The DASH Prototype: Logic Overhead and Performance

Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy

Abstract—**The fundamental premise behind the DASH project is that it is feasible to build large-scale shared-memory multiprocessors with hardware cache coherence. While paper studies and software simulators are useful for understanding many high-level design tradeoffs, prototypes are essential to ensure that no critical details are overlooked. A prototype provides convincing evidence of the feasibility of the design, allows one to accurately estimate both the hardware and the complexity cost of various features, and provides a platform for studying real workloads. A 48-processor prototype of the DASH multiprocessor is now operational. In this paper, we first examine the hardware overhead of directory-based cache coherence in the prototype. The data show that the overhead is only about 10–15%, which appears to be a small cost for the ease of programming offered by coherent caches and the potential for higher performance. We then discuss the performance of the system and show the speedups obtained by a variety of parallel applications running on the prototype. Using a sophisticated hardware performance monitor, we also characterize the effectiveness of coherent caches and the relationship between an application's reference behavior and its speedup. Finally, we present an evaluation of the optimizations incorporated in the DASH protocol in terms of their effectiveness on parallel applications and on atomic tests that stress the memory system.**[1]

Index Terms— **Directory-based cache coherence, implementation cost, multiprocessor, parallel architecture, performance analysis, shared-memory.**

I. Introduction

FOR parallel architectures to achieve widespread usage it is important that they efficiently run a wide variety of applications without excessive programming difficulty. To maximize both high performance and wide applicability, we believe a parallel architecture should provide i) scalability to support hundreds to thousands of processors, ii) high-performance individual processors, and iii) a single shared address space.

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, and can provide the impetus for programmers to port their sequential applications to parallel architectures. High performance processors are important to achieve both high total system performance and general applicability. A single address space greatly aids in 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 provides better support for parallelizing compilers, standard operating systems, multiprogramming, and incremental tuning of parallel applications.

One important question that arises in the design of such large-scale single-address-space machines is whether or not to allow caching of shared writeable data. The advantage of caching is that it allows higher performance to be achieved by reducing memory latency; the disadvantage is the problem of cache coherence. While solutions to the cache coherence problem are well understood for small-scale multiprocessors, they are unfortunately not so clear for large-scale machines. In fact, no large-scale machine currently supports cache coherence, and it has so far not been clear whether it is feasible to do so, what the benefits are, and what the costs will be.

For the past several years, the DASH (Directory Architecture for SHared memory) project has been exploring the feasibility of building large-scale single-address-space machines with coherent caches. The key ideas are to distribute the main memory among the processing nodes to provide scalable memory bandwidth, and to use a distributed directory-based protocol to support cache coherence. To test our ideas, we have been constructing a prototype DASH machine. The final prototype is to consist of sixty-four 33 MHz MIPS R3000/R3010 processors, delivering up to 1600 MIPS and 600 scalar MFLOPS. A forty-eight processor prototype has recently begun working.

This paper examines the hardware overhead and performance of the prototype DASH system. We begin in Section II with a review of the DASH architecture and machine organization. Section III introduces the structure of the DASH prototype and the components that execute the directory-based coherence protocol. Section IV details the overhead of the directory logic exhibited in the prototype. Section V outlines the structure and function of the performance monitor logic incorporated into each DASH node. Section VI focuses on the performance of the DASH system. The system performance is given in terms of basic memory latency, the speedup on parallel applications, the reference behavior that leads to the observed speedup, and the effectiveness of the optimizations to the coherence protocol. We conclude in

Manuscript received December 4, 1991; revised July 20, 1992. This work was supported by DARPA Contract N00039-91-C-0138. D. Lenoski is supported by Tandem Computers Inc. A. Gupta is also supported by a Presidential Young Investigator Award.

The authors are with the Computer Systems Laboratory, Stanford University, Stanford University, CA 94305.

IEEE Log Number 9203540.

[1] Portions of the text in the abstract and Sections I through VI-B2, along with Figs. 1 through 5 and Tables I through III originally appeared in D. Lenoski *et al.* "The DASH Prototype: Implementation and Performance," in *Proc. 19th Int. Symp. Comput. Architecture,* pp. 92–103, Copyright 1992, Association for Computing Machinery, Inc. Reprinted with permission.

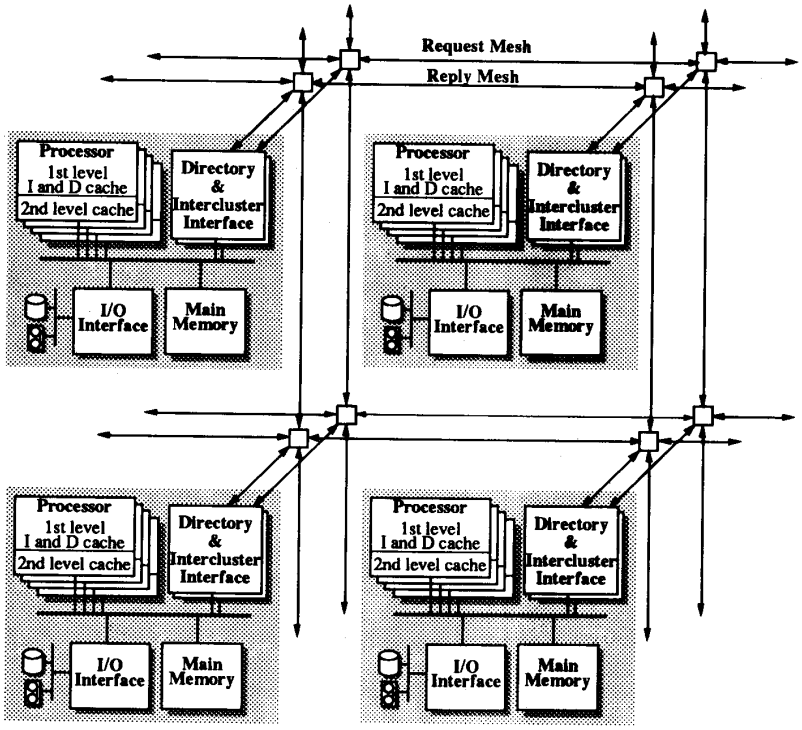


Fig. 1. Block diagram of a 2 × 2 DASH prototype.

Section VII with a summary of our experience with the DASH prototype.

II. The DASH Architecture

The DASH architecture has a two-level structure shown in Fig. 1. At the top level, the architecture consists of a set of processing nodes (clusters) connected through a mesh interconnection network. In turn, each processing node is a bus-based multiprocessor. Intra-cluster cache coherence is implemented using a snoopy bus-based protocol, while inter-cluster coherence is maintained through a distributed directory-based protocol.

The cluster functions as a high-performance processing node. A bus-based cache protocol is chosen for implementing small-scale shared-memory multiprocessors because the bus bandwidth is sufficient to support a small number of processors. The grouping of multiple processors on a bus within each cluster amortizes the cost of the directory logic and network interface among a number of processors. Furthermore, this grouping reduces the directory memory requirements by keeping track of cached lines at a cluster as opposed to processor level.

The directory-based protocol implements an invalidation-based coherence scheme. A memory location may be in one of three states: uncached, that is not cached by any processing node at all; shared, that is in an unmodified state in the caches of one or more nodes; or dirty, that is modified in the cache of some individual node. The directory keeps the summary information for each memory line, specifying the clusters that are caching it [9].

The DASH memory system can be logically broken into the four level hierarchy shown in Fig. 2. The level closest to the processor is the processor cache and is designed to match the speed of the processor. A request that cannot be serviced by the processor cache is sent to the second level in the hierarchy, the local cluster level. This level consists of 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 directory home level. The home level consists of the cluster that contains the directory and physical memory for a given memory address. For some accesses, the local and home cluster are the same and the second and third level access occur simultaneously. In general, however, the request will travel through the interconnect to the home cluster. The home cluster can usually satisfy the request, but if the directory entry is in the dirty state, or in the shared state when the requesting processor requires exclusive access, the fourth, remote cluster level, must be accessed. The remote cluster level responds directly to the local cluster level while also updating the directory level.

In addition to providing coherent caches to reduce memory latency, DASH supports several other techniques for hiding and tolerating memory latency. DASH supports the release consistency model [5], that helps hide latency by allowing buffering and pipelining among memory requests. DASH also supports software-controlled nonbinding prefetching to help hide latency of read operations [7]. Finally,

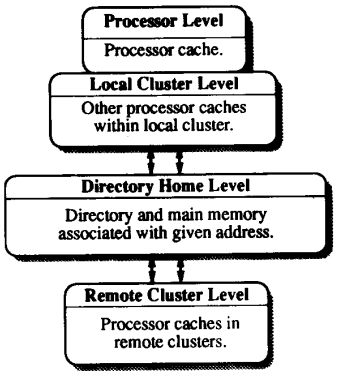


Fig. 2. Logical memory hierarchy of DASH.

DASH supports efficient spin locks in hardware and fetch-and-increment/decrement primitives to help reduce the overhead of synchronization. These optimizations will be discussed in more detail in Section VI-C.

III. The DASH Prototype

To focus our effort on the novel aspects of the design and to speed the completion of a usable system, the base cluster hardware of 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 system uses a Silicon Graphics POWER Station 4D/340 as the base cluster [2]. The 4D/340 system consists of four MIPS R3000 processors and R3010 floating-point coprocessors running at 33 MHz. Each R3000/R3010 combination can sustain execution rates of 25 VAX MIPS and 10 MFLOPS. Each CPU contains a 64 Kbyte 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 4 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 with a MESI (Illinois) protocol [14], and inclusion of the first-level cache by the second-level. The main advantage of using the Illinois protocol in DASH is the cache-to-cache transfers specified in this protocol. 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 effectively act 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 4 bus clocks with a latency of 6 bus clocks. This results in a maximum bandwidth of 64 Mbytes/s.

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 inter-cluster 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 which 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.

While the prototype could scale to support hundreds of processors, it is limited to a maximum configuration of 16 clusters and 64 processors. This limit was dictated primarily by the physical memory addressability of the 4D/340 system (256 Mbytes) which would severely limit the memory per processor in a larger system.

The directory logic in DASH is responsible for implementing the directory-based coherence protocol and interconnecting the clusters within the system. Pictures of the directory boards are shown in Fig. 3. The directory logic is split between the two boards along the lines of the logic used for outbound and inbound portions of inter-cluster transactions.

The DC board contains three major subsections. The first section is the directory controller (DC) itself, which includes the directory memory associated with the cachable main memory contained within the cluster. The DC logic initiates all out-bound network requests and replies. The second section is the performance monitor which can count and trace a variety of intra- and inter-cluster events. The third major section is the request and reply outbound network logic together with the X-dimension of the network itself. The second board is the RC board which also contains three major sections. The first section is the reply controller (RC) 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 is responsible for buffering incoming requests and issuing these requests onto 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.

Directory memory is accessed on each bus transaction. 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 organization is similar to the original directory scheme proposed by Censier and Feautrier [3]. Directory pointers are stored as a bit vector with 1 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

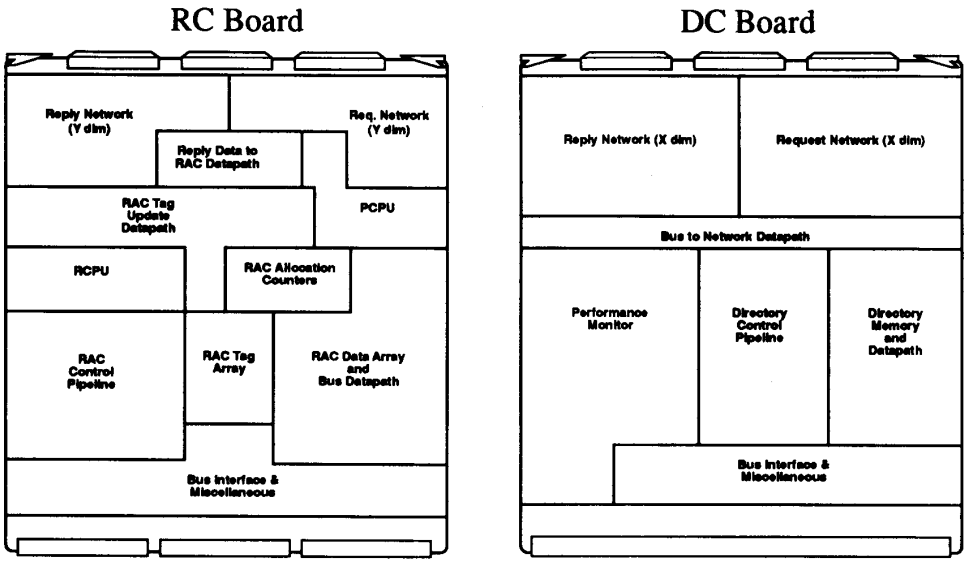


Fig. 3. Directory and reply controller boards.

direct measurements of the caching behavior of the machine. 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 DRAM technology, but performs all necessary actions within a single bus transaction.

