be the root of the tree. The second and third nodes issuing read requests will be the children of the first one. Similarly, the fourth and fifth nodes making a read request will be the children of the second node. Continuing the same procedure, a balanced tree is formed. Basically, a read request is first sent to the home memory, and the reply contains the data and the ID of C_L. The requester will later become the new C_L. The requester sends another request to C_L. C_L then sends the ID of C_L's father to the requester and set the forward pointer to the requester. On receiving the reply from C_L, the requester set its backward pointer to C_L and sends another message to C_L's father, asking permission to join the tree as its son. If C_L's father already owns two sons, the read requester will exchange two more messages with the successor of C_L's father and join the tree as its son.

On a write miss, a write-miss message is sent to the home memory module. The home memory then sends a message to C_L to check if the last addition of a node is complete and gets an acknowledgment from it. Then the invalidation process following the tree structure can be performed. The replacement of a cache copy A is done by replacing A with C_L.

This protocol attains a logarithmic time invalidation process by constructing a balanced tree, but paying the price of generating too many messages for read misses. Since most of the requests in an application are read misses, the protocol performs poorly when the degree of sharing is low or write misses is infrequent.

STEM Tree Extension to SCI. To improve the performance for widely shared data, this scheme has the consensus of the SCI working group for use as an extension to SCI, officially IEEE P1596.2 (19). This scheme is a binary tree protocol that allows parallel tree insertion and deletion, while maintaining a reasonably balanced tree for write operations. The height of the tree is balanced during insertions by the AVL tree rotation algorithm. The AVL tree balancing property is that the heights of two subtrees of a node differ by at most one. The

directory at the memory contains one pointer to the most recently inserted node, called insertion point of the binary tree. The insertion point is one of the leaf nodes in the balanced binary tree. The nodes traversed by forward pointers from the insertion point to the root of the tree are called *list nodes.* As in SCI, four states are encoded in the directory. Each cache block contains three pointers called *forward, backward,* and *downward* pointers, a 5-bit *height* field, and a 10-bit *state* field. In contrast to the AVL balancing property, this scheme requires that the right height of every list cache block in the binary tree be strictly greater than the right height of its backward neighbor, called *strictly growing* property.

On a read miss, a read-miss request is first sent to the home memory. After receiving the read-miss request, the home memory module updates its pointer field in the directory to point to the requesting node and sends a response message back. The requesting cache now becomes the new insertion point of the sharing list. It then sends a rotation request to the old insertion point. If the strictly growing property is not satisfied, the node at the old insertion point sends a rotation reject message back to the requesting node and then the insertion operation is complete. Otherwise, a rotation is performed and a rotation-accept message is sent back to the requesting node. After receiving a rotation-accept message, the requesting node sends another rotation request to the forward neighbor for asking a possible rotation. This process repeats until the strictly growing property is satisfied.

On a write miss, the requesting node first becomes the node at the insertion point, using the deletion and/or insertion protocols if necessary. Recall that the node at the insertion point is a leaf node of the balanced binary tree. The requesting node sends the invalidation message to the list node pointed to by its forward pointer (its parent), again to the list node pointed to by its parent's forward pointer (its grand parent), until the root list node. Then the subtrees rooted at these list nodes can be purged in parallel.

The replacement of a victim cache block is solved as follows. If the victim cache block has zero or one child, then the child (if any) is connected to the parent (or the directory), as in the SCI's replacement policy. However, if the victim has two children, the victim walks down the tree to a leaf node, detaches the leaf from its parent, and finally replaces itself with the leaf node.

As described above, the read miss overhead of this scheme is similar to STP. Thus, it does not perform well for the applications with a low degree of data sharing and less frequent write misses. The write latencies grow logarithmically as the number of nodes in the shared list.

GLOW Tree Extension to SCI. In addition to STEM, GLOW is another kiloprocessor extension to SCI (20). GLOW extensions are intended to be used in SCI multiprocessor systems that are comprised of multiple SCI rings connected through SCI bridges. Only accesses to widely shared data use the GLOW extension protocol by special request commands, while other accesses to data with low degree of sharing are left to standard SCI cache coherence protocol. The extensions are implemented in the bridges that connect the SCI rings. GLOW is a k-ary tree protocol. Recall that in SCI and STEM protocols, caches join the sharing list based on their arriving order. In contrast, GLOW extensions construct the tree in a predetermined way. GLOW maps the underlying topology onto a k-ary tree such that the nodes in physical proximity become neighbors in the sharing list. All GLOW protocol processes take place in strategically selected bridges called *agents* in the network topology. In general, the agents form the k-ary tree. Caches on a ring form a circular doubly linked list (called a child list) that is appended to the agent on the same ring. Since an agent is a bridge that connects many SCI rings, it can have as many child lists as the SCI rings connected. In a child list, the agent acts as its virtual head and tail. All read requests for a data block from nodes of an SCI ring will be routed to a remote memory through the agent. If this agent contains directory information or a copy of the requested data block, these requests are intercepted and satisfied locally on the ring. The agents only intercept the specially tagged requests for widely shared data.

