Cache-Coherent Distributed Shared Memory: Perspectives on Its Development and Future Challenges

Article *in* Proceedings of the IEEE · June 1999

DOI: 10.1109/5.747863 · Source: CiteSeer

John L. Hennessy
Stanford University
293 PUBLICATIONS **28,021** CITATIONS
SEE PROFILE

Mark Heinrich
University of Central Florida
55 PUBLICATIONS **1,809** CITATIONS
SEE PROFILE

Anoop Gupta
Stanford University
183 PUBLICATIONS **19,685** CITATIONS
SEE PROFILE

Cache-Coherent Distributed Shared Memory: Perspectives on Its Development and Future Challenges

JOHN HENNESSY, FELLOW, IEEE, MARK HEINRICH, AND ANOOP GUPTA, MEMBER, IEEE

Invited Paper

Distributed shared memory is an architectural approach that allows multiprocessors to support a single shared address space that is implemented with physically distributed memories. Hardware-supported distributed shared memory is becoming the dominant approach for building multiprocessors with moderate to large numbers of processors. Cache coherence allows such architectures to use caching to take advantage of locality in applications without changing the programmer's model of memory. We review the key developments that led to the creation of cache-coherent distributed shared memory and describe the Stanford DASH multiprocessor, the first working implementation of hardware-supported scalable cache coherence. We then provide a perspective on such architectures and discuss important remaining technical challenges.

Keywords**—***Cache coherence, directory-based cache coherence, distributed shared memory, multiprocessor architecture, scalable multiprocessors.*

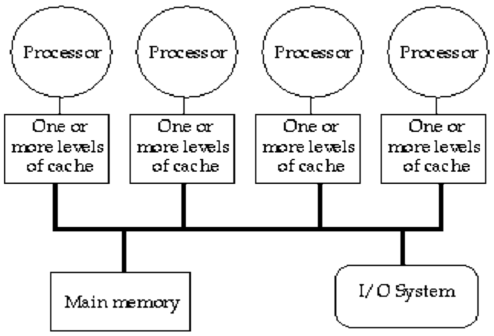

Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Main memory
I/O System

Fig. 1. A symmetric, shared memory multiprocessor. The most common interconnect in such multiprocessors is a bus, used both to allow access to the common memory and to implement cache coherence by using the bus as a broadcast medium. The role of cache coherence is discussed in more detail in Section II.

I. Motivations

In the 1980's, multiprocessors were designed with two major architectural approaches. For small numbers of processors (typically less than 16 or 32), the dominant architecture was a single shared memory with multiple processors, interconnected with a bus, as shown in Fig. 1. These machines were called bus-based multiprocessors or symmetric multiprocessors (SMP's), since all processors have an equal relationship with the centralized main memory. Bus-based, shared memory multiprocessors remain the dominant multiprocessor architecture for small processor counts.

To scale to larger numbers of processors, designers distributed the memory throughout the machine and used a scalable interconnect to enable processor–memory pairs to communicate, as shown in Fig. 2. The primary form of these multiprocessors through the early 1990's were message-passing architectures, named for the method by which processors communicate. Message-passing architectures also have been called multicomputers because they consist of separate computing nodes with no shared structure, other than the interconnect. The name distributed address space architecture is also sometimes used for these machines. In the 1980's, a small number of architectures with physically distributed memory but using a shared memory model were also developed. We discuss these early distributed shared memory architectures in Section I-A.

Each of these two primary approaches offered advantages. The shared memory architectures supported the traditional programming model, which saw memory as a single, shared address space. The shared memory machines also had lower communication costs since the processors communicated through shared memory rather than through a software layer. On the other hand, the distributed address space architectures had advantages in scalability, since such architectures did not suffer from the limits of a single,

Manuscript received September 1, 1997; revised June 16, 1998. This work was supported by the Defense Research Projects Agency (DARPA).

J. Hennessy is with the Computer Systems Laboratory, Stanford University, Stanford, CA 94305 USA.

M. Heinrich is with the School of Electrical Engineering, Cornell University, Ithaca, NY 14853 USA.

A. Gupta is with Microsoft Research, Microsoft, Inc., Redmond, WA 98052-6399 USA.

Publisher Item Identifier S 0018-9219(99)01744-2.

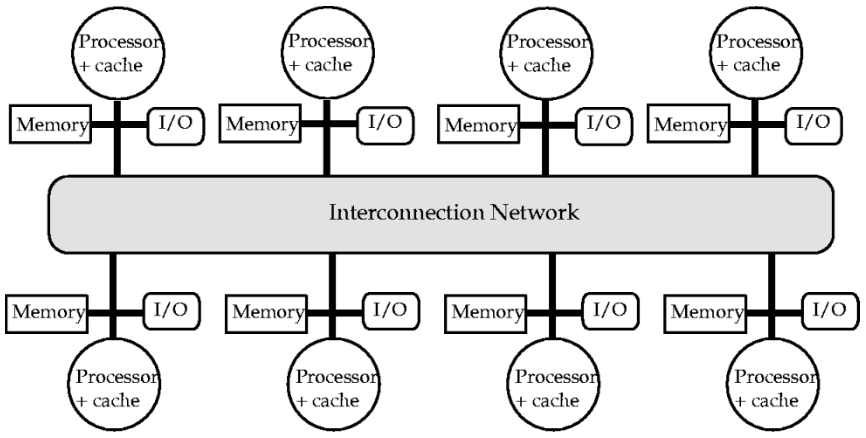


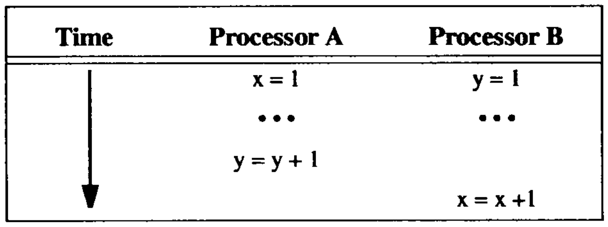
Fig. 2. A distributed memory multiprocessor. The key distinction between this and the shared bus design is the distribution of memory with the processors. The use of multiple memory modules and a scalable network allows the machine to scale to larger numbers of processors. In addition, because the memory is distributed with the processors, local memory accesses do not consume global bandwidth and can achieve lower access times.

centralized shared memory or bus. Despite these scalability advantages, the difference in programming model from the dominant small-scale, shared memory multiprocessors has severely limited the success of message-passing computers, especially at lower processor counts (e.g., less than 64 processors).

A. Development of Distributed Shared Memory (DSM)

DSM is an architectural approach designed to overcome the scaling limitations of symmetric shared memory multiprocessors while retaining a shared memory model for communication and programming. DSM multiprocessors achieve this by using a memory that is physically distributed but logically implements a single shared address space, allowing the processor to communicate through, and share the contents of, the entire memory. DSM multiprocessors have the same basic organization as the machines in Fig. 2.