The reply controller stores the state of on-going requests in the remote access cache (RAC). The RAC's primary role is the coordination of replies to inter-cluster 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 in-bound reply network while the other snoops on bus transactions. The RAC is lock-up free in that it can handle several outstanding remote requests from each of the local processors. RAC entries are allocated when a remote request is initiated by a local processor and persist until all inter-cluster 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, instead, in DASH the RAC takes the data in a dirty-shared state) and operations such as prefetch (the RAC holds the prefetched data).

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 Caltech where the datapaths have been extended from 8 to 16 bits [4]. 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 (50 ns per hop). The bandwidth of each self-timed link is limited by the round trip delay of the request-acknowledge signals. In the prototype flits are transferred at approximately 30 MHz, resulting in a peak bandwidth of 120 Mbytes/s in and out of each cluster.

IV. Gate Count Summary

One important result of building the DASH prototype is that it provides a realistic model of the cost of directory-based cache coherence. While some of these costs are tied to the specific prototype implementation (e.g., the full DRAM directory vector), they provide a complete picture of one system.

At a high level, the cost of the directory logic can be estimated by the fact that a DASH cluster includes six logic cards, four of which represent the base processing node and two of which are used for directory and inter-cluster coherence. This is a very conservative estimate, however, because Silicon Graphics' logic, in particular the MIPS processor chips and Silicon Graphics' gate arrays, are more highly integrated than the MSI PAL's and LSI FPGA's used in the directory logic.

Table I summarizes the logic for a DASH cluster at a more detailed level. The table gives the percent of logic for each section and the totals in terms of thousands of 2-input gates, kilobytes of static RAM, megabytes of dynamic RAM, and 16-pin IC equivalents. RAM bytes include all error detecting or correcting codes and cache tags. 16-pin IC equivalent is a measure of board area (0.36 sq. inch), assuming through-hole technology (i.e. DIP's and PGA's) was used throughout the

TABLE I
PERCENT OF ALL LOGIC IN A DASH CLUSTER

Section	Gates (thous.)	SRAM (KB)	DRAM (MB)	IC Equiv
Processors/Caches	69.3%	86.0%	0.0%	32.4%
Main Memory	3.3%	0.0%	75.8%	12.7%
IO Board	8.3%	0.0%	1.4%	13.5%
Directory Controller	4.4%	0.0%	12.0%	8.2%
Reply Controller	8.4%	7.3%	0.0%	12.0%
PCPU	0.5%	0.0%	0.0%	1.2%
Network Outbound	2.8%	0.7%	0.0%	4.3%
Network Inbound	1.1%	0.7%	0.0%	2.4%
Performance Mon.	1.8%	5.3%	10.8%	3.5%
Total	535	2420	83	2603

TABLE II
PERCENT OF CORE LOGIC IN A DASH CLUSTER

Section	Gates (thous.)	SRAM (KB)	DRAM (MB)	IC Equiv
Processors/Caches	70.8%	90.8%	0.0%	45.2%
Main Memory	3.9%	0.0%	86.3%	13.9%
IO Board	5.1%	0.0%	0.0%	10.3%
Directory Controller	5.2%	0.0%	13.7%	9.0%
Reply Controller	9.9%	7.7%	0.0%	13.1%
PCPU	0.6%	0.0%	0.0%	1.3%
Network Outbound	3.3%	0.8%	0.0%	4.7%
Network Inbound	1.3%	0.8%	0.0%	2.6%
Performance Mon.	0.0%	0.0%	0.0%	0.0%
Total	456	2292	73	2286

design. (Actually about 1/4 of the CPU logic is implemented in surface mount technology, but the IC Equivalent figures used here assume through-hole since all of the logic could have been designed in surface mount.) The number of 2-input gates is an estimate based on the number of gate-array 2-input gates needed to implement each function. For each type of logic used in the prototype the equivalent gate complexity was calculated as:

Custom VLSI	Estimate based on part documentation.
CMOS Gate Array	Actual gate count or estimate based on master-slice size and complexity.
PAL	Translation of 2-level minimized logic into equivalent gates.
PROM	Espresso minimized PROM files translated into 2-input gates. This includes the primary state machines in the DC and RC, but not the boot EPROM's for the CPU's.
TTL	Gates in equivalent gate array macros.

The numbers in Table I are somewhat distorted by the extra logic in the base Silicon Graphics' hardware and the directory boards that is not needed for normal operations. This includes i) the performance monitor logic on the directory board; ii) the diagnostic UART's and timers attached to each processor; iii) the Ethernet and VME bus interfaces on the Silicon Graphics' I/O board.[2] Table II shows the percentage of this core logic assuming the items mentioned above are removed.

As expected, only when measured in terms of IC equivalents (i.e., board area), is the cost of the directory logic approximately 33%. When measured in terms of logic gates the portion of the cluster dedicated to the directory is 20%, and the SRAM and DRAM portions are 13.9% and 13.7% respectively.

Note that in the above analysis, we do not account for the hardware cost of snooping on the local bus separately because these costs are very small. In particular, the processor's two-level cache structure does not require duplicate snooping tags, and the processor's bus interface accounts for only 3.2% of the gates in a cluster. Even if the second-level cache tags were duplicated, it would represent only 4.0% of a cluster's SRAM. In practice, we expect most future systems will use microprocessors with integrated first-level caches (e.g., MIPS R4000, DEC Alpha, etc.) and to incorporate an external second-level cache (without duplicate tags) to improve uniprocessor performance. Thus, the extra SRAM cost for snooping (e.g., extra state bits) is expected to be negligible.

[2]Each I/O board would still include a SCSI interface for disk interfacing.

Looking at the numbers in Table II in more detail also shows additional areas where the directory overhead might be improved. In particular, the prototype's simple bit vector directory grows in proportion to the number of clusters in the system, and in inverse proportion to cache line size. Thus, increasing cache line size from 16 to 32 or 64 bytes would reduce the directory DRAM overhead to 6.9% and 3.4% respectively, or it could allow the system to grow to 128 or 256 processors with the same 13.7% overhead. For larger systems, a more scalable directory structure [1], [8], [13] could be used to keep the directory overhead at or below the level in the prototype. The directory's overhead in SRAM could also be improved. The 128 KB remote access cache (RAC) is the primary use of SRAM in the directory. The size of the RAC could be significantly reduced if the processor caches were enhanced to be lockup-free, support a dirty-shared state, and track outstanding requests. With enhanced processor caches, the primary use of the RAC would be to collect invalidation acknowledgments, receive granted locks, and merge requests to the same memory line. This would allow a reduction in size by at least a factor of four, and result in an SRAM overhead of less than 2%. Likewise, a closer coupling of the base cluster logic and bus protocol to the inter-cluster protocol might reduce the directory logic overhead by as much as 25%. Thus, the prototype represents a conservative estimate of directory overhead. A more ideal DASH system would have a logic overhead of 18–25%, an SRAM overhead of 2–8% and a DRAM overhead 3–14%. This is still significant,[3] but when amortized over the cluster the overhead is reasonable.

The prototype logic distributions can also be extrapolated to consider other system organizations. For example, if the DASH cluster-based node was replaced by a uniprocessor node, the overhead for directory-based cache-coherence would be higher. Ignoring the potential growth in directory storage (that would need to track individual processor caches instead of clusters), the percent of directory logic in a uniprocessor node would grow to 44% (a 78% overhead). Thus, a system based on uniprocessor nodes would lose almost a factor of two in cost/performance relative to a uniprocessor or a small-scale multiprocessor.

Another possible system organization is one based on a general memory or messaging interconnect, but without support for global hardware cache coherence (e.g., the BBN TC2000 or Intel Touchstone). An optimistic assumption for such a system is that it would remove all of the directory DRAM and support, the RAC and its datapath, and 90% of the RC and DC control pipelines. Under these assumptions, the fraction of logic dedicated to the inter-cluster interface falls to 10% of a cluster, and the memory overhead becomes negligible. Thus, the cost of adding inter-cluster coherence to a large-scale, noncache coherent system is approximately 10%. If more than a 10% performance gain is realized by this addition, then the overall cost/performance of the system will improve. Our measurements on DASH indicate that caching improves performance by far more than 10%, and support for global cache coherence is well worth the extra cost.

[3] Approximately equal to the complexity of an entire processor with its caches.

Finally, by examining the required gates, memory bits and connectivity requirements, one can estimate how the DASH prototype might be integrated into a small number of VLSI chips. As examined in detail in [11], state-of-the-art VLSI technology could be used to integrate the prototype's cluster logic into seven VLSI chips plus memory to form a system with up to 512 clusters (2048 total processors) while maintaining a similar logic and memory overhead to that given in Table II.

V. Performance Monitor

One of the prime motivations for building the DASH prototype was to study real applications with large data sets running on a large ensemble of processors. The alternative, simulation, results in a real-time slow down in the range of 1000–100 000 [6]. Because many of the applications have an execution time measured in tens of minutes on actual hardware, simulation is prohibitively expensive. To enable more insight into these applications when running on the prototype, we have dedicated over 20% of the DC board to a hardware performance monitor. Integration of the performance monitor with the base directory logic allows noninvasive measurements of the complete system without extra test hardware. The performance hardware is controlled by software, and it provides low-level information on software reference characteristics and hardware resource utilizations. The monitor hardware can trace and count a variety of bus, directory and network events. Event selection is controlled by a software programmable Xilinx gate array (FPGA) [17]. This allows flexible event selection and sophisticated event preprocessing.

A block diagram of performance logic is shown in Fig. 4. It consists of three major blocks. First, the FPGA which selects and preprocesses events to be measured and controls the rest of the performance logic. Second, two banks of 16 K $\times$ 32 SRAM's and increment logic that count event occurrences. Third, a 2M $\times$ 36 trace DRAM which captures 36 or 72 bits of information on each bus transaction.

The counting SRAM's together with the FPGA support a wide variety of event counting. The two banks of SRAM are addressed by events selected by the FPGA. They can be used together to trace events that occur on a cycle by cycle basis, or the banks can be used independently to monitor twice as many events. By summing over all addresses with a particular address bit high or low the number of occurrences of that event can be determined. Likewise, the conjunction or disjunction of any set of events can be determined by summing over the appropriate address ranges. Another use of the count SRAM is as a histogram array. In this mode, certain events are used to start, stop and increment a counter inside the FPGA. The stop event also triggers the counter to be used as a SRAM address to increment.

The current use of the counting SRAM in the prototype increments the two banks of SRAM independently on each bus transaction. The address of the first bank is the directory controller's state machine PROM address. The resulting SRAM values give a complete distribution of bus transaction types and

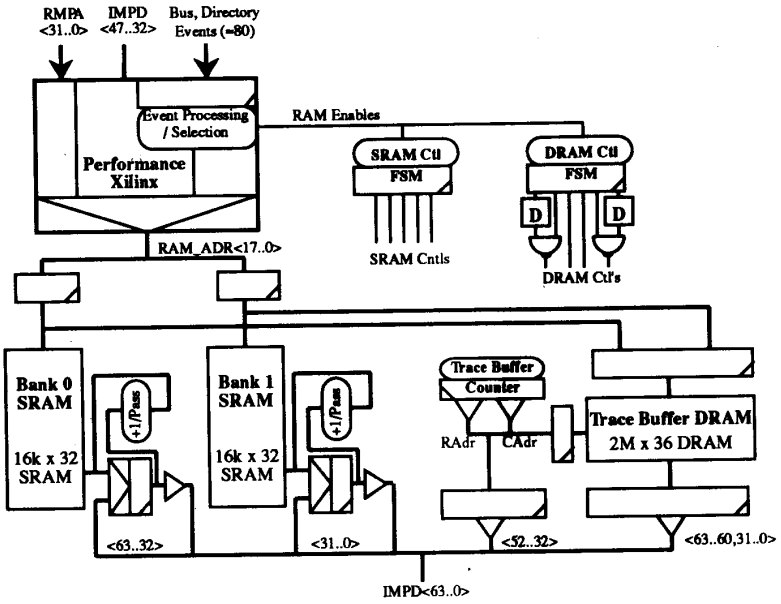


Fig. 4. Block diagram of performance monitor logic.

initiators, directory states, RAC states and local/remote status. From this data access type frequency, bus utilization, access locality, RAC performance, and remote caching statistics can be calculated. Indirectly, the output of the PROM's can be determined from the input address to calculate the frequency of network messages and network utilization. The second bank of SRAM is addressed with the local cache snoop results and histogram counters of remote latency. The snoop data allows one to determine the effectiveness of cache-to-cache sharing within the cluster. The remote latency histogram dedicates an internal counter to each CPU. An individual counter is reset until a processor is forced to retry for a remote access. When the processor is released the counter value is used to index the SRAM and increment a histogram bucket. The result is that a distribution of remote access latencies can be determined. Furthermore, when combined with the count of local bus cycles an estimate of processor utilization can be determined.