On a read miss, the request is intercepted at the first agent. The intercept causes a lookup in the agent's directory to find information about the requested data block. If the lookup results in a miss, the agent sends another special request upward along the predetermined tree structure to the home memory. The requesting node is informed by the agent to form a circular linked list which is then attached to the agent. After receiving the requested data, the agent passes the data to the requesting node. If the lookup results in a hit, the requesting node gets the data from either the agent or the previous head of the associated doubly linked list.

On a write miss, the requesting node must first become the head of the top-level child list connected directly to the memory directory. In this position, the requesting node is the root of the sharing tree. A node has to roll out from the sharing tree before becoming root. Invalidation messages are forwarded from the root down the tree in parallel. Each agent invalidates the caches on its child list by using the SCI invalidation protocol. When the invalidation messages reach the tails of the lists, they invalidate themselves and send an acknowledgment back. GLOW agents wait for acknowledgments of all the invalidation messages they forwarded, invalidate themselves, and return their own acknowledgments.

For the replacement of a cache block, it follows the standard SCI protocol. Agents roll out because of conflicts in their directory (or data) storage or because they are left childless. Rollout of an agent is based on concatenating its child lists and subsequently substituting the concatenated child lists in place of the agent in a tree. Thus, the child lists may scatter over multiple rings after agents roll out. This replacement policy is more effective than invalidating all of the agent's descendants because of possible shrashing effects.

As we should see, the write overhead is similar to that of STEM extension. The write latencies grow only logarithmically as the number of sharing nodes grows. Since it incurs a great deal of overhead for read operations, it is intended to be used only for the accesses to the data with high degree of sharing. That is, GLOW must be used in combination with the standard SCI protocol, or even other full-map protocols. The read overhead will be the same as the SCI protocol for the data with a low degree of sharing and much higher than SCI for the data with a high degree of sharing.

The Hybrid Protocol. The hybrid protocol combines a limited directory scheme with a tree-based scheme. It is similar to the limited directory protocol, except in the cases when the number of caches requesting read copies of a particular data

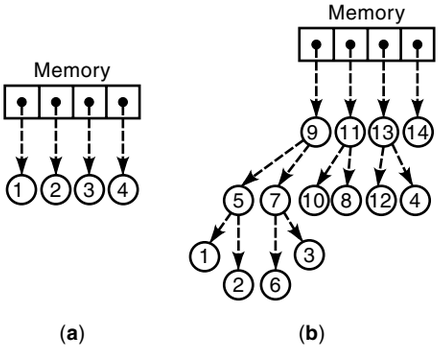

Figure 7. Two organizations of trees constructed under the Dir_4Tree_2 scheme.

block from the main memory module is more than that of the pointers available in the memory directory. The design of the protocol aims at minimizing the communication overhead for constructing optimal or near-optimal trees from the caches having copies of data. Based on the trees constructed, a logarithmic time algorithm can be developed for invalidating all the shared caches when a write miss occurs. Thus, the hybrid protocol possesses the advantages of the bit-map protocol and the tree-based linked list protocol, namely, small read miss latency (two network messages), logarithmic write latency, and scalable directory memory requirement.

As in the limited directory protocol, i pointers are maintained in each memory block. However, each cache block has k pointers, where k must be less than or equal to i. As described later, k-ary trees can be constructed by using the i pinters in the memory directory and k pointers in the cache directory. We call it a Dir_iTree_k scheme. The subscript k must be less than or equal to i. For example, two organizations of trees with 4 and 14 caches having shared copies of data constructed under the Dir_4Tree_2 scheme are illustrated in Fig. 7, where the numbers in the circles denote the arriving sequence of the read requests. The construction of the trees is explained in detail under read misses. The memory requirement is $O(n \cdot B \cdot i \log n + n \cdot C \cdot k \log n)$ in an n-processor system, where B and C are the numbers of memory and cache blocks per processor, respectively.

The empirical results in Ref. 31 suggest that in many applications, the number of shared copies of a piece of data is lower than four, regardless of the system size. Thus, we feel justified in using $i = 4$ and $k = 2$ to construct binary trees. The write operation can be implemented by employing either an invalidation or an update protocol. We illustrate the hybrid scheme by using an invalidation protocol and a strong consistency memory model. In addition to the i pointers in the memory directory, a *level* field is also associated with each pointer to record the heights of the trees for facilitating the construction of optimal or near-optimal trees.

In general, the coherence operations are similar to those in the full-map protocol. The states of cache blocks are *exclusive, valid,* and *invalid.* The states of the memory blocks are *valid* and *dirty,* the same as those in the full-map directory protocol. The major differences between Dir_iTree_k and the full-map protocol lie in how trees are constructed by using the limited number of pointers and in the actions taken for block replacements. As in the full-map directory protocol, the requested block is always provided by the home memory. We discuss the coherence operations for the read misses, write misses, and cache replacements in detail below.

