The Stanford Dash Multiprocessor

Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam
Stanford University

Directory-based cache coherence gives Dash the ease-of-use of shared-memory architectures while maintaining the scalability of message-passing machines.

The Computer Systems Laboratory at Stanford University is developing a shared-memory multiprocessor called Dash (an abbreviation for Directory Architecture for Shared Memory). The fundamental premise behind the architecture is that it is possible to build a scalable high-performance machine with a single address space and coherent caches.

The Dash architecture is scalable in that it achieves linear or near-linear performance growth as the number of processors increases from a few to a few thousand. This performance results from distributing the memory among processing nodes and using a network with scalable bandwidth to connect the nodes. The architecture allows shared data to be cached, thereby significantly reducing the latency of memory accesses and yielding higher processor utilization and higher overall performance. A distributed directory-based protocol provides cache coherence without compromising scalability.

The Dash prototype system is the first operational machine to include a scalable cache-coherence mechanism. The prototype incorporates up to 64 high-performance RISC microprocessors to yield performance up to 1.6 billion instructions per second and 600 million scalar floating point operations per second. The design of the prototype has provided deeper insight into the architectural and implementation challenges that arise in a large-scale machine with a single address space. The prototype will also serve as a platform for studying real applications and software on a large parallel system.

This article begins by describing the overall goals for Dash, the major features of the architecture, and the methods for achieving scalability. Next, we describe the directory-based coherence protocol in detail. We then provide an overview of the prototype machine and the corresponding software support, followed by some

0018-9162/92/0300-0063$03.00 © 1992 IEEE

preliminary performance numbers. The article concludes with a discussion of related work and the current status of the Dash hardware and software.

Dash project overview

The overall goal of the Dash project is to investigate highly parallel architectures. For these architectures to achieve widespread use, they must run a variety of applications efficiently without imposing excessive programming difficulty. To achieve both high performance and wide applicability, we believe a parallel architecture must provide scalability to support hundreds to thousands of processors, high-performance individual processors, and a single shared address space.

The gap between the computing power of microprocessors and that of the largest supercomputers is shrinking, while the price/performance advantage of microprocessors is increasing. This clearly points to using microprocessors as the compute engines in a multiprocessor. The challenge lies in building a machine that can scale up its performance while maintaining the initial price/performance advantage of the individual processors. Scalability allows a parallel architecture to leverage commodity microprocessors and small-scale multiprocessors to build larger scale machines. These larger machines offer substantially higher performance, which provides the impetus for programmers to port their sequential applications to parallel architectures instead of waiting for the next higher performance uniprocessor.

High-performance processors are important to achieve both high total system performance and general applicability. Using the fastest microprocessors reduces the impact of limited or uneven parallelism inherent in some applications. It also allows a wider set of applications to exhibit acceptable performance with less effort from the programmer.

A single address space enhances the programmability of a parallel machine by reducing the problems of data partitioning and dynamic load distribution, two of the toughest problems in programming parallel machines. The shared address space also improves support for automatically parallelizing compilers, standard operating systems, multiprogramming, and incremental tuning of parallel applications — features that make a single-address-space machine much easier to use than a message-passing machine.

Caching of memory, including shared writable data, allows multiprocessors with a single address space to achieve high performance through reduced memory latency. Unfortunately, caching shared data introduces the problem of cache coherence (see the sidebar and accompanying figure).

While hardware support for cache coherence has its costs, it also offers many benefits. Without hardware support, the responsibility for coherence falls to the user or the compiler. Exposing the issue of coherence to the user would lead to a complex programming model, where users might well avoid caching to ease the programming burden. Handling the coherence problem in the compiler is attractive, but currently cannot be done in a way that is competitive with hardware. With hardware-supported cache coherence, the compiler can aggressively optimize programs to reduce latency without having to rely purely on a conservative static dependence analysis.

The major problem with existing cache-coherent shared-address machines is that they have not demonstrated the ability to scale effectively beyond a few high-performance processors. To date, only message-passing machines have shown this ability. We believe that using a directory-based coherence mechanism will permit single-address-space machines to scale as well as message-passing machines, while providing a more flexible and general programming model.

The Dash team

Many graduate students and faculty members contributed to the Dash project. The PhD students are Daniel Lenoski and James Laudon (Dash architecture and hardware design); Kourosh Gharachorloo (Dash architecture and consistency models); Wolf-Dietrich Weber (Dash simulator and scalable directories); Truman Joe (Dash hardware and protocol verification tools); Luis Stevens (operating system); Helen Davis and Stephen Goldschmidt (trace generation tools, synchronization patterns, locality studies); Todd Mowry (evaluation of prefetch operations); Aaron Goldberg and Margaret Martonosi (performance debugging tools); Tom Chanak (mesh routing chip design); Richard Simoni (synthetic load generator and directory studies); Josep Torrellas (sharing patterns in applications); Edward Rothberg, Jaswinder Pal Singh, and Larry Soule (applications and algorithm development). Staff research engineer David Nakahira contributed to the hardware design.

The faculty associated with the project are Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam.

Dash system organization

Most existing multiprocessors with cache coherence rely on snooping to maintain coherence. Unfortunately, snooping schemes distribute the information about which processors are caching which data items among the caches. Thus, straightforward snooping schemes require that all caches see every memory request from every processor. This inherently limits the scalability of these machines because the common bus and the individual processor caches eventually saturate. With today's high-performance RISC processors this saturation can occur with just a few processors.

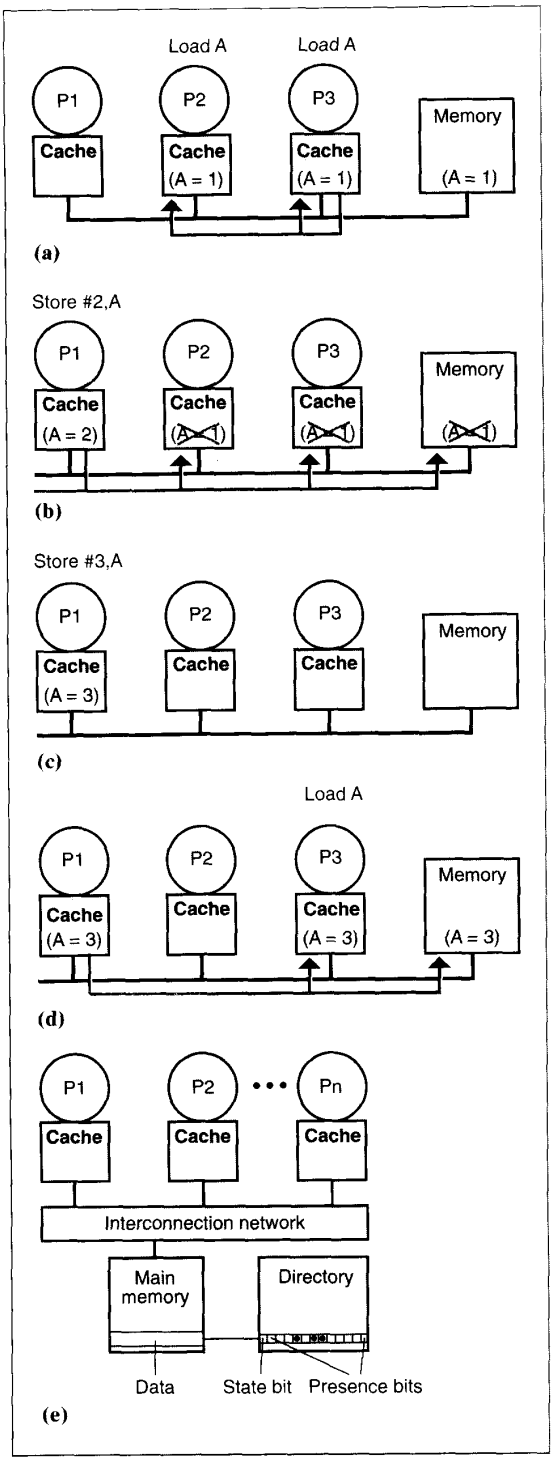
Directory structures avoid the scalability problems of snoopy schemes by removing the need to broadcast every memory request to all processor caches. The directory maintains pointers to the processor caches holding a copy of each memory block. Only the caches with copies can be affected by an access to the memory block, and only those caches need be notified of the access. Thus, the processor caches and interconnect will not saturate due to coherence requests. Furthermore, directory-based coherence is not dependent on any specific interconnection network like the bus used by most snooping schemes. The same scalable, low-latency networks such as Omega networks or k-nary n-cubes used by non-cache-coherent and

Cache coherence

Cache-coherence problems can arise in shared-memory multiprocessors when more than one processor cache holds a copy of a data item (a). Upon a write, these copies must be updated or invalidated (b). Most systems use invalidation since this allows the writing processor to gain exclusive access to the cache line and complete further writes into the cache line without generating external traffic (c). This further complicates coherence since this dirty cache must respond instead of memory on subsequent accesses by other processors (d).

Small-scale multiprocessors frequently use a snoopy cache-coherence protocol,[1] which relies on all caches monitoring the common bus that connects the processors to memory. This monitoring allows caches to independently determine when to invalidate cache lines (b), and when to intervene because they contain the most up-to-date copy of a given location (d). Snoopy schemes do not scale to a large number of processors because the common bus or individual processor caches eventually saturate, since they must process every memory request from every processor.

The directory relieves the processor caches from snooping on memory requests by keeping track of which caches hold each memory block. A simple directory structure first proposed by Censier and Feautrier[2] has one directory entry per block of memory (e). Each entry contains one presence bit per processor cache. In addition, a state bit indicates whether the block is uncached, shared in multiple caches, or held exclusively by one cache (that is, whether the block is dirty). Using the state and presence bits, the memory can tell which caches need to be invalidated when a location is written (b). Likewise, the directory indicates whether memory's copy of the block is up to date or which cache holds the most recent copy (d). If the memory and directory are partitioned into independent units and connected to the processors by a scalable interconnect, the memory system can provide scalable memory bandwidth.



References

1. J. Archibald and J.-L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Trans. Computer Systems*, Vol. 4, No. 4, Nov. 1986, pp. 273-298.
2. L. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. Computers*, Vol. C-27, No. 12, Dec. 1978, pp. 1,112-1,118.

message-passing machines can be employed.