The other primary function of the performance monitor is a 2 M × 36 trace array. Again, which information is traced can vary based on programming of the FPGA, but the current use of the trace logic has two modes. In the first configuration, up to 2 M memory addresses together with the issuing processor number and read/write status are captured. The second mode can capture only 1 M addresses, but adds the directory controller's input PROM address and a bus idle count to each trace entry. With software assistance the tracer can be used to capture much longer traces with minimal distortion. The trace information can be used to do detailed analysis of reference behavior or as input to another memory simulator. The only restriction is that only bus references are captured, not references satisfied by the processor cache. If a complete address trace is desired, it can be generated by using only uncached memory spaces. Some distortion of the access pattern may result, however, because the speed of memory will be substantially slower than normal, and will not include any of the effects of cache hits interspersed with cache misses.

VI. Prototype Performance

This section examines the performance of the initial hardware prototype of DASH which includes 48 processors in twelve clusters. The first part summarizes the memory latencies measured on the prototype hardware. The second part describes the speedups obtained by parallel applications run on the actual machine. The third part discusses the effect of optimizations included in the coherence protocol.

A. Processor Issue Bandwidth and Latency

Although the coherent caches in DASH significantly reduce the number of remote accesses made by a processor, it is still essential to minimize the latency when misses do occur. Table III lists the processor bandwidth and latency for cache memory operations in DASH assuming no contention. (PClocks refer to processor clocks which are 30 ns in the prototype.) The delays are average for a 16 cluster DASH configuration and are based on hardware measurements. In the table, the best-case numbers assume stride-one access, with one cache miss every four references (cache lines are 16 bytes). The worst-case numbers assume stride-four accesses with no reuse of cache lines.

Table III presents data separately for reads and writes. For reads, the access latency is given by the last column of the table. The latency can vary by more than two orders of magnitude depending on where a read access is serviced in the memory hierarchy. The read bandwidth also varies considerably, from a high of 133 Mbytes/s from the primary cache to a meager 4 Mbytes/s if all of the data is dirty in a

TABLE III
CACHE OPERATION BANDWIDTH AND LATENCIES

Cache Operation	Best Case		Worst Case	
	MByte / Sec	PClock / Word	MByte / Sec	PClock / Word
Read from 1st-level Cache	133.3	1.0	133.3	1.0
Cache fill from 2nd-level cache	29.6	4.5	8.9	15.0
Cache fill from local bus	16.7	8.0	4.6	29.0
Cache fill from remote bus	5.1	26.0	1.3	101.0
Cache fill from dirty-remote bus	4.0	33.8	1.0	132.0
Write retired in 2nd-level cache	32.0	4.2	32.0	4.2
Write retired on local bus	18.3	7.3	8.0	16.7
Write retired on remote bus	5.3	25.3	1.5	88.7
Write retired on dirty remote bus	4.0	33.0	1.1	119.7

remote nonhome cluster. While, beyond a point, not much can be done about reducing the latency in large-scale machines, the bandwidth can be increased via pipelining, and it is for this reason we have provided nonblocking prefetch operations in DASH. The times given for store operations are the rate at which writes are retired from the write buffer into the second-level cache after acquiring ownership. Release consistency is assumed so that the processor need not wait for the write to retire, and invalidations do not affect write latency.

A more detailed break-down of the latency for local and remote cache misses is given in Fig. 5. The latency for a local miss that is serviced within the cluster is based entirely on the base SGI hardware (i.e., the hardware we have added to the SGI clusters does not slow the system down). In the prototype, a simple remote miss (i.e., a miss that is serviced by a remote home cluster) takes 3.5 times longer than a local miss. The final case illustrated in Fig. 5 represents the latency for fetching a location that is dirty in a cluster other than its home. In this case, an extra 1 μs of delay is incurred in forwarding the request to the dirty cluster. The DASH protocol supports the direct transfer of the dirty data between the dirty and requesting cluster, reducing latency by 20% over a simpler protocol that first causes a writeback to the home cluster and then replies to the requesting processor.

While latencies in the DASH prototype (when measured in microseconds) are far from optimal, we believe that the delays when measured in processor clocks are quite indicative of what we expect to see in future large-scale machines. The reason is that while state-of-the-art technology (with integration and optimization) would allow us to reduce the prototype's latencies by a factor of about three [11], state-of-the-art processor clock rates are also about three times the 33 MHz used in the prototype. As a result, we expect that exploiting cache and memory locality will continue to be important in future large scale machines, as will mechanisms that help hide or tolerate latency.

B. Parallel Application Performance

This subsection outlines the performance actually achieved on the prototype for a number of parallel applications. We begin by describing the software environment available on the prototype and how the measurements were made. We then present the speedup for ten parallel programs representing a variety of application domains. Four of these applications are studied more in-depth using data captured by the performance monitor.

1) Application Runtime Environment: The operating system running on the prototype DASH is a modified version of IRIX; a variant of UNIX System V.3 developed by Silicon Graphics. The applications for which we present results are coded in C and use the Argonne National Labs (ANL) parallel macros [12] to control synchronization and sharing.

Before giving the speedup results in the next subsection, we first state the assumptions used in measuring the speedups. The speedups were measured as the time for the uniprocessor to execute the parallel version of the application code (i.e., not all synchronization code is removed) divided by the time for the parallel application to run on a given number of processors. We timed the entire application for most of the applications. The four exceptions are Cholesky, Radiosity, LocusRoute, and VolRend. We discuss our decision to time only a portion of these applications later in this section.

For our measurements, each application process is attached to a processor for its lifetime, and we fully use one cluster before assigning processors to new clusters. Physical memory pages used by the application are allocated only from the clusters that are actively being used, as long as the physical memory in those clusters is enough. Thus, for an application running with 4 processes, all memory is allocated from the local cluster, and all misses cost about 30 clocks. However, with 8 processes, some misses may be to a remote cluster and cost over 100 clock cycles. Most of the programs allocate shared data randomly or in a round-robin fashion from the clusters being actively used, but some include explicit system calls to control memory allocation.[4] Finally, all applications were run under processor consistency mode, i.e., writes were not retired from the write-buffer until all invalidation acknowledgments had been received.

2) Application Speedups: Fig. 6 gives the speedup for ten parallel applications running on the hardware prototype using from 1 to 48 processors. The applications cover a variety of domains. There are some scientific applications (Barnes-Hut, FMM, and Water), several engineering applications (MP3D, PSIM4, Cholesky, and LocusRoute), two graphics applications (Radiosity and VolRend) and one kernel (matrix multiply).

[4] Currently all operating system code and data are allocated from cluster-0's memory. This causes cluster 0 to become a hot spot for OS misses, and causes some degradation in speedups. We are in the process of fixing this problem.

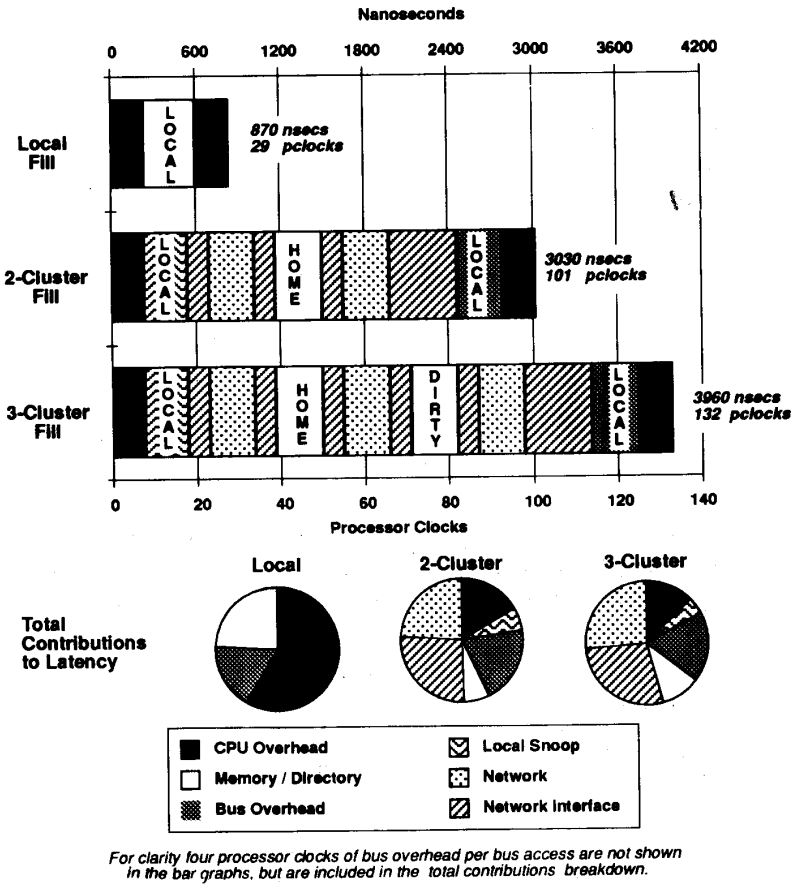


Fig. 5. Cache fill latency in the DASH prototype.

Five of the programs (Barnes-Hut, Water, LocusRoute, MP3D and Cholesky) are applications taken from the SPLASH parallel benchmark suite [15]. The characteristics of the applications give some insight into why they achieve good or bad speedups. We begin with a quick overview of all ten applications and then go into detail on the performance for four of them.

Starting with the applications with the best speedup, Matmult is a blocked matrix multiply, which uses 88 x 88 blocks for a roughly 1000x1000 matrix. The size of the matrix is slightly modified with the number of processors to ensure an integral number of blocks per processor. Matmult gets almost perfect speedup and the overall performance with 48 processors is 363 double-precision MFLOPS. The second application is VolRend from the domain of computer graphics. It performs volume rendering for a 256x256x113 computer tomography dataset. Rendering is performed by ray tracing, casting one ray per pixel. An octree combined with early ray termination is used to reduce rendering time. VolRend is intended to be an interactive application. There is a fixed, large initialization overhead, which can be amortized over an arbitrary number of frames. Since the intention of VolRend is to allow real-time rendering and viewing of an image, we time a single frame in its entirety (including rendering and copying the image to the frame buffer). VolRend achieved a speedup of over 45 on 48 processors for this frame. With 48 processors, DASH renders this image at approximately 2 frames per second.

The next application is the FMM code [16], which represents an n-body galactic simulation solved using the fast multipole method. The input consists of two Plummer-model clustered galaxies with 16 384 bodies each. The structure of the application and its data is complex, however, we see that caches work quite well and we get a speedup of over 40 with 48 processors. Immediately below is the Water code (the parallelized version of the MDG code from the Perfect Club benchmarks), a molecular dynamics code. We run it using 1728 water molecules and 45 time steps. We will discuss this application in detail later in this section.

