Stanford DASH

The DASH Prototype: Implementation and Performance

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

Computer Systems Laboratory
Stanford University, CA 94305

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 16-processor prototype of the DASH multiprocessor has been operational for the last six months. In this paper, the hardware overhead of directory-based cache coherence in the prototype is examined. We also discuss the performance of the system, and the speedups obtained by parallel applications running on the prototype. Using a sophisticated hardware performance monitor, we characterize the effectiveness of coherent caches and the relationship between an application's reference behavior and its speedup.

1.0 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) the ability to support hundreds to thousands of processors, (ii) high-performance individual processors, and (iii) a single shared address space.

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 course, is that caching 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, large-scale machines currently do not support cache coherence, and it has not been clear what the benefits and 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 evaluate these ideas, we have constructed a prototype DASH machine. The full prototype will consist of sixty-four 33MHz MIPS R3000/R3010 processors, delivering up to 1600 MIPS and 600 scalar MFLOPS. An initial 16-processor prototype has been working for the past several months, and we are currently expanding this to the full 64-processor configuration.

This paper examines the hardware cost and performance characteristics of the prototype DASH system. Cost is measured in terms of the logic gates and the bytes of dynamic and static memory in the base system and the added directory logic. Performance is measured in terms of memory system bandwidth and latency, and in terms of parallel application speedups. For a representative set of the measured applications, we also present detailed reference statistics and relate these statistics to the observed application speedups. Finally, we describe the structure of the performance monitor logic which was used to take the detailed reference measurements.

The paper is organized as follows. Section 2 gives an overview of the DASH architecture. Section 3 introduces the DASH prototype and describes the logic used for the directory-based coherence protocol. Section 4 details the hardware costs of the system. Section 5 outlines the structure and function of the performance monitor logic, and Section 6 presents the performance of the memory system, and the speedups obtained by parallel applications running on the prototype. We conclude in Section 7 with a summary of our experience with the DASH prototype.

2.0 The DASH Architecture

The DASH architecture has a two-level structure shown in Figure 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. In addition, the grouping of multiple processors on a bus within each cluster amortizes the cost of the directory logic and the network interface. This grouping also reduces the directory memory requirements by keeping track of cached lines at a cluster as opposed to processor level. (We will more concretely discuss the role of clustering in reducing overhead in Section 4).

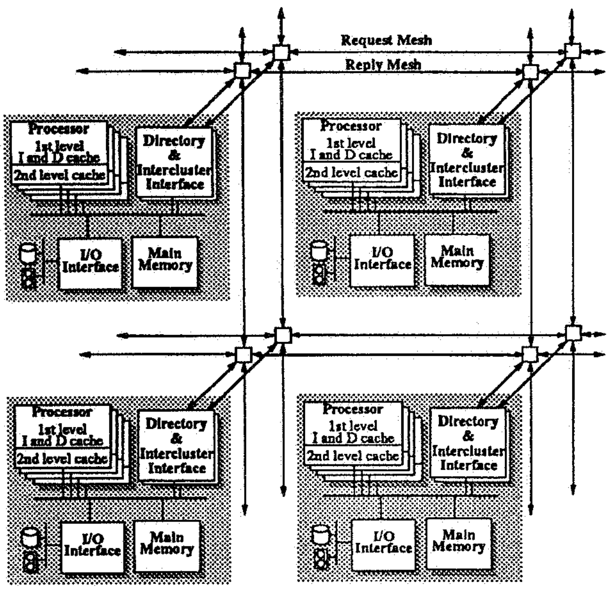


Figure 1. Block diagram of a 2x2 DASH prototype.

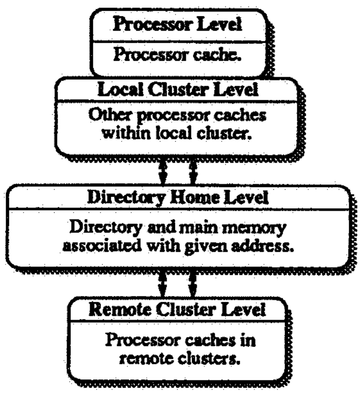


Figure 2. Logical memory hierarchy of DASH.

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.

The DASH memory system can be logically broken into the four level hierarchy shown in Figure 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 addresses, 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 a 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, that helps hide latency by allowing buffering and pipelining among memory requests. DASH also supports *software-controlled non-binding prefetching* to help hide latency of read operations. Finally, DASH supports efficient spin locks in hardware and fetch-and-incr/decr primitives to help reduce the overhead of synchronization. Since we will primarily be focussing on the basic cache coherence protocol in this paper, we will not describe the details of these optimizations. For a more detailed discussion of the protocol and the optimizations, see [7, 8].

3.0 The DASH Prototype

To focus our effort on the novel aspects of the design and speed completion of a usable system, the base cluster hardware of the prototype is a commercially available bus-based multiprocessor. While there are some constraints and compromises imposed by the given hardware, the prototype still makes an interesting research vehicle.

The prototype system is based on a Silicon Graphics POWER Station 4D/340 as the base cluster [3]. The 4D/340 system consists of four MIPS R3000 processors and R3010 floating-point coprocessors running at 33 MHz. Each R3000/R3010 combination can achieve execution rates up to 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 an independent 16 MHz memory bus clock.

The second-level processor caches are responsible for bus snooping and maintaining coherence among the caches in the cluster. Since the first-level caches satisfy most memory requests, the second-level caches do not need duplicate snooping tags. Coherence is maintained with a MESI (Illinois) protocol[12], 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/sec.

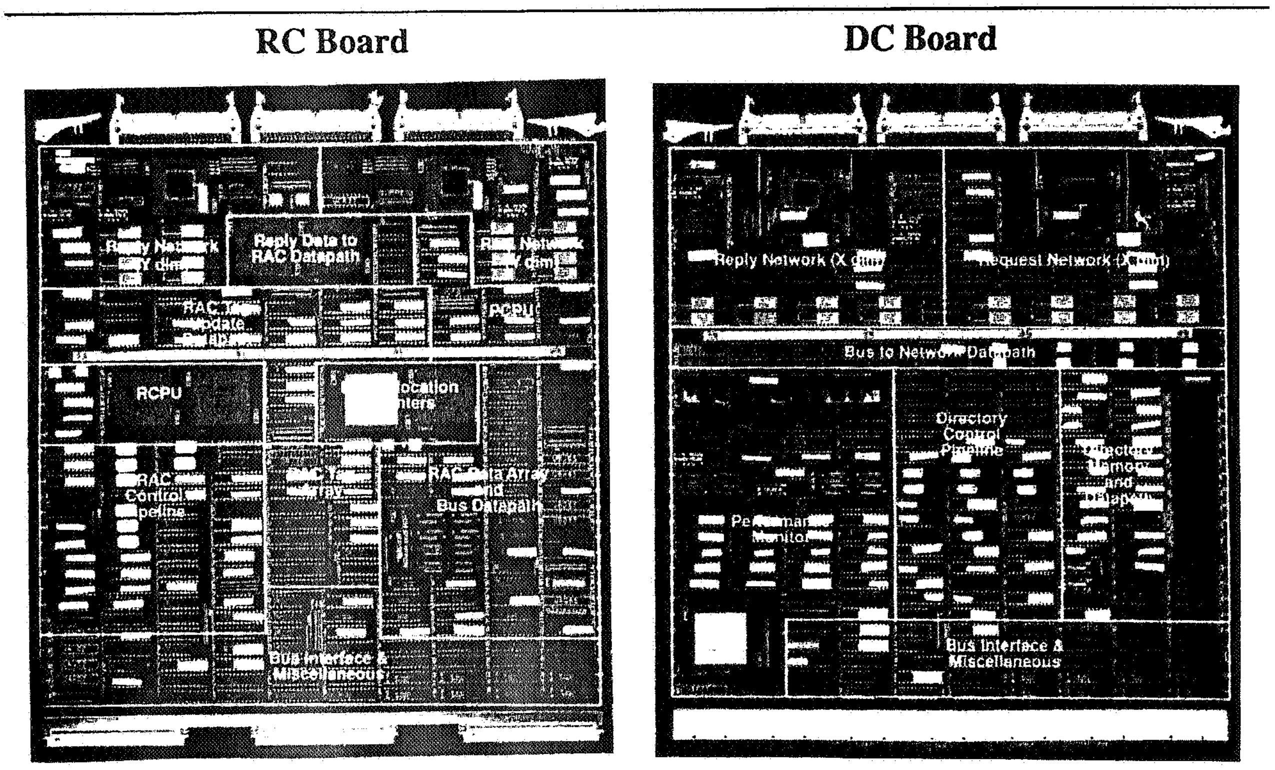

RC Board
DC Board
Reply Data to RAC Datapath
RCPU
PCPU
Bus Datapath
Bus Interface & Miscellaneous
Bus to Network Datapath
Directory Control Pipeline
Bus Interface & Miscellaneous

Figure 3. Directory and Reply Controller boards.

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, with minor modifications, could scale to support hundreds of processors, the current version 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 Figure 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 [4]. 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 [2, 6, 11, 1] given the size of the prototype, and it allows for more 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 128Kbyte direct-mapped snoopy cache with 16byte cache lines. One port of the RAC services the in-bound reply network while the other snoops on bus transactions. The RAC is lockup-free in that it can handle several outstanding remote requests from each of the local processors. RAC entries are allocated when a 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 within the cluster, and it takes advantage of the cache-to-cache transfer protocol supported between the local processors. The snoopy structure also allows the RAC to supplement the function of the processor caches. This includes support for a dirty-sharing state for a cluster (normally the Illinois protocol would force a write-back) and operations such as prefetch.

As stated in the architecture section, the DASH coherence protocol does not rely on a particular interconnection network topology. 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[5]. 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 (≈ 50ns per hop in our network). 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/sec in and out of each cluster.

4.0 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 PALs and LSI FPGAs used in the directory logic.

Table 1 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. DIPs and PGAs) was used throughout the 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 EPROMs for the CPUs.
TTL	Gates in equivalent gate array macros.

Table 1. Percent of all logic in a DASH cluster.

Section	Gates (000's)	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

The numbers in Table 1 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 UARTs and timers attached to each processor; (iii) the Ethernet and VME bus interfaces on the Silicon Graphics' I/O board.[1] Table 2 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 approximately33%. When measured in terms of logic gates the portion of the cluster dedicated to the directory is 20%, and the SRAM and DRAM overhead is 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 doesn't 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.

1. Each I/O board would still include a SCSI interface for disk interfacing.

Table 2. Percent of *core* logic in a DASH cluster.

Section	Gates (000's)	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	1268	73	2286

Looking at the numbers in Table 2 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 [2, 6, 11, 1] 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 lockup-free. With enhanced processor caches, the primary use of the RAC would be to collect invalidation acknowledgments and to receive granted locks. 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,[2] 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 very high. 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 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, non-cache 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.

Finally, by examining the required gate, memory, and connectivity requirements, one can estimate how the prototype logic might be integrated into a small number of VLSI components. As examined in detail in [9], such a system could consist of clusters based on the following:

- Four single-chip microprocessors with direct control of their second-level caches and a their interface to the cluster's snoopy bus.
- A single memory control chip interfaced to local DRAM and the cluster bus.
- An I/O interface chip connecting the cluster to a high-speed fiber optic I/O links (e.g., fibre channel).
- A single directory controller chip interfacing to an external mixed SRAM-DRAM sparse directory, the cluster bus, and a network chip.[3]
- A single mesh routing chip supporting two logical 3-D meshes.

This integrated system could maintain a similar directory logic and memory overhead as the prototype, while supporting cache coherence for 2K processors and 128 GBytes of memory.

5.0 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. To enable more insight into the behavior of 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 non-invasive measurements of the complete system without any external 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. The monitor is controlled by a software programmable Xilinx gate array (FPGA) [15] allowing flexible event selection and sophisticated event preprocessing.

A block diagram of performance logic is shown in Figure 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 16Kx32 SRAMs and increment logic that count event occurrences. Third, a 2M x 36 trace DRAM which captures 36 or 72 bits of information on each bus transaction.

The counting SRAMs 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 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 con-

2. Approximately equal to the complexity of a entire processor with its caches.

3. A small associative RAC would be kept on-chip.

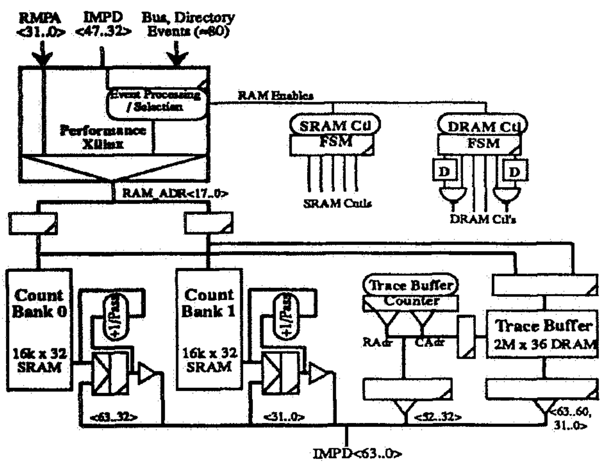


Figure 4. Block diagram of the performance monitor logic

junction 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 data in the first bank allows access type frequencies, bus utilization, access locality, RAC performance, remote caching statistics and network message frequency to be calculated. 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 FPGA counter to each CPU which is enabled whenever a processor is waiting for a remote access. This allows a complete distribution of remote access latencies to be determined. Furthermore, when combined with the count of local bus cycles an estimate of processor utilization can be made.

The other resource of the performance monitor is a 2Mx36 trace array. Again, what 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 2M memory addresses together with the issuing processor number and read/write status are captured. The second mode can capture only 1M addresses, but adds additional bus and directory state, and a bus idle count to each trace entry. This trace information can be used to do detailed analysis of reference behavior or as input to a memory simulator. With software assistance the tracer can be used to capture much longer traces and trace all memory references.

6.0 Prototype Performance

This section examines the performance of the initial hardware prototype of DASH which includes 16 processors in four 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.

6.1 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 3 lists the processor bandwidth and latency for cache memory operations in DASH assuming no contention. *(PClocks* refer to processor clocks which are 30ns in the prototype.) The delays are based on measurements of the hardware, but extrapolated to a full 4x4 cluster configuration. 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.

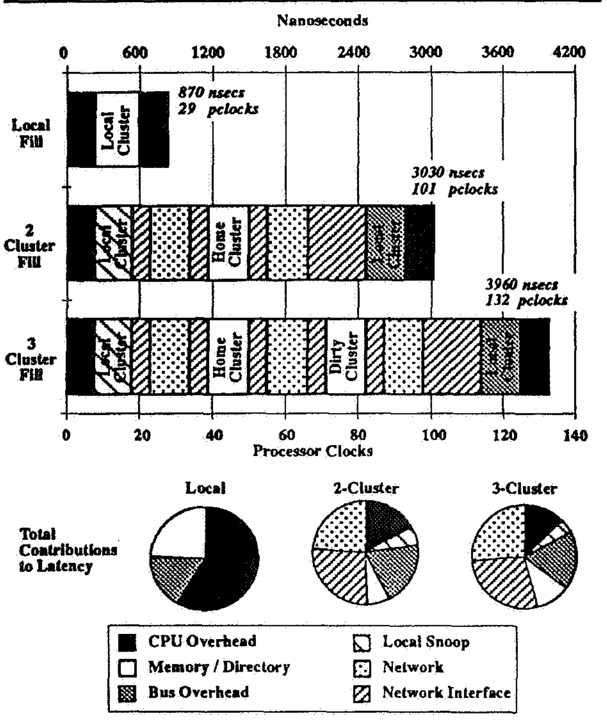
The table presents data separately for reads and writes. For reads, the access latency is given by the last column of the table. The latency can vary by more than two orders of magnitude depending on where a read access is serviced in the memory hierarchy. The read bandwidth also varies considerably,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

Table 3. Cache operation bandwidth and latencies.

Cache Operation	Best Case		Worst Case	
	MB/ sec	Clock/ word	MB/ sec	Clock/ word
Read from 1st-level cache	133.3	1.0	133.3	1.0
Fill from 2nd-lev. cache	29.6	4.5	8.9	15.0
Fill from local bus	16.7	8.0	4.6	29.0
Fill from remote	5.1	26.0	1.3	101.0
Fill from dirty-remote	4.0	33.8	1.0	132.0
Write retired in 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	5.3	25.3	1.5	88.7
Write retired on dirty-rem.	4.0	33.0	1.1	119.7

A more detailed break-down of the latency for local and remote cache misses is given in Figure 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 Figure 5 represents the latency for fetching a location that is dirty in a cluster other than its home. In this case, an extra 30 clocks (or 1 μsec) 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



For clarity four processor clocks of bus overhead per bus transaction are not shown in the bar graphs, but are included in the total contribution breakdown.

Figure 5. Cache fill latency in the DASH prototype.

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 [9], 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.

6.2 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 nine parallel programs representing a variety of application domains. Three of these applications are studied more in greater detail using data captured by the performance monitor.

6.2.1 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 that has been augmented with the Argonne National Labs (ANL) parallel macros[10] to control a MIMD, shared-memory programming model.

Before giving the speedup results in the next subsection, we first state the assumptions used in measuring the speedups. The speedup 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. The runs were averaged over a number of executions to eliminate any scheduling anomalies in UNIX. In some applications, the serial start-up time is ignored from the measurement because the runs used shortened executions of the application (less time steps, etc.) to reduce measurement time. In production runs the start-up time would be negligible.

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, and no prefetching has been added.

6.2.2 Application Speedups

Figure 6 gives the speedup for nine parallel applications running on the hardware prototype using from 1 to 16 processors. The applications cover a variety of domains. There are some scientific applications (Barnes-Hut and Water), several engineering applications (Radiosity, MP3D, PSIM4, Locusroute) and two kernels (Cholesky, Matrix Multiply and Mincut). Four of the programs (Water, LocusRoute, MP3D and Cholesky) are taken from the SPLASH parallel application suite[14]. We begin with a quick overview of all nine applications and then present the detailed reference behavior and performance of three of the applications (Barnes-Hut, Water, and LocusRoute).

Starting with the applications with the best speedup, the Radiosity application is from the domain of computer graphics. It computes the 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. While the data structures used are a complex oct-tree and a binary-space partition tree, we see that caches work quite well and we get a speedup of over 14 with 16 processors. The second application, Barnes-Hut, is an N-body galactic simulation solved using the Barnes-Hut algorithm (an $O(N\log N)$ algorithm). Again we see that although the structure of the program is complex, good speedups are obtained.

The next application is Mincut. It performs graph partitioning using parallel simulated annealing. We ran Mincut to find the minimum bisection of a graph containing 500 nodes. Mincut achieves good speedups because there is extensive cache reuse as processors traverse the graph and repeatedly consider moving nodes as the graph partition and annealing temperature change.

The next application is a scaled matrix multiply. It uses 88x88 square matrices on a single processor (obtaining 8.8 DP MFLOPS on a single processor) and 1408x1408 matrices on 16 processors (obtaining 125 DP MFLOPS). The application below that is the Water code (the parallelized version of the MDG code from the

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.

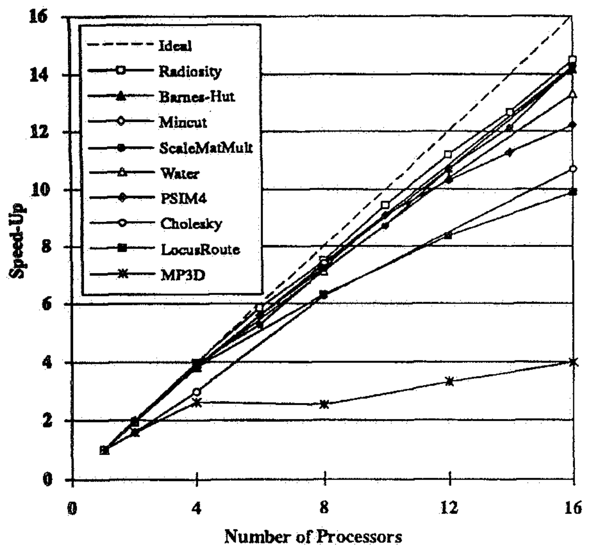


Figure 6. Speedup of applications on the DASH prototype.

Perfect Club benchmarks), a molecular dynamics code. We measured runs using 512 water molecules.

PSIM4 is an application from NASA-Ames, that is a particle-based simulator of a wind tunnel. PSIM4 is an enhanced version of the MP3D code (the application with the poorest speedup), both in terms of functionality (it models multiple types of gases and it models chemistry) and in terms of locality of memory accesses (it uses spatial decomposition of simulated space to distribute work among processors). The 16 processor run is done with over 100,000 particles and it achieves a scaled speedup of over 12 in contrast to a speedup of about 4 achieved by the older MP3D code. While the changes in functionality make a direct comparison of absolute execution time for PSIM4 and MP3D impossible, the speedup improvements of PSIM4 over MP3D are very encouraging.

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 a 256x256 grid problem. We note that much of the fall off in speedup that we see 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. The next application is LocusRoute which performs global routing of standard cells. It will be discussed later in this section.

Finally, as stated before, MP3D is a particle-based wind-tunnel simulator. The measured runs simulated 40,000 particles. The speedups for MP3D are poor because the particles are statically allocated to processors, but 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 dirty in another processor's cache. Thus, even with four processors, the speedup is poor. When a second cluster is added, speedup is flat because roughly half of the misses are now remote. MP3D's speedups improve for 12 and 16 processors, but this does not compensate for the initial inefficiencies encountered with 4 and 8 processors.

Overall, we see that many applications achieve good speedups, even though they have not been specially optimized on the DASH prototype. Almost all get over ten times improvement on sixteen processors, and some get above fourteen times speedup.

6.2.3 Detailed Case Studies

To get a better understanding of the detailed reference behavior of these applications, we now examine Barnes-Hut, Water, and LocusRoute in more detail. We also extend the results for these applications with preliminary results from the 32-processor DASH system (which has just been up for the last two weeks). As expected from Figure 6, Barnes-Hut and Water achieve good speedups on 32 processors (27.5 and 24.5 respectively), but the speedups for LocusRoute fall-off significantly (9.9 at 16, but only 13.2 at 32). These applications were chosen because they achieved a range of speedups, and we were able to get results for them on the larger 32-processor system. We expect that once the 32-processor kernel is better tuned the speedups will improve.

6.2.3.1 Barnes-Hut

The Barnes-Hut program[13] models the dynamic evolution of a system of galaxies under gravitational forces. Using the Barnes-Hut algorithm, the possible N^2 interactions are reduced to $N\log N$ by a hierarchical decomposition of the galaxies and by approximating groups of distant bodies by a single point at their center of mass. The input to the measured runs consisted of two interacting Plummer-model galaxies with 16384 bodies each.

Table 4 gives a detailed memory reference profile of Barnes-Hut running on DASH as measured by the hardware performance monitor.[5] Table 5 is broken into four sections: (i) overall performance and processor utilization; (ii) memory request distribution; (iii) request locality and latency and (iv) bus and network utilization.

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

As indicated by Table 4, Barnes-Hut has a high and nearly constant cache-hit ratio, and a large fraction of its misses are local. Thus, processor efficiency and speedup are very good as the number of

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.

processors increases. There is a small drop in efficiency as the first remote cluster is added, but the degradation beyond this is slight.

The second section of Table 4 gives a breakdown of the memory reference types. This breakdown indicates the type of accesses that cause bus transactions and whether synchronization references are significant. In Barnes-Hut, cache read misses dominate and there are few synchronizations.

The third section of Table 5 lists the fraction of local cache fills, the fraction of remote fills satisfied by a dirty-remote cluster (ratio of 3 cluster to 2 cluster fills), and the latency for local and remote cache fills. The locality figure counts any references satisfied in a single bus transaction as local, while any that must be repeated are considered remote. Thus, a remote reference that was satisfied by a local cache-to-cache transfer between processors would be considered local, and a local reference that was dirty-remote would be considered remote.[7]

For Barnes-Hut, cluster locality is high and decreases only slightly as clusters are added. The actual locality of references (not given in the table) does decrease as clusters are added, but most communication is with nearby processors. Thus, even though the home for the data is remote, many remote accesses are satisfied by a local cache-to-cache transfer from another processor in the cluster. In addition, of the references that are remote, most misses are serviced by the home cluster. This results from the fact that most of these misses are to global data, and only the first processor needs to fetch this data from the producing (i.e. dirty) cluster. The others read the shared data from the home.

The final section of Table 4 indicates the load on the cluster bus and on the bisection of the mesh networks. Bus utilization is measured directly by the performance monitor, while the network bisection utilizations are estimates assuming uniform network traffic. The bisection utilizations are calculated by knowing the total number of network messages sent, assuming half of the messages cross the bisection of the mesh, and dividing by the bandwidth provided across the bisection.[8] For Barnes-Hut, both the buses and networks are lightly loaded.

In Barnes-Hut, as in the other applications studied, the bus loading is higher than the network bisection loading. This is due to the relatively slow cluster bus used in the prototype, and the retry mechanism used for the remote references which implies that each remote reference includes at least three bus accesses, but only one network request and one network reply. For the size of the prototype, the individual cluster buses limit total memory bandwidth. In larger systems with more nodes and faster split-transaction cluster buses, we expect that the network bisection will limit aggregate memory bandwidth if accesses are uniformly distributed.

6.2.3.2 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. The algorithm is $O(N^2)$ in that each molecule interacts with all other molecules in the system. As shown in Figure 6, the Water application achieves good speedup on the DASH hardware.

Table 5 shows that cache-locality is high in Water, and the time between processor stalls indicates that it is not highly sensitive to memory latency. The table does show some reduction in the busy clocks between stalls, especially when going from 2 to 4 processors, but the decrease is slight after this initial drop. In comparison with Barnes-Hut, Water achieves a slightly lower speedup due to lower cluster locality which increases the average miss penalty. As indicated by the fraction of dirty-remote cache reads, many more of the misses in Water are satisfied directly by the producing processor than in Barnes-Hut.

Looking at the breakdown of memory reference types 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 or release references the bus in the prototype. Given the percentage of locks and unlocks are almost identical, this data also indicates that lock contention is not a problem in Water.

Finally, we see that like Barnes-Hut, Water does not put a heavy load on the memory system. While the table does indicate an increase in remote memory latency, this is due to the increasing fraction of remote misses that are dirty-remote as opposed to larger queuing delays.

6.2.3.3 LocusRoute

LocusRoute is a standard-cell placement tool that uses actual routed area to evaluate the quality of a given placement. Thus, the task that LocusRoute performs 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. The runs shown in Figure 6 were for the route of a circuit consisting of 3817 wires and 20 routing channels.

LocusRoute achieves a respectable 9.9 times speedup on 16 processors, but speedup does not increase linearly when more than 16 processors are used (only 13.2 times speedup on 32 processors). Comparing the reference behavior of LocusRoute shown in Table 6 with that of Water, it is clear why LocusRoute does not achieve the same speedup. First, its cache hit rate (busy pclocks between stalls) is lower than Water. This makes it more sensitive to the increase in memory latency when going to multiple clusters. Second, as in Water, locality falls off with more processors and the fraction of dirty-remote references also increases.

Considering the application data structures and algorithms, these effects are not surprising. In LocusRoute, most misses are due to the *cost array* which tracks the number of signals routed in a given section of a channel. The cost array is actively updated by each processor, and there is only limited reuse of the data structure as wires are routed. The result is a lower hit rate than Water and more sensitivity to locality.

Looking at system loading, it is clear that LocusRoute puts more stress on the memory system than Barnes-Hut or Water. Bus utilization is moderate (35-40%). The large latencies for remote memory references when using 24 or 32 processors are due in part to hot spotting for the cost array. In the current implementation, this array is allocated only out of cluster 0's memory. For the 32-processor run, this cluster has over 65% bus utilization (as compared to the average of 37%), and we suspect there is substantial queuing at the PCPU. While removing this hot spot should improve performance, we expect that LocusRoute will achieve significantly better speedup on more than 16 processors only with larger problems.

7. The 20-25% dirty-remote fills when using 1-4 processors are background OS activity. This is only 20-25% of the 0.5-3.0% of the misses that are remote. Local misses caused by the application dominate. When using 8 processors, the dirty-remote percentages are low because there are never any three cluster fills because there is only two active clusters.

8. The factor of two increase when going from 12 to 16 processors arises because the 32-processor DASH is arranged in a 4x2 grid. Currently, processors are numbered and allocated first in the X-dimension so that the 16 processors run in a 4x1 ensemble. Thus, a single X-dimension path sees the load of all four clusters.

Table 4. Barnes-Hut memory access characteristics

Execution Attribute	1 Proc.	2 Proc.	4 Proc.	8 Proc.	12 Proc.	16 Proc.	24 Proc.	32 Proc.
Speedup	1.0	2.0	3.9	7.4	10.7	14.2	20.6	27.5
Efficiency (relative to uniprocessor)	1.00	0.99	0.96	0.92	0.89	0.89	0.86	0.86
Busy Pclks between Proc. Stalls	526.3	633.8	602.7	606.8	534.6	602.7	564.5	560.9
Est. Processor Utilization (%)	94.6	95.6	95.5	94.2	92.9	93.4	92.5	92.3
Cache Read (%)	85.6	90.3	88.5	91.5	92.0	90.7	90.5	90.4
Cache Read Exclusive (%)	9.7	6.8	10.1	6.5	6.1	6.9	6.7	6.7
Cache Lock (%)	2.3	1.4	0.7	1.1	1.2	1.6	2.0	2.0
Cache Unlock (%)	2.1	1.3	0.7	0.8	0.7	0.8	0.6	0.6
Fraction of Reads Local (%)	97.8	99.0	99.5	89.8	86.8	84.9	82.6	82.5
Fraction of Rem. Rds. Dirty-Rem (%)	21.8	21.0	21.2	9.1	6.4	5.7	5.5	5.3
Avg Local Cache Fill (Pclks)	29.2	29.2	29.3	29.4	29.4	29.4	29.4	29.4
Avg Rem Cache Fill (Pclks)	106.6	107.4	105.7	104.2	106.2	109.1	110.9	113.1
Bus Utilization (%)	5.2	6.3	9.1	9.8	11.0	10.2	10.9	11.0
Req. Net Bisection Util. (%)	0.6	0.7	0.7	0.8	0.9	1.7	1.9	2.0
Reply Net Bisection Util. (%)	0.5	0.5	0.5	0.9	1.1	2.2	2.5	2.6

Table 5. Water memory access characteristics

Execution Attribute	1 Proc.	2 Proc.	4 Proc.	8 Proc.	12 Proc.	16 Proc.	24 Proc.	32 Proc.
Speedup	1.0	2.0	3.8	7.1	10.4	13.3	19.3	24.6
Efficiency (relative to uniprocessor)	1.00	0.99	0.95	0.89	0.87	0.83	0.80	0.77
Busy Pclks between Proc. Stalls	935.9	955.0	614.6	528.3	515.0	523.7	553.8	506.3
Est. Processor Utilization (%)	97.5	97.7	97.2	93.2	91.1	90.6	89.2	88.0
Cache Read (%)	46.0	45.2	46.7	46.9	45.2	45.2	44.6	47.0
Cache Read Exclusive (%)	11.9	7.3	25.1	29.6	30.9	30.9	31.7	30.8
Cache Lock (%)	20.9	23.8	14.1	12.1	12.4	12.5	12.3	11.8
Cache Unlock (%)	20.8	23.7	14.1	11.4	11.6	11.4	11.5	10.5
Fraction of Reads Local (%)	96.9	97.2	99.4	68.9	55.1	51.6	40.9	42.8
Fraction of Rem. Rds. Dirty-Rem (%)	21.4	21.6	22.6	9.2	18.3	27.0	48.0	50.0
Avg Local Cache Fill (Pclks)	29.2	29.2	29.4	29.6	29.7	29.7	29.7	29.8
Avg Rem Cache Fill (Pclks)	102.8	107.5	106.9	104.5	108.1	111.6	118.3	120.6
Bus Utilization (%)	4.6	5.7	10.2	14.5	16.3	16.6	17.7	18.8
Req. Net Bisection Util. (%)	0.7	0.8	0.7	1.5	2.0	4.6	6.1	6.7
Reply Net Bisection Util. (%)	0.5	0.6	0.5	1.8	2.5	5.3	6.3	6.7

Table 6. LocusRoute memory access characteristics

Execution Attribute	1 Proc.	2 Proc.	4 Proc.	8 Proc.	12 Proc.	16 Proc.	24 Proc.	32 Proc.
Speedup	1.0	2.0	3.8	6.3	8.4	9.9	11.9	13.2
Efficiency (relative to uniprocessor)	1.00	0.99	0.95	0.79	0.70	0.62	0.50	0.41
Busy Pclks between Proc. Stalls	342.8	312.4	258.0	196.1	179.4	175.4	179.9	181.2
Est. Processor Utilization (%)	92.7	92.9	92.0	79.1	73.1	69.5	64.8	59.4
Cache Read (%)	74.0	70.3	68.8	64.5	69.0	69.3	69.8	70.1
Cache Read Exclusive (%)	15.8	21.9	25.8	31.3	27.5	27.2	26.5	26.0
Cache Lock (%)	5.0	3.9	2.7	2.3	2.0	2.1	2.4	2.7
Cache Unlock (%)	5.0	3.8	2.6	2.0	1.5	1.4	1.3	1.0
Fraction of Reads Local (%)	97.1	98.4	99.6	63.7	60.7	54.4	46.5	41.3
Fraction of Rem. Rds. Dirty-Rem (%)	24.9	24.7	22.8	2.9	20.8	28.0	33.6	35.0
Avg Local Cache Fill (Pclks)	29.1	29.3	29.7	30.4	30.8	31.0	31.0	30.9
Avg Rem Cache Fill (Pclks)	108.2	106.9	107.3	109.1	119.6	128.7	149.6	183.0
Bus Utilization (%)	5.9	8.9	17.1	30.8	36.1	38.2	38.5	36.8
Req. Net Bisection Util. (%)	0.6	0.6	0.7	3.8	5.9	14.2	16.1	16.1
Reply Net Bisection Util. (%)	0.5	0.5	0.5	4.9	6.7	15.4	16.8	16.6

6.2.4 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 for a number of real applications. Applications with the best speedup have good cache and cluster locality. Since most of the degradation in memory latency occurs when adding the first remote cluster, we expect most of the applications will perform well on 32 and 64 processors. Speedup for many of the applications should be more than 24 on 32 processors, and more than 45 on 64 processors, especially if problem size is increased.

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 with the SPLASH benchmarks[14] indicates that the reference rates for Water and LocusRoute to shared data (with 32 processors) is roughly one reference every 20 and 11 instructions respectively. Given the number of busy clocks between misses for Water and LocusRoute given in Table 5 and Table 6 (or even assuming that the number of processor instructions between stalls is optimistic by a factor of two due to internal stalls), caches are satisfying 92% of the shared references in Water (i.e. there are at least $506/2/20 \approx 12$ shared references for every miss), and 88% of the shared references in LocusRoute. Thus, processor utilization without caching would be only 26% in Water and only 13% in LocusRoute.[9] Overall, this implies that caching of shared data improves performance by a factor of 3.4-4.5 in these applications, but as shown earlier, only adds 10% to system cost.[10]

Even with caches, however, locality is still important. If locality is very low and communication misses are frequent (as in MP3D), then speedup will be poor. However, for many applications, the natural locality of the applications is enough (e.g., Barnes-Hut, Radiosity, Water) that good speedups can be achieved without algorithmic or programming contortions. Even in applications where natural locality is limited, DASH's shared-address space model 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.

7.0 Conclusions

This paper has outlined our experience in building and starting to use the DASH prototype system. The first result from building 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 have shown that the logic overhead for supporting distributed shared memory (without coherence) is about 10%. Supporting cache coherence adds another 10%-14% in logic and memory overhead if clustering is used.

The second result of building the prototype has been an analysis of the memory system performance. At the lowest level, it is clear that remote memory latencies are significant. We believe this will remain true as processor speeds increase relative to the inherent delays of a large system. Thus, both cache and cluster locality are important in this class of machines, and latency hiding techniques (e.g., prefetch) may be very useful.

The prototype system has also allowed us to measure applications with large data sets using the performance monitor hardware. A number of parallel applications have been run and most achieve good speedup. Many of these applications achieve better than 12 times speedup on 16 processors, and the preliminary results with the 32-processor machine indicate that many will also work well with 32 and 64 processors.

9. This assumes the locality of shared references is the same as the locality given in the table. This is optimistic since there would be no cache-to-cache sharing.

10. The extra overhead 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.

Acknowledgments

This research has been supported by DARPA contract N00039-91-C-0138. Dan Lenoski is supported by Tandem Computers Inc. James Laudon is supported by IBM. Anoop Gupta is partly supported by a NSF Presidential Young Investigator Award.

References

[1] Agarwal, A., B.-H. Lim, D. Kranz, and J. Kubiatowicz. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proc. Fourth Int. Conf. on Architectural Support Programming Languages and Operating Systems.* pp. 224-234, 1991.

[2] Agarwal, A., R. Simoni, J. Hennessy, and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proc. 15th Int. Symp. on Computer Architecture.* pp. 280-289, 1988.

[3] Baskett, F., 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.* pp. 468-471, 1988.

[4] Censier, L. and P. Feautrier, A New Solution to Coherence Problems in Multicache Systems. *IEEE Trans. on Computers,* C(27):1112-1118, 1978.

[5] Flaig, C.M., *VLSI Mesh Routing Systems.* Technical Report 5241:TR:87, California Institute of Technology, May 1987.

[6] Gupta, A., W.-D. Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *Proc. 1990 Int. Conf. on Parallel Processing.* pp. I:312-321, 1990.

[7] Lenoski, D., J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. 17th Int. Symp. on Computer Architecture.* pp. 148-159, 1990.

[8] Lenoski, D., J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam, The Stanford DASH Multiprocessor. *Computer,* 25(3), 1992.

[9] Lenoski, D.E., *The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor.* Ph.D. Thesis. Stanford University. 1991. Also available as Stanford University Technical Report CSL-TR-92-507

[10] Lusk, E., R. Overbeek, J. Boyle, R. Butler, T. Disz, B. Glickfeld, J. Patterson, and R. Stevens, *Portable Programs for Parallel Processors.* Holt, Rinehard and Winston, Inc.1987.

[11] O'Krafka, B.W. and A.R. Newton. An Empirical Evaluation of Two Memory-Efficient Directory Methods. In *Proc. 17th Int. Symp. on Computer Architecture*. pp. 138-147, 1990.

[12] Papamarcos, M.S. and J.H. Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proc. 11th Int. Symp. on Computer Architecture*. pp. 348-354, 1984.

[13] Singh, J.P., C. Holt, T. Totsuka, A. Gupta, and J.L. Hennessy, *Load Balancing and Data Locality in Parallel N-body Techniques.* Technical Report CSL-TR-92-505, Stanford University, 1991.

[14] Singh, J.P., W.-D. Weber, and A. Gupta, *SPLASH: Stanford Parallel Applications for Shared Memory.* Technical Report CSL-TR-91-469, Stanford University, 1991.

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

Chapter 4

Simulation Design and Implementation

4.1 Overview

This section discusses some of the design decisions made during the initial phase of the project including:

- The simulation construction rationale.
- The level of detail to be simulated.

4.1.1 Simulation Construction Rationale

After assimilating material regarding the operation of the DASH multiprocessor the next task was to decide upon the structure of the HASE simulation. It was decided to adopt a top-down approach to simulation *design* as this provided a useful way of hiding technical detail in the initial design stages. By starting with a single simulation entity called 'The DASH System' it was simple to break down the architecture by a series of refinements. For example, the level below this most abstract starting point was the 'Cluster Level' in which we considered DASH clusters and their interconnection.

It is interesting to note that whilst the design phase took a top-down approach the implementation of the simulation proceeded bottom-up with the lowest level entities being 'tied' together via HASE's group facility (discussed later).

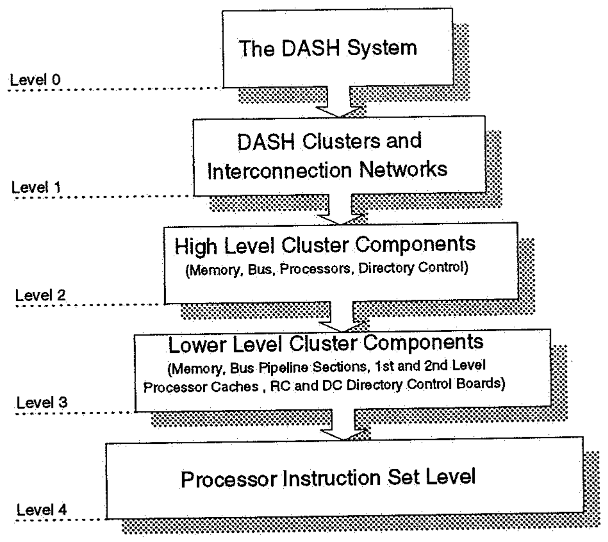

The DASH System
Level 0
DASH Clusters and Interconnection Networks
Level 1
High Level Cluster Components
(Memory, Bus, Processors, Directory Control)
Level 2
Lower Level Cluster Components
(Memory, Bus Pipeline Sections, 1st and 2nd Level Processor Caches , RC and DC Directory Control Boards)
Level 3
Processor Instruction Set Level
Level 4

Figure 4.1: Initial Simulation Decomposition Hierarchy.

4.1.2 Simulation Detail

Obviously the top-down refinement of any design must stop at a point suitable for the task in hand (for example, in our simulation we try to offer a demonstration tool to illustrate features of DASH such as the cache coherency mechanisms, thus a low-level silicon simulation is irrelevant to our particular needs).

After the initial top-down design the resulting simulation hierarchy was that shown in Figure 4.1.

It was decided that the highest level of abstraction served only to bind the lower levels together to a single reference point and would provide no useful information within a simulation. For this reason the top level was discarded early in the design process.

Following this assessment of the upper level of simulation abstraction, attention was turned to the suitability of the suggested lower bound. During the initial design it seemed logical to stop refinement at the instruction set level. However after experimentation with HASE is was soon realised that this level of implementation would be very time consuming (implementing the MIPS R3000 at the instruction set level could be a project in itself!). Clearly some abstraction was required from the instruction set level whilst still aiming to retain enough detail to allow accurate modelling of the architecture's salient features (for example the coherency mechanisms and interconnection networks).

Faced with this problem a list of requirements was drafted:

- It was desirable for users of the simulation to be able to observe the effect of various memory requests throughout the system.

- There should be a differentiation between read and write requests to memory in order to demonstrate fully the coherency protocols.
- Memory requests should be specified with explicit addresses so that local, home and remote cluster addressing mechanisms could be realised.
- Users should be able to define their own input MIPS trace files so as to observe the effect of different addressing orders on the simulation (for example to see how well the system exploits the locality principle). This meant that any input file should be simple to generate.

Given the above criteria, it was decided that rather than implement the actual MIPS instruction set it would be possible to implement an abstract instruction format which could drive the simulation with a level of accuracy adequate for our needs. The new format need only allow specification of a read/write and main memory address pair.

By modelling the address generation of the MIPS processor in such an abstract way however, the ability to distinguish between instruction requests and data requests[1] is lost. Although this may seem to be a large inaccuracy, the simulation can still demonstrate the DASH coherency protocols as well as the use of primary and secondary processor caches. This is possible as these features are based around the classification of accesses as reads or writes. However, this compromise in simulation detail makes the modelling of actual processor behaviour impossible. In effect the MIPS processor entity only functions as a simple address generating box.

4.2 Modelling the Entity Hierarchy in HASE

Having now decided upon the scope of simulation abstraction the next phase of simulation design involved connecting entities together so as to define data paths in the model.

The HASE on-screen designer provides a useful environment for such experimentation. Ports and links are drawn on-screen between the icons representing simulation entities (as described in section 2.2.2). HASE also provides facilities to group entities together in a hierarchical manner. This feature was used to describe the simulation model in terms of the previously discussed abstraction hierarchy. At this point the HASE model contains no actual functionality – this is added at a later stage. However it is possible to use the

[1]Apart from noting that all instruction requests will be reads

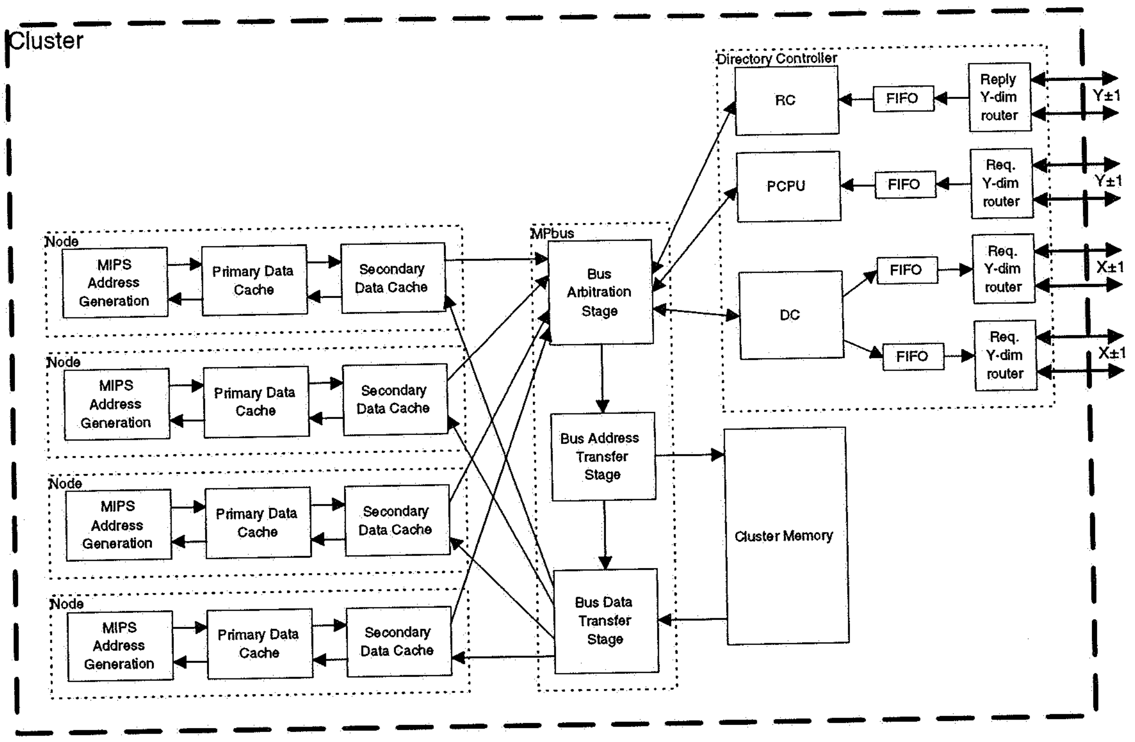


Figure 4.2: Three Levels of Architectural Abstraction.

HASE functions `up_level` and `expand` to navigate the abstraction hierarchy in order to test the final appearance of the on-screen simulation.

A three-level entity hierarchy was finally constructed. This three-level model corresponds to levels 1,2 and 3 shown in Figure 4.1; the actual HASE screen output related to this three-level construction can be found in Appendix A of this report. Level 4 has been replaced by the abstract instruction format discussed above in section 4.1.2. This instruction format is realised in Sim++ code.

Figure 4.2 shows the three level entity hierarchy found in a DASH cluster. The dotted lines surrounding groups of entities indicate a collection of entities which have been grouped together to form a single icon at a higher level of abstraction. For example, a MIPS address generation box, primary cache and secondary cache constitute a processing *node*. Similarly all the entities in Figure 4.2 are encompassed in the highest level of abstraction – the cluster.

Another valuable feature of HASE worth mentioning at this point is the facility to allow multiple levels of abstraction to be viewed simultaneously. This viewing mechanism is facilitated via the use of HASE 'free-ports'. These are entity connection ports which transcend multiple levels of the object hierarchy. Consider Figure 4.3 which shows the simulation entity tree exhibiting all three levels of design abstraction simultaneously. The

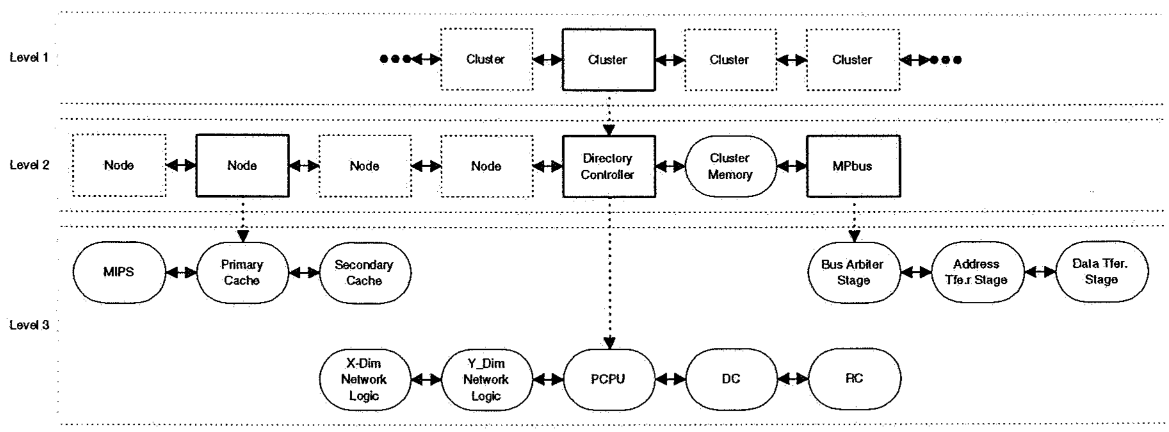


Figure 4.3: HASE Abstraction through Free Ports.

tree has been expanded to allow a detailed inspection of one particular DASH cluster (abstraction level 3 of Figure 4.1) whilst the remaining clusters are visible at a high level (level 1 of Figure 4.1) of abstraction.

In Figure 4.3 an entity selected for expansion is shown by a thick bordered box, unexpanded entities are denoted via a dotted surround. The lowest level entities (those which in our simulation will eventually contain the Sim++ functionality) are shown as oval shapes. In this figure HASE free ports would be found connecting lower level components together into a medium level grouping. At this medium level composite entities from the lower levels and medium level entities (such as the cluster memory entity) are grouped together and connected to the highest abstraction level, also by free ports.

Free ports can be used to transcend more than one level of hierarchy. For example, the lowest level entities used for describing the mesh interconnection logic have free ports which propagate upwards to the highest abstract level (the cluster) to allow clusters to be joined together.

4.3 HASE Mechanisms for Describing Entity Behaviour

One of the major challenges in the implementation of the DASH simulation was the use of the new GUI based HASE entity specification interface. When defining a simulation in the HASE environment many attributes of the simulation need be specified. These include:

- **Entity layout design**: (such as the layout design discussed above in section 4.2) This includes specifying the bitmap used represent the entity on-screen, defining port icons and giving co-ordinates for dynamic state information placement (for example a cache unit in our simulation possesses a dynamically updated text display in the centre of the cache icon displaying read/write and hit/miss information).

- **Port and link specification**: All communication points for a given entity must be specified in terms of the protocol they will carry and their position on screen. Ports are also defined as being source or destination points, however this is at present just a labelling convention and in actuality there is no checking performed by HASE ensuring that each link comprises of a source and destination.

- **Global Simulation Parameters**: These parameters specify simulation parameters which are accessible by all simulation entities. They are useful in varying simulation conditions. For more detail readers are referred to [PHV95].

- **Each entity's operational parameters**: This is state information to be used locally by a given entity.

- **Port/link protocols**: having defined and joined ports together with communication links the simulation designer must specify the format of the data to be passed over the link.

Prior to the release of HASE 5.4/5 *all* this information was specified in a C++ file which was translated into an ObjectStore database at compilation time. This C++ file was known as an ADF (Architecture Description File) and required the programmer to have a detailed understanding of HASE's internal object hierarchy before he/she could proceed with simulation design/implementation work.

In a move to make the design aspect of HASE more user-friendly the HASE development team implemented a set of X11 menus and dialogs[2] to allow all the information previously specified in the ADF to be entered in a more intuitive way. This yielded the benefit that designers could start work under HASE without the previously required knowledge of HASE's internal operation.

4.4 Event Handling Strategy

When programming an entity's Sim++ body code various event handling strategies may be adopted. After a period of initial experimentation with HASE and Sim++, several

[2]Two of these dialogs are shown in sections 2.2.1 and 2.2.2.

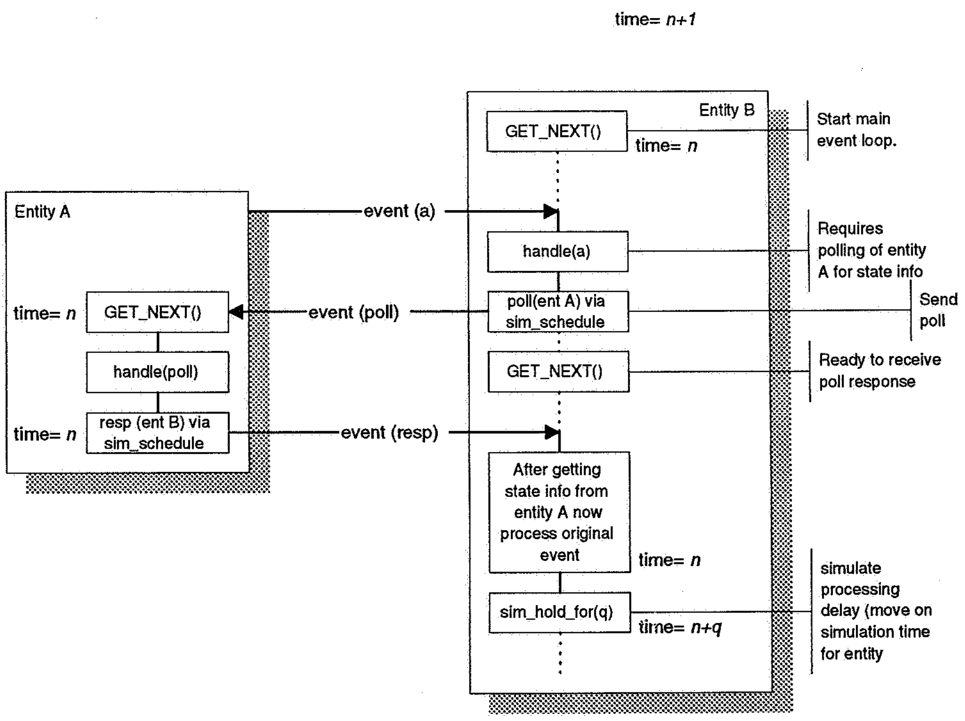
event handling strategies were found to work well in differing situations. However the final DASH simulation uses the two following generalised techniques only:

Fully Event-Driven Approach : In this scheme an entity's state information is set up at simulation time zero. Immediately after this initialisation the entity's code enters an infinite loop. This loop proceeds to wait for the next event sent to the entity (via *any* port) and then calls an appropriate event handler (which may in turn call further class methods to process the event). This technique ensures that there is only ever one call per iteration to the HASE macro `GET_NEXT()` and that each event must be fully processed, and a suitable simulation delay incurred, before the next iteration can take place.

This proved important in making debugging tolerable. Previous DASH prototype simulations often employed several calls to `GET_NEXT` per iteration. This proved 'messy' and meant that events could be received in the wrong section of simulation code if extreme care was not taken.

Zero Time Event Handling : The only exception to the above 'one `GET_NEXT` call per iteration' approach was made when dealing with 'zero-time' message exchanges. Often when programming the behaviour of a simulation entity one requires to gather some state information from another entity before a decision can be made as to the outcome of an event. Sim++ allows messages to be passed between entities with a delay of zero time which provides a useful basis for the gathering of the previously mentioned state information. For example, when implementing the MPbus state information is often required from all the attached processors (say, if they are waiting for bus control). By performing a poll of all processors (via some agreed protocol) in zero time the required information can be gathered. A side-issue worth noting is that the zero-time messages are sent using the Sim++ function `sim_schedule` to prevent the animator displaying the packets on the links used (if the animator were to do this some very strange displays would result!).

A polling example similar to that cited above is illustrated in Figure 4.4. Notice the way that through the use of `sim_schedule` the poll and response is done in zero simulation time units.



time= n+1
Entity B
GET_NEXT()
time= n
Start main event loop.
Entity A
event (a)
handle(a)
Requires polling of entity A for state info
time= n
GET_NEXT()
event (poll)
poll(ent A) via sim_schedule
Send poll
handle(poll)
GET_NEXT()
Ready to receive poll response
time= n
resp (ent B) via sim_schedule
event (resp)
After getting state info from entity A now process original event
time= n
sim_hold_for(q)
time= n+q
simulate processing delay (move on simulation time for entity

Figure 4.4: An Example of Zero Time Polling.

4.5 Implementation Prototypes

Before arriving at the final simulation model a number of prototypes proved useful in the construction and testing of major sections of the simulation. An overview of these prototypes' functionality is given below:

(a). Single Node : The initial prototype simulated a single processing node (MIPS R3000) with the following functionality:

- The ability to generate read/write requests to memory.
- Accurate primary and secondary processor data caches (direct-mapped utilising write-though and write-back policies respectively).
- A main memory unit.
- Dummy MPbus entity which passed requests from the single processing node to the memory unit.

(b). Four Incoherent Nodes and MPbus : After creating the first prototype the resultant model of the MIPS address generation box and its associated caches were available for use in the second prototype. The second prototype's goal was to connect four MIPS nodes together via a common bus. This entailed:

- Modification of MPbus ports.
- Understanding and modelling the pipelined operation of the MPbus.
- Defining bus arbitration protocols.
- An upgrade to the processing node's secondary level cache entity to allow it to understand the arbitration protocol.

(c). **Coherent Cluster** : The next logical step in the prototype evolution was to make the four processor cluster coherent according to the MESI snooping mechanism used in DASH. This work took considerable time and is discussed in detail later in this chapter. Other work undertaken for this prototype included the addition of state data panels on all caches (to give hit/miss information dynamically as a simulation animation progresses), implementing a bus 'shut-down' mechanism (to stop the simulation when all processors finish processing their input trace files) and updating the MPbus code to allow for the MESI protocol's use of cache-to-cache data transfers.

(d). **Cluster & Directory Control Logic** : The final simulation model developed was that of a four cluster DASH multi-processor. This required that entities representing the DC and RC boards be implemented. Also careful on-screen design work was required to allow a meaningful layout of the double mesh interconnection network. Unfortunately this simulation model was not fully completed although much of the infrastructure was in place at the end of the project's implementation phase. The level of completion attained is discussed later in section 4.6.8

4.6 Final Entity Descriptions

This, the largest section of the chapter, offers a description of each of the main simulation entities found in the final demonstration model.

The order in which entities are discussed follows closely the order in which they were implemented in the various prototypes discussed above.

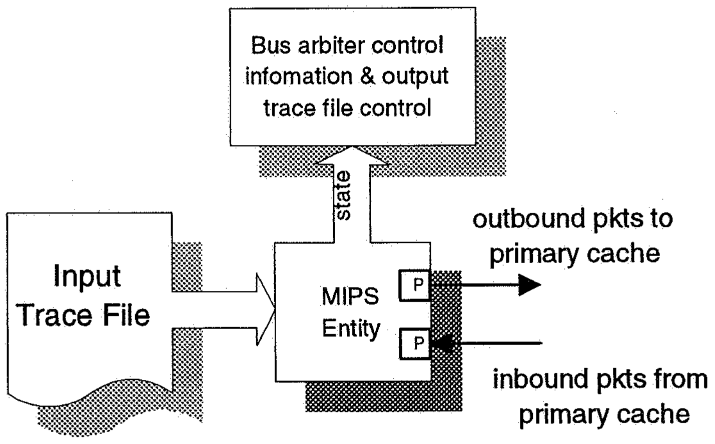
Except for illustrative fragments no Sim++ code listings for entities are given here[3].

[3]Readers wishing to examine the Sim++ body code of a typical entity are referred to Appendix E which gives the code for the MPbus arbiter entity.

4.6.1 MIPS Address Generation Unit

The MIPS entity (as illustrated in Figure 4.5) acts as an address generation box for the simulation. It communicates with the primary level cache via two HASE ports using one of the three general purpose protocols used throughout the simulation for inter-entity message passing.

We note at this point a convention used in the entity description diagrams (such as Figure 4.5) throughout this section. Any port based communication is represented via thin black lines connected to a port box (marked with a 'p' for a standard port and a 'f' for a HASE free port) whereas other communication (be it file I/O, state information being written to the trace file or communication through global variables) is denoted by a wide white arrow.



Bus arbiter control infomation & output trace file control
state
Input Trace File
MIPS Entity
P
P
outbound pkts to primary cache
inbound pkts from primary cache

Figure 4.5: The MIPS Address Generation Entity.

Apart from the address generation role of the MIPS entity, its other responsibilities include:

- **Communication** (via a global state variable) with the bus arbiter to indicate when the processor has exhausted the input trace file. This is achieved by all processors incrementing an integer counter when they have finished trace file processing. When the bus arbiter entity sees that the counter value is equal to the number of simulated processors the bus arbiter can stop. This artificial stopping of the arbiter is required as otherwise the MPbus would keep creating simulation events (via a regular poll) for ever and the simulation would never terminate.

- **The writing of state information** to the output trace file for later generation of simulation timing diagrams. The processor state changes between IDLE, BUSY and STOPPED and these changes of state must be reflected in the output trace file to allow timing diagrams to be constructed. This is achieved via use of the HASE `dump_state()` function at appropriate points in the Sim++ body code.

4.6.2 Primary Data Cache

Having defined the address generation mechanism the next entity to be created was the primary data cache; this was a natural choice as it is the next component in the path towards the cluster memory.

The primary cache was the first entity of any substance to be implemented. The primary cache is direct-mapped and operates a write-through policy. Aside from these DASH-dictated attributes, the demonstration tool cache was designed to allow the user to redefinable the entity's other operational parameters. For example, the cache entity allows the user to specify the size of the cache (in 16-byte lines) and the delay associated with a cache access.

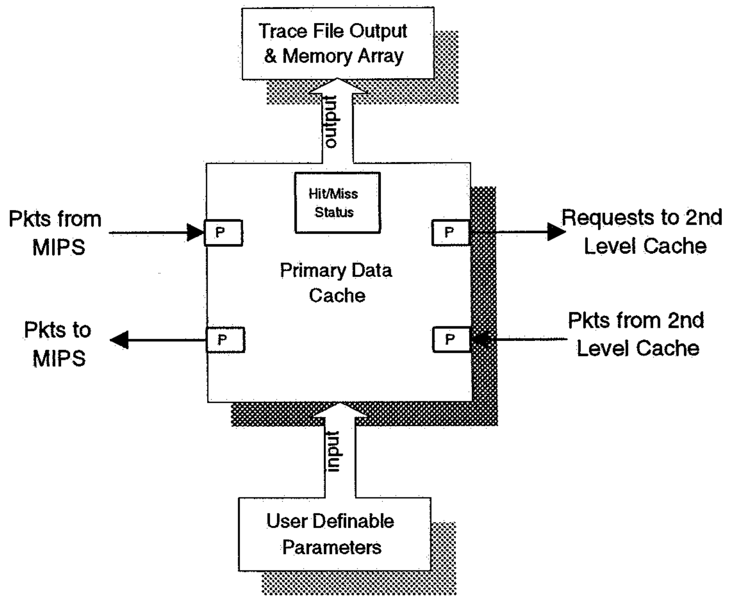

Trace File Output & Memory Array
output
Hit/Miss Status
Pkts from MIPS
P
P
Requests to 2nd Level Cache
Primary Data Cache
Pkts to MIPS
P
P
Pkts from 2nd Level Cache
input
User Definable Parameters

Figure 4.6: The Primary Level Processor Cache Entity.

The primary cache entity is shown in Figure 4.6. We see that the unit has four communication ports (two out, two in) and an on-screen display which changes its text value according to the outcome of the most recent access (it displays a two letter code stating the access type and hit status. For example a read miss would display `RM`).

The data structure central to the operation of this entity is a HASE memory array which represents the cache memory contents. Each line of this array holds a cache entry

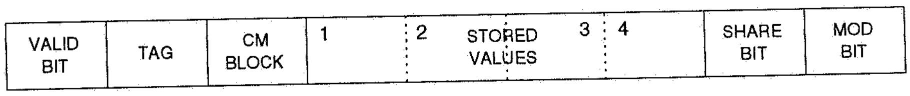


Figure 4.7: Primary/Secondary Cache Line Format.

(type `t_ca_line_struct`) in the format of Figure 4.7.

This format is shared with the secondary cache unit (see below); the only difference in use is that the primary cache never has need to use the share bit. On receipt of an incoming packet a table lookup is performed and validity bit and tag checks are made. If a hit occurs a delay is initiated before sending the result back to the MIPS entity. On a miss the packet is referred (after the miss delay) to the secondary cache entity.

Throughout the simulation the cache's state[4] (a value from an enumerated type `P_CACHE_STATE`) is recorded in the output trace file. These values are used in the construction of timing diagrams which show the state of the cache with respect to simulation time.

4.6.3 Secondary Data Cache

Following the data path towards the cluster memory unit we next encounter the secondary level processor cache. In terms of caching operation this entity is identical to that of the primary cache unit. Once again the user can define cache size and latency through the use of entity parameters.

However, this unit also hosts the snoopy-bus cache coherency logic and as such is one of the most complicated units in the cluster implementation. In order to understand the operation of the snooping mechanism it is also necessary to understand the operation of the MPbus arbiter and its associated communication protocols. For this reason the snoopy coherence mechanism will be discussed in the section describing the bus arbiter.

4.6.4 Node

The node (Figure 4.8) was the first composite entity encountered during simulation implementation. The node entity consists of the MIPS address generation unit and its associated primary and secondary level caches. The node entity provides abstraction from the processor cache level by showing a single entity on-screen from which one can observe addresses which have 'missed' at both cache levels (or values being sent to memory as a

[4]Appendix C includes a table detailing the state values used for each simulation entity

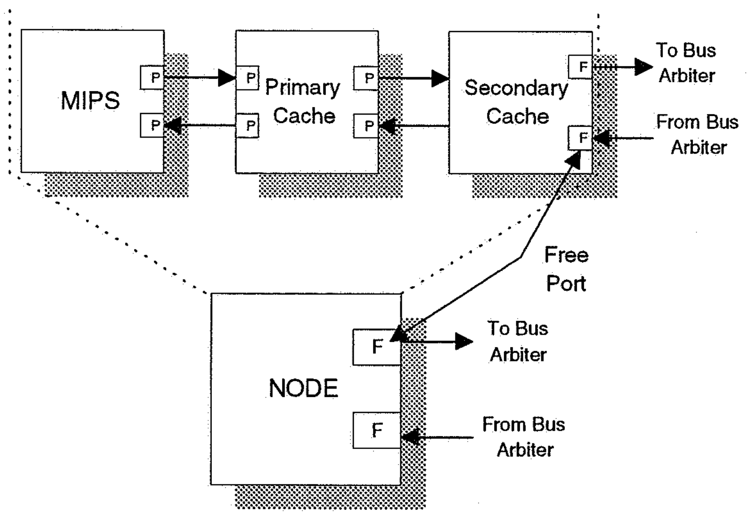


Figure 4.8: Composite Node Entity.

consequence of a write-back) being issued across the MPbus.

This entity also demonstrates the concept of HASE 'free-ports'. Notice that when the simulation is run at the 'node' level of abstraction the links between the caches and the MIPS address generator become `internal` to the node entity, however the ports interfacing the node to the MPbus are still visible. These are actually the same ports that connect the secondary processor cache to the bus. The ports are only defined within the secondary cache entity and automatically propagate to the node level when entities are grouped together. This propagation is forced on any ports which are 'free' (i.e. not connected to another entity) at the time of the group operation. Therefore the designer of a HASE simulation must be *very* careful when grouping entities together. He/she must ensure that all ports not required to act as 'free-ports' are already connected to their peer entities before the group operation is performed. If a mistake is made in this grouping it is currently (version 5.5) *impossible* to undo the group operation (this is a serious oversight which needs rectifying in future releases of HASE).

4.6.5 Communication Issues.

At this point we shall divert our attention from the operation of simulation entities in order to look more closely at the communication that takes place on the links which connect them. So far we have simply said that entities communicate by sending to, and receiving from, ports which are connected by links. We shall now consider the format of the data passed over these links.

When defining a simulation in HASE it is necessary to associate a packet type with a link[5]. This packet is usually designed to reflect the communication task in hand.

The remainder of this section highlights:

- The three packet formats currently used within the DASH simulation.
- The rationale behind the packet design.
- The scope within which the various packet types are used within the simulation model[6].

Node Level Packets.

This is the most basic of the three formats (referred to in the Sim++ code as packet type `t_p1_struct`) used in the DASH simulation. It was designed to carry address requests from the MIPS address generator through the processor caches and on to the MPbus. The packet contains the three fields illustrated in Figure 4.9.

Address	Read/Write	Inst/Data
`int`	`char[]`	`char`

Figure 4.9: Protocol P1 Structure.

The address field always carries the demanded/returned address. The second field was initially designed simply to indicate whether the address issue was with respect to a read or a write, however this field has become somewhat 'overloaded' and is now used to pass a variety of control/polling information between the secondary level cache and MPbus arbiter.

It would have been possible to define multiple packets types for this extra control/polling information however the new HASE interface method for specifying links contains something of a flaw. In order to change the packet format used on any already defined link the user must systematically delete *every* link in the simulation which will carry the newly defined packet and then re-build all links in order for the packet format changes to propagate down to the actual simulation links. This is a laborious task and rather than pursue this line of development it was decided that an overloading of the existing packet format would suffice.

[5]We note also that links can support multiple packet types although this facility was not used in the DASH simulation.
[6]Appendix B contains a scope map for the simulation packet types

The final field of this packet type indicates whether the address generated corresponds to a data or instruction request. Although only the data access path is accurately modelled in this simulation (see section 4.1.2) this field was built in for possible future use.

Cluster Level Packets

The second type of packet (type `t_p2_struct`) used within the simulation is pictured in Figure 4.10. Once again we see the address, read/write and data/instruction fields as featured in packet format `t_p1_struct` along side a new field `device_id`.

Address	Read/Write	Inst/Data	Device ID
int	char[]	char	int

Figure 4.10: Protocol P2 Structure.

This field reflects the fact that this protocol is used within the MPbus logic for routing address requests within a cluster. When a packet reaches the MPbus it enters a multi-processor environment for the first time. As such it needs to be identifiable as belonging to a particular processor. When the bus arbiter receives a packet from one of its five input ports (4 processor ports + 1 PCPU (Pseudo-CPU) port) it translates the `t_p1_struct` packet into a `t_p2_struct` and assigns an integer label to the `device_id` field of the packet reflecting the source of the request.

Inter-Cluster Level Packets

These packets are used at the highest level of abstraction and therefore must reflect the scope of the entire DASH prototype – this means being able to identify not only an individual processor but the cluster to which it belongs.

We shall see in later sections that the first two packet designs had fields 'overloaded' to add support for protocols that were not envisaged at the time of their design. To avoid this situation occurring again the inter-cluster packet design includes three extra, as yet unused fields, for future expansion. The inter-cluster packet format is shown in Figure 4.11.

Address	Read/Write	Inst/Data	Device ID	Cluster ID	Reserved1	Reserved2	Reserved3
int	char[]	char	int	int	char	char	int

Figure 4.11: Protocol P3 Structure.

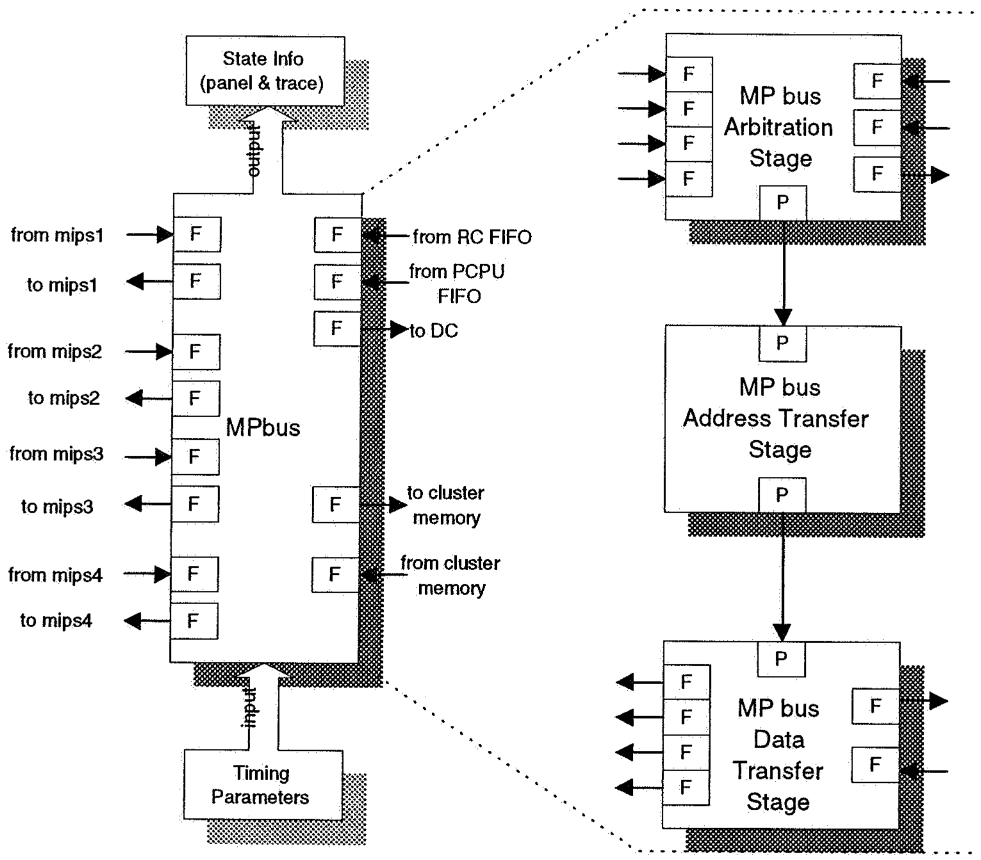