The concept of directory-based cache coherence is not new. It was first proposed in the late 1970s. However, the original directory structures were not scalable because they used a centralized directory that quickly became a bottleneck. The Dash architecture overcomes this limitation by partitioning and distributing the directory and main memory, and by using a new coherence protocol that can suitably exploit distributed directories. In addition, Dash provides several other mechanisms to

reduce and hide the latency of memory operations.

Figure 1 shows Dash's high-level organization. The architecture consists of a number of processing nodes connected through directory controllers to a low-latency interconnection network. Each processing node, or *cluster*, consists of a small number of high-performance processors and a portion of the shared memory interconnected by a bus. Multiprocessing within the cluster can be viewed either as increasing the power of each processing node or as reducing the cost of the directory and network interface by amortizing it over a larger number of processors.

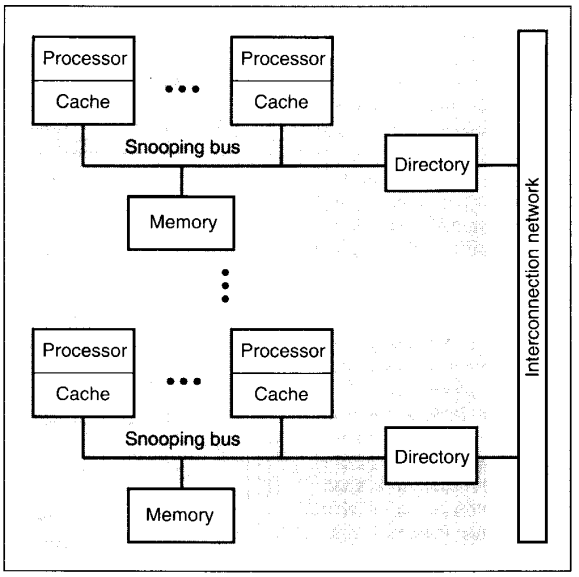


Figure 1. The Dash architecture consists of a set of clusters connected by a general interconnection network. Directory memory contains pointers to the clusters currently caching each memory line.

Distributing memory with the processors is essential because it allows the system to exploit locality. All private data and code references, along with some of the shared references, can be made local to the cluster. These references avoid the longer latency of remote references and reduce the bandwidth demands on the global interconnect. Except for the directory memory, the resulting system architecture is similar to many scalable message-passing machines. While not optimized to do so, Dash could emulate such machines with reasonable efficiency.

Scalability of the Dash approach

We have outlined why we believe a single-address-space machine with cache coherence holds the most promise for delivering scalable performance to a wide range of applications. Here, we address the more detailed issues in scaling such a directory-based system. The three primary issues are ensuring that the system provides scalable memory bandwidth, that the costs scale reasonably, and that mechanisms are provided to deal with large memory latencies.

Scalability in a multiprocessor requires the total memory bandwidth to scale linearly with the number of processors. Dash provides scalable bandwidth to data objects residing in local memory by distributing the physical memory among the clusters. For data accesses that must be serviced remotely, Dash uses a scalable interconnection network. Support

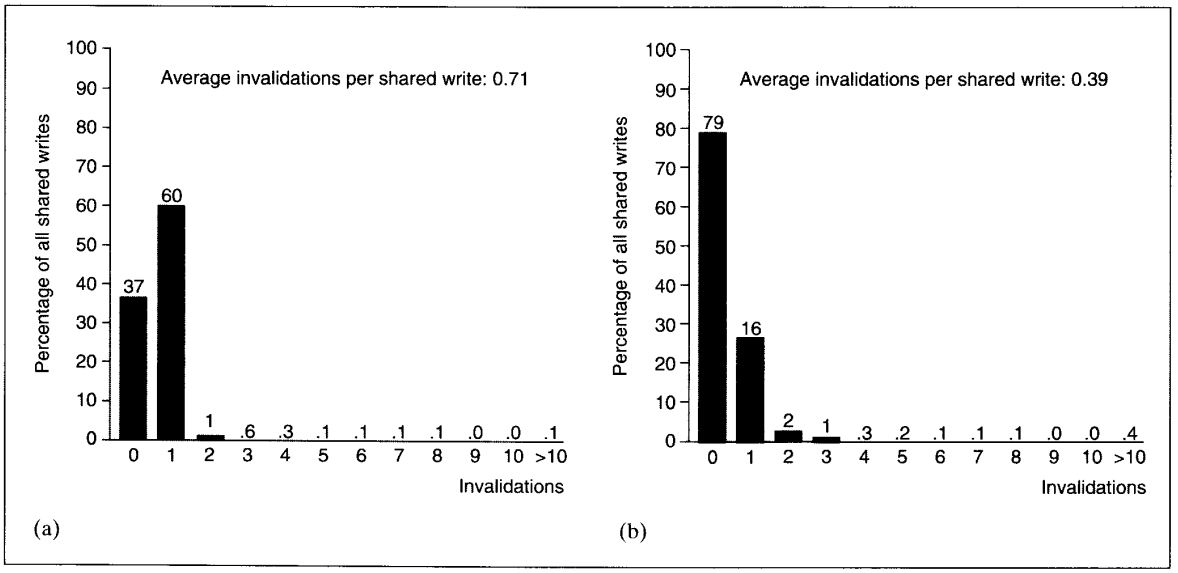


Figure 2. Cache invalidation patterns for MP3D (a) and PThor (b). MP3D uses a particle-based simulation technique to determine the structure of shock waves caused by objects flying at high speed in the upper atmosphere. PThor is a parallel logic simulator based on the Chandy-Misra algorithm.

of coherent caches could potentially compromise the scalability of the network by requiring frequent broadcast messages. The use of directories, however, removes the need for such broadcasts and the coherence traffic consists only of point-to-point messages to clusters that are caching that location. Since these clusters must have originally fetched the data, the coherence traffic will be within some small constant factor of the original data traffic. In fact, since each cached block is usually referenced several times before being invalidated, caching normally reduces overall global traffic significantly.

This discussion of scalability assumes that the accesses are uniformly distributed across the machine. Unfortunately, the uniform access assumption does not always hold for highly contended synchronization objects and for heavily shared data objects. The resulting *hot spots* — concentrated accesses to data from the memory of a single cluster over a short duration of time — can significantly reduce the memory and network throughput. The reduction occurs because the distribution of resources is not exploited as it is under uniform access patterns.

To address hot spots, Dash relies on a combination of hardware and software techniques. For example, Dash provides special extensions to the directory-based protocol to handle synchronization references such as queue-based locks (discussed further in the section, "Support for synchronization"). Furthermore, since Dash allows caching of shared writable data, it avoids many of the data hot spots that occur in other parallel machines that do not permit such caching. For hot spots that cannot be mitigated by caching, some can be removed by the coherence protocol extensions discussed in the section, "Update and deliver operations," while others can only be removed by restructuring at the software level. For example, when using a primitive such as a barrier, it is possible for software to avoid hot spots by gathering and releasing processors through a tree of memory locations.

Regarding system costs, a major scalability concern unique to Dash-like machines is the amount of directory memory required. If the physical memory in the machine grows proportionally with the number of processing nodes, then using a bit-vector to keep track of all clusters caching a memory block does not scale well. The total amount of directory memory needed is $P^2 \times M/L$ megabits, where P is the number of clusters, M is the megabits of memory per cluster, and L is the cache-line size in bits. Thus, the fraction of memory devoted to keeping directory information grows as P/L. Depending on the machine size, this growth may or may not be tolerable. For example, consider a machine that contains up to 32 clusters of eight processors each and has a cache (memory) line size of 32 bytes. For this machine, the overhead for directory memory is only 12.5 percent of physical memory as the system scales from eight to 256 processors. This is comparable with the overhead of supporting an error-correcting code on memory.

For larger machines, where the overhead would become intolerable, several alternatives exist. First, we can take advantage of the fact that at any given time a memory block is usually cached by a very small number of processors. For example, Figure 2 shows the number of invalidations generated by two applications run on a simulated 32-processor machine. These graphs show that most writes cause invalidations to only a few caches. (We have obtained similar results for a large number of applications.) Consequently, it is possible to replace the complete directory bit-vector by a small number of pointers and to use a limited broadcast of invalidations in the unusual case when the number of pointers is too small. Second, we can take advantage of the fact that most main memory blocks will not be present in any processor's cache, and thus there is no need to provide a dedicated directory entry for every memory block. Studies[1,2] have shown that a small directory cache performs almost as well as a full directory. These two techniques can be combined to support machines with thousands of processors without undue overhead from directory memory.

The issue of memory access latency also becomes more prominent as an architecture is scaled to a larger number of nodes. There are two complementary approaches for managing latency: methods that reduce latency and mechanisms that help tolerate it. Dash uses both approaches, though our main focus has been to reduce latency as much as possible. Although latency tolerating techniques are important, they often require additional application parallelism to be effective.

Hardware-coherent caches provide the primary latency reduction mechanism in Dash. Caching shared data significantly reduces the average latency for remote accesses because of the spatial and temporal locality of memory accesses. For references not satisfied by the cache, the coherence protocol attempts to minimize latency, as shown in the next section. Furthermore, as previously mentioned, we can reduce latency by allocating data to memory close to the processors that use it. While average memory latency is reduced, references that correspond to interprocessor communication cannot avoid the inherent latencies of a large machine. In Dash, the latency for these accesses is addressed by a variety of latency hiding mechanisms. These mechanisms range from support of a relaxed memory consistency model to support of nonblocking prefetch operations. These operations are detailed in the sections on "Memory consistency" and "Prefetch operations."

We also expect software to play a critical role in achieving good performance on a highly parallel machine. Obviously, applications need to exhibit good parallelism to exploit the rich computational resources of a large machine. In addition, applications, compilers, and operating systems need to exploit cache and memory locality together with latency hiding techniques to achieve high processor utilization. Applications still benefit from the single address space, however, because only performance-critical code needs to be tuned to the system. Other code can assume a simple uniform memory model.

The Dash cache-coherence protocol

Within the Dash system organization, there is still a great deal of freedom in selecting the specific cache-coherence protocol. This section explains the basic coherence protocol that Dash uses for normal read and write operations, then outlines the resulting memory consistency model visible to the programmer and compiler. Finally, it details extensions to the protocol that support latency hiding and efficient synchronization.

Memory hierarchy. Dash implements an invalidation-based cache-coherence protocol. A memory location may be in one of three states:

- *uncached* — not cached by any cluster;
- *shared* — in an unmodified state in the caches of one or more clusters; or
- *dirty* — modified in a single cache of some cluster.