As mentioned earlier, DSM architectures first appeared in the late 1970's and through the 1980's, embodied in three early multiprocessors: the Carnegie Mellon Cm* [23]; the IBM RP3 [21]; and the BBN Butterfly [4]. All these computers implemented a shared address space where the time to access a datum depended on its location. Hence, the name nonuniform memory access (NUMA) computers was also given to such architectures. (The symmetric multiprocessors, as shown in Fig. 1, are called UMA's for uniform memory access.) Although in NUMA architectures the actual time to access a datum could depend on exactly which memory contained the desired address, in reality the big difference was between addresses in local memory and addresses in remote memory. Because the access times could differ by a factor of ten or more and there were no simple mechanisms to hide these differences, it proved difficult to program these early DSM multiprocessors. In uniprocessors, this long access time to memory is largely hidden through the use of caches. However, adapting caches to work in a multiprocessor environment is challenging, as we discuss next.



Time	Processor A	Processor B
↓	x = 1	y = 1
	• • •	• • •
	y = y + 1	
		x = x +1

Fig. 3. Coherence problems that arise when shared data are cached. Assume both x and y are initially zero and that copies are cached in both A and B. If the caches are not coherent, it is possible that Processor A does not see the changed value of y (thus assigning y the value one) and the Processor B does not see the changed value of x (thus assigning x the value one). This problem arises with either write-through caches, if the values of x and y are in the caches to start, or write-back caches irrespective of the initial cached state of x and y.

B. The Problem of Cache Coherence

When used in a multiprocessor, caching introduces an additional problem: cache coherence, which arises when different processors cache and update values of the same memory location, as shown in Fig. 3. Introducing caches without dealing with the coherence problem does little to simplify the programming model, since the programmer must worry about the potentially inconsistent views of memory.

A clever solution, which builds on the bus interconnect, was developed to address the cache-coherence problem in small-scale shared memory multiprocessors. The basic idea is to enforce the property that before a memory location is written, all other copies of the location, which may be present in other caches, are invalidated. Thus, the system allows multiple copies of a memory location to exist when it is being read, but only one copy when it is being written. The unit for enforcing coherence is a block in the cache, typically 16–128 bytes.

As shown in Fig. 4, the bus is the key to implementing the most common coherence protocols, which are called snoopy protocols. When a processor wants to write into a

Processor A Program action	Processor A System Action	Processor B Program Action	Processor B System Action
x = 1	Broadcast cache invalidation for x		
			Receive invalidation and eliminate x from cache
		t = x + 1	Broadcast cache miss for x
	Receive cache miss; recognize that only copy of x is in cache and place value of x on bus		
			Receive value of x (=1) from bus, place into cache, and continue

Fig. 4. A snoopy-based cache-coherence scheme. This example shows the order of events as they occur in a coherent system with write-back caches, assuming that the variable x is present in both caches at the start. This is an invalidation protocol, similar to those used in most real systems. If two processors attempt to write the same data at the same time, the bus acts as a serialization point, allowing one to go forward first. Some early multiprocessors also implemented an update protocol, whereby data were updated on writes rather than invalidated. Update is not supported in most recent multiprocessors because of its potential inefficiency.

cache block that may be shared, a snoopy protocol transmits the request on the bus, and all caches that have a copy of the cache block simply invalidate the copy. For write-through caches, this is the only addition needed to a standard cache protocol, since the memory is always up to date. For write-back caches, an added complication arises since the most recent copy of a data item may reside in a cache. If so, read misses must snoop in the caches and possibly retrieve data from another cache.

Whether the cache uses a write-through or a write-back mechanism, the snooping operations are implemented by placing the request on the bus and having all the caches read the address and (if the address matches something in the processor's cache) either perform an invalidation or supply data from the cache. Because all requests must be placed on the bus, which carries only one request at a time, the bus breaks the tie when two processors try to write at the same time. This serialization of all requests via the bus imposes an ordering on all writes (including those to the same address) and is critical to maintaining coherence.

The snoopy cache-coherence schemes and the bus-based interconnect used in small-scale shared memory multiprocessors work well together for three reasons. First, the cache-coherence scheme makes the cache functionally transparent, allowing the system to cache both shared and private data without changing the shared memory programming model. Second, the use of caches reduces the bandwidth requirements on the bus and memory, thus allowing the processors to share a single memory and bus. Third, the use of a bus, which broadcasts all memory requests to all processor-cache modules, makes it relatively easy to implement the snoopy coherence protocols.

The reduced programming complexity offered by cache coherence, together with its relatively low cost of implementation, led to cache coherence being included in all small-scale, bus-based multiprocessors. In the last few years, microprocessors have included support for cache coherence and interconnecting small numbers of processors (two to four) within the microprocessor die, further reducing the cost of small-scale multiprocessors and significantly increasing their popularity.

Unfortunately, the snoopy schemes used in small-scale SMP's do not scale. The problem goes beyond the use of a bus, since any potential memory request must be seen and snooped by all the caches in the system. Thus, when the first DSM multiprocessors were developed, the designers did not attempt to cache shared data, but instead forced the programmers to deal with long access times for shared data residing in remote memories. This made the early DSM machines not much easier to program than the message-passing architectures. Furthermore, the lack of cache coherence created a schism in the programming approach used for the widespread small-scale, cache-coherent multiprocessors and the programming approach used for large-scale shared memory architectures without cache coherence. The solution to this problem was to develop a coherence mechanism that could be efficiently extended to the DSM architectural approach.

II. Directory-Based Cache Coherence

As we saw in Section I-B, the small-scale bus-based multiprocessors rely on snoopy cache coherence mechanisms, which inherently use broadcast. Such schemes were not, however, the first protocols developed for cache coherence. Before the snoopy schemes were developed, directory-based protocols had been proposed [7]. Directory-based schemes rely on an extra structure, called the directory, which tracks which processors have cached any given block in main memory. The initial directory schemes assumed a single, monolithic directory, and we explain the basic operation of directory coherence using this assumption.

Because the directory tracks which caches have copies of any given memory block, a coherence protocol can use the directory to maintain a consistent view of memory. To maintain coherence, the state of each cache block is tracked in the cache and additional information is kept in the directory for each block. A simple cache coherence protocol can operate with three states for each cache block.