On a read miss, a read-miss request is first sent to the home memory module. The operations to serve a read miss are the same as that in the full-map scheme if a null pointer in the memory directory is available for the request. For example, consider the memory block containing the requested data in the valid state. Upon receiving the read-miss request, the home memory module sets a null pointer to point to the requester and then sends the data to the requester. If there is no null pointer available, two pointers are selected based on the heights of the trees and sent to the requesting node along with the requested data. The nodes which were pointed to by the selected pointers will become the children of the

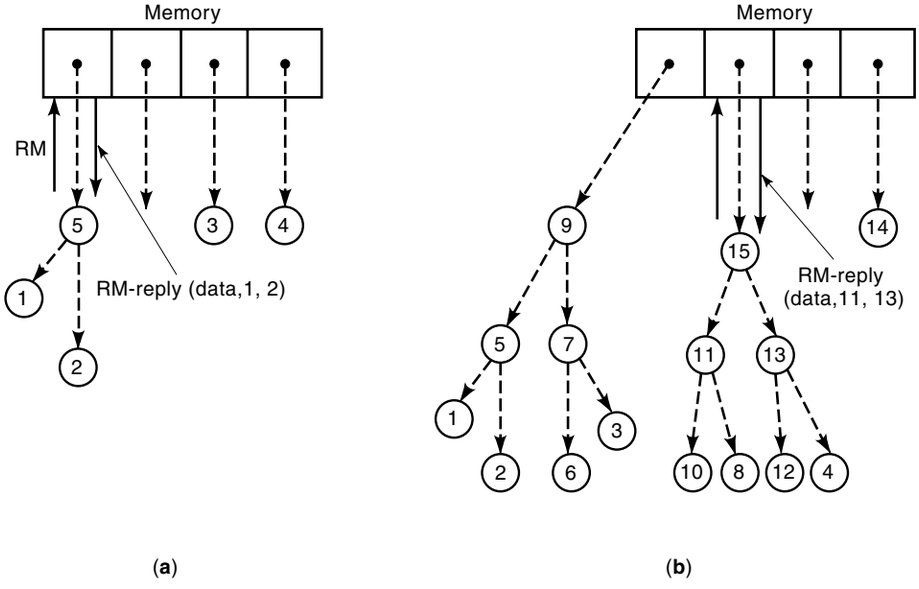

Figure 8. Message movements for a read miss. (a) After the read miss is completed, nodes 1 and 2 become the children of node 5. (b) After the read miss is completed, nodes 11 and 13 become the children of node 15.

```
for (i = 0..3)
   if (p[i] == requester) {
       (data, null, null) → requester; level[i]=1; return; }
for (i = 0..3)
   if (p[i] == null) {
       (data, null, null) → requester;
       p[i] = requester; level[i]=1; return; }
if (level[i] == level[j]), where i, j ∈ {0..3} and i ≠ j {
    (data, p[i], p[j]) → requester;
    p[i] = requester; level[i]++;
    p[j] = null; level[j] = 0; return; }
if (level[i] ≤ level[j]) for all j ≠ i {
    (data, p[i], null) → requester;
    p[i] = requester; level[i]++; return; }
```

Figure 9. Cache coherence operations for a read miss.

requesting node. One of these two pointers in the memory directory is reset to pointing to the requester and the other is reset to null. Figure 8 shows the tree construction process indicated by the involved messages while the fifth and the fifteenth node having a read miss try to join the sharing trees shown in Fig. 7. It can be seen that after the read miss is completed, nodes 1 and 2 become the children of node 5 in Fig. 8(a). Similarly, nodes 11 and 13 become the children of node 15 in Fig. 8(b).

Figure 9 lists in detail the coherence operations for serving a read miss at memory directory. Pointer $p[i]$ and $level[i]$ field for $i = 0.3$ are initialized to null and 0, respectively. The operation $(data, x, y) \rightarrow p$ means that the data along with two pointers x and y are sent to node p. Four different situations are considered in Fig. 9. First (D), the coherence protocol, checks whether or not the requesting node has already been pointed to by one of the i pointers in the memory directory. This dangling pointer problem might occur when a cached block was replaced, and later on it is requested by the same node again. Other situations regarding dangling pointers are addressed when we discuss the replacement policy. The second situation considers the case when a node has a read miss the first time, and there is a null pointer available in the memory directory. As in the bit-map scheme, the available pointer is set to pointing to the requester before sending out the data. The third and forth situations consider the cases when there is no null pointer available in the directory for the next incoming read request. If there are two pointers pointing to two trees with the same height, these two pointers will be sent to the requesting node and the nodes pointed to by these two pointers become the children of the requesting node. Then, one of these two pointers is set to pointing to the requesting node and the corresponding level field is incremented by one. The other pointer is reset to null and the level is reset to 0. The last situation considers the case when there are no two pointers which point to the trees with the same height. The pointer with the smallest level will be selected and sent to the requesting node. The node pointed to by the selected pointer becomes the only child of the requesting node. Then the selected pointer is set to pointing to requesting processor, and the level of the pointer is incremented by one.

Since there are only limited number of pointers in the directory, trees generated by $\mathrm{Dir}_i\mathrm{Tree}_k$ may not be balanced. Table 1 lists the maximum number of processors in the sharing list versus the level of the trees for $\mathrm{Dir}_2\mathrm{Tree}_2$, $\mathrm{Dir}_4\mathrm{Tree}_2$, and STP or SCI binary tree extension. For $\mathrm{Dir}_4\mathrm{Tree}_2$ in Table 1, it can be easily checked that when there are 16 processors in the sharing list, the highest tree is of three levels, which is even smaller than a balanced binary tree of 16 nodes. Similarly, if a 1024-node system is built, the highest tree generated by $\mathrm{Dir}_4\mathrm{Tree}_2$ is of 12 levels, which is only one level higher than the balanced binary tree.