The directory keeps the summary information for each memory block, specifying its state and the clusters that are caching it.

The Dash memory system can be logically broken into four levels of hierarchy, as illustrated in Figure 3. The first level is the processor's cache. This cache is designed to match the processor speed and support snooping from the bus. A request that cannot be serviced by the processor's cache is sent to the second level in the hierarchy, the *local cluster*. This level includes the other processors' caches within the requesting processor's cluster. If the data is locally cached, the request can be serviced within the cluster. Otherwise, the request is sent to the *home cluster* level. The home level consists of the cluster that contains the directory and physical memory for a given memory address. For many accesses (for example, most private data references), the local and home cluster are the same, and the hierarchy collapses to three levels. In general, however, a request will travel through the interconnection network to the home cluster. The home cluster can usually satisfy the request immediately, but if the directory entry is in a dirty state, or in shared state when the requesting processor requests exclusive access, the fourth level must also be accessed. The *remote cluster* level for a memory block consists of the clusters marked by the directory as holding a copy of the block.

To illustrate the directory protocol, first consider how a processor read traverses the memory hierarchy:

- *Processor level* — If the requested location is present in the processor's cache, the cache simply supplies the data. Otherwise, the request goes to the local cluster level.
- *Local cluster level* — If the data resides within one of the other caches within the local cluster, the data is supplied by that cache and no state change is required at the directory level. If the request must be sent beyond the local cluster level, it goes first to the home cluster corresponding to that address.
- *Home cluster level* — The home cluster examines the directory state of the memory location while simultaneously fetching the block from main memory. If the block is clean, the data is sent to the requester and the directory is updated to show sharing by the requester. If the location is dirty, the request is forwarded to the remote cluster indicated by the directory.
- *Remote cluster level* — The dirty cluster replies with a shared copy of the data, which is sent directly to the requester. In addition, a sharing write-back message is sent to the home level to update main memory and change the directory state to indicate that the requesting and remote cluster now have shared copies of the data. Having the dirty cluster respond directly to the requester, as opposed to routing it through the home, reduces the latency seen by the requesting processor.

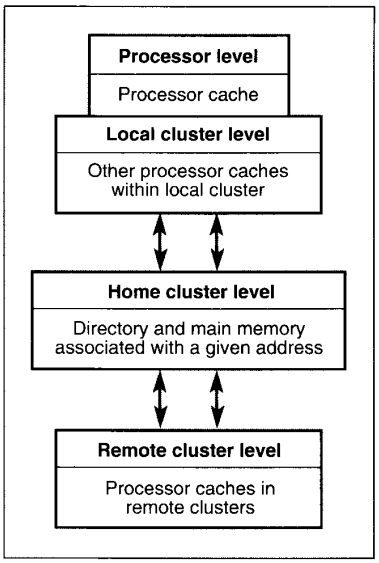


Figure 3. Memory hierarchy of Dash.

Now consider the sequence of operations that occurs when a location is written:

- *Processor level* — If the location is dirty in the writing processor's cache, the write can complete immediately. Otherwise, a read-exclusive request is issued on the local cluster's bus to obtain exclusive ownership of the line and retrieve the remaining portion of the cache line.
- *Local cluster level* — If one of the caches within the cluster already owns the cache line, then the read-exclusive request is serviced at the local level by a cache-to-cache transfer. This allows processors within a cluster to alternately modify the same memory block without any intercluster interaction. If no local cache owns the block, then a read-exclusive request is sent to the home cluster.
- *Home cluster level* — The home cluster can immediately satisfy an ownership request for a location that is in the uncached or shared state. In addition, if a block is in the shared state, then all cached copies must be invalidated. The directory indicates the clusters that have the block cached. Invalidation requests are sent to these clusters while the home concurrently sends an exclusive data reply to the requesting cluster. If the directory indicates that the block is dirty, then the read-exclusive request must be forwarded to the dirty cluster, as in the case of a read.
- *Remote cluster level* — If the directory had indicated that the memory block was shared, then the remote clusters receive an invalidation request to eliminate their shared copy. Upon receiving the invalidation, the remote clusters send an acknowledgment to the requesting cluster. If the directory had indicated a dirty state, then the dirty cluster receives a read-exclusive request. As in the case of the read, the remote cluster responds directly to the requesting cluster and sends a dirty-transfer message to the home indicating that the requesting cluster now holds the block exclusively.

When the writing cluster receives all the invalidation acknowledgments or the reply from the home or dirty cluster, it is guaranteed that all copies of the old data have been purged from the system. If the processor delays completing the write until all acknowledgments are received, then the new write value will become available to all other processors at the same time. However, invalidations involve round-trip messages to multiple clusters, resulting in potentially long delays. Higher processor utilization can be obtained by allowing the write to proceed immediately after the

ownership reply is received from the home. Unfortunately, this may lead to inconsistencies with the memory model assumed by the programmer. The next section describes how Dash relaxes the constraints on memory request ordering, while still providing a reasonable programming model to the user.

Memory consistency. The memory consistency model supported by an architecture directly affects the amount of buffering and pipelining that can take place among memory requests. In addition, it has a direct effect on the complexity of the programming model presented to the user. The goal in Dash is to provide substantial freedom in the ordering among memory requests, while still providing a reasonable programming model to the user.

At one end of the consistency spectrum is the *sequential consistency* model,[3] which requires execution of the parallel program to appear as an interleaving of the execution of the parallel processes on a sequential machine. Sequential consistency can be guaranteed by requiring a processor to complete one memory request before it issues the next request.[4] Sequential consistency, while conceptually appealing, imposes a large performance penalty on memory accesses. For many applications, such a model is too strict, and one can make do with a weaker notion of consistency.

As an example, consider the case of a processor updating a data structure within a critical section. If updating the structure requires several writes, each write in a sequentially consistent system will stall the processor until all other cached copies of that location have been invalidated. But these stalls are unnecessary as the programmer has already made sure that no other process can rely on the consistency of that data structure until the critical section is exited. If the synchronization points can be identified, then the memory need only be consistent at those points. In particular, Dash supports the use of the *release consistency* model,[5] which only requires the operations to have completed before a critical section is released (that is, a lock is unlocked).

Such a scheme has two advantages. First, it provides the user with a reasonable programming model, since the programmer is assured that when the critical section is exited, all other processors will have a consistent view of the modified data structure. Second, it permits reads to bypass writes and the invalidations of different write operations to overlap, resulting in lower latencies for accesses and higher overall performance. Detailed simulation studies for processors with blocking reads have shown that release consistency provides a 10- to 40-percent increase in performance over sequential consistency.[5] The disadvantage of the model is that the programmer or compiler must identify all synchronization accesses.

Release consistency provides a 10- to 40-percent increase in performance over sequential consistency.

The Dash prototype supports the release consistency model in hardware. Since we use commercial microprocessors, the processor stalls on read operations until the read data is returned from the cache or lower levels of the memory hierarchy. Write operations, however, are nonblocking. There is a write buffer between the first- and second-level caches. The write buffer queues up the write requests and issues them in order. Furthermore, the servicing of write requests is overlapped. As soon as the cache receives the ownership and data for the requested cache line, the write data is removed from the write buffer and written into the cache line. The next write request can be serviced while the invalidation acknowledgments for the previous write operations filter in. Thus, parallelism exists at two levels: the processor executes other instructions and accesses its first-level cache while write operations are pending, and invalidations of multiple write operations are overlapped.

The Dash prototype also provides fence operations that stall the processor or write-buffer until previous operations complete. These fence operations allow software to emulate more stringent consistency models.

Memory access optimizations. The use of release consistency helps hide the latency of write operations. However, since the processor stalls on read operations, it sees the entire duration of all read accesses. For applications that exhibit poor cache behavior or extensive read/write sharing, this can lead to significant delays while the processor waits for remote cache misses to be filled. To help with these problems Dash provides a variety of prefetch and pipelining operations.

Prefetch operations. A prefetch operation is an explicit nonblocking request to fetch data before the actual memory operation is issued. Hopefully, by the time the process needs the data, its value has been brought closer to the processor, hiding the latency of the regular blocking read. In addition, nonblocking prefetch allows the pipelining of read misses when multiple cache blocks are prefetched. As a simple example of its use, a process wanting to access a row of a matrix stored in another cluster's memory can do so efficiently by first issuing prefetch reads for all cache blocks corresponding to that row.

Dash's prefetch operations are nonbinding and software controlled. The processor issues explicit prefetch operations that bring a shared or exclusive copy of the memory block into the processor's cache. Not binding the value at the time of the prefetch is important in that issuing the prefetch does not affect the consistency model or force the compiler to do a conservative static dependency analysis. The coherence protocol keeps the prefetched cache line coherent. If another processor happens to write to the location before the prefetching processor accesses the data, the data will simply be invalidated. The prefetch will be rendered ineffective, but the program will execute correctly. Support for an exclusive prefetch operation aids cases where the block is first read and then updated. By first issuing the exclusive prefetch, the processor avoids first obtaining a shared copy and then having to rerequest an exclusive copy of the block. Studies have shown that, for certain applications, the addition of a small number of prefetch instructions can increase processor utilization by more than a factor of two.[6]

Update and deliver operations. In some applications, it may not be possible for the consumer process to issue a prefetch early enough to effectively hide the latency of memory. Likewise, if multiple

consumers need the same item of data, the communication traffic can be reduced if data is multicast to all the consumers simultaneously. Therefore, Dash provides operations that allow the producer to send data directly to consumers. There are two ways for the producing processor to specify the consuming processors. The *update-write* operation sends the new data directly to all processors that have cached the data, while the *deliver* operation sends the data to specified clusters.

The *update-write* primitive updates the value of all existing copies of a data word. Using this primitive, a processor does not need to first acquire an exclusive copy of the cache line, which would result in invalidating all other copies. Rather, data is directly written into the home memory and all other caches holding a copy of the line. These semantics are particularly useful for event synchronization, such as the release event for a barrier.

The *deliver* instruction explicitly specifies the destination clusters of the transfer. To use this primitive, the producer first writes into its cache using normal, invalidating write operations. The producer then issues a deliver instruction, giving the destination clusters as a bit vector. A copy of the cache line is then sent to the specified clusters, and the directory is updated to indicate that the various clusters now share the data. This operation is useful in cases when the producer makes multiple writes to a block before the consumers will want it or when the consumers are unlikely to be caching the item at the time of the write.