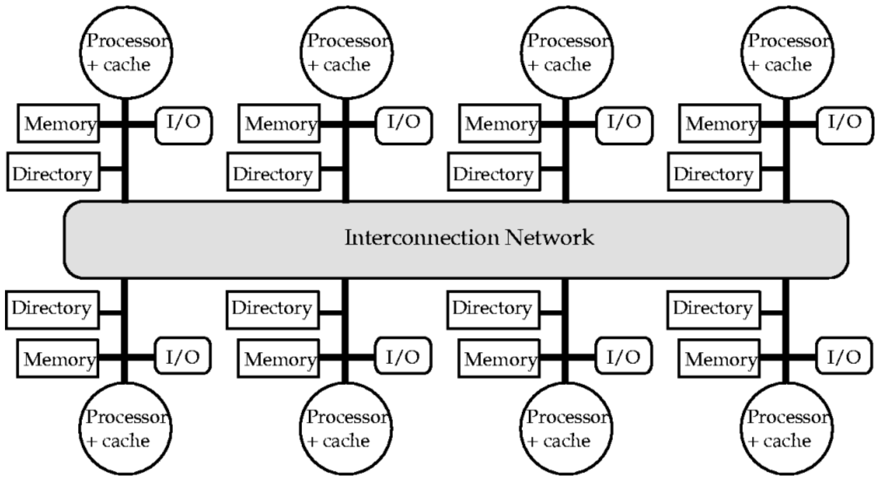


Fig. 5. Structure of a DSM multiprocessor using distributed directories. Each processing node includes memory and a directory. The directory tracks copies of the memory locations in the memory associated with that node. The directory information is kept on a per-block basis.

1) *Invalid:* The cache block is invalid and cannot be used by the processor.
2) *Shared:* The cache block is readable but may be present in the cache of another processor. In this case, the directory entry for this block contains a list of the other processors whose caches have a copy of this block. When a block is shared, it can be read but not written.
3) *Exclusive:* The cache block is only cached in this cache and hence is writable.

The protocol ensures consistency by invalidating all the caches that have a copy of a cache block before allowing a cache block to enter the exclusive state. The key difference between a directory protocol and a snoopy protocol is that the directory protocol gets the information about which processors are sharing a copy of the data from a known location (the directory) rather than interrogating all the processors by a broadcast. The directory also serves to serialize writes, just as a bus does in a snoopy scheme. To see this, consider what happens when two processors decide to write into the same block. In a directory scheme, the potential race is prevented when the requests serialize on their way to the directory. Since one request is processed before the other, the first request will cause the other processor's data to be invalidated. We will examine the details of a directory protocol shortly, after we discuss an important enhancement.

The original directory protocols were never widely embraced because, in a small-scale machine, a bus interconnect suffices and the snoopy schemes can be easily implemented. The use of a directory avoids the need to broadcast to interrogate all the caches, but a single centralized directory still cannot scale to larger numbers of processors. Using a single centralized directory would simply shift the bottleneck from the bus to the directory. The key insight to circumvent this limitation was to distribute the directory and extend and adapt the protocols to deal with a distributed directory.

A. Distributed Directory Protocols

Cache-coherent DSM architectures build on the directory concept but distribute the directory just as the memory is distributed. The directory information for a memory location is kept in the directory associated with the memory containing the location. Fig. 5 shows the structure of a typical cache-coherent DSM multiprocessor.

Although the extension of the directory protocol to a distributed directory implementation is conceptually simple, the implementation introduces many complexities, since few of the protocol actions can be atomic (e.g., acquiring exclusive access to a memory location). Instead, the protocol is implemented by sending messages among:

1) the requesting processor node, called the local node;
2) the node containing the address of the block that the local node desires to read or write, called the home node;
3) a possible third node, called the owner or remote node, which contains the cache block when it is in the exclusive state.

Fig. 6 shows an example of such a protocol in operation.

One complication in distributed directory protocols is that satisfying a remote request requires at least two messages: from the local to the home node to request a cache block and then from the home to the local node to reply with the data. In the case of a remotely cached data item in the exclusive state, at least three messages are required (local to home, home to remote, and remote to local). Furthermore, a request may require many more messages when the request requires invalidating a heavily shared block. The need for multiple messages to complete what is architecturally an atomic operation introduces significant potential for deadlock. Deadlock occurs when the machine enters a state from which it cannot continue. The distributed nature of the implementation, together with finite resources such as message buffers, makes deadlock a danger. Avoiding such situations requires significant attention in the implementation of the protocol. The avoidance of broadcast and a

Processor A Program action	Processor A System Action	Processor B Program Action	Processor B System Action
x = 1	Send ownership request to home node. Home node sends invalidations to all processors that cache x.		
			Receive invalidation, eliminate x from cache, and acknowledge invalidation.
	A has an exclusive copy of x; store1 into x.		
		t = x +1	Send message to home node, which forwards request to A for value.
	Receives request from home node; changes cache state to shared; sends value to both B (for its use) and to the home node (for writing back into memory).		
			Obtain the value of x from A, and change state of x to shared.
		x = t	Send message to home node requesting ownership of x. Home node sends invalidation to A.
	Receive invalidation, eliminate x from cache, and acknowledge invalidation.		
			B has an exclusive copy of x; store t into x.

Fig. 6. A distributed directory cache-coherence protocol. This chart shows the events in the order they occur, as initiated by the program actions.

single point of serialization, which is forced by the bus of a snoopy coherence scheme, complicates the coherence protocols; however, avoiding these features is also the key to scalable cache coherence.

III. An Example: The Stanford DASH Multiprocessor

To see how these architectural concept work in a real machine, let us briefly look at the Stanford DASH prototype [17], the first operational multiprocessor to support a scalable coherence protocol. DASH was designed by a team of faculty, staff, and students at Stanford University and became operational in late 1991. DASH implements a DSM architecture supporting cache coherence with a distributed directory. The DASH prototype was built using four-processor bus-based multiprocessors as the building block, as shown in Fig. 7.

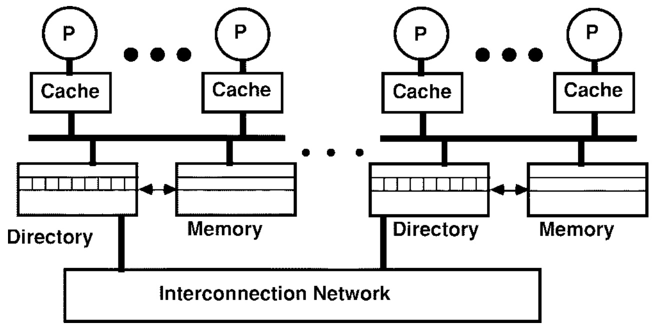


Fig. 7. The Stanford DASH Architecture. Each node in the multiprocessor was a four-processor, bus-based multiprocessor using snoopy cache-coherence on a bus. The prototype consisted of 16 such nodes, for a total of 64 processors. The directory structure was implemented using a bit vector representation. The use of a snoopy-bus-based multiprocessor for the nodes of DASH was an implementation decision. In retrospect, this choice complicated the coherence protocol and introduced a potential performance bottleneck (the bus). When several of the DASH designers worked on the SGI Origin, they decided to avoid this problem by using a simpler two-processor node design.