When a write miss occurs, the write request is first sent to the home memory module. Invalidation messages are then sent out to the root nodes of the trees by following the pointers in the directory. The other nodes caching the data are invalidated by the messages originating from their roots. In order to speed up the invalidation process further, the nodes pointed to by odd-numbered pointers receive invalidation message from the nodes pointed to by even-numbered pointers. The home memory module only receives at most half the number of acknowledgments, and thus the possibility of the home node becoming a bottleneck reduces. An example of a write miss operation is shown in Fig. 10, where 15 shared copies are in the shared tree before a write miss occurs. The invalidation message received at node 15 is from node 9. The acknowledgments which are omitted from the figure to preserve clarity follow the reverse direction of the invalidation paths. It can be seen that $\mathrm{Dir}_4\mathrm{Tree}_2$ has a three-level tree which is shorter than the four-level binary tree with 10 nodes

Table 1. Maximum Number of Nodes Constructed in $\mathrm{Dir}_2\mathrm{Tree}_2$ and $\mathrm{Dir}_4\mathrm{Tree}_2$ as a Function of Level

Level	$\mathrm{Dir}_2\mathrm{Tree}_2$	$\mathrm{Dir}_4\mathrm{Tree}_2$	Binary Tree (SCI or STP)
3	9	16	7
4	14	43	15
5	20	75	31
6	27	99	63
7	35	163	127
8	44	256	255
9	54	386	511
10	65	562	1023
11	77	794	2047
12	90	1093	4095

maintained by an STP protocol with binary trees or the SCI tree extensions.

When a miss occurs, a cache block must be selected for storing the requested data before a request is sent to the home memory module for service. If the selected cache block currently holds a valid or exclusive copy of data with a different address, a replacement operation needs to be performed. We propose that when a valid or exclusive cached block is being replaced, the subtree rooted at the replaced cache block be invalidated without informing the home directory. The rationale of doing this is as follows. First, as noted in Ref. 31, most of the time, the number of shared copies of a memory block is less than four. Thus, our replacement operations will perform as well as the bit-map scheme because the replaced cache block does not have any child most of the time. Second, the replacements are not frequent if the set size of an associative cache memory increases. Third, even when the trees grow bigger, most of the replaced cache blocks are positioned as the leaf nodes of the trees. The probability that the replaced cache blocks are nonleaf nodes is low. It should be noticed that the parent of a cache of a tree is not informed when it is being replaced. It causes a cache pointing to another cache that does not have a valid data block. It is the dangling pointer problem which can be easily solved by the same method as in the bit-map protocol. That is, if a cache receives an invalidation message for invalidating the data that it does not have in its cache blocks, it just replies with an acknowledgment to the sender without any further action. The proposed replacement action is simple and easy to implement. It is worthwhile to note that the only possible communication overhead of the hybrid scheme comes from the replacements.

We summarize the number of messages generated by a read or a write miss for the various protocols in Table 2. In Table 2, we can see that the full-map, Dir$_i$NB, LimitLESS, and hybrid protocols generate the smallest number of network messages for a read miss. A smaller number of network messages reflects a smaller amount of time taken to complete a read miss. From the table, we also can see that the singly linked list protocol generates a smaller number of messages than the other protocols for a write miss. However, it is not necessary to mean that the singly linked list protocol must take a smaller amount of time to complete a write miss than

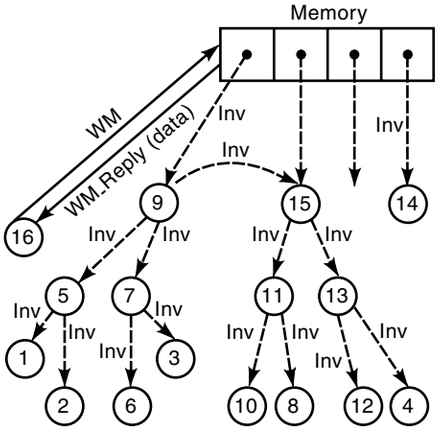

Figure 10. Message movements for a write miss. (For clarity, acknowledgments are omitted.)