Support for synchronization. The access patterns to locations used for synchronization are often different from those for other shared data. For example, whenever a highly contended lock is released, waiting nodes rush to grab the lock. In the case of barriers, many processors must be synchronized and then released. Such activity often causes hot spots in the memory system. Consequently, synchronization variables often warrant special treatment. In addition to update writes, Dash provides two extensions to the coherence protocol that directly support synchronization objects. The first is queue-based locks, and the second is fetch-and-increment operations.

Most cache-coherent architectures handle locks by providing an atomic test&set instruction and a cached test-and-test&set scheme for spin waiting. Ideally, these spin locks should meet the following criteria:

- minimum amount of traffic generated while waiting,
- low latency release of a waiting processor, and
- low latency acquisition of a free lock.

Cached test&set schemes are moderately successful in satisfying these criteria for low-contention locks, but fail for high-contention locks. For example, assume there are N processors spinning on a lock value in their caches. When the lock is released, all N cache values are invalidated, and N reads are generated to the memory system. Depending on the timing, it is possible that all N processors come back to do the test&set on the location once they realize the lock is free, resulting in further invalidations and rereads. Such a scenario produces unnecessary traffic and increases the latency in acquiring and releasing a lock.

The *queue-based locks* in Dash address this problem by using the directory to indicate which processors are spinning on the lock. When the lock is released, one of the waiting clusters is chosen at random and is granted the lock. The grant request invalidates only that cluster's caches and allows one processor within that cluster to acquire the lock with a local operation. This scheme lowers both the traffic and the latency involved in releasing a processor waiting on a lock. Informing only one cluster of the release also eliminates unnecessary traffic and latency that would be incurred if all waiting processors were allowed to contend. A time-out mechanism on the lock grant allows the grant to be sent to another cluster if the spinning process has been swapped out or migrated. The queued-on-lock-bit primitive described in Goodman et al.[7] is similar to Dash's queue-based locks, but uses pointers in the processor caches to maintain the list of the waiting processors.

The *fetch-and-increment* and *fetch-and-decrement* primitives provide atomic increment and decrement operations on uncached memory locations. The value returned by the operations is the value before the increment or decrement. These operations have low serialization and are useful for implementing several synchronization primitives such as barriers, distributed loops, and work queues. The serialization of these operations is small because they are done directly at the memory site. The low serialization provided by the fetch-and-increment operation is especially important when many processors want to increment a location, as happens when getting the next index in a distributed loop. The benefits of the proposed operations become apparent when contrasted with the alternative of using a normal variable protected by a lock to achieve the atomic increment and decrement. The alternative results in significantly more traffic, longer latency, and increased serialization.

The Dash implementation

A hardware prototype of the Dash architecture is currently under construction. While we have developed a detailed software simulator of the system, we feel that a hardware implementation is needed to fully understand the issues in the design of scalable cache-coherent machines, to verify the feasibility of such designs, and to provide a platform for studying real applications and software running on a large ensemble of processors.

To focus our effort on the novel aspects of the design and to speed the completion of a usable system, the base cluster hardware used in the prototype is a commercially available bus-based multiprocessor. While there are some constraints imposed by the given hardware, the prototype satisfies our primary goals of scalable memory bandwidth and high performance. The prototype includes most of Dash's architectural features since many of them can only be fully evaluated on the actual hardware. The system also includes dedicated performance monitoring logic to aid in the evaluation.

Dash prototype cluster. The prototype system uses a Silicon Graphics Power Station 4D/340 as the base cluster. The 4D/340 system consists of four Mips R3000 processors and R3010 floating-point coprocessors running at 33 megahertz. Each R3000/R3010 combination can reach execution rates up to 25 VAX MIPS and 10 Mflops. Each

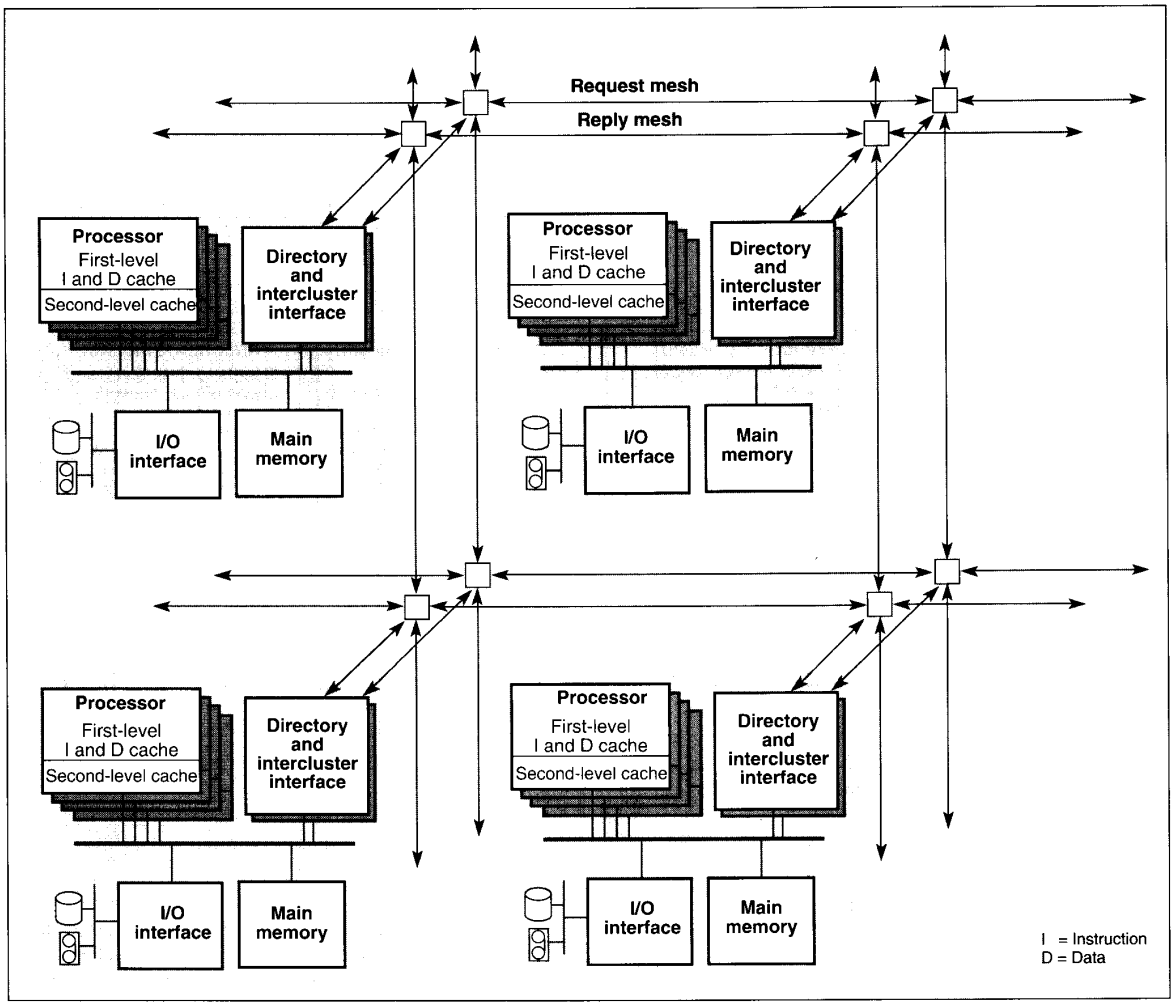


Figure 4. Block diagram of a 2 × 2 Dash system.

CPU contains a 64-kilobyte instruction cache and a 64-Kbyte write-through data cache. The 64-Kbyte data cache interfaces to a 256-Kbyte second-level write-back cache. The interface consists of a read buffer and a four-word-deep write buffer. Both the first- and second-level caches are direct-mapped and support 16-byte lines. The first level caches run synchronously to their associated 33-MHz processors while the second level caches run synchronous to the 16-MHz memory bus.

The second-level processor caches are responsible for bus snooping and maintaining coherence among the caches in the cluster. Coherence is maintained using an Illinois, or MESI (modified, exclusive, shared, invalid), protocol. The main advantage of using the Illinois protocol in Dash is the cache-to-cache transfers specified in it. While they do little to reduce the latency for misses serviced by local memory, local cache-to-cache transfers can greatly reduce the penalty for remote memory misses. The set of processor caches acts as a cluster cache for remote memory. The memory bus (MPbus) of the 4D/340 is a synchronous bus and consists of separate 32-bit address and 64-bit data buses. The MPbus is pipelined and supports memory-to-cache and cache-to-cache transfers of 16 bytes every four bus clocks with a latency of six bus clocks. This results in a maximum bandwidth of 64 Mbytes per second. While the MPbus is pipelined, it is not a split-transaction bus.

To use the 4D/340 in Dash, we have had to make minor modifications to the existing system boards and design a pair of new boards to support the directory memory and intercluster interface. The main modification to the existing boards is to add a bus retry signal that is used when a request requires service from a remote cluster. The central bus arbiter has also been modified to accept a mask from the directory. The mask holds off a processor's retry until the remote request has been serviced. This effectively creates a split-transaction bus protocol for requests requiring remote service. The new directory controller boards contain the directory memory, the intercluster coherence state machines and buffers, and a local section of the global interconnection network. The interconnection network consists of a pair of wormhole routed meshes, each with 16-bit wide channels. One mesh is dedicated to the request messages while the other handles replies. Figure 4 shows a block diagram of four clusters connected to form a 2 × 2 Dash system. Such a system could scale to support hundreds