A. Other Innovations in DASH

In addition to being the first implementation of distributed directory-based coherence, DASH experimented with three concepts that increase the latency tolerance of the processor. The first concept was the use of a relaxed memory consistency model. A key question for multiprocessors implementing shared memory is how consistent a view of memory should be provided. In particular, if Processor A writes a memory location, when must the system guarantee that Processor B will have seen the value written? The strictest definition of memory consistency that has been implemented for a machine with DSM is sequential consistency, described by Lamport [13]. Sequential consistency effectively says that a processor will not perform additional reads or writes of shared data until it knows that all other processors have seen the most recent write. Fig. 6 assumes sequential consistency, since Processors A and B wait for invalidations to be acknowledged. Unfortunately, sequential consistency generally requires that a write must potentially wait until all processors have been guaranteed to see the invalidation caused by the write. Thus, while sequential consistency offers a simple model of shared memory, it is more strict than most programs require.

In practice, most programs use explicit synchronization operations to ensure that Processor B does not try to read the value of a location before Processor A has written it. If programs use this type of synchronization, then the system can implement a relaxed model of memory consistency without affecting the results that the programmer sees. DASH implemented a relaxed memory consistency model called release consistency. Under release consistency, the system need not guarantee that a written value will be seen by another processor until both processors perform a synchronization operation. This allows the system to perform writes without having to wait until all the invalidations have been performed. If the processor reaches a synchronization operation, it must wait for the outstanding writes to complete any pending invalidations before continuing.

If, however, programmers do not rely on standard synchronization primitives or build programs that are not strictly synchronized, the use of a weaker consistency model makes writing a correct program significantly more challenging. Relaxed consistency models have had an interesting effect on commercial multiprocessor architectures. While a number of architectures have adopted some sort of relaxed model for the multiprocessor architecture specification, most commercial DSM multiprocessors have implemented sequential consistency, using a variety of buffering and speculative techniques. Nonetheless, in the long term, relaxed models will offer advantages in hiding memory latency and are likely to be used in future multiprocessors.

The second key technique for hiding memory latency was the introduction of support for software prefetch. Prefetch operations allow a processor to specify the address of a data item so that the system can fetch that data item before the processor actually needs it. Although prefetch was implemented in a number of earlier machines, the incorporation of cache coherence allowed DASH to include a nonbinding prefetch. A nonbinding prefetch brings a copy of the data into the cache but maintains the coherence of the data; that is, if the prefetched data are written by another processor before the prefetching processor uses it, the data are invalidated—causing the prefetching processor to refetch the most recent value when it actually accesses the data. Unlike binding prefetch, a nonbinding prefetch does not change the program semantics and thus can be freely inserted by the compiler affecting only program performance. Several studies have since documented the efficacy of nonbinding prefetch [6], [9], [18].

Third, DASH introduced a remote access cache (RAC) to allow remote accesses to be combined and buffered within the individual nodes. This cache, also called a cluster cache, stored remote data that were recently accessed. If the remote data requested by a processor in a node are in the RAC, the data can be retrieved from the RAC, eliminating the need for a remote memory access. In DASH, the RAC also served to track pending requests for a cache block, thus eliminating duplicate requests that could lead to protocol failure.

An RAC can capture remote memory accesses under two circumstances: when data are fetched by one processor and used by another in the same node and when the data are simply too large to be kept in the local cache but fit in a larger RAC. Unfortunately, the RAC cannot eliminate misses caused by invalidations (called coherence or communication misses), since these misses and requests are needed to maintain coherence. Instead, an RAC can capture remote capacity misses, changing such misses into local misses to the RAC. As we will see shortly, many recent systems use large cluster caches to reduce the frequency of remote memory access.

Type of Access	Processor Clocks (30 ns each)
Local Fill	29
Fill from Home	101
Fill from Remote	132

Fig. 8. Memory access times in DASH. These are for an uncontended access.

B. DASH: System and Application Performance

A challenge in any scalable coherent machine is to achieve good performance given the inherently long delays associated with remote memory access. Fig. 8 shows the time to access different levels in the memory hierarchy for DASH [15]. In recent cache-coherent DSM multiprocessors, such as the HP Exemplar [3] and SGI Origin [16], the absolute times for these accesses have all decreased, though the remote access time relative to the processor clock cycle time has increased compared to DASH, and in most systems the relative cost of remote access compared to local access has also increased.

The key question for DSM multiprocessors is whether applications can achieve good performance given the long delays to remote memory. The answer to this question is illustrated in Fig. 9, which shows speed-up curves for several applications running on DASH. Overall, we can see that DASH achieves significant speed-ups for a range of applications.

IV. Perspectives and Lessons Learned

When we started the exploration of the ideas that led to DASH in the late 1980's, we did so based on two hypotheses. First, shared memory machines would be easier to program than message-passing machines, since shared memory allowed programmers to share data structures in flexible ways at different granularities. Second, cache coherence would be vital to allow shared data to be cached in complex applications. We did not come to this second belief easily. In fact, we started our project with the goal of enforcing coherence by the compiler. We concluded, after significant exploration, that it was unlikely that the compiler could efficiently solve the coherence problem across a wide range of applications. While significant progress has been made for well-structured scientific problems, efficient software coherence for more dynamically structured applications or for systems software has not yet been achieved.

To explore these hypotheses, we included a significant applications effort as part of the DASH project. Our experience in developing the SPLASH applications has confirmed