The next application, Barnes-Hut [16] represents an n-body galactic simulation solved using the Barnes-Hut algorithm (an $O(N \log N)$ algorithm). The input consists of the same two Plummer-model clustered galaxies used for FMM, and again we see that although the structure of the program is complex, good speedups are obtained. We will discuss Barnes-Hut in detail later in this section. Below it is Radiosity [16], from the domain of computer graphics. It computes the

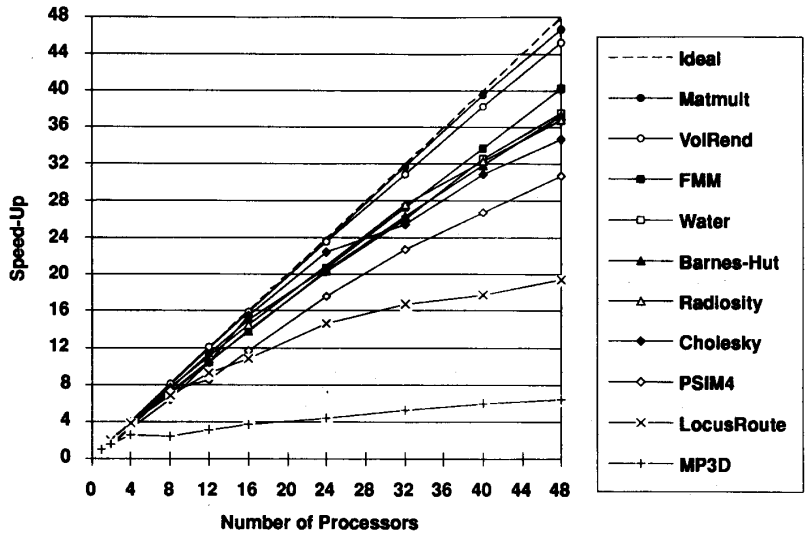


Fig. 6. Speedup of parallel applications run on the DASH prototype.

global illumination in a room given a set of surface patches and light sources using a variation of hierarchical n-body techniques. The particular problem instance solved starts with 364 surface patches, and ends with over 10 000 patches. Radiosity has a serial initialization portion consisting primarily of generating a large array of random numbers. We do not time the initialization, as it could be parallelized, but has not been in the current version. While the data structures used in Radiosity are a complex octree and a binary-space partition tree, the application achieves a speedup of 37 on 48 processors.

The next application is Cholesky. It performs factorization of sparse positive-definite matrices using supernodal techniques (data blocking techniques that enhance the performance of caches both for uniprocessors and multiprocessors). Here it is used to solve the BCSSTK33 problem from the Boeing-Harwell benchmark set, which contains 2.54 million nonzero matrix elements. The Cholesky numbers give speedups for the most important and most time-consuming phase of the computation, the numerical factorization. Runtimes for pre- and post-processing phases are not included because these phases have not yet been parallelized. Each phase is quite involved and would require a significant amount of parallelization effort. Parallelization of these phases is important and appears possible, but the required effort has simply not yet been expended. We note that much of the fall off in speedup for Cholesky is due to the trade-off between large data block sizes (which increase processor efficiency, but decrease available concurrency and cause load balancing problems) and small data block sizes. As we go to a large number of processors, we are forced to use smaller block sizes unless the problem size is scaled to unreasonably large sizes. With 48 processors, Cholesky achieves 190 MFLOPS. We feel that this is very good given the relatively low floating-point performance of the MIPS R3000 processors.

Next is PSIM4, an application from NASA, that is a particle-based simulator of a wind tunnel. PSIM4 is an enhanced version of the MP3D code, both in functionality and locality of memory accesses. The 48 processor run is done with over 300 000 particles and it achieves a speedup of over 31. In contrast, the older MP3D code achieves a speedup of under 7 with 48 processors simulating 40 000 particles. We discuss both PSIM4 and MP3D in detail later in this section. Finally, the LocusRoute application does global routing of standard cells using routed area to evaluate the quality of a given placement. The inner loop of LocusRoute is the routing of a given cell placement. LocusRoute exploits parallelism at two levels. First, multiple wires are routed simultaneously. Second, different routes for the same wire are evaluated in parallel. LocusRoute was run over a circuit consisting of 3817 wires and 20 routing channels. We ignore the serial initialization portion of LocusRoute, timing only the parallel routing. The serial initialization in LocusRoute consists of two portions: the reading of the design input file, and the data structure initialization. LocusRoute is intended to be the routing phase of a CAD tool, and the design would already be resident in memory. In addition, the data structure initialization could be parallelized, but this has not been done yet.

Overall, we see that most applications achieve good speedups, even though they have not been specially optimized for DASH. More than half the applications get over 36 times improvement on 48 processors and two of the applications get near-linear improvement.

3) Detailed Case Studies: To get a better understanding of the detailed reference behavior of some of the applications, we now examine the Water, Barnes-Hut, MP3D, and PSIM4 applications in more detail. These applications were chosen because they represent programs that achieved a range of good and bad speedups. Results are presented for only the parallel portions of the applications (with the exception of the speedup numbers). Statistics were ignored during the sequential initialization portion to avoid lowering the bus and network utilization.

TABLE IV
WATER MEMORY ACCESS CHARACTERISTICS

Execution Attribute	1 PE	2 PEs	4 PEs	8 PEs.	12 PEs	16 PEs	24 PEs	32 PEs	40 PEs	48 PEs
Speedup	1.00	1.98	3.85	7.13	10.42	13.95	20.27	26.05	32.58	37.57
Busy Pclks between Proc. Stalls	686.6	620.1	486.2	425.2	432.2	453.3	412.8	440.8	442.0	428.8
Est. Processor Utilization (%)	96.5	96.2	95.8	91.5	90.9	89.6	88.2	87.3	87.3	86.9
Cache Read (%)	60.9	67.9	62.6	60.7	58.0	54.7	58.4	56.2	58.1	58.9
Cache Read Exclusive (%)	3.9	2.8	15.6	20.9	23.9	25.3	23.8	24.9	24.0	23.5
Cache Lock (%)	17.6	14.6	10.8	9.2	9.1	10.0	9.0	9.5	9.0	8.9
Cache Unlock (%)	17.4	14.6	10.8	9.2	9.0	10.0	8.9	9.4	8.9	8.7
References to Local Memory (%)	99.9	99.6	99.9	56.0	37.1	29.0	20.5	16.5	12.6	11.3
Local Ref's satisfied Locally (%)	98.3	99.4	99.8	93.8	90.3	84.1	83.5	80.9	80.0	77.9
Remote Ref's satisfied Locally (%)	0.0	0.0	0.0	49.4	55.7	48.0	50.1	43.4	46.0	47.2
Remote Ref's satisfied in Home (%)	100.0	100.0	100.0	96.8	97.3	94.2	93.5	89.4	89.3	89.7
Measured Local Cache Fill (Pclks)	29.2	29.2	29.5	29.6	29.7	29.7	29.8	29.8	29.8	29.8
Unloaded Remote Cache Fill (Pclks)	101.0	101.0	101.0	103.3	104.9	109.5	111.5	116.5	117.4	116.9
Measured Remote Cache Fill (Pclks)	112.8	109.9	112.0	107.5	111.3	116.1	119.6	125.5	127.8	128.3
Bus Utilization (%)	4.9	6.9	11.9	16.1	16.7	17.7	19.1	19.8	19.6	19.8
Req. Net Bisection Util. (%)	0.6	0.8	0.8	1.6	1.9	4.6	5.4	6.4	6.5	6.5
Reply Net Bisection Util. (%)	0.5	0.7	0.7	2.0	2.2	5.2	5.8	6.4	6.3	6.4

a) Water: Water is a molecular dynamics code from the field of computational chemistry. The application computes the interaction between a set of water molecules over a series of time steps. For the problem size that we consider here, 1728 molecules, the algorithm is essentially $O(N^2)$ in that each molecule interacts with all other molecules in the system. As shown in Fig. 6, the Water application achieves good speedup on the DASH hardware.

Table IV gives a detailed memory reference profile of Water running on DASH as measured by the hardware performance monitor.[5] Table IV is broken into five sections: i) overall performance and processor utilization; ii) memory request distribution; iii) request locality; iv) memory latency characteristics and v) bus and network utilization.

The first section of the table gives the overall speedup and estimated processor utilization. Unfortunately, processor utilization cannot be measured directly from the bus, so what is given in the table assumes that the processor is doing active work whenever it is not waiting for a bus transaction to complete. This ignores stalls due to first-level cache misses satisfied by the second-level, TLB miss handling and floating-point interlocks.[6] Processor utilization (third row of table) and busy clocks between stalls (second row) give an indication of the level of cache locality in the application and its sensitivity to memory latency.

For Water, cache-locality is very high and the time between processor stalls indicates that Water is not highly sensitive to memory latency. The table shows a reduction in the busy clocks between stalls as the number of processors increases from 2 to 4 processors, but this levels off afterwards. The application computes the forces on molecules one at a time, and incurs misses only between force computations. The amount of work done for each force computation is essentially independent of the number of processors, which explains why the number of busy clocks between stalls levels off.

The second section of Table IV gives a breakdown on the memory reference types. This breakdown indicates the type of accesses that cause bus transactions and whether synchronization references are significant. In Water, the percentage of synchronization references is fairly high. This is due in part to the high cache hit rates, and to the fact that every successful lock acquire and release reference the bus in the prototype. Given the percentage of locks and unlocks are very close, this data also indicates that lock contention is not a severe problem in Water.

The third section of Table IV gives a breakdown of cache fill locality. The first line indicates the fraction of all cache misses which refer to local memory (based on physical address). The next line shows the percentage of these local references which are satisfied locally. Note that a local memory reference may have to go to a remote cluster if the data is dirty in that cluster. The third line shows the percentage of remote references (based on physical address) that are satisfied locally. A remote reference may be satisfied locally if another processor's cache contains that data (using a cache-to-cache transfer) or if the RAC contains that data. The final line in this section shows the fraction of remote references (out of the remote references which are not satisfied locally) that are satisfied by the home cluster, that is, the home does not forward the request to another cluster where the data are dirty. As can be seen from Table IV, the Water application does not use any specific memory placement strategy—the fraction of references to local memory fall off in inverse proportion to the number of clusters. However, nearly 50% of remote references are satisfied locally for all the multiple-cluster runs. This good cluster locality agrees with simulations that indicate that Water should achieve over 50 times speedup on a 64 processor DASH [10].

The fourth section of Table IV shows the latencies for remote and local cache fills. Both unloaded and loaded latencies are given for remote fills. The latency figures are based on the

[5]The results given in the table are averaged over all active clusters. This implies some inaccuracies for the one and two processor runs due to the overhead of the idle processors running the UNIX scheduler and daemons (e.g., slightly lower processor utilization for 1 and 2 processors than for 4 processors).

[6]Since writes are buffered, they are not assumed to stall the processor directly. Instead, it is assumed that the processor can execute for 20 clocks before stalling. This delay is an estimate of the time for the processor to fill the other 3 words to the write-buffer (i.e. assuming 15% instructions are writes). In reality, the processor may not issue writes at this rate, or may stall earlier due to a first-level cache miss.

TABLE V
BARNES-HUT MEMORY ACCESS CHARACTERISTICS

Execution Attribute	1 PE	2 PEs	4 PEs	8 PEs.	12 PEs	16 PEs	24 PEs	32 PEs	40 PEs	48 PEs
Speedup	1.00	1.60	2.48	2.33	3.12	3.74	4.42	5.27	5.99	6.42
Busy Pclks between Proc. Stalls	82.5	46.1	43.6	44.2	38.6	49.7	56.0	58.9	61.5	84.5
Est. Processor Utilization (%)	73.8	67.5	62.0	34.6	24.8	27.2	26.5	26.2	26.4	32.1
Cache Read (%)	98.3	72.9	67.5	57.7	56.9	54.0	51.8	52.2	53.4	51.6
Cache Read Exclusive (%)	0.9	26.9	32.2	42.0	42.7	45.5	47.3	46.6	45.2	46.3
Cache Lock (%)	0.3	0.1	0.0	0.0	0.2	0.3	0.5	0.8	1.0	1.5
Cache Unlock (%)	0.3	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
References to Local Memory (%)	100.0	100.0	100.0	46.5	29.3	23.0	16.0	11.6	8.9	8.4
Local Ref's satisfied Locally (%)	99.8	99.9	100.0	79.3	74.7	67.9	62.3	63.3	66.1	59.3
Remote Ref's satisfied Locally (%)	0.0	0.0	0.0	30.8	18.9	17.1	12.9	10.0	8.1	10.4
Remote Ref's satisfied in Home (%)	100.0	100.0	100.0	99.3	89.0	80.5	71.6	71.0	73.3	63.1
Measured Local Cache Fill (Pclks)	29.4	30.8	37.0	37.2	38.1	36.6	36.0	35.7	35.2	34.1
Unloaded Remote Cache Fill (Pclks)	101.0	101.0	101.0	101.3	108.6	114.7	120.3	119.8	117.5	124.6
Measured Remote Cache Fill (Pclks)	109.5	117.5	130.4	138.7	160.9	172.2	186.8	191.3	192.7	201.6
Bus Utilization (%)	16.6	42.9	78.6	77.3	78.6	73.6	71.0	69.6	67.6	62.2
Req. Net Bisection Util. (%)	0.5	0.6	0.9	10.7	15.0	30.6	32.5	33.1	32.5	30.6
Reply Net Bisection Util. (%)	0.4	0.5	0.9	13.7	15.6	30.1	29.6	29.3	28.7	26.3

weighted average of remote and dirty-remote references. The remote reference latency figures indicate that Water does not heavily load the system; there is very little queueing. For 48 processors, less than 12 extra cycles are added due to queueing delays.