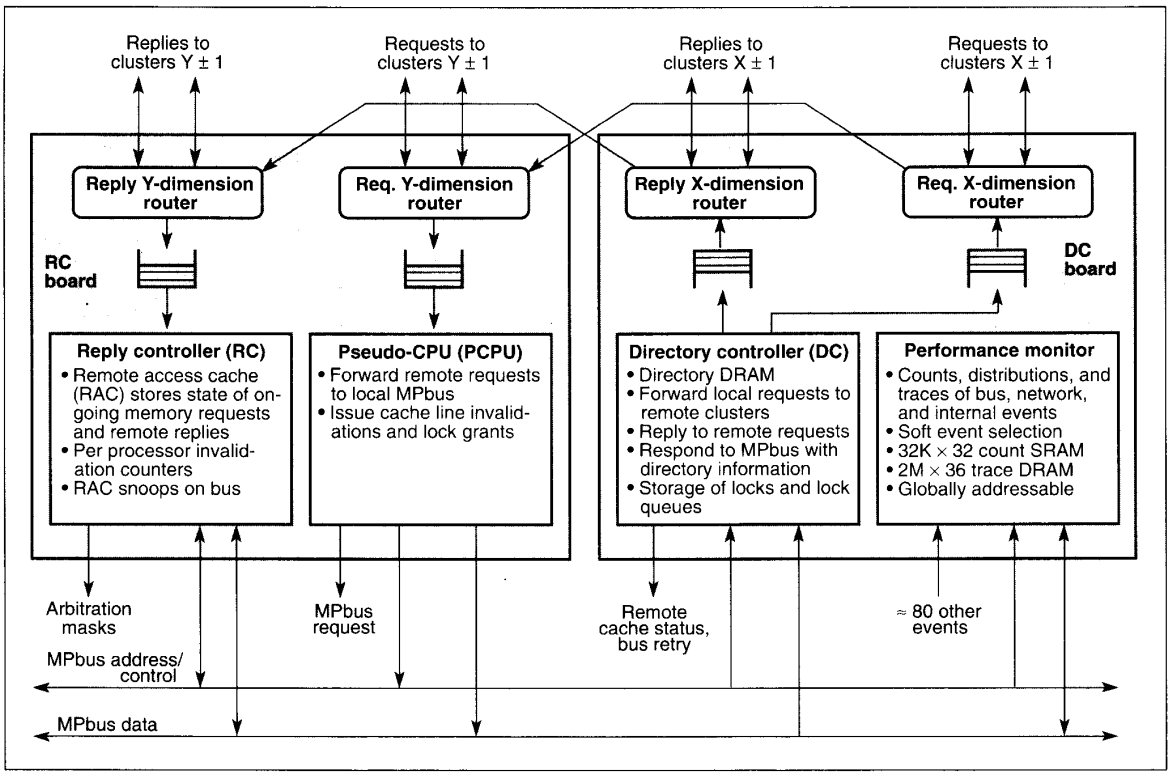


Figure 5. Block diagram of directory boards.

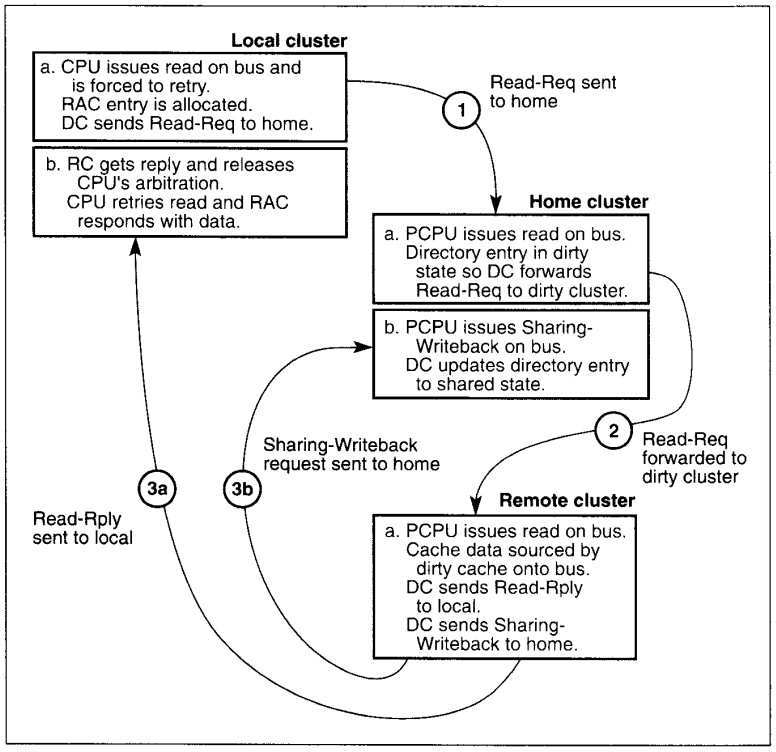


Figure 6. Flow of a read request to remote memory that is dirty in a remote cluster.

of processors, but the prototype will be limited to a maximum configuration of 16 clusters. This limit was dictated primarily by the physical memory addressability (256 Mbytes) of the 4D/340 system, but still allows for systems up to 64 processors that are capable of 1.6 GIPS and 600 scalar Mflops.

Dash directory logic. The directory logic implements the directory-based coherence protocol and connects the clusters within the system. Figure 5 shows a block diagram of the directory boards. The directory logic is split between the two logic boards along the lines of the logic used for outbound and inbound portions of intercluster transactions.

The directory controller (DC) board contains three major sections. The first is the directory controller itself, which includes the directory memory associated with the cachable main memory contained within the cluster. The DC logic initiates all outbound network requests and replies. The second section is the performance monitor, which can count and trace a variety of intra- and intercluster events. The third major section is the request and reply outbound

network logic together with the X-dimension of the network itself.

Each bus transaction accesses directory memory. The directory information is combined with the type of bus operation, the address, and the result of snooping on the caches to determine what network messages and bus controls the DC will generate. The directory memory itself is implemented as a bit vector with one bit for each of the 16 clusters. While a full-bit vector has limited scalability, it was chosen because it requires roughly the same amount of memory as a limited pointer directory given the size of the prototype, and it allows for more direct measurements of the machine's caching behavior. Each directory entry contains a single state bit that indicates whether the clusters have a shared or dirty copy of the data. The directory is implemented using dynamic RAM technology, but performs all necessary actions within a single bus transaction.

The second board is the reply controller (RC) board, which also contains three major sections. The first section is the reply controller, which tracks outstanding requests made by the local processors and receives and buffers replies from remote clusters using the remote access cache (RAC). The second section is the pseudo-CPU (PCPU), which buffers incoming requests and issues them to the cluster bus. The PCPU mimics a CPU on this bus on behalf of remote processors except that responses from the bus are sent out by the directory controller. The final section is the inbound network logic and the Y-dimension of the mesh routing networks.

The reply controller stores the state of ongoing requests in the remote access cache. The RAC's primary role is the coordination of replies to intercluster transactions. This ranges from the simple buffering of reply data between the network and bus to the accumulation of invalidation acknowledgments and the enforcement of release consistency. The RAC is organized as a 128-Kbyte direct-mapped snoopy cache with 16-byte cache lines.

One port of the RAC services the inbound reply network while the other snoops on bus transactions. The RAC is lockup-free in that it can handle several outstanding remote requests from each of the local processors. RAC entries are allocated when a local processor initiates a remote request, and they persist until all intercluster transactions relative to that request have completed. The snoopy nature of the RAC naturally lends itself to merging requests made to the same cache block by different processors and takes advantage of the cache-to-cache transfer protocol supported between the local processors. The snoopy structure also allows the RAC to supplement the function of the processor caches. This includes support for a dirty-sharing state for a cluster (normally the Illinois protocol would force a write-back) and operations such as prefetch.

Interconnection network. As stated in the architecture section, the Dash coherence protocol does not rely on a particular interconnection network topology. However, for the architecture to be scalable, the network itself must provide scalable bandwidth. It should also provide low-latency communication. The prototype system uses a pair of *wormhole* routed meshes to implement the interconnection network. One mesh handles request messages while the other is dedicated to replies. The networks are based on variants of the mesh routing chips developed at the California Institute of Technology, where the data paths have been extended from 8 to 16 bits. Wormhole routing allows a cluster to forward a message after receiving only the first flit (flow unit) of the packet, greatly reducing the latency through each node. The average latency for each hop in the network is approximately 50 nanoseconds. The networks are asynchronous and self-timed. The bandwidth of each link is limited by the round-trip delay of the request-acknowledge signals. The prototype transfers flits at approximately 30 MHz, resulting in a total bandwidth of 120 Mbytes/second in and out of each cluster.

An important constraint on the network is that it must deliver request and reply messages without deadlocking. Most networks, including the meshes used in Dash, are guaranteed to be deadlock-free if messages are consumed at the receiving cluster. Unfortunately, the Dash prototype cannot guarantee this due, first, to the limited buffering on the directory boards and also to the fact that a cluster may need to generate an outgoing message before it can consume an incoming message. For example, to service a read request, the home cluster must generate a reply message containing the data. Similarly, to process a request for a dirty location in a remote cluster, the home cluster needs to generate a forwarding request to that cluster. This requirement adds the potential for deadlocks that consist of a sequence of messages having circular dependencies through a node.

Dash avoids these deadlocks through three mechanisms. First, reply messages can always be consumed because they are allocated a dedicated reply buffer in the RAC. Second, the independent request and reply meshes eliminate request-reply deadlocks. Finally, a back-off mechanism breaks potential deadlocks due to request-request dependencies. If inbound requests cannot be forwarded because of blockages on the outbound request port, the requests are rejected by sending negative acknowledgment reply messages. Rejected requests are then retried by the issuing processor.