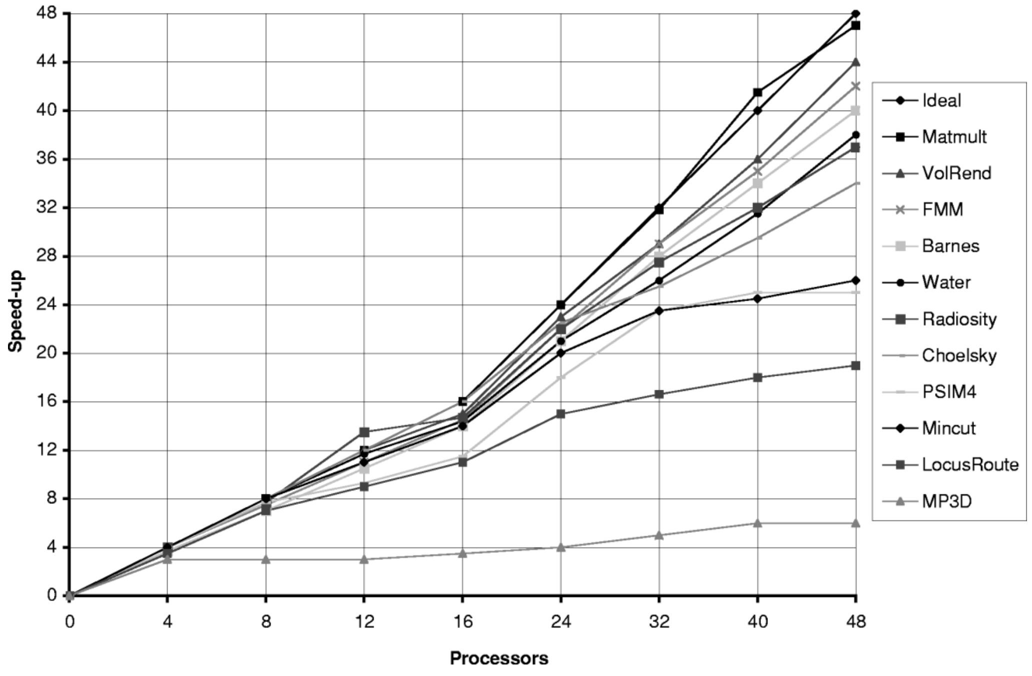


Fig. 9. Speed-up curves on DASH for a variety of applications. The basis (i.e., a speed-up of one) is normalized to the best single processor version, so the curves indicate real reductions in running time. The applications are chosen largely from the SPLASH suite [25] and vary from kernels (such as matrix multiply) to partial differential equation solvers (such as Ocean) to applications using a variety of n-body modeling techniques: Barnes–Hut, fast multipole method (FMM), and radiosity. Although the DASH prototype has 64 processors, reliability problems in the interconnect make it difficult to obtain useful measurements with more than 48 processors.

our hypotheses, although in somewhat unexpected ways. First, we found that achieving the highest level of performance often required careful planning of remote data accesses; sometimes down to the level of understanding the impact of cache line size. We did find, however, that the coherence mechanisms often helped exploit locality with little extra work, and that when careful attention was needed, the effort often focused on a small kernel of the application. We also found that exploiting the natural locality of scientific problems, even when using dynamic solutions techniques, was sometimes surprisingly easy. The n-body applications illustrate this best: the system being modeled has natural locality due to the underlying physical phenomena, but exploiting that locality requires fine-grain and time-varying communication. This is the type of environment where a cache-coherent shared address space shows its advantages most strongly. As a side note, we often found that when scaling applications, bottlenecks arose in different areas, including load balancing, limited parallelism, synchronization, and data locality. In general, we found that scaling up applications to use more processors is more difficult than it appears at first glance. Much of this experience is documented in our description of SPLASH-2 [27].

We began the DASH project based on the belief that a shared memory cache-coherent programming model used both by small machines (using bus-based, snoopy protocols) and larger machines (using cache-coherent DSM) would be vital to the development of applications. This belief motivated our key objective: to demonstrate that scalability and cache coherence were not incompatible. While this has proved correct and is probably the most significant accomplishment of the DASH project, our understanding of the role of cache coherence has also grown.

In particular, based on our experience, as well as recent experience with the Cray T3D/T3E [24], we believe that the effective use of shared memory programming requires support for global cache coherence. This hypothesis is based on a simple observation: caches provide a functionally transparent way to allow the memory system to amortize the cost of remote access. In shared memory programming, the latency and bandwidth efficiency of remote access is crucial. Cache coherence allows both the latency and bandwidth of a remote access to be amortized across a cache block.

Without such support, the natural unit of access of a load or store is a single word, which is extremely inefficient. Overcoming this inefficiency and accessing multiple words without cache coherence requires the compiler or programmer to guarantee the functional correctness of retrieving a block of data rather than a single word. We believe this capability for functionally transparent buffering of remote memory accesses is key to programming shared memory machines in a manner that is efficient in both performance and the use of programmer time.

V. Important Commercial Developments

While DASH demonstrated that distributed directory coherence could be implemented efficiently and that cache-coherent DSM architectures could be programmed to achieve high performance, several important innovations were developed in the early industrial machines and by the scalable coherent interference (SCI) coherence protocol. After discussing these innovations, we turn to a discussion of recent cache-coherent DSM multiprocessors.

A. Efficient Scaling of Directory Structures

Although distributed directories conceptually eliminate the scaling barriers inherent in snoopy-based schemes, the initial implementation of DASH used a nonscalable implementation of directories. In particular, DASH used a bit vector with 1 bit for every node. Since each node contains four processors, a 16-bit vector handled up to 64 processors. Besides the implementation difficulties of handling a very wide bit vector for larger processor counts, using a flat bit vector has a fundamental problem: the amount of directory storage scales quadratically as a function of processor count. In practice, this problem is not an issue for machines with less than 64 nodes (or 64–256 processors), but it becomes a major problem for very large processor counts.

There are two primary methods that have been used by the commercial DSM machines to avoid inefficient scaling of directory memory: storing full directory information only for blocks that are actually in a processor cache and using a sparse representation of the directory information. DASH uses a directory that maintains the state of each memory block in a single unique location tracking exactly the nodes that have copies of the associated memory block. By distributing the information for each actively cached memory block, the amount of directory memory can be reduced to an amount that grows linearly in the processor count. Sparse schemes reduce the directory memory required either by changing the representation of sharing information or by keeping track of only those blocks that are actively shared. The directory memory for sparse schemes also scales linearly in processor count. Compared to the simple bit vector scheme, these more scalable schemes sacrifice either simplicity or performance, and often both.

The protocol used in the largest number of multiprocessor architectures is the SCI [11] protocol, an IEEE standard. In SCI, the directory information is stored in a two-way linked list that is distributed among the nodes that share a cache block. The node that is the home also contains an entry for each word in its memory, and this entry provides access to the head of the list of sharers for a particular block. Coherence is maintained by adding processors that read a cache block to the linked sharing list and by invalidating all the processors on the list when writing a shared data block. Because the list is distributed, invalidations must traverse the list. The protocol specifies that when a sharer receives an invalidation it acknowledges the invalidation and returns the identity of the next sharer to which an invalidation must be sent. In practice, the protocol is usually implemented by having the invalidation operations sent down the sharing list and having each sharer forward the invalidation to the next sharer. The distribution of the directory information for a single block in SCI decreases the storage overhead and reduces the contention for accessing the information; however, SCI incurs greater overhead and requires more complex protocol operations in return for these advantages. The Sequent NUMA-Q [14], HP Exemplar [26], and Data General NUMALiine [8] all use variations of SCI.

An alternative method for directory scaling is to use a sparse directory representation. This can be done either by caching the directory information or by using a representation that becomes coarse as the processor count increases. A directory cache might keep the information in bit vector form but only keep entries for a subset of the memory. When a cache-mapping conflict occurs and an entry in the directory cache must be replaced, the system sends invalidations for the conflicting block to all processors sharing the block and then replaces the entry with the requested block. The HAL S-1 [26] uses a cached directory structure.