The final section of the table also shows that the memory system is not heavily loaded. With the low reference rate, both the bus and the request and reply networks have a low utilization. Bus utilization is measured directly by the performance monitor, while the network bisection utilizations are estimates assuming uniform network traffic. The bisection utilization is calculated by knowing the total number of network messages sent, assuming the half of the messages cross the midline of the mesh, and dividing by the bandwidth provided across the bisection. For Water, the bisection load is negligible.

b) Barnes-Hut: Barnes-Hut is a galactic N-body simulation solved using the hierarchical Barnes-Hut algorithm. This algorithm reduces the complexity of force computation among N bodies from $O(N^2)$ to $O(N \log N)$ by approximating groups of distant bodies by equivalent single points when possible. To do this, it recursively subdivides the computational domain to obtain a tree structured representation. The force on each body is computed by a partial depth-first traversal of the tree data structure which terminates at cells that are considered far enough away from the body. As a result, a body computes interactions directly with other bodies that are close to it, and with larger and larger cells as they get further away from it. The input for the runs in Table V consists of two Plummer-model clustered galaxies with 16 384 bodies each.

Barnes-Hut achieves a respectable 37.2 speedup on 48 processors. The processor utilization remains essentially constant as more processors are added up to 48 processors. Excellent cache locality is achieved despite the nonuniform and dynamically changing problem domain, because the nature of the tree traversals ensures that most of the interactions computed are between bodies in the same processor's partition.

c) MP3D: The MP3D application is a particle-based wind-tunnel simulator developed at NASA-Ames. In each time step, particles are moved according to their velocity vectors and collisions are modeled. The results reported in Table VI correspond to a run with 40 000 particles and 300 time steps. The speedups are poor for MP3D because of frequent sharing of data that is actively being updated. While the particles are statically allocated to processors, the space cells (representing physical space in the wind tunnel) are referenced in a relatively random manner depending on the location of the particle being moved. Since each move operation also updates the corresponding space cell, as the number of processors increases, it becomes more and more likely that the space cell being referenced will be in remote-dirty state. Thus, as shown in Table VI cache hit rates fall (see busy clocks between processor stalls) and the average miss latencies go up.

The high miss rates together with low locality are the primary cause for the performance decrease that occurs when going from 4 to 8 processors. When all of the processors are within the same cluster, the miss latencies are 30 processor clocks. As soon as we go to two clusters (8 processors), there is a 50% chance that a space cell miss will be handled by a remote cluster, which takes over 100 clock cycles. Since the miss rates are high in MP3D, these larger miss latencies nullify the benefits of the extra processors. (As can be seen from Table VI, when we go to two clusters, the fraction of references to local memory goes down from 100% to 46%, and even the fraction of local references that are satisfied locally goes down from 100% to 79%.) Speedup when going from 8 to 12 processors is also poor because there is now a good possibility that a space cell cache miss will be satisfied by a dirty-remote cluster (132 versus 100 clock unloaded latency). Table VI shows this is occurring, with 10% of remote references for the 12 processor run being satisfied by a dirty-remote cluster. Thus, processor utilization falls further when going from 8 to 12 processors, and speedup is less than ideal. Interestingly enough, however, the busy time between processor stalls actually increases as more processors are added. This is not because the application is performing better, but instead because of load balancing problems. The processors spend more time spinning in their cache, waiting on barriers. This barrier time is counted as processor busy time by the performance monitor.

TABLE VI
MP3D MEMORY ACCESS CHARACTERISTICS

Execution Attribute	1 PE	2 PEs	4 PEs	8 PEs.	12 PEs	16 PEs	24 PEs	32 PEs	40 PEs	48 PEs
Speedup	1.00	1.96	3.81	7.29	10.59	13.82	20.42	26.33	31.84	37.24
Busy Pclks between Proc. Stalls	528.9	519.8	514.1	468.8	528.4	529.0	510.4	545.4	562.5	638.9
Est. Processor Utilization (%)	94.7	94.6	94.7	92.9	92.8	92.6	92.0	92.2	92.4	93.1
Cache Read (%)	91.8	93.8	95.3	94.3	93.6	93.4	93.6	92.7	92.5	91.7
Cache Read Exclusive (%)	4.2	3.9	3.3	4.4	5.0	5.1	4.8	5.4	5.5	6.0
Cache Lock (%)	2.0	1.1	0.7	0.7	0.8	0.8	0.9	1.1	1.2	1.4
Cache Unlock (%)	1.9	1.1	0.7	0.6	0.7	0.7	0.5	0.6	0.6	0.7
References to Local Memory (%)	99.1	99.5	99.5	50.5	33.6	25.7	17.6	12.7	10.6	8.7
Local Ref's satisfied Locally (%)	98.7	99.4	99.8	99.6	99.4	99.1	99.2	98.8	98.8	98.5
Remote Ref's satisfied Locally (%)	0.0	0.0	0.0	80.1	76.2	77.4	77.6	77.7	77.3	76.8
Remote Ref's satisfied in Home (%)	100.0	100.0	100.0	97.0	98.1	98.6	99.0	99.1	99.1	98.9
Measured Local Cache Fill (Pclks)	29.2	29.2	29.4	29.4	29.4	29.4	29.4	29.4	29.4	29.4
Unloaded Remote Cache Fill (Pclks)	101.0	101.0	101.0	103.9	102.6	102.4	102.1	102.2	102.2	102.5
Measured Remote Cache Fill (Pclks)	111.8	110.3	111.3	107.5	109.8	113.1	114.7	118.0	118.8	122.2
Bus Utilization (%)	5.2	6.9	10.1	11.5	11.1	11.2	11.5	11.1	11.0	10.2
Req. Net Bisection Util. (%)	0.6	0.6	0.6	0.8	0.8	1.7	1.7	1.8	1.8	1.7
Reply Net Bisection Util. (%)	0.5	0.5	0.5	0.9	1.1	2.2	2.4	2.3	2.3	2.2

TABLE VII
PSIM4 MEMORY ACCESS CHARACTERISTICS

Execution Attribute	1 PE	2 PEs	4 PEs	8 PEs.	12 PEs	16 PEs	24 PEs	32 PEs	40 PEs	48 PEs
Speedup	1.00	1.53	3.47	7.42	8.54	11.66	17.61	22.77	26.79	30.76
Busy Pclks between Proc. Stalls	166.5	180.4	191.9	217.5	252.2	281.9	315.8	397.2	413.7	432.1
Est. Processor Utilization (%)	80.6	82.9	82.0	88.8	84.5	86.9	88.9	91.1	91.5	92.1
Cache Read (%)	78.3	79.5	79.2	82.1	79.6	79.8	80.0	80.0	79.8	79.0
Cache Read Exclusive (%)	18.3	17.7	18.3	15.3	17.5	17.2	16.6	16.1	16.0	16.3
Cache Lock (%)	1.7	1.4	1.2	1.3	1.6	1.6	1.8	2.1	2.4	2.7
Cache Unlock (%)	1.7	1.4	1.2	1.3	1.3	1.4	1.5	1.7	1.8	2.0
References to Local Memory (%)	78.0	82.2	77.2	92.5	66.1	70.7	70.8	69.2	67.9	69.2
Local Ref's satisfied Locally (%)	99.7	99.8	99.9	99.9	99.8	99.9	99.9	99.9	99.8	99.9
Remote Ref's satisfied Locally (%)	1.0	1.5	0.8	48.0	24.4	24.7	37.2	44.6	49.2	51.3
Remote Ref's satisfied in Home (%)	100.0	100.0	100.0	100.0	98.0	97.8	98.6	98.1	98.0	98.2
Measured Local Cache Fill (Pclks)	29.2	29.4	30.0	29.9	30.1	29.9	29.8	29.6	29.6	29.6
Unloaded Remote Cache Fill (Pclks)	101.0	101.0	101.0	101.0	101.8	101.9	101.8	102.5	102.9	102.9
Measured Remote Cache Fill (Pclks)	104.3	105.1	107.6	109.4	114.5	114.1	116.4	120.0	121.4	124.3
Bus Utilization (%)	8.6	13.8	24.4	21.7	24.7	22.0	19.2	15.8	15.2	14.5
Req. Net Bisection Util. (%)	0.1	0.2	0.4	0.4	2.6	4.1	3.0	2.2	2.1	1.9
Reply Net Bisection Util. (%)	0.0	0.0	0.0	0.5	3.0	4.6	3.2	2.4	2.2	2.0

Looking at Table VI, it is also clear that MP3D puts a heavy load on the memory system. The load on the cluster bus is very high (up to 79%) when multiple clusters are used. The load on the network is not nearly as high (up to 33%). This heavy memory system load results in large queueing delays for remote references, increasing the latency by up to 63% for the 48 processor run (while the unloaded remote reference latency should have been 125 cycles, the measured remote latency was 202 cycles). The utilization data also confirms calculations, based on total bandwidth and uniform loading, that the cluster bus in the prototype limits memory system bandwidth more than the network. The bus utilization also indicates that, in the prototype, latency hiding techniques such as prefetching will not help applications like MP3D. A better solution is to restructure the application to achieve better locality of the space cells by not strictly tying particles to processors. This restructured MP3D is called PSIM4, and from Table VII we can see that this time the speedups are significantly better (more than 30-fold with 48 processors). PSIM4 provides a good example of how a poorly performing application can be restructured to achieve much better performance on DASH, and we discuss it next.

d) PSIM4: PSIM4 is an enhanced version of the MP3D code, both in terms of functionality and in terms of locality of memory accesses. The enhanced functionality includes modeling multiple types of gases and including molecular chemistry. To improve locality, PSIM4 uses spatial decomposition of simulated space to distribute work among processors. Thus, both particles and the space cells in a large spatial chunk are assigned to the same processor and the data allocated from its corresponding local memory.

While the results of PSIM4 are not directly comparable with MP3D (PSIM4 models the wind tunnel much more accurately and is solving a more realistic problem than MP3D), we can see from Table VII that the spatial decomposition of PSIM4 works quite well. The percentage of references to local memory does fall from 8 to 12 processors, but levels off at approximately 70% from 12 to 48 processors. In addition to this improved memory locality, PSIM4 is busy much longer between processor stalls than MP3D, mainly due to improved cache usage (some gains occur from performing more computation per particle to model the different molecule types and chemistry). The improved cache and memory locality results in very good speedups. The 48 processor run is done with over 300 000 particles running for 600 time steps and achieves a speedup of nearly 31, in contrast to the speedup of 6.4 for 48 processors for the old MP3D code.

4) Application Speedup Summary: Overall, a number of conclusions can be drawn from the speedup and reference statistics presented in previous sections. First, it is possible to get near linear speedup on DASH with a number of real applications. Applications with the best speedup have good cache locality, but even those with moderate miss rates (e.g., PSIM4) can achieve reasonable speedup.