Coherence examples. The following examples illustrate how the various structures described in the previous sections interact to carry out the coherence protocol. For a more detailed discussion of the protocol, see Lenoski et al.[8]

Figure 6 shows a simple read of a memory location whose home is in a remote cluster and whose directory state is dirty in another cluster. The read request is not satisfied on the local cluster bus, so a Read-Req (message 1) is sent to the home. At this time the processor is told to retry, and its arbitration is masked. A RAC entry is allocated to track this message and assign ownership of the reply. The PCPU at the home receives the Read-Req and issues a cache read on the bus. The directory memory is accessed and indicates that the cache block is dirty in another cluster. The directory controller in the home forwards the Read-Req (message 2) to the dirty remote cluster. The PCPU in the dirty cluster issues the read on the dirty cluster's bus and the dirty processor's cache responds. The DC in the dirty cluster sends a Read-Rply (message 3a) to the local cluster and a Sharing-Writeback (message 3b) request to the home to update the directory and main memory. The RC in the local cluster receives the reply into the RAC, releases the requesting CPU for arbitration, and then sources the data onto the bus when the processor retries the read. In parallel,

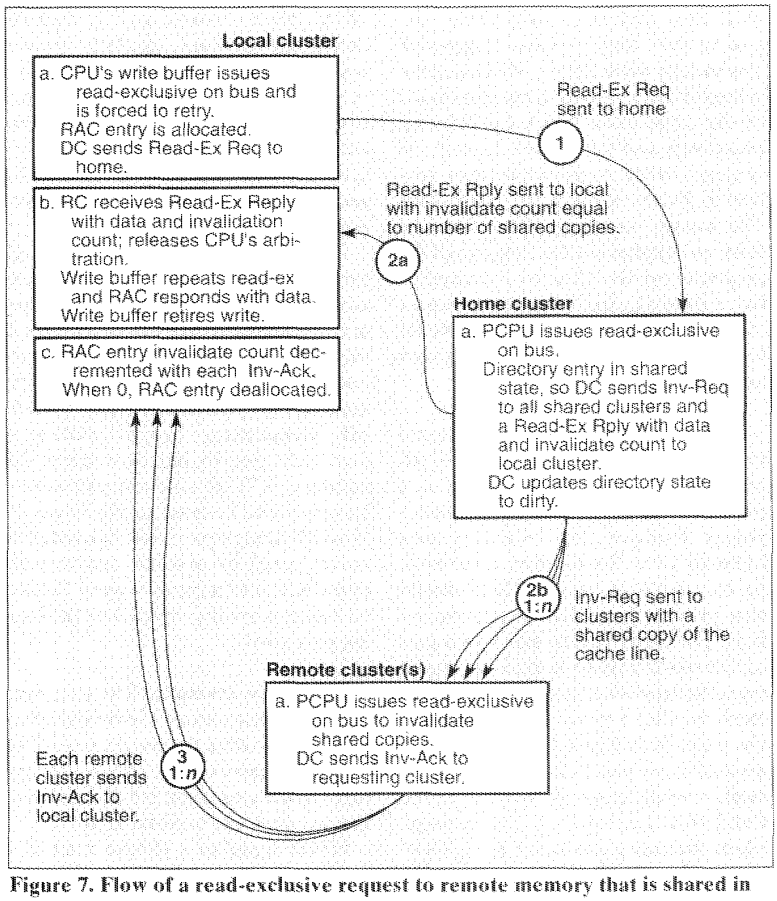


Figure 7. Flow of a read-exclusive request to remote memory that is shared in remote clusters.

the Sharing-Writeback request is received by the home PCPU, which issues it onto the bus. The sharing writeback updates the directory to a shared state indicating that the local and dirty clusters now have a read-only copy of the memory block.

Figure 7 shows the corresponding sequence for a store operation that requires remote service. The invalidation-based protocol requires the processor (actually the write buffer) to acquire exclusive ownership of the cache block before completing the store. Thus, if a store is made to a block that the processor does not have cached, or only has cached in a shared state, the processor issues a read-exclusive request on the local bus. In this case, no other cache holds the block dirty in the local cluster so a Read-Ex Req (message 1) is sent to the home cluster. As before, a RAC entry is allocated in the local cluster. At the home, the PCPU issues the read-exclusive request to the bus. The directory indicates that the line is in the shared state. This results in the DC sending a Read-Ex Rply (message 2a) to the local cluster and invalidation requests (Inv-Req, messages 2b) to the sharing clusters. The home cluster owns the block, so it can immediately update the directory to the dirty state indicating that the local cluster now holds an exclusive copy of the memory line. The

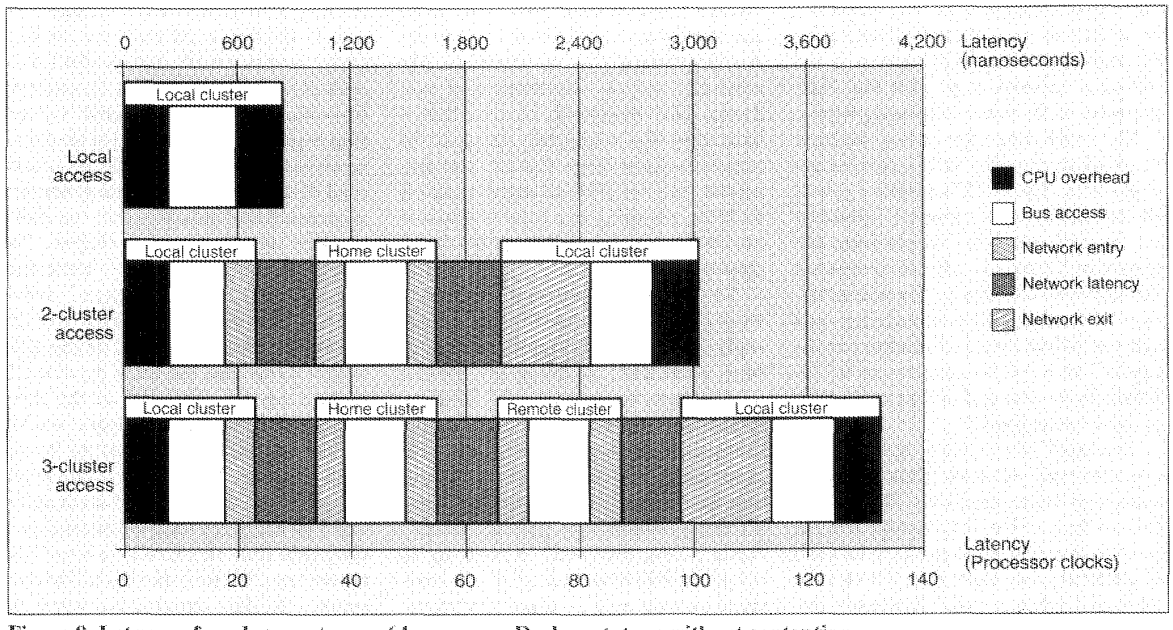


Figure 8. Latency of read requests on a 64-processor Dash prototype without contention.

Read-Ex Rply message is received in the local cluster by the RC, which can then satisfy the read-exclusive request. To assure consistency at release points, however, the RAC entry persists even after the write-buffer's request is satisfied. The RAC entry is only deallocated when it receives the number of invalidate acknowledgments (Inv-Ack, message 3) equal to an invalidation count sent in the original reply message. The RC maintains per-processor RAC allocation counters to allow the hardware to stall releasing synchronization operations until all earlier writes issued by the given processor have completed systemwide.

An important feature of the coherence protocol is its forwarding strategy. If a cluster cannot reply directly to a given request, it forwards responsibility for the request to a cluster that should be able to respond. This technique minimizes the latency for a request, as it always forwards the request to where the data is thought to be and allows a reply to be sent directly to the requesting cluster. This technique also minimizes the serialization of requests since no cluster resources are blocked while intercluster messages are being sent. Forwarding allows the directory controller to work on multiple requests concurrently (that is, makes it multithreaded) without having to retain any additional state about forwarded requests.

Software support

A comprehensive software development environment is essential to make effective use of large-scale multiprocessors. For Dash, our efforts have focused on four major areas: operating systems, compilers, programming languages, and performance debugging tools.

Dash supports a full-function Unix operating system. In contrast, many other highly parallel machines (for example, Intel iPSC2, Ncube, iWarp) support only a primitive kernel on the node processors and rely on a separate host system for program development. Dash avoids the complications and inefficiencies of a host system. Furthermore, the resident operating system can efficiently support multiprogramming and multiple users on the system. Developed in cooperation with Silicon Graphics, the Dash OS is a modified version of the existing operating system on the 4D/340 (Irix, a variation of Unix System V.3). Since Irix was already multithreaded and worked with multiple processors, many of our changes have been made to accommodate the hierarchical nature of Dash, where processors, main memory, and I/O devices are all partitioned across the clusters. We have also adapted the Irix kernel to provide access to the special hardware features of Dash such as prefetch, update write, and queue-based locks. Currently, the modified OS is running on a four-cluster Dash system, and we are exploring several new algorithms for process scheduling and memory allocation that will exploit the Dash memory hierarchy.

At the user level, we are working on several tools to aid the development of parallel programs for Dash. At the most primitive level, a parallel macro library provides structured access to the underlying hardware and operating-system functions. This library permits the development and porting of parallel applications to Dash using standard languages and tools. We are also developing a parallelizing compiler that extracts parallelism from programs written for sequential machines and tries to improve data locality. Locality is enhanced by increasing cache utilization through *blocking* and by reducing remote accesses through *static partitioning* of computation and data. Finally, *prefetching* is used to hide latency for remote accesses that are unavoidable.

Because we are interested in using Dash for a wide variety of applications, we must also find parallelism beyond the loop level. To attack this problem we have developed a new parallel language called Jade, which allows a programmer to easily express dynamic coarse-grain parallelism. Starting with a sequential program, a programmer simply augments those sections of code to be parallelized with side-effect information. The compiler and runtime system use this information to execute the program concurrently while respecting the program's data dependence constraints. Using Jade can significantly reduce the time and effort required to develop a parallel version of a serial application. A prototype of Jade is operational, and applications developed with Jade include sparse-matrix Cholesky factorization, Locus Route (a printed-circuit-board routing algorithm), and MDG (a water simulation code).

To complement our compiler and language efforts, we are developing a suite of performance monitoring and analysis tools. Our high-level tools can identify portions of code where the concurrency is smallest or where the most execution time is spent. The high-level tools also provide information about synchronization bottlenecks and load-balancing problems. Our low-level tools will couple with the built-in hardware monitors in Dash. As an example, they will be able to identify portions of code where most cache misses are occurring and will frequently provide the reasons for such misses. We expect such noninvasive monitoring and profiling tools to be invaluable in pinpointing critical regions for optimization to the programmer.

Dash performance

This section presents performance data from the Dash prototype system. First, we summarize the latency for memory accesses serviced by the three lower levels of the memory hierarchy. Second, we present speedup for three parallel applications running on a simulation of the prototype using one to 64 processors. Finally, we present the actual speedups for these applications measured on the initial 16-processor Dash system.