Table 2. Number of Messages Generated by a Read or Write Miss for Various Schemes, Where *P* is the Number of Nodes in the Sharing List

Protocol	Read Miss	Write Miss
Full-map	2	$2P + 2$
Dir$_i$NB	2	$2P + 2$ plus unnecessary invalidations and read misses
LimitLESS$_4$	2	$2P + 2$ plus $(P - 4)$ software handler delay
Singly linked list	3	$P + 2$
SCI	4	$2P + 2$
SCI tree extensions	4 to 2 log P	More than $2P + 4$
STP (binary)	4 to 8	$2P + 4$
Hybrid Dir$_4$Tree$_2$	2	$2P + 2$

the other protocols. The reason is that the invalidation process for the singly linked list protocol is sequential. In other words, the caches in the shared list of the singly linked list protocol must be invalidated one after another. The amount of time taken to invalidate a cache in the shared list is the sum of the time for creating a message, transferring the message across the network, receiving the message, and processing the message. Thus, the sequential invalidation process is slow. Note that the invalidation process for SCI protocol is also sequential. The invalidation processes for the full-map, Dir$_i$NB, and LimitLESS protocols may be sequential or parallel. If the memory module requires that the next invalidation message cannot be sent out until the acknowledgment of the previous invalidation message is received, the process is sequential. Otherwise, the invalidation process is parallel. Note that if the memory module in the full-map, Dir$_i$NB, and LimitLESS protocols is responsible for sending and receiving all the invalidation and acknowledgment messages, it may become the bottleneck of the invalidation process if the size of the shared list is large. On the other hand, the tree-based protocols, such as the SCI extension, STP, and hybrid protocols, distribute the load of the memory module over many other nodes. In addition to the parallel message transfers over the network, the tree-based invalidation process is faster than the other protocols. The pros and cons of each protocol are also given in Table 3.

PERFORMANCE EVALUATION

We use four benchmark applications to compare the performance of the hybird Dir$_i$Tree$_k$ coherence scheme with that of the full-map and the limited directory schemes. The applications comprise MP3D, LU decomposition, the Floyd Washall algorithm, and a fast Fourier transformation (FFT) program. We give a brief description of each program, indicating its purpose and the data structure employed as follows.

Simulation Methodology

We ported the hybrid coherence scheme to PROTEUS (32), which is an execution-driven simulator for shared memory multiprocessors. The simulator can be configured to either bus-based or k-ary n-cube networks. The networks use a wormhole routing technique. The specification of the simu-

Table 3. Pros and Cons for Various Protocols

Protocol	Pro	Con
Full-map	Simple to implement	High memory overhead
	No replacement overhead	Sequential invalidation process
	Low read miss overhead	
Dir_iNB	Simple to implement	High invalidation overhead
	Low memory overhead	Sequential invalidation
	Low read miss overhead	
$LimitLESS_4$	Low memory requirement (hardware)	Sequential invalidation
		Slow software handler
Single-link chain	Moderate memory overhead	Sequential invalidation
Double-link chain	Moderate memory overhead	Sequential invalidation
SCI extensions	Logarithmic invalidation	High read miss overhead
		High replacement overhead
STP	Logarithmic invalidation	High read miss overhead
		High replacement overhead
Dir_iTree_k	Low read miss overhead	Replacement overhead
	Logarithmic invalidation	
	Low memory overhead	

lated network and the cache memory is given in Table 4. We compare the normalized execution time for each application running with the various schemes as mentioned above, where the normalized execution time is defined as the relative execution time to that of the full-map scheme. The examined schemes are Dir_nNB, Dir_iNB, and Dir_iTree_2 for $i = 1, 2, 4, 8$.

MP3D. The MP3D application is taken from the SPLASH parallel benchmark suite (33). MP3D solves problems in rarefied fluid flow simulation that are useful for aerospace researchers who study the forces exerted on space vehicles as they pass through the upper atmosphere at hypersonic speeds. MP3D employs a five-degree-of-freedom simulation of idealized diatomic molecules in a three-dimensional space. Two large arrays of structures are used to store the state information for each molecule and the properties of each cell in the three-dimensional space. The work is partitioned by molecules, which are statically scheduled on processors. MP3D is notorious for its low speedups (34). For our simulation, we used 3000 particles and ran the application in 10 steps. The results are given in Fig. 11 for 8, 16, and 32 processors. As expected, the performance of limited protocols (Dir_8NB and Dir_4NB) is the worst due to the delay caused by unnecessary invalidations. The protocol Dir_1Tree_2 creates a linear sharing list instead of a tree-like list. The invalidation process for the linear sharing list becomes sequential, and thus results in worse performance. For 8-processor and 16-processor systems, the full-map scheme is the best because the degree of sharing for most shared blocks in MP3D is low (31). It is shown that Dir_4Tree_2 is only less than 5% slower than the full-map scheme and much faster than the limited directory schemes Dir_4NB and Dir_8NB.

However, as the size of the system increases from 16 to 32 processors, Dir_2Tree_2 and Dir_4Tree_2 perform even better than the full-map scheme. The reason is as follows. As the size of the system increases, it is quite possible for different processors to access a given space cell during the same time-step. Thus, the number of shared blocks with larger degree of sharing also increases. It takes less time for Dir_2Tree_2 and Dir_4Tree_2 to invalidate the shared blocks with larger degree of sharing than the full-map scheme.