In absolute terms, the number of busy clocks between bus accesses indicates that caching of shared data improves performance significantly. For example, earlier simulation work [10] with the Water application indicated that the reference rate to shared data is roughly one reference every 25 instructions. Even assuming that the number of processor instructions between stalls given in Table IV are optimistic by a factor of two, caches are satisfying 88% of the shared references in Water (i.e., there are at least 428/2/25 or 8.6 shared references for every miss). Thus, processor utilization without caching would be 26%[7]. Overall, this implies that caching of shared data improves performance by a factor of three, but as shown earlier only adds 10% to system's hardware cost.[8]

Even with caches, however, locality is still important. As shown by applications like MP3D, if locality is very low and the communication misses are high; speedup will be poor. However, for many applications the natural locality is enough (e.g., FMM, Barnes-Hut, VolRend, Water) that very good speedups can be achieved without algorithmic or programming contortions. Even in applications where natural locality is limited, DASH allows the programmer to focus on the few critical data structures that are causing loss in performance, rather than having to explicitly manage (replicate and place) all data objects in the program.

Looking at the system resource usage, the applications confirm that bus bandwidth is the primary limitation to overall memory system bandwidth in the prototype. Even without latency hiding techniques, MP3D was able to drive the buses to over 70% utilization. Oppositely, the network bisection had a much smaller loading (2–3 times less) and does not appear to be a major contributor to queueing. Finally, the loading on the request and reply networks appears to be well balanced.

C. Protocol Effectiveness

This section examines in detail how the protocol features and alternative memory operations provided in DASH improve system performance. Features of the base protocol optimizations are examined in the context of the parallel applications, while the alternative memory operations are studied with stand-alone atomic tests. The atomic tests allow the alternative operations to be studied in a more controlled environment where their effects can be easily decoupled from an application's ability to utilize the operation.

1) Base Protocol Features: While there are many facets of the base coherency protocol that could be examined in detail, this section focuses on three of the most important and novel aspects of the DASH protocol: request forwarding, cache-to-cache sharing, and the use of the remote access cache. The protocol optimizations are examined on the four parallel applications that we have made performance monitor measurements on: Water, Barnes-Hut, MP3D, and PSIM4. The results in this section were derived from the reference statistics captured using the hardware performance monitor together with the measured latency of operations given in Section VI-A.

a) Request Forwarding: Request forwarding in DASH refers to the protocol's ability to forward remote requests from the home cluster to a third cluster that currently caches the requested memory block, and for the third cluster to respond directly to the cluster that originated the request. Two important uses of forwarding in DASH are the forwarding of memory requests to a dirty-remote cluster, and the forwarding of invalidations to clusters sharing a memory block. In both of these cases, the incoming memory request generates new outgoing request(s), and the destination cluster(s) reply directly to the cluster that originated the request. The primary advantage of forwarding is that it reduces the latency of these requests (usually by one cluster/network transaction, which is approximately 1 μs without contention). Fowarding also reduces hardware complexity by eliminating the need to buffer information about pending requests in the home. The home directory controller reacts to a request based on the current state of the directory, and when necessary, forwards responsibility for the request to the remote cluster(s) that should be able to complete it. The disadvantage of forwarding is that it requires a mechanism to avoid network deadlock[9] and complicates recovery from network errors. Forwarding can also lead to performance loss when requests are forwarded to dirty clusters that cannot satisfy the requests due to pending operations or changes in block ownership (the retry penalty is approximately 3 μs).

Table VIII summarizes the effectiveness of forwarding cache reads and read-exclusives to dirty-remote clusters for the benchmark applications using 48 processors. The improvement if the forward is successful is assumed to be 1 μs, while a retry has a cost of 3 μs. With these assumptions, Water and MP3D experience an improvement of approximately 11% in read fill times. Barnes-Hut and PSIM4 rarely fetch data from a dirty-remote cluster so neither of these see a large improvement from forwarding. Read-exclusive requests rarely need forwarding to a remote dirty cluster, and thus they see little benefit (two clusters writing a cache-line before reading it is usually the result of false sharing).

Table IX details the effect of forwarding write invalidation requests to shared-remote clusters. When release consistency is used, this type of forwarding is not as critical as dirty-remote forwarding because the receipt of invalidation acknowledgments can be overlapped with processor execution. When processor consistency is used, however, a read-exclusive request is not satisfied until all invalidations are received. Thus, in this case invalidation forwarding saves at least 1 μs

[7]This assumes the locality of shared references is the same as the locality seen for cache misses.

[8]The added costs for caching is conservative because we ignore any added costs in the uncached system that would be necessary to increase the bandwidth to main memory.

[9]Forwarding adds a dependency to the acceptance of incoming requests on the ability of the node to send outgoing requests. Without provisions to break this dependence, there could be deadlock due to buffer overflow of the PCPU's input FIFO that blocks a node's outgoing requests.

TABLE VIII
EFFECTIVENESS OF DIRTY-REMOTE FORWARDING

	Water	Barnes-Hut	PSIM4	MP3D
% Remote Reads, Dirty-Rem	51.2%	4.9%	6.0%	76.2%
Success Rate	100.0%	93.1%	99.7%	98.7%
Avg. Read Fill Improvement	**11.7%**	**1.0%**	**1.6%**	**10.7%**
% Remote Read-Ex's, Dirty-Rem	1.1%	5.3%	3.4%	5.1%
Success Rate	99.1%	99.4%	92.7%	82.6%
Avg. Read-Ex. Improvement	**0.3%**	**1.3%**	**0.7%**	**0.2%**

TABLE IX
EFFECTIVENESS OF INVALIDATION FORWARDING

	Water	Barnes-Hut	PSIM4	MP3D
% Read-Exclusive to Shared	76.4%	15.5%	1.5%	84.2%
Read-Ex Fill Improvement	**16.6%**	**3.9%**	**0.4%**	**11.8%**

TABLE X
EFFECTIVENESS OF LOCAL CACHE TO CACHE SHARING

	Water	Barnes-Hut	PSIM4	MP3D
Rem. Reads satisfied locally	24.9%	46.0%	11.6%	10.5%
Read savings from Snoop	14.4%	24.9%	5.7%	6.5%
Rem Read-Ex Satisfied Locally	0.04%	0.66%	1.41%	0.70%
Read-Ex savings from Snoop	-3.2%	-2.7%	-2.1%	-1.3%
Combined Read and Read-Ex	**9.9%**	**23.3%**	**4.3%**	**2.9%**

per write to a shared-remote location (again, one network and cluster hop).[10] Forwarding of invalidations has a significant effect in Water and MP3D (> 12%), but is unimportant for Barnes-Hut and PSIM4 where writes to shared variables are infrequent.

b) Local Cache to Cache Sharing: As illustrated in Fig. 5, all remote accesses are delayed in the originating cluster until the results of the local snoop are known. Performing this local snoop has the advantage that some remote cache and cache-lock requests are satisfied by a local cache-to-cache transfer and do not incur the latency of fetching the data from a remote home cluster. Of course, it has the disadvantage of increasing latency[11] for remote accesses not satisfied by a local transfer. Table X summarizes the measured gains from the local snoop on the 48 processor benchmark runs in terms of the percent improvement of remote memory latency. As with forwarding to dirty-remote clusters, reads benefit more than writes from the local sharing. The local snoop actually results in a performance loss for read-exclusive requests, but the overall effect is a decrease in remote latency because of local cache-to-cache sharing for read requests. This performance gain, together with the hardware simplifications that result from making all the network request generation timing the same (in local, home and dirty-remote clusters), imply that local cache sharing was well worth supporting in the prototype.

c) Remote Access Cache: Another design feature in DASH is the implementation of the reply controller's pending request buffer as a direct-map snoopy cache (RAC). While the RAC structure serves many purposes (e.g., enforcement of release consistency), it has two direct performance benefits. First, by serving as an additional cluster cache, the RAC supports cache-to-cache sharing and a cluster-level dirty-sharing cache state. Second, by merging outstanding requests, the RAC reduces the average latency for remote locations, and it can reduce hot-spot traffic. The disadvantage of the RAC arises from its cache structure which can cause conflicts for individual RAC entries. In these cases, the outstanding request must be satisfied before the new request can be initiated.

Table XI provides a summary of the performance impact of the RAC. The effectiveness of the RAC as a cluster cache is somewhat overstated in the table because for all requests that could be satisfied by either a local processor cache or the RAC, the RAC is given the credit (due to limits in the current performance monitor). This implies that the improvements from the cache sharing listed in Table X are understated, and that the savings given in the two tables can be added. The combined savings from the two types of cluster caching are significant. Cluster caches reduce the effective remote miss penalty by 6% in MP3D, by more than 20% in Water and PSIM4, and over 30% in Barnes-Hut. Merging of outstanding remote requests by the RAC has a much smaller impact on performance. While a number of requests are merged, they do not significantly reduce the fill time of the merged requests because latencies are not dramatically reduced. Thus, any benefits from merging are made indirectly through a reduction in hot-spot traffic.

2) Alternative Memory Operations: This section investigates the performance potential of the alternative memory operations such as prefetch, granting locks, Fetch&Op and update writes. While it would be desirable to present the effects of these operations on a variety of real benchmarks, such

[10] The data in Table IX also estimates the gain from release consistency over processor consistency in the prototype since the gain from release consistency is to allow a write to retire as soon as ownership is returned, even if invalidations are pending. The savings from release-consistency 1 μs as in forwarding, but this is compared with a nominal delay of 3 μs instead of 4 μs in the case of invalidation forwarding (implying gains are approximately 1/3 larger).

[11] The local cache snoop adds 2–4 processor clocks in the prototype.

TABLE XI
EFFECTIVENESS OF THE RAC

	Water	Barnes-Hut	PSIM4	MP3D
Rem Reads satisfied by RAC	32.6%	35.9%	51.9%	4.3%
RAC Read conflicts	0.00%	0.12%	0.32%	0.05%
Avg Read conflict penalty (Pclk)	45.2	21.4	23.1	51.9
Read savings from RAC	20.0%	21.4%	28.3%	3.4%
Rem Read-Ex satisfied by RAC	35.1%	1.0%	10.5%	4.7%
RAC Read-Ex conflicts	0.51%	0.11%	0.00%	0.01%
Avg Read-Ex cnflct. penalty (Pclk)	18.8	18.8	0.0	56.5
Read-Ex savings from RAC	23.2%	0.8%	8.2%	4.1%
Savings from RAC caching	**20.8%**	**20.3%**	**24.8%**	**3.7%**
Rem Reads merged by RAC	0.0%	0.1%	0.2%	0.1%
Avg Read merge fill time (Pclk)	85.2	80.5	86.1	89.1
Read savings from merging	0.0%	0.0%	0.1%	0.1%
Rem Read-Ex merged by RAC	0.3%	1.2%	0.9%	0.1%
Avg Read-Ex merge fill time (Pclk)	45.2	18.8	23.1	51.9
Read-Ex savings from merging	0.1%	0.2%	0.2%	0.0%
Savings from RAC merging	**0.0%**	**0.0%**	**0.1%**	**0.0%**
Total RAC/Merge Savings	**20.9%**	**20.3%**	**24.9%**	**3.8%**

optimized benchmarks are not readily available. Furthermore, if only these results were given, it would be difficult to distinguish the performance potential of the operations from the particular applications' ability to exploit them. Instead, this section quantifies the effectiveness of the alternative operations using small, atomic tests. These tests demonstrate how the operations would be commonly used and stress the individual operations heavily. In most cases, the results can be considered an upper bound on performance gain when using the operation in a real application. All of the atomic tests were run on the actual hardware, under a simple stand-alone multiprocessor monitor which is supplied with the SGI hardware.

a) Synchronization Operations: DASH provides four mechanisms for performing atomic read-modify-writes: uncached locks, invalidating locks, queued locks, and Fetch&Op. Uncached locks provide uncached test&set and clear operations, but spin waiting with uncached locks uses cluster and network resources. Invalidating locks are simple cached test&set locks. Unlocks, however, force waiting processors to rush for an contended lock when their cache copy is invalidated. Queued-locks use the directory state to avoid needlessly invalidating all waiting processors when a lock is released. Instead, a grant cluster is chosen at random from one of the nodes caching (spinning on) the lock, and this cluster is allowed to acquire the lock with a local cluster operation. Thus, the granted lock reduces both traffic and latency when a contended lock is released. Fetch&Op operations perform noncombining Fetch&Increment or Fetch&Decrement on uncached memory locations. These operations can be very useful because they are serialized only by the cycle time of local memory. For all of these operations, the atomic synchronization update is done at memory due to limitations of the base SGI hardware. While this precludes a processor from holding a lock in an exclusive state in its cache, which is ideal for locks with low contention and high processor affinity, memory updates are more efficient for contended locks or locks with little processor affinity.

The test used to evaluate sychronization operations is the serialization time and network traffic when multiple processors update a simple counter variable. When using locks, the counter is kept in a cachable memory and protected by a spin lock. When Fetch&Inc is used, the Fetch&Inc location is updated in memory and no mutual exclusion is necessary. The test was run with the lock and counter variable in a cluster that was remote to all contending processors. Keeping these variables remote gives results that are more indicative of a large system.

To reduce latency for the counter in the lock case, the cache line with the count variable is brought into the cache in an exclusive state by writing to another variable in the same cache line. This eliminates the need to fetch the memory block twice (first to get a shared copy of the block to read the current value, and then to refetch to get an exclusive copy for the update). Likewise, the cache line with the counter is flushed back to memory after the mutual exclusion region is unlocked to avoid having the next incrementer fetch the data from a dirty-remote cache. These optimizations reduce the serialization by about 1 μs over simply leaving the counter variable uncached or in the updating processor's cache.

The left graph in Fig. 7 shows the time for one counter update when running the lock test with 1 to 44 processors spread across 1 to 11 clusters. The uniprocessor results show minimum time for the processor to execute one iteration of the loop which includes at least one remote memory fetch. The uncached lock case spins over the network waiting on the lock. Thus, as processors are added, the serialization actually decreases because the lock is sampled without waiting for an indication that lock is free. When going from 2 or 3 clusters, however, the advantage from sampling is lost because the excess memory traffic interferes with the execution of the critical section. For invalidating locks, latency is relatively constant except for the first increase when going from 1

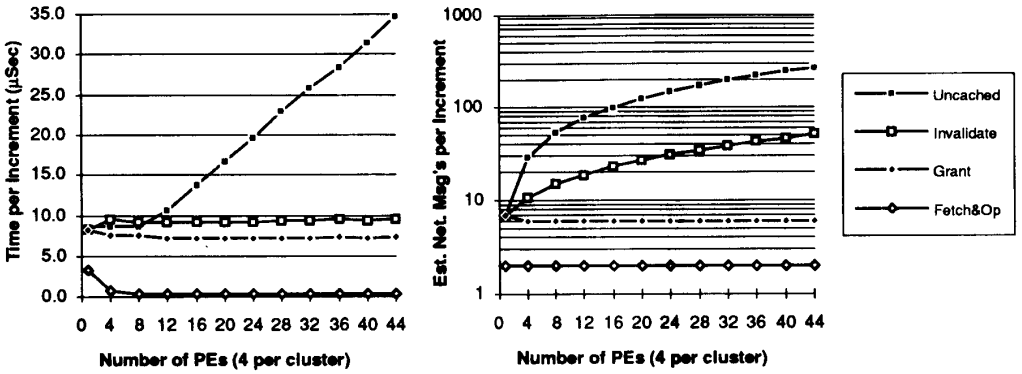


Fig. 7. Serialization time and network messages for an atomic counter increment.

to 4 processors. This increase results from the processor immediately acquiring the lock in the uniprocessor case, while receiving a locked copy of the lock from a locally spinning processor and waiting for a cache invalidation in the multiprocessor case. In contrast, the serialization of a granted lock decreases when going between 1 and 4 processors. This is because the lock grant takes place in parallel with the counter flush in the four processor case, while in the uniprocessor case, the processor is not ready to fetch the lock immediately after the unlock. For the Fetch&Inc case, serialization is low and throughput constantly increases. For the uniprocessor case, throughput is limited by the latency to remote memory. When using a single cluster, the requesting cluster's bus limits throughput. Above the three clusters throughput approaches the reply network outbound serialization limit in the home cluster.

Looking at Fig. 7, Fetch&Inc is obviously the best when it is applicable. For the more general case, the queue-based lock offers approximately 33% less latency for an unlock/lock pair than the invalidating unlock. Furthermore, one important feature not revealed directly by this test is the decrease in traffic when using queue-based locks. As shown in the right graph of Fig. 7 the amount of traffic per counter increment grows with the number of processors when using uncached or invalidating locks,[12] but is constant for granting locks and Fetch&Ops. Whether these latency or traffic reduction gains are important depends upon the frequency of synchronization and the amount of lock contention, but most applications will be adversely affected by added synchronization overhead or increased memory traffic.

b) Update Write: Update write operations allow the producer of data to communicate data to consuming processors without forcing the consuming processors to fetch the data from the producing processor's cache. Update writes are best suited to communicating small data items that the consumers have already cached. They are more appropriate for small data items because each word written, as opposed to each cache line, generates a separate set of coherence messages. Likewise, update writes only deliver data to processors that are caching the block at the time of the write.

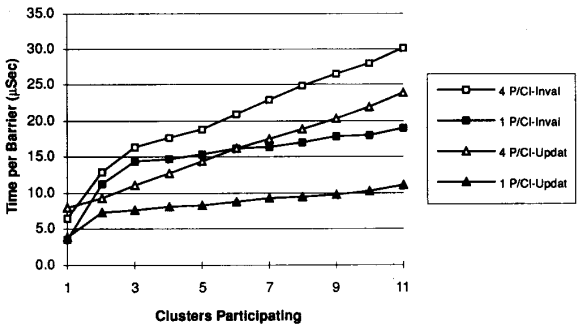


Fig. 8. Barrier synchronization with update writes and Fetch&Op variables.

A useful application for update writes is in implementing the release from a barrier synchronization. Fig. 8 summarizes the timing of a single barrier while varying both the number of clusters (1 to 11) and number of processors per cluster (1 or 4). As in other tests, we assume that the variables for the barrier reside in a remote cluster. The implementation of the barrier uses a Fetch&Op variable to count the incoming processors, and a flag to release the waiting processors. In the case that the processors are all in one cluster, an invalidating write is actually superior to an update because the release flag becomes dirty in the local cluster. Beyond this degenerate case, however, update write reduces the barrier time by approximately 6 μs or about 20–40%.[13] Update writes also reduce serialization and traffic for release notification because the update is serialized only by the network send time. In the invalidation case, the refetch of the release flag is serialized by the first reader causing the dirty release flag to be written back to memory, and then by the other clusters serializing on the outbound reply network port in the home.

The barrier test is also useful in illustrating the power of the Fetch&Op variables. Shown in Fig. 9 is the time for the same

[12] Actual traffic growth for uncached locks can be unbounded since there is traffic generated during the entire time the critical region is held. The graph in Fig. 7 assumes these retries are limited to 3 per processor per iteration since each lock attempt takes approximately 3 μs and the critical region is held for approximately 9 microseconds.

[13] The 6 microsecond difference comes from the savings of reading the release flag from a processors second-level cache instead of the dirty-remote cache of the cluster that issued the write.

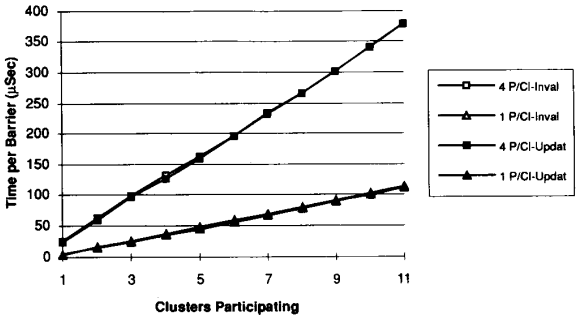


Fig. 9 Barrier synchronization without Fetch&Op variables.

barrier test when the Fetch&Op operations are replaced by a simple memory counter protected by an test&set lock (using simple invalidating unlocks). As expected from the atomic counter test (see Fig. 7), without the Fetch&Op variables, per processor serialization is greatly increased. The benefit of the update write becomes lost in the large increase in the time to count the arrival of the processors at the barrier. By comparing Fig. 8 and Fig. 9 it is clear that Fetch&Op variables are key to efficient barrier synchronization, and that without Fetch&Op, barriers with a large fan-in would require some form of software gather tree to count the synchronizing processors.

c) Prefetch Tests: DASH supports software-controlled nonbinding prefetch operations which can either bring data closer to the issuing processor or send data to a set of specified clusters. Consumer prefetch operations bring a shared or exclusive copy of the requested memory block closer to the issuing processor. Producer prefetch sends a shared copy of the given block to the set of remote clusters specified as an input operand to the operation. Prefetches are software-controlled in the sense that application software issues an explicit operation to prefetch a given memory block. They are nonbinding in a sense that they only move data between coherent caches (into the RAC on the prototype). Thus, prefetched blocks are always kept coherent, and the value of a location is only bound when an actual load operation is issued. Not binding the value of the prefetch is important because it allows the user or compiler to aggressively prefetch data without a conservative dependency analysis and without affecting the memory consistency model. Prefetch operations can increase the amount of computation-communication and communication-communication overlap substantially because they replace blocking cache load operations with nonblocking operations. This overlap is especially useful when accessing a large data set that does not fit into cache or when accessing data generated by another processor. In the prototype, the prefetch operations have the effect of converting the penalty of a remote cache miss to that of a local miss. Thus, throughput can theoretically be increased from one cache line every 101 processor clocks to one block every 29 clocks (nearly a 3.5x improvement).

The first atomic test used to stress prefetch operations uses a varying number of processors that read blocks (128 bytes or 8 cache lines) of data from remote memory. This test models a computation where at least one of the data structures is large and must always be fetched from memory. On each iteration, the processor picks a random remote cluster and reads a data block which is guaranteed to miss in the cache. When prefetching is enabled, the new block is prefetched while reading the previous block.

The results of the test are presented in Fig. 10 when using 1 to 48 processors fetching data from any of the active clusters (i.e. ones with enabled processors). The results are given in terms of number of processor clocks per cache line read, and the total megabytes per second read by all the processors. The code to read a block includes approximately 20 instructions. This instruction count corresponds with the measured time for the loop[14] which includes the 20 instructions plus one local (30 clock) or remote (100 clock) cache miss. Thus, prefetch results in a speedup of approximately 2.4 when the memory system is not saturated. As the number of processors increases beyond 12, the benefits of prefetching decrease because the memory system begins to saturate. In the prototype, this is due to saturation of the cluster buses (as opposed to the mesh network bisection) which are relatively slow (each cluster bus can sustain 20 Mbytes/s when doing remote references that require a total of 3 bus transactions per reference). Thus, on the prototype bandwidth limits throughput to approximately one prefetch every 20 instructions / cluster.

Another use of prefetch is in emulation of message-passing. To test the ability of prefetch to improve message-passing performance, a test was written to do simple one-way communications using a shared circular buffer. In the test, the sending processor checks for a free buffer in the message queue, and if available, writes a message into the buffer and increments the buffer full pointer. Similarly, the receiving processor polls the full pointer, reads each word of the message, and then increments the free buffer pointer. A large quantity of data is sent through the limited-size buffer, so throughput is determined by either the sender or receiver together with the latency to communicate that the message buffer has been filled or emptied.

Fig. 11 shows the throughput (in Mbytes/s) of running the message test with various optimizations and home clusters for the message queue. The test sends 64 000 64-byte messages, using a queue capable of holding 32 messages. The first version of the test uses the normal invalidating protocol (with release consistency) and achieves a throughput of 2.2–2.6 Mbytes/s. In this simple version, throughput is limited by the receiver who must fetch the data from the sending processor's cache. The first enhancement to the code uses writes with update coherence to store the incremented full and empty block pointers. This enhancement increases performance to 2.7–3.4 Mbytes/s because the receiver can read the updated buffer pointer from its cache, not the remote processor's cache. The third version of the test uses consumer prefetch for both the sender and receiver. The receiver prefetches all four blocks of the message as soon as it is notified that they are available, and the producer prefetches buffers exclusively when they are freed. Prefetch improves performance by a factor of 1.8–2.1

[14] When using a small number of processors (1 to 12).

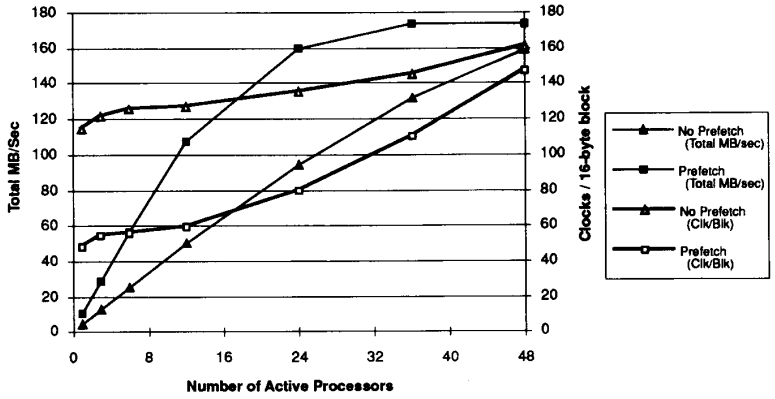


Fig. 10. Effectiveness of prefetch operations when doing block memory reads.

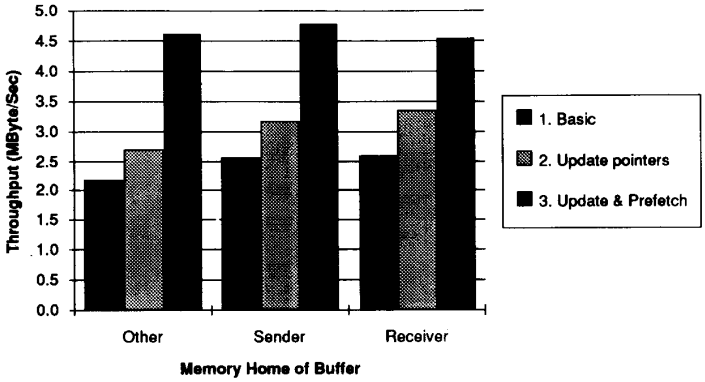


Fig. 11 Message passing emulation using prefetch operations.

to a rate of 4.6–4.8 Mbytes/s. This improvement comes from converting the four remote misses to read the message to a single remote cache miss and three local misses (the miss time for these remote misses are pipelined with the miss time for the first block). In this test, using producer-prefetch instead of consumer prefetch would add little because 3/4 of the remote miss latency has already been hidden, and forcing the producer to send the data to the consumer, in addition to its other tasks, makes the sender the bottleneck.

As shown by these tests, prefetch can successfully overlap a significant amount of interprocessor communication delay. Furthermore, even when remote latency cannot be completely hidden, prefetch allows misses to be pipelined, and greatly reducing the cost of communicating items that are larger than a single cache line. Overall, when memory bandwidth does not limit performance, we expect prefetch to be very useful in improving the performance of interprocessor communication.

3) Summary of Protocol Effectiveness: The parallel benchmark reference statistics and the atomic tests indicate that many of DASH's protocol optimizations can be quite effective. Forwarding requests to dirty-clusters reduces remote latency an average of 5%, and forwarding invalidations for write requests reduced ownership latency by 11%. Likewise, cache-to-cache sharing within a cluster reduces the average latency for remote memory by 27%.

The atomic tests of the alternative memory operations indicate that many of these operations are well-worth supporting. In particular, nonbinding prefetch can reduce effective memory latency by more than a factor of 2.5 and can improve message-passing throughput by a factor of 2. The latency gains from queue-based locks are more modest, but the significant reduction in traffic should make them very useful in applications with significant synchronization. Likewise, the low serialization of noncombining Fetch&Op accesses can be very useful in implementing common operations such as distributed loops and barriers. In contrast, the performance benefits of other alternative operations may not warrant the complexity they add to the memory system protocol. For example, the effectiveness of consumer prefetch operations may preclude the need for producer prefetch (unless multicast is supported in the network), and invalidating and uncached locks offer little compared with queue-based locks.

VII. Conclusions

This paper has outlined the architecture of DASH and our initial experience in building and using the

prototype system. The goals of the DASH architecture are to support scalability to hundreds of processors, high-performance processing elements and a single address space. Key to these goals is a scalable directory-based cache coherence mechanism. The DASH prototype is an experimental system built to investigate the feasibility and performance of this architecture. It supports up to 64 RISC processors to achieve up to 1.6 GIPS and 600 MFLOPS of performance. The prototype utilizes a commercially available small-scale multiprocessor as a computing node, and interconnects the system by a pair of logic boards that implement the global coherence protocol and interconnection network.

The first result from building and using the prototype is that such systems are feasible. While the coherence protocol and hardware are not trivial, such systems can be built. Looking in more detail at the logic and memory costs exhibited by the prototype, we show that the logic overhead for supporting distributed shared memory (without coherence) is about 10%. Supporting scalable cache coherence adds another 10% in logic and 14% in memory overhead. Our experience also shows that using a small-scale multiprocessor as the base processing node, rather than a uniprocessor processing node, significantly reduces the per-processor overhead of supporting cache coherence.

Our second goal in building the prototype is to analyze the performance of the DASH architecture. Toward this goal we have added performance monitoring hardware to the directory logic in each cluster. This monitor uses a static RAM array with increment logic, and a large DRAM buffer to flexibly count and trace bus, directory and network events. This logic allows us to correlate application speedups to reference behavior, and to monitor hardware resource utilization.

We have begun to analyze the performance of the system in more detail. The first result has been to determine accurate memory system delays. In the prototype, memory access times for first-level cache, second-level cache, local memory, remote memory, and dirty-remote memory are 1:15:29:101:132 process clocks (30 ns) respectively. While the absolute delays can be improved, processor clock rates are also increasing. Thus, long latencies in terms of processor clocks are quite unavoidable, and the benefits from exploiting cache and memory locality will continue to be important for this class of systems.

The performance of several parallel applications has also been studied on the prototype. While not highly tuned for the prototype, over half of the applications achieve over 36-fold speedup on 48 processors. Many of these applications use complex and dynamically updated data structures, such as octrees, and the task of programming them on a noncache coherent machine would have been extremely difficult. The reference behavior of the applications indicates that best speedup is achieved if both cache and memory locality are high.

Finally, many of the optimizations in the DASH protocol appear to be effective. In particular, request forwarding and local cache-to-cache sharing can reduce remote latency by up to 25%. Prefetch operations can help overlap communication traffic with other communication and computation, and low serialization operations such as Fetch&Op can be very useful in implementing common operations such as distributed loops and barriers.

Overall, our initial experience with the DASH prototype has been very positive. Scalable, directory-based coherence can be achieved with a modest hardware overhead and can deliver good performance on parallel applications.

ACKNOWLEDGMENT

We are especially grateful for the help we have received with applications. J. P. Singh helped with Barnes-Hut, FMM, Water, and Radiosity applications, E. Rothberg with MatMul and Cholesky applications, A. Erlichson with the PSIM4 application, and J. Nieh with the Volume Rendering application. D. Ofelt helped gather the application performance results.

REFERENCES

[1] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz, "LimitLESS directories: A scalable cache coherence scheme," in *Proc. Fourth Int. Conf. Architectural Support Programming Languages and Oper. Syst.*, Apr. 1991, pp. 224–234.

[2] F. Baskett, T. Jermoluk, and D. Solomon, "The 4D-MP graphics superworkstation: Computing + graphics = 40 MIPS + 40 MFLOPS and 100,000 lighted polygons per second," in *Proc. Compcon Spring 88*, Feb. 1988, pp. 468–471.

[3] 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.

[4] C. M. Flaig, "VLSI mesh routing systems," Tech. Rep. 5241:TR:87, California Institute of Technology, May 1987.

[5] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *Proc. 17th Int. Symp. Comput. Architecture*, May 1990, pp. 15–26.

[6] S. Goldschmidt and H. Davis, "Tango introduction and tutorial," Tech. Rep. CSL-TR-90–410, Stanford Univ., 1990.

[7] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber, "Comparative evaluation of latency reducing and tolerating techniques," in *Proc. 18th Int. Symp. Comput. Architecture*, May 1991, pp. 254–263.

[8] A. Gupta, W.-D. Weber, and T. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," in *Proc. 1990 Int. Conf. Parallel Processing*, Aug. 1990, pp. I:312–321.

[9] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the DASH multiprocessor," in *Proc. 17th Int. Symp. Comput. Architecture*, May 1990.

[10] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam, "The Stanford DASH Multiprocessor," *IEEE Comput. Mag.*, vol. 25, no. 3, Mar. 1992.

[11] D. Lenoski, "The design and analysis of DASH: A scalable shared-memory multiprocessor," Ph.D. dissertation. Stanford Univ., Dec. 1991.

[12] E. Lusk, R. Overbeek, J. Boyle, R. Butler, T. Disz, B. Glickfeld, J. Patterson, and R. Stevens, *Portable Programs for Parallel Processors*. New York: Holt, Rinehart and Winston, 1987.

[13] B. W. O'Krafka and A.R. Newton, "An empirical evaluation of two memory-efficient directory methods," in *Proc. 17th Int. Symp. Comput. Architecture*, May 1990, pp. 138–147.

[14] M. S. Papamarcos and J. H. Patel, "A low overhead coherence solution for multiprocessors with private cache memories," in *Proc. 11th Int. Symp. Comput. Architecture*, May 1984, pp. 348–354.

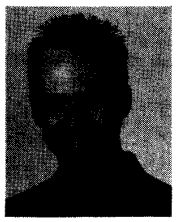
[15] J. P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford parallel applications for shared memory," Tech. Rep. CSL-TR-91–469, Stanford Univ., 1991.

[16] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy, "Load balancing and data locality in hierarchical N-body methods," Tech. Rep. CSL-TR-92–505, Stanford Univ., 1992.

[17] Xilinx, *The Programmable Gate Array Data Book*, 1991.

Daniel Lenoski (M'86) received the B.S.E.E. degree from the California Institute of Technology in 1983 and the M.S.E.E. degree from Stanford in 1985. He received his Ph.D. in electrical engineering from Stanford University in December 1991.

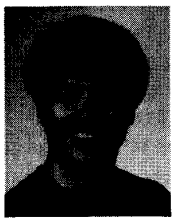
He is a senior staff engineer at Sun Microsystems working on high-performance processor and system design. His research efforts concentrated on the design and implementation of DASH, and other issues related to scalable multiprocessors. He also spent nine years at Tandem Computers where his work included the architectural definition and design of the CLX series of processors.



James Laudon received the B.S. degree in electrical engineering from the University of Wisconsin–Madison in 1987 and the M.S. degree in electrical engineering from Stanford University in 1988. He is a Ph.D. candidate in the Department of Electrical Engineering at Stanford University.

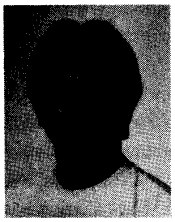
His research interests focus on multiprocessor architectures, with an emphasis on latency tolerating techniques.

Mr. Laudon is a member of the IEEE Computer Society and the Association for Computing Machinery.



Truman Joe received the B.S. degree in electrical and computer engineering from Rice University in 1987 and the M.S. degree in electrical engineering from Stanford University in 1989. He is a Ph.D. candidate in the Department of Electrical Engineering at Stanford University. His research interests are in the area of multiprocessor architecture with particular emphasis on high performance memory systems.

Mr. Joe is a member of the Association for Computing Machinery, Eta Kappa Nu, and Tau Beta Pi.



David Nakahira received the B.A. degree in computer science from the University of California at Santa Cruz in 1984.

He is a staff research engineer in the Computer Systems Laboratory at Stanford University. His interests are in hardware design for large-scale multiprocessors. He worked in the computer industry for five years prior to joining Stanford's DASH project.

Luis Stevens received the Bachelor of Engineering degree in 1986, and the Civil Electronics Engineering degree in the area of computer systems in 1988, from the Universidad Tecnica Federico Santa Maria, Valparaiso, Chile. In 1990 he received the M.S.E.E. degree from Stanford University.

He is currently pursuing his Ph.D. in electrical engineering at Stanford University, doing research in the area of multiprocessor operating systems. He is also interested in distributed operating systems, compilers, and automatic control.

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, He was on the research faculty of Carnegie Mellon University, where he received the Ph.D. in 1986.

Dr. Gupta was the recipient of a DEC faculty development from 1987–1989, and he received the NSF Presidential Young Investigatory Award in 1990. He currently holds the Robert Noyce faculty scholar chair in the School of Engineering at Stanford.

John Hennessy (F'91) is a Professor of Electrical Engineering and Computer Science at Stanford University. His current research interests are in exploiting parallelism at all levels to build higher performance computer systems.

Dr. Hennessy is the recipient of a 1984 Presidential Young Investigatory Award, and in 1987 was named the Willard and Inez K. Bell Professor of Electrical Engineering and Computer Science. He is a member of the National Academy of Engineering.