While caches reduce the effective access time of memory, the latency of main memory determines the sensitivity of processor utilization to cache and cluster locality and indicates the costs of interprocessor communication. Figure 8 shows the unloaded latencies for read misses that are satisfied within the local cluster, within the home cluster, and by a remote (that is, dirty) cluster. Latencies for read-exclusive requests issued by the write buffer are similar. A read miss to the local cluster takes 29 processor clocks (870 ns), while a remote miss takes roughly 3.5 times as long. The delays arise primarily from the relatively slow bus in the 3D/340 and from our implementation's conservative technology and packaging. Detailed simulation has shown that queuing delays can add 20 to 120 percent to these delays. While higher levels of integration could reduce the absolute time of the prototype latencies, we believe

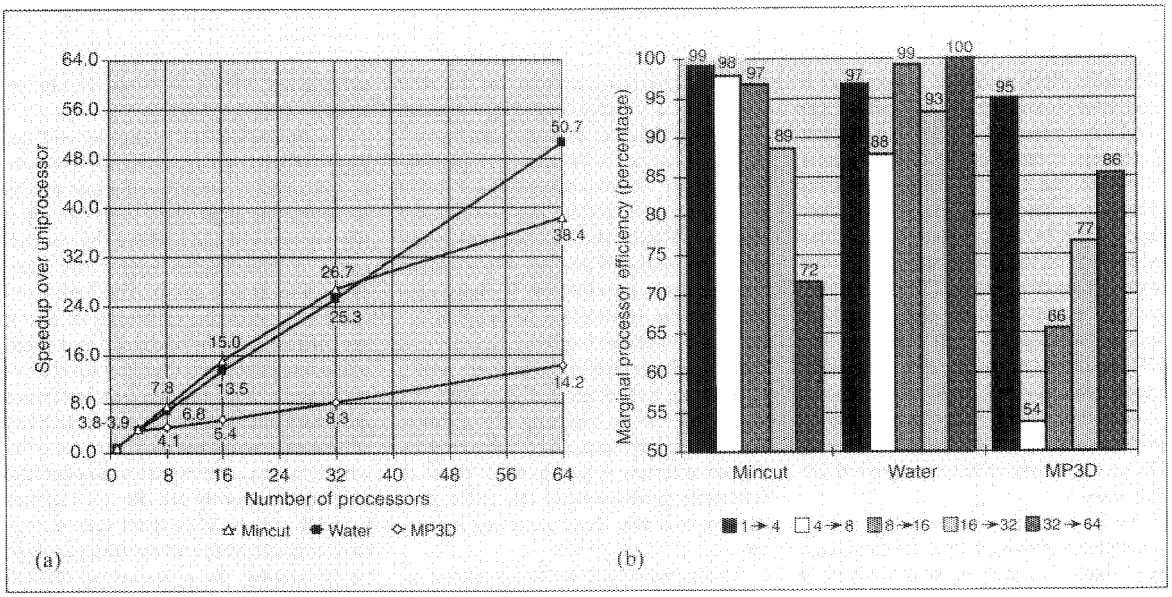


Figure 9. Speedup of three parallel applications on a simulation of the Dash prototype with one to 64 processors: (a) overall application speedup; (b) marginal efficiency of additional clusters.

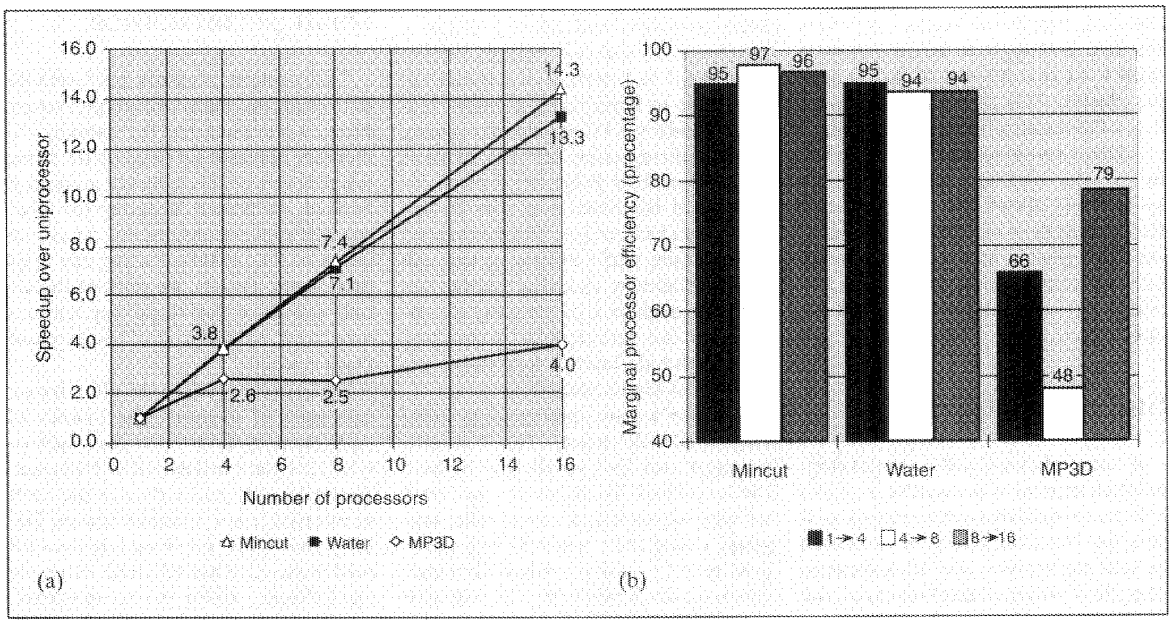


Figure 10. Speedup of three parallel applications on the actual Dash prototype hardware with one to 16 processors: (a) overall application speedup; (b) marginal efficiency of additional clusters.

the increasing clock rate of microprocessors implies that the latencies measured in processor clocks will remain similar.

Applications for large-scale multiprocessors must utilize locality to realize good cache hit rates, minimize remote accesses, and achieve high processor utilization. Figure 9 shows the speedup and processor efficiency for three applications on simulated Dash systems consisting of one to 64 processors (that is, one to 16 clusters). The line graph shows overall application speedup, while the bar chart shows the marginal efficiency of additional clusters. The marginal efficiency is defined as the average processor utilization, assuming processors were 100 percent utilized at the previous data point. The three applications simulated are Water, Mincut, and MP3D. Water is a molecular-dynamics code that computes the energy of a system of water molecules. Mincut uses parallel simulated annealing to solve a graph-partitioning problem. MP3D models a wind tunnel in the upper atmosphere, using a discrete particle-based simulation.

The applications were simulated using a combination of the Tango multi-

processor simulator and a detailed memory simulator for the Dash prototype. Tango allows a parallel application to run on a uniprocessor and generates a parallel memory-reference stream. The detailed memory simulator is tightly coupled with Tango and provides feedback on the latency of individual memory operations.

On the Dash simulator, Water and Mincut achieve reasonable speedup through 64 processors. For Water, the reason is that the application exhibits good locality. As the number of clusters increases from two to 16, cache hit rates are relatively constant, and the percent of cache misses handled by the local cluster only decreases from 69 to 64 percent. Thus, miss penalties increase only slightly with system size and do not adversely affect processor utilizations. For Mincut, good speedup results from very good cache hit rates (98 percent for shared references). The speedup falls off for 64 processors due to lock contention in the application.

MP3D obviously does not exhibit good speedup on the Dash prototype. This particular encoding of the MP3D application requires frequent interprocessor communication, thus resulting in frequent cache misses. On average, about 4 percent of the instructions executed in MP3D generate a read miss for a shared data item. When only one cluster is being used, all these misses are serviced locally. However, when we go to two clusters, a large fraction of the cache misses are serviced remotely. This more than doubles the average miss latency, thus nullifying the potential gain from the added processors. Likewise, when four clusters are used, the full benefit is not realized because most misses are now serviced by a remote dirty cache, requiring a three-hop access.

Reasonable speedup is finally achieved when going from 16 to 32 and 64 processors (77 percent and 86 percent marginal efficiency, respectively), but overall speedup is limited to 14.2. Even on MP3D, however, caching is beneficial. A 64-processor system with the timing of Dash, but without the caching of shared data, achieves only a 4.1 speedup over the cached uniprocessor. For Water and Mincut the improvements from caching are even larger.

Figure 10 shows the speedup for the three applications on the real Dash hardware using one to 16 processors. The applications were run under an early version of the Dash OS. The results for Water and Mincut correlate well with the simulation results, but the MP3D speedups are somewhat lower. The problem with MP3D appears to be that simulation results did not include private data references. Since MP3D puts a heavy load on the memory system, the extra load of private misses adds to the queuing delays and reduces the multiprocessor speedups.

We have run several other applications on our 16-processor prototype. These include two hierarchical n-body applications (using Barnes-Hut and Greengard-Rokhlin algorithms), a radiosity application from computer graphics, a standard-cell routing application from very large scale integration computer-aided design, and several matrix-oriented applications, including one performing sparse Cholesky factorization. There is also an improved version of the MP3D application that exhibits better locality and achieves almost linear speedup on the prototype.

Over this initial set of 10 parallel applications, the harmonic mean of the speedup on 16 processors in 10.5 Furthermore, if old MP3D is left out, the harmonic mean rises to over 12.8. Overall, our experience with the 16-processor machine has been very promising and indicates that many applications should be able to achieve over 40 times speedup on the 64-processor system.

Related work

There are other proposed scalable architectures that support a single address space with coherent caches. A comprehensive comparison of these machines with Dash is not possible at this time, because of the limited experience with this class of machines and the lack of details on many of the critical machine parameters. Nevertheless, a general comparison illustrates some of the design trade-offs that are possible.

Encore GigaMax and Stanford Paradigm. The Encore GigaMax architecture[9] and the Stanford Paradigm project[10] both use a hierarchy-of-buses approach to achieve scalability. At the top level, the Encore GigaMax is composed of several clusters on a global bus. Each cluster consists of several processor modules, main memory, and a cluster cache. The cluster cache holds a copy of all remote locations cached locally and also all local locations cached remotely. Each processing module consists of several processors with private caches and a large, shared, second-level cache. A hierarchical snoopy protocol keeps the processor and cluster caches coherent.

The Paradigm machine is similar to the GigaMax in its hierarchy of processors, caches, and buses. It is different, however, in that the physical memory is all located at the global level, and it uses a hierarchical directory-based coherence protocol. The clusters containing cached data are identified by a bit-vector directory at every level, instead of using snooping cluster caches. Paradigm also provides a lock bit per memory block that enhances performance for synchronization and explicit communication.

The hierarchical structure of these machines is appealing in that they can theoretically be extended indefinitely by increasing the depth of the hierarchy. Unfortunately, the higher levels of the tree cannot grow indefinitely in bandwidth. If a single global bus is used, it becomes a critical link. If multiple buses are used at the top, the protocols become significantly more complex. Unless an application's communication requirements match the bus hierarchy or its traffic-sharing requirements are small, the global bus will be a bottleneck. Both requirements are restrictive and limit the classes of applications that can be efficiently run on these machines.