Table 4. Simulation Model

Data cache	16 kbytes
Block size	8 bytes
Cache associativity	Fully associative
Network type	Binary n-cube
Network size	8, 16, 32 processors
Network bandwidth	8 bits
Switch/wire delay	1 cycle
Memory access latency	5 cycles
Cache access latency	1 cycle

LU Decomposition. The LU application is also taken from the SPLASH parallel benchmark suite (33). It is a parallel version of the dense blocked LU factorization which factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The dense $n \times n$ matrix A is divided into an $N \times N$ array of $B \times B$ blocks ($n = NB$) to exploit temporal locality on submatrix elements. We use a 128×128 matrix in our simulation study. Figure 12 shows the performance results for LU. As expected, the Dir_1NB and Dir_2NB protocols give the worst performance for all cases. The difference between other protocols is within 1%. The reason is that the time spent on waiting synchronization points exceeds 35% of overall execution time for LU. Thus, all the protocols excluding Dir_1NB and Dir_2NB do not show much difference on the normalized overall execution time.

Floyd Washall. Floyd Washall is a program that computes the shortest distance between every pair of nodes in a network. The network employed is a random graph of 32 nodes. The basic data structures in the Floyd Washall algorithm are two-dimensional arrays for representing the predecessor matrix and the distance matrix. An additional two-dimensional array is also used for recording the computed path. Each pro-

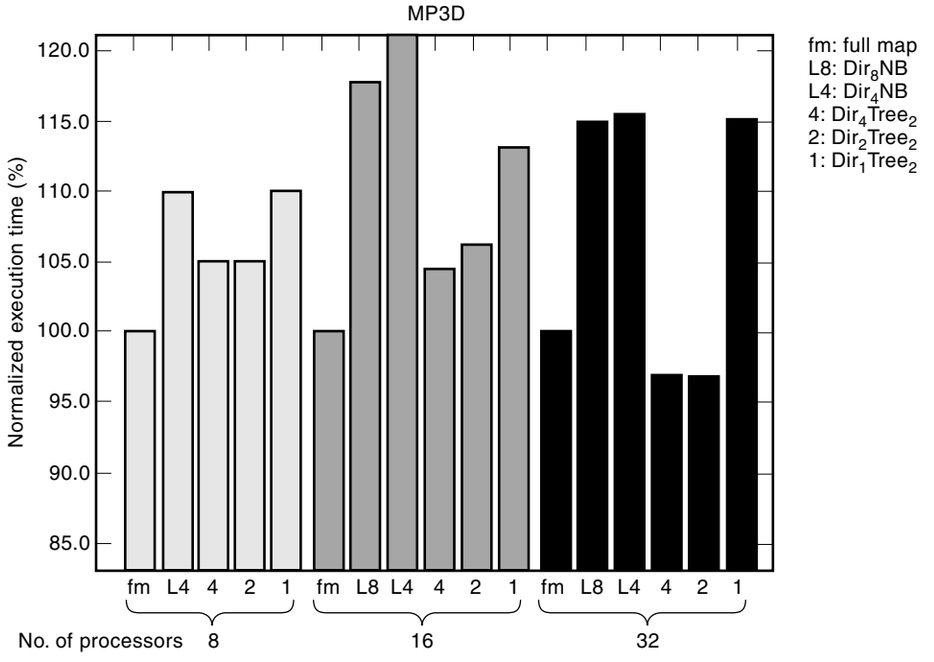

Figure 11. Normalized execution time for MP3D.

cessor is responsible for updating a few rows of the distance matrix. The entire matrix is declared as a shared array. Updating the distance matrix requires reading the entire shared array, which incurs a large degree of data sharing. Figure 13 shows the performance plot for the Floyd Washall program. The performance of both Dir_8Tree_2 and Dir_4Tree_2 is very similar to that of the full-map scheme. The performance difference between Dir_4Tree_2 and the full-map scheme is less than 2%.

FFT. Figure 14 gives the results for the FFT application. Except for Dir_1Tree_1, all the other schemes perform very well. However, the hybrid schemes Dir_4Tree_2 and Dir_8Tree_2 perform better than the full-map and the limited directory schemes. The improvement in case of the hybrid schemes increases when the system becomes bigger. The improvement stems from the fact that not much communication overhead is caused by replacements.