An alternative to caching is to use a representation of the directory information that becomes coarser as processor count grows. For example, suppose we wanted to scale DASH to more than 64-processors but maintain only 16-bits of directory bit vector. We could use each bit to represent an increasing number of processors. At 64 processors, each bit represents one node, which is four processors. At 256 processors, each bit would represent four nodes, or 16 processors. The directory memory would then scale linearly with processor count. The disadvantage, however, is that the sharing information is coarse, so that each bit in directory of a 256-processor machine represents 16 processors and indicates that any of the 16 processors may be sharing the data. On a write, invalidations must conservatively be sent to all 16 processors, only one of which may actually have the data. The SGI Origin [16] uses a coarse bit vector for configurations with more than 128 processors; the coarseness is fixed at 8 processors/bit, allowing a maximum of 1024 processors.

B. Extending Caching Further: The Cache Only Memory Architecture (COMA) Concept

In DASH, remote memory latency was significantly larger than local memory latency. The small remote access cache on DASH could help reduce this latency, and many recent machines have incorporated such caches, typically at much larger sizes (4–16 MB). The Kendall Square Research machine, the KSR-1 [5], introduced an even more ambitious scheme for reducing the remote latency of data that could not be accommodated in the processor caches. This scheme, widely called COMA, was simultaneously investigated by a research group at the Swedish Institute of Computer Science [10].

The key idea in a COMA architecture is to use the memory within each node of the multiprocessor as a cache, migrating and replicating data in the memory, just as you would in the caches. The key advantage of COMA is the ability to capture remote capacity misses as hits in the local memory. If a data item initially allocated in a remote memory is heavily used by a processor, the data block can be replicated in the memory of the node where it is being referenced. In the ultimate COMA system, data blocks may not even exist in the memory in which they were initially allocated; however, such flexibility incurs additional complexity, since the system must ensure that it always has at least one copy of every memory block. Because data migrate to where they are used, COMA reduces the need to worry about initial memory allocation, which can be important with large data structures that have high capacity miss rates. Implementing a COMA scheme requires that larger tags be allocated for main memory and also incurs significant added protocol complexity, but COMA's potential for reducing costly remote misses can provide performance advantages for some applications. The KSR-1 is the only instance of a commercial COMA architecture, but several research implementations of the COMA idea are underway.

C. Commercial Cache-Coherent DSM Multiprocessors

This paper does not allow for a complete description of the wide variety of commercial multiprocessors that have been developed. Instead, we briefly discuss the architecture of three typical DSM multiprocessors: the HP Exemplar [3] (formerly the Convex Exemplar), the SGI Origin [16], and the Sequent NUMA-Q [14].

The HP Exemplar is organized as a two-dimensional torus containing up to 32 nodes, each of which contains 16 processors and memories connected with a crossbar switch. The torus is formed by sets of rings, each connecting eight nodes and using an SCI protocol. In addition, each node contains a large cluster cache, called the CTI cache, used to reduce the fraction of remote misses. Internode coherence is maintained using an SCI protocol, while a bit-vector directory scheme is used for the 16-processor nodes.

The SGI Origin is organized as a fat hypercube consisting of up to 512 nodes, each containing two processors, a memory, and an I/O system. A fat hypercube yields a bisection bandwidth that grows linearly with processor count, through the addition of extra links. The bisection bandwidth is the lowest bandwidth across the middle of the multiprocessor and is commonly used as a measure of the interconnect capacity.

As mentioned earlier, the Origin uses a bit-vector directory scheme for up to 128 processors and a coarse vector for larger numbers. One of the novel features in the Origin is hardware support for page migration. This support, which consists of bits that can cause traps when migrated pages are accessed, reduces the overhead for page migration. Page migration, like the COMA and cluster cache techniques, can reduce the remote capacity misses incurred by a processor.

The Sequent NUMA-Q, like the DG NUMALiine, the HAL S-1, and the DASH prototype, is built from a standard four-processor bus-based multiprocessor. The four-processor nodes maintain a snoopy coherence scheme internally. Internode coherence uses SCI protocols on the Sequent and DG multiprocessors and a directory cache for the HAL multiprocessor. The NUMA-Q and NUMALiine multiprocessors use an SCI ring to connect up to 32 processors. The HAL design also uses a ring interconnect, which supports up to 16 processors.

System	Local memory	Remote memory (clean at home)	Remote memory (dirty at a third node)
HP Exemplar	450	1250	1860
SGI Origin	200	710	1055
DG NUMA-Liine	165	2400	3400
HAL S-1	180	1005	1305

Fig. 10. The local and remote access times in nanoseconds for several commercial systems. All configurations are 32 processors, except the HAL system, which has only 16 processors. Times are shown for a local memory access, a remote access that is clean in the home node's memory, and an access requiring intervention to a third processor's cache to retrieve an exclusive copy. The latencies shown assume no contention, a cache and cluster cache miss, and are an average value for the multiprocessors where multiple network hops are involved.

Though all these computers use distributed directories for implementing cache coherence, tradeoffs among different directory protocols and in interconnection technology result in very different performance characteristics. Fig. 10 shows one of the most important performance metrics: memory latency for these commercial DSM multiprocessors. The wide variety in access times shows the significance in the tradeoffs among different organizations and coherence protocols.

VI. Challenges

The DASH experiment and the recent commercial DSM multiprocessors demonstrated that cache-coherent DSM computers can be built efficiently and can perform well. Nonetheless, a variety of interesting technical challenges both in hardware and software remain; we pose these as a series of questions.

What are the best alternatives for designing scalable, cache-coherent multiprocessors? Although individual parts of the design space for coherence protocols have been explored, it is unclear how best to design protocols that span a variety of processor counts and provide robust and efficient performance across the range. In addition to the choice of protocols, it remains unclear what are the best combinations of hardware and software for implementing such protocols. Several research multiprocessors, including the MIT Alewife [2], MIT Star-T [20] and StarT-Voyager [1], and Wisconsin Typhoon [22], as well as the NUMA-Q design have used a combination of hardware and software to implement coherence. Most other commercial multiprocessors use hardwired hardware implementations. The flexible architecture for shared memory (FLASH) multiprocessor, discussed below, uses a completely programmable, but specialized, protocol processor. Which combinations are most effective for what classes of applications?

What should the programming model be for shared memory multiprocessors and what software tools are needed?

Clearly, to obtain good performance programmers must understand the importance of spatial and temporal locality and possibly the single-writer nature of coherence protocols, but programmers should not need to understand the details of a cache organization or coherence algorithm. The challenge lies in developing a programming model that helps programmers reason about their code and develop applications that require tuning rather than rewriting to achieve good performance. Once a programming model is broadly accepted, the challenge of developing tools for programming parallel applications can be tackled. We need better languages for conveying parallelism and expressing data locality issues. We need compilers that automate the process of optimizing the parallelism decomposition as well as the data allocation. Finally, the process of tuning a code for larger processor counts and specific architectures requires significant advances. The parallel programming task broadly defined—including programming models, load balancing, synchronization, and memory locality optimization—is the most critical challenge facing more effective use of parallel computing.