IEEE Scalable Coherent Interface. The IEEE P1596 Scalable Coherent Interface (SCI) is an interface standard that also strives to provide a scalable system model based on distributed directory-based cache coherence.[11] It differs from Dash in that it is an interface standard, not a complete system design. SCI only specifies the interfaces that each processing node should implement, leaving open the actual node design and exact interconnection network. SCI's role as an interface standard gives it somewhat different goals from those of Dash, but systems based on SCI are likely to have a system organization similar to Dash.

The major difference between SCI and Dash lies in how and where the directory information is maintained. In SCI, the directory is a distributed sharing list maintained by the processor caches

themselves. For example, if processors A, B, and C are caching some location, then the cache entries storing this location include pointers that form a doubly linked list. At main memory, only a pointer to the processor at the head of the linked list is maintained. In contrast, Dash places all the directory information with main memory.

The main advantage of the SCI scheme is that the amount of directory pointer storage grows naturally as new processing nodes are added to the system. Dash-type systems generally require more directory memory than SCI systems and must use a limited directory scheme to scale to a large configuration. On the other hand, SCI directories would typically use the same static RAM technology as the processor caches while the Dash directories are implemented in main memory DRAM technology. This difference tends to offset the potential storage efficiency gains of the SCI scheme.

The primary disadvantage of the SCI scheme is that the distribution of individual directory entries increases the latency and complexity of the memory references, since additional directory-update messages must be sent between processor caches. For example, on a write to a shared block cached by N processors (including the writing processor), the writer must perform the following actions:

- detach itself from the sharing list,
- interrogate memory to determine the head of the sharing list,
- acquire head status from the current head, and
- serially purge the other processor caches by issuing invalidation requests and receiving replies that indicate the next processor in the list.

Altogether, this amounts to $2N + 6$ messages and, more importantly, $N + 1$ serial directory lookups. In contrast, Dash can locate all sharing processors in a single directory lookup, and invalidation messages are serialized only by the network transmission rate.

The SCI working committee has proposed several extensions to the base protocol to reduce latency and support additional functions. In particular, the committee has proposed the addition of directory pointers that allow sharing lists to become sharing trees, support for request forwarding, use of a clean cached state, and support for queue-based locks. While these extensions reduce the differences between the two protocols, they also significantly increase the complexity of SCI.

MIT Alewife. The Alewife machine[12] is similar to Dash in that it uses main memory directories and connects the processing nodes with mesh network. There are three main differences between the two machines:

- Alewife does not have a notion of clusters — each node is a single processor.
- Alewife uses software to handle directory pointer overflow.
- Alewife uses multicontext processors as its primary latency-hiding mechanism.

The size of clusters (one processor, four processors, or more) is dictated primarily by the engineering trade-offs between the overhead of hardware for each node (memory, network interface, and directory) and the bandwidth available within and between clusters. Techniques for scaling directories efficiently are a more critical issue. Whether hardware techniques, such as proposed in O'Krafka and Newton[2] and Gupta et al.,[1] or the software techniques of Alewife will be more effective remains an open question, though we expect the practical differences to be small. Multiple contexts constitute a mechanism that helps hide memory latency, but one that clearly requires additional application parallelism to be effective. Overall, while we believe that support for multiple contexts is useful and can complement other techniques, we do not feel that its role will be larger than other latency-hiding mechanisms such as release consistency and nonbinding prefetch.[13]

We have described the design and implementation decisions for Dash, a multiprocessor that combines the programmability of single-address-space machines with the scalability of message-passing machines. The key means to this scalability are a directory-based cache-coherence protocol, distributed memories and directories, and a scalable interconnection network. The design focuses on reducing memory latency to keep processor performance high, though it also provides latency-hiding techniques such as prefetch and release consistency to mitigate the effects of unavoidable system delays.

At the time of this writing, the 2×2 Dash prototype is stable. It is accessible on the Internet and used daily for research into parallel applications, tools, operating systems, and directory-based architectures. As indicated in the performance section, results from this initial configuration are very promising. Work on extending the 2×2 cluster system to the larger 4×4 (64-processor) system is ongoing. All major hardware components are on hand and being debugged. By the time this article is in print, we expect to have an initial version of the Unix kernel and parallel applications running on the larger machine. ■

Acknowledgments

This research was supported by DARPA contracts N00014-87-K-0828 and N00039-91-C-0138. In addition, Daniel Lenoski is supported by Tandem Computers, James Laudon and Wolf-Dietrich Weber are supported by IBM, and Kourosh Gharachorloo is supported by Texas Instruments. Anoop Gupta is partly supported by a National Science Foundation Presidential Young Investigator Award.

We also thank Silicon Graphics for their technical and logistical support and Valid Logic Systems for their grant of computer-aided engineering tools.

References

1. A. Gupta, W.-D. Weber, and T. Mowry, "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes," *Proc. 1990 Int'l Conf. Parallel Processing*, IEEE Computer Society Press, Los Alamitos, Calif., Order No. 2101, pp. 312-321.

2. B.W. O'Krafka and A.R. Newton, "An Empirical Evaluation of Two Memory-Efficient Directory Methods," *Proc. 17th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 2047, 1990, pp. 138-147.

3. L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Trans. Computers*, Sept. 1979, Vol. C-28, No. 9, pp. 241-248.

4. C. Scheurich and M. Dubois, "Dependency and Hazard Resolution in Multiprocessors," *Proc. 14th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 776, 1987, pp. 234-243.

5. K. Gharachorloo, A. Gupta, and J. Hennessy, "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM, New York, 1991, pp. 245-257.

6. T. Mowry and A. Gupta, "Tolerating Latency Through Software in Shared-Memory Multiprocessors," *J. Parallel and Distributed Computing*, Vol. 12, No. 6, June 1991, pp. 87-106.

7. J.R. Goodman, M.K. Vernon, and P.J. Woest, "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors," *Proc. Third Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, IEEE CS Press, Los Alamitos, Calif., Order No. 1936, 1989, pp. 64-73.

8. D. Lenoski et al., "The Directory-Based Cache Coherence Protocol for the Dash Multiprocessor," *Proc. 17th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 2047, 1990, pp. 148-159.

9. A.W. Wilson, Jr., "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors," *Proc. 14th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 776, 1987, pp. 244-252.

10. D.R. Cheriton, H.A. Goosen, and P.D. Boyle, "Paradigm: A Highly Scalable Shared-Memory Multicomputer Architecture," *Computer*, Vol. 24, No. 2, Feb. 1991, pp. 33-46.

11. D.V. James et al., "Distributed-Directory Scheme: Scalable Coherent Interface," *Computer*, Vol. 23, No. 6, June 1990, pp. 74-77.

12. A. Agarwal et al., "Limitless Directories: A Scalable Cache Coherence Scheme," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM, New York, 1991, pp. 224-234.

13. A. Gupta et al., "Comparative Evaluation of Latency Reducing and Tolerating Techniques," *Proc. 18th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 2146, 1991, pp. 254-263.

Daniel Lenoski is a research scientist in the Processor and Memory Group of Tandem Computers. He recently completed his PhD in electrical engineering in the Computer Systems Laboratory at Stanford University. His research efforts concentrated on the design and implementation of Dash and other issues related to scalable multiprocessors. His prior work includes the architecture definition of Tandem's CLX 600, 700, and 800 series processors.

Lenoski received a BSEE from the California Institute of Technology in 1983 and an MSEE from Stanford in 1985.

James Laudon is a PhD candidate in the Department of Electrical Engineering at Stanford University. His research interests include multiprocessor architectures and algorithms.

Laudon received a BS in electrical engineering from the University of Wisconsin-Madison in 1987 and an MS in electrical engineering from Stanford University in 1988. He is a member of the IEEE Computer Society and ACM.

Kourosh Gharachorloo is a PhD candidate in the Computer Systems Laboratory at Stanford University. His research interests focus on techniques to reduce and tolerate memory latency in large-scale shared-memory multiprocessors.

Gharachorloo received the BS and BA degrees in electrical engineering and economics, respectively, in 1985 and the MS degree in electrical engineering in 1986, all from Stanford University.

Wolf-Dietrich Weber is a PhD candidate in the Computer Systems Laboratory at Stanford University. His research interests focus on directory-based cache coherence for scalable shared-memory multiprocessors.

Weber received the BA and BE degrees from Dartmouth College in 1986. He received an MS degree in electrical engineering from Stanford University in 1987.

Anoop Gupta is an assistant professor of computer science at Stanford University. His primary interests are in the design of hardware and software for large-scale multiprocessors.

Prior to joining Stanford, Gupta was on the research faculty of Carnegie Mellon University, where he received his PhD in 1986. Gupta was the recipient of a DEC faculty development award from 1987-1989, and he received the NSF Presidential Young Investigator Award in 1990.

John Hennessy is a professor of electrical engineering and computer science at Stanford University. His research interests are in exploiting parallelism at all levels to build higher performance computer systems.

Hennessy is the recipient of a 1984 Presidential Young Investigator Award. In 1987, he was named the Willard and Inez K. Bell Professor of Electrical Engineering and Computer Science. In 1991, he was elected an IEEE fellow. During a leave from Stanford in 1984-85, he cofounded Mips Computer Systems where he continues to participate in industrializing the RISC concept as chief scientist.

Mark Horowitz is an associate professor of electrical engineering at Stanford University. His research interests include high-speed digital integrated circuit designs, CAD tools for IC design, and processor architecture. He is a recipient of a 1985 Presidential Young Investigator Award. During a leave from Stanford in 1989-90, he cofounded Rambus, a company that is working on improving memory bandwidth to and from DRAMs.

Horowitz received the BS and SM degrees in electrical engineering and computer science from the Massachusetts Institute of Technology and his PhD in electrical engineering from Stanford University.

Monica S. Lam has been an assistant professor in the Computer Science Department at Stanford University since 1988. Her current research project is to develop a compiler system that optimizes data locality and exploits parallelism at task, loop, and instruction granularities. She was one of the chief architects and compiler designers for the CMU Warp machine and the CMU-Intel's iWarp.

Lam received her BS from University of British Columbia in 1980 and a PhD in computer science from Carnegie Mellon University in 1987.

Readers may contact Daniel Lenoski at Tandem Computers, 19333 Vallco Parkway, MS 3-03, Cupertino, CA 95014; e-mail lenoski_dan @tandem.com. Anoop Gupta can be reached at Stanford Univ., CIS-212, Stanford, CA 94305; e-mail ag@pepper.stanford.edu.