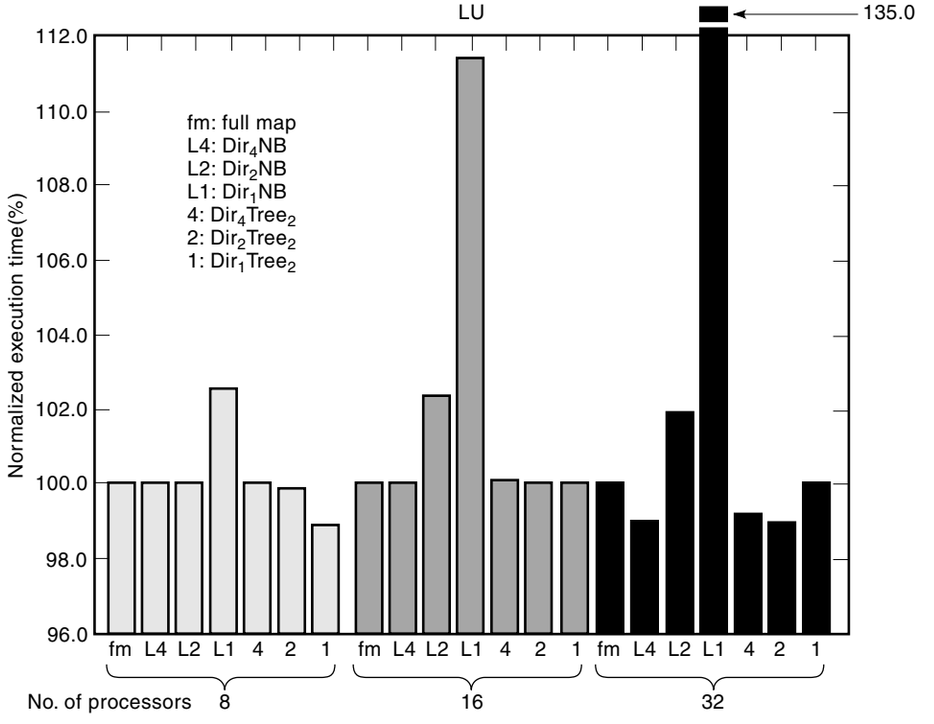

Figure 12. Normalized execution time for LU.

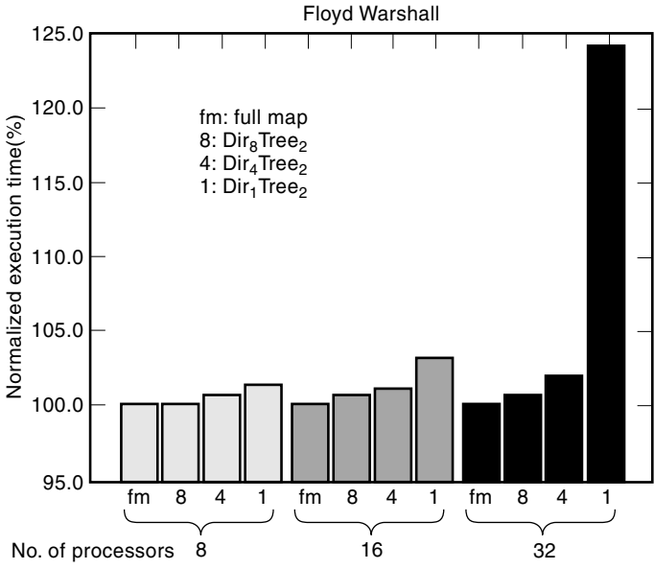

Figure 13. Normalized execution time for Floyd Washall.

CONCLUSION

In this article we reviewed the existing cache coherence protocols, namely, the full-map, the linked list, and the hybrid protocols for shared memory multiprocessors. The hybrid protocol combines the features of the limited directory schemes with tree protocols. It utilizes a limited number of pointers to construct trees to reduce the directory size and the invalidation latency. Compared to the STP and the SCI tree extensions, the hybrid scheme has lower read miss overhead, which is just two messages. At the same time, it retains the low invalidation properties of a tree protocol for a large degree of sharing. The trees constructed by the hybrid scheme are nearly balanced. Extensive execution-driven simulation on PROTEUS has shown that the hybrid scheme is very close in performance to that of the full-map scheme. When the number of processors is large, the new scheme performs even better than the full-map scheme in some cases. However, the hybrid scheme requires less directory space than the full-map scheme.

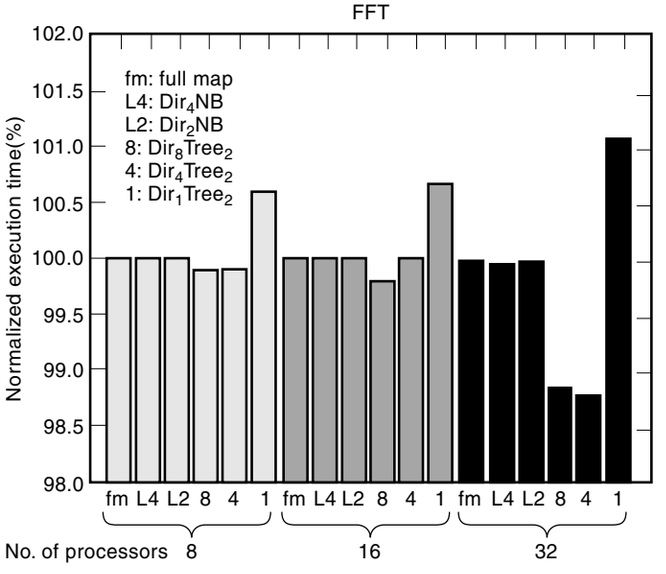

Figure 14. Normalized execution time for FFT.