How should synchronization be supported? The underlying cache-coherence structure provides one mechanism for implementing simple synchronization primitives such as locks and barriers. Our explorations and those of other researchers have shown that these simple synchronization primitives can have poor performance in larger multiprocessors, especially under high contention. Designing more flexible synchronization mechanisms that have low latency in low contention situations, but also scale well under high contention, is becoming more important as processor counts grow.

How can large-scale multiprocessors deal with the challenges of reliability and fault tolerance? Scalable multiprocessors offer the advantage of scaling up performance by adding processors. If, however, the reliability of a multiprocessor decreases as it becomes larger, the attractiveness of such computers is decreased. Ideally, the reliability seen by an individual application should depend on the resources (processors, memory, etc.) that the application uses rather than on the total size of the computer. Unfortunately, this is not the case today, and faults in one node can easily affect other nodes that are not involved in the computation. Furthermore, the tighter coupling and more implicit sharing that are supported in a cache-coherent shared memory environment complicate the problem of fault containment and recovery.

How should DSM multiprocessors deal with increasing remote latency? DASH explored several approaches: relaxed consistency; prefetching; and cluster caches. COMA provides another method for reducing some types of remote latency. The Alewife project explored the use of multithreading for hiding remote latency. (Multithreading uses multiple threads of execution, automatically changing which thread is executing when a long latency event is incurred.) The best approach or combination of approaches remains open.

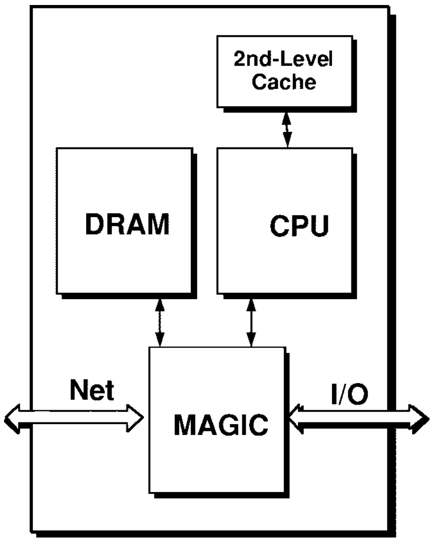


Fig. 11. Each node in the FLASH architecture includes a programmable communication controller, called MAGIC. MAGIC handles all internode communication, including coherence and synchronization. MAGIC consists of a superscalar processor with custom interfaces and special instruction support for protocol code operations.

A. The Stanford FLASH Multiprocessor

To explore a number of these issues, our research group at Stanford began a project called FLASH [12]. Fig. 11 shows a picture of a FLASH node. The key innovation in FLASH is the use of a programmable communications controller called memory and general interconnect controller (MAGIC) in each node.

A programmable controller has several benefits. First, the controller can implement different scalable coherence protocols, based on processor count and application characteristics, including complex protocols unsuitable for hardwired implementation. Second, MAGIC can also provide support for high-performance message-passing protocols that are integrated with the coherence mechanisms. Third, the replacement of special-purpose hardware with a programmable processor reduces parts count while retaining flexibility that can be used for many purposes, such as implementing synchronization. Fourth, a programmable controller can incorporate support for fault detection, encapsulation, and recovery. Last, such a controller can include performance-monitoring code in the core of the communications path [19]. The key challenge we are exploring in FLASH is whether we can achieve these benefits without unreasonable sacrifices in performance or cost.

VII. Concluding Remarks

The development of cache-coherent DSM has been an interesting process. Like other engineering research projects, it was motivated by the need to solve a problem and, in the process, has led to new understanding of multiprocessor architecture as well as parallel software systems. While cache-coherent DSM was originally developed for large-scale parallel multiprocessors, continuing rapid increases in microprocessor performance and the resultant demand on memory bandwidth both lead to the observation that DSM techniques are likely to be used for smaller numbers of processors. Already, several of the major vendors of bus-based multiprocessors have switched to DSM designs. It is likely that cache-coherent DSM will soon be the dominant architectural approach for multiprocessors with as few as four or more processors.

We believe that moderate-scale DSM multiprocessors (16–64 processors) are likely to become one of the most important architectures for large-scale commercial computing. These computers can provide high performance, fault containment and recovery, and efficient use of resources. Whether such multiprocessors will be built with custom nodes or whether they will be built from standard two to four processor clusters will be determined by the potential performance and functional advantages of a custom node versus the cost advantages of a commodity two to four processor node.

Economic issues will also play a key role in the future of very large DSM multiprocessors (above 64 processors). The key question for such computers is whether the market is significantly large to justify the engineering investment to develop the coherence mechanisms, interconnect, and operating systems support for these large-scale multiprocessors. It appears that the technical issues for such multiprocessors are reasonably well understood (at least for a few hundred processors), and a large-scale integrated cache-coherent multiprocessor could be built today. Alternatively, very large processor counts might be achieved by hooking together DSM nodes with 64 to 128 processors using standard off-the-shelf interconnect. Although such "clustered" multiprocessors will probably have a lower design and manufacturing cost, they are likely to provide lower performance, introduce significant new performance challenges, and require new operating systems support. In the end, the needs of applications and the economic resources available to invest in multiprocessors for these applications are likely to dictate how they will be built.

Acknowledgment

The DASH and FLASH teams involved a large number of faculty and students contributing to the architecture, hardware implementation, and software. Key members of the DASH team included: D. Lenoski; J. Laudon; D. Nakahira; and K. Gharachorloo. J. P. Singh was responsible for the development of SPLASH and the application-focused evaluations. Key members of the FLASH team included: J. Kuskin; D. Ofelt; M. Heinrich; J. Baxter; J. Bergman; J. Solomon; H. Kapadia; R. Simoni; J. Heinlein; M. Rosenblum; and M. Horowitz. The authors would like to acknowledge that without the efforts of a dedicated team of students, staff, and faculty, and the support of DARPA, these ideas might still be unproven.

References

[1] B. Ang, D. Chiou, D. Rosenband, M. Erlich, L. Rudolph, and Arvind, "StarT-Voyager: A flexible platform for exploring scalable SMP issues," in *Proc. SuperComputing '98,* Orlando, FL.

[2] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, "The MIT Alewife Machine: Architecture and performance," in *Proc. 22nd Int. Symp. Computer Architecture,* Santa Margherita Liguire, Italy, June 1995, pp. 2–13.

[3] T. Brewer and G. Astfalk, "The evolution of the HP/convex exemplar," in *Proc. COMPCON Spring'97: 42nd IEEE Computer Society Int. Conf.,* Feb. 1997, pp. 81–86.

[4] "Butterfly parallel processor," BBN Lab., Cambridge, MA, Tech. Rep. 6148, 1986.