BIBLIOGRAPHY

1. M. Dubois, C. Scheurich, and F. A. Briggs, Synchronization, coherence, and event ordering in multiprocessors, *IEEE Comput.,* **21** (2): 9–21, 1988.
2. J. Handy, *The Cache Memory Book,* New York: Academic Press, 1993.
3. W. Stallings, *Computer Organization and Architecture: Designing for Performance,* Upper Saddle River, NJ: Prentice-Hall, 1996.
4. D. J. Lilja, Cache coherence in large-scale shared-memory multiprocessors: Issues and comparisons, *ACM Comput. Surv.,* **25** (3): 303–338, 1993.
5. J. R. Goodman, Using cache memory to reduce processor-memory traffic, *Proc. Int. Symp. Comput. Archit.,* 1983, pp. 124–131.
6. G. Fielland and D. Rodgers, 32-bit Computer system shares load equally among up to 32 processors, *Electron. Des.,* **32** (18): 153–168, 1984.
7. M. Hill et al., Dragon decisions in SPUR, *IEEE Comput.,* **19**: 8–22, 1986.
8. M. Papamarcos and J. Patel, A low overhead coherence solution for multiprocessors with private cache memories, *Proc. Int. Symp. Comput. Archit.,* 1984, pp. 348–354.
9. S. J. Frank, Tightly coupled multiprocessor systems speed memory access times, *Electronics,* **57**: 164–169, 1984.
10. C. P. Thacker, L. C. Stewart, and E. H. S. Jr., Firefly: A multiprocessor workstation, *ASPLOS-V Proc.,* 1987, pp. 164–172.
11. E. McCreight, The dragon computer system: An early overview, *NATO Adv. Study Inst. Microarchit. VLSI Comput.,* 1984.
12. Q. Yang, L. N. Bhuyan, and B.-C. Liu, Analysis and comparison of cache coherence protocols for a packet-switched multiprocessor, *IEEE Trans. Comput.,* **38** (8): 1143–1153, 1989.
13. L. A. Barroso and M. Dubois, Cache coherence on a slotted ring, *Proc. Int. Conf. Parallel Process.,* 1991, pp. 230–237.
14. Kendall Square Research Corporation, *Kendall Square Research: Technical Summary,* Waltham, MA: Kendall Square Research Corporation, 1992.
15. D. Chaiken, J. Kubiatowicz, and A. Agarwal, LimitLESS directories: A scalable cache coherence scheme, *ASPLOS-IV Proc.,* 1991, pp. 224–234.
16. IEEE, *IEEE Std 1596-1992: IEEE Standard for Scalable Coherent Interface,* IEEE, New York, 1993.
17. M. Thapar, B. Delagi, and M. J. Flynn, Linked list cache coherence for scalable shared memory multiprocessors, *Proc. Int. Parallel Process. Symp.,* 1993, pp. 34–43.
18. H. Nilsson and P. Stenstrom, The scalable tree protocol—A cache coherence approach for large-scale multiprocessors, *Proc. Int. Symp. Parallel Distributed Process.,* 1992, pp. 498–506.
19. R. E. Johnson, *Extending the scalable coherent interface for large-scale shared-memory multiprocessors,* PhD thesis, Univ. Wisconsin—Madison, 1993.
20. S. Kaxiras, Kiloprocessor extensions to SCI, *Proc. Int. Parallel Process. Symp.,* 1996.
21. Y. Chang and L. N. Bhuyan, An efficient hybrid cache coherence protocol for shared memory multiprocessors, *Proc. Int. Conf. Parallel Process.,* 1996, pp. I172–I179.

22. A. Agarwal et al., An evaluation of directory schemes for cache coherence, *Proc. Int. Symp. Comput. Archit.,* 1988, pp. 280–289.
23. D. Chaiken and A. Agarwal, Software-extended coherent shared memory: Performance and cost, *Proc. Int. Symp. Comput. Archit.,* 1994, pp. 314–324.
24. D. V. James et al., Distributed directory scheme: Scalable coherent interface, *IEEE Comput.,* **23**: 74–77, 1990.
25. L. M. Censier, A new solution to coherence problems in multicache systems, *IEEE Trans. Comput.,* **C-27** (12): 1112–1118, 1978.
26. M. Hill et al., Cooperative shared memory: Software and hardware for scalable multiprocessors, *ASPLOS-V Proc.,* 1992, pp. 262–273.
27. D. Wood et al., Mechanisms for cooperative shared memory, *Proc. Int. Symp. Comput. Archit.,* 1993, pp. 156–167.
28. A. Gupta et al., Reducing memory and traffic requirements for scalable directory-based cache coherence schemes, *Proc. Int. Conf. Parallel Process.,* 1990.
29. W. Michael, A scalable coherence system with a dynamic pointing scheme, *Proc. Supercomput.,* 1992, pp. 358–367.
30. M. Thapar and B. Delagi, Stanford distributed directory protocol, *IEEE Comput.,* **23** (6): 78–80, 1990.
31. W.-D. Weber and A. Gupta, Analysis of cache invalidation patterns in multiprocessors, *ASPLOS-III Proc.,* 1989, p. 243–256.
32. E. A. Brewer et al., PROTEUS: A high-performance parallel architecture simulator, Tech. Rep. MIT/ICS/TR516, MIT, Cambridge, MA, 1991.
33. J. P. Singh, W. D. Weber, and A. Gupta, SPLASH: Stanford parallel applications for shared memory, Tech. Rep. CSL-TR-92-526, Stanford Univ., Stanford, CA, 1992.
34. D. Lenoski et al., The DASH prototype: Logic overhead and performance, *IEEE Trans. Parallel Distrib. Syst.,* **4** (1): 41–60, 1993.

YEIMKUAN CHANG
Chung-Hua University

LAXMI N. BHUYAN
Texas A&M University

CACHE SIMULATION. See STACK SIMULATION.