[5] H. Burkhardt III, S. Frank, B. Knobe, and J. Rothnie, "Overview of the KSR-1 computer system," Kendall Square Research, Boston, MA, Tech. Rep. KSR-TR-9202001, Feb. 1992.

[6] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *Proc. 4th Int. Conf. Architectural Support for Programming Languages and Operating Systems,* Santa Clara, CA, Apr. 1991, pp. 40–52.

[7] L. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *IEEE Trans. Comput.,* vol. C(27), pp. 1112–1118, Dec. 1978.

[8] R. Clark, "SCI interconnect chipset and adapter: Building large scale enterprise servers with Pentium Pro SHV nodes," Data General, Tech. Rep., 1996.

[9] E. Gornish, E. Granston, and A. Veidenbaum, "Compiler-directed data prefetching in multiprocessors with memory hierarchies," in *Proc. Int. Conf. Supercomputing,* Nov. 1990, pp. 354–368.

[10] E. Hagersten, A. Landin, and S. Haridi, "DDM—A cache-only memory architecture," *IEEE Comput.,* vol. 25, pp. 44–54, Sept. 1992.

[11] *Scalable Coherent Interface*, IEEE Standard 1596-1992, Aug. 1993.

[12] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, "The Stanford FLASH multiprocessor," in *Proc. 21st Int. Symp. Computer Architecture,* Chicago, IL, Apr. 1994, pp. 302–313.

[13] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.,* vol. C-28, pp. 241–248, Sept. 1979.

[14] T. Lovett and R. Clapp, "STiNG: A CC-NUMA computer system for the commercial marketplace," in *Proc. 23rd Int. Symp. Computer Architecture,* Philadelphia, PA, May 1996, pp. 308–317.

[15] D. Lenoski, "The design and analysis of DASH: A scalable directory-based multiprocessor," Ph.D. dissertation, Stanford Univ., Sanford, CA, Tech. Rep. CSL-TR-92-507, Feb. 1992.

[16] J. Laudon and D. Lenoski, "The SGI origin: A ccNUMA highly scalable server," in *Proc. 24th Int. Symp. Computer Architecture,* Denver, CO, June 1997, pp. 241–251.

[17] D. Lenoski, J. Laudon, K. Gharachorloo, W. D. Weber, A. Gupta, and J. Hennessy, "The Stanford DASH multiprocessor," *IEEE Comput.,* vol. 25, pp. 63–79, Mar. 1992.

[18] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proc. 5th Int. Conf. Architectural Support for Programming Languages and Operating Systems,* Cambridge, MA, Oct. 1992, pp. 26–36.

[19] M. Martonosi, D. Ofelt, and M. Heinrich, "Integrating performance monitoring and communication in parallel computers," in *Proc. SIGMETRICS Int. Conf. Measurement and Modeling of Computer Systems,* May 1996, pp. 138–147.

[20] R. S. Nikhil, G. M. Papadopoulos, and Arvind, "*T: A multi-threaded massively parallel architecture," in *Proc. 19th Annu. Symp. Computer Architecture,* Gold Coast, Australia, May 1992, pp. 156–167.

[21] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "The IBM research parallel processor prototype (RP3): Introduction and architecture," in *Proc. 1985 Conf. Parallel Processing,* 1985, pp. 764–771.

[22] S. Reinhardt, J. Larus, and D. Wood, "Tempest and typhoon: User-level shared memory," in *Proc. 21st Int. Symp. Computer Architecture,* Chicago, IL, Apr. 1994, pp. 325–336.

[23] R. J. Swan, A. Bechtolsheim, K. W. Lai, and J. K. Ousterhout, "The implementation of the Cm* multi-microprocessor," in *Proc. AFIPS NCC,* 1977, pp. 645–654.

[24] S. Scott, "Synchronization and communication in the Cray T3E multiprocessor," in *Proc. 7th Int. Conf. Architectural Support for*

Programming Languages and Operating Systems, Cambridge, MA, 1996, pp. 26–36.

[25] J. P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford parallel applications for shared-memory," *Comput. Architecture News,* vol. 20, no. 1, pp. 5–44, Mar. 1992.

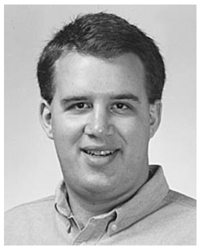
[26] W. D. Weber, S. Gold, P. Helland, T. Shimizu, T. Wicki, and W. Wilke, "The Mercury interconnect architecture: A cost-effective infrastructure for high-performance servers," in *Proc. 24th Int. Symp. Computer Architecture,* Denver, CO, June 1997, pp. 98–107.

[27] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. 22nd Annu. Int. Symp. Computer Architecture,* Santa Margherita Liguire, Italy, June 1995, pp. 24–36.

John Hennessy (Fellow, IEEE) received the B.E. degree in electrical engineering from Villanova University, Villanova, PA, in 1973. He received the Master's and Ph.D. degrees in computer science from State University of New York, Stony Brook, in 1975 and 1977, respectively.

Since September 1977, he has been with Stanford University, Stanford, CA, where he is currently a Professor of Electrical Engineering and Computer Science and the Frederick Emmons Terman Dean of Engineering.

Dr. Hennessy is the recipient of the 1983 John J. Gallen Memorial Award from Villanova University. He is the recipient of a 1984 National Science Foundation Presidential Young Investigator Award, and in 1987 he was named the Willard and Inez K. Bell Professor of Electrical Engineering and Computer Science. In 1991, he received the Distinguished Alumnus Award from the State University of New York, Stony Brook. He is a member of the National Academy of Engineering, a Fellow of the American Academy of Arts and Sciences, and a Fellow of the Association for Computing Machinery. He is the recipient of the 1994 IEEE Piore Award.



Mark Heinrich received the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 1998, the M.S. degree from Stanford University in 1993, and the B.S. degree in electrical engineering and computer science from Duke University, Durham, NC, in 1991.

He is an Assistant Professor in the School of Electrical Engineering, Cornell University, Ithaca, NY. His research interests include parallel computer architecture, data-intensive computing, cache-coherence protocol design, hardware/software codesign, and multiprocessor simulation methodology.

Anoop Gupta (Member, IEEE) received the Ph.D. degree from Carnegie Mellon University, Pittsburgh, PA, in 1986.

He is a Senior Researcher at Microsoft, Redmond, WA. From 1987 to 1998, he was an Associate Professor of Computer Science and Electrical Engineering at Stanford University, Stanford, CA. He has worked in the areas of computer architecture, operating systems, programming languages, performance debugging tools, and parallel applications. He co-led the design and construction of the Stanford DASH machine, one of the first scalable distributed shared memory multiprocessors, and he has worked on the follow-up FLASH project. He has published close to 100 papers in major conferences and journals, including several award papers.

Dr. Gupta received the NSF Presidential Young Investigator Award and held the Robert Noyce Faculty Scholar Chair at Stanford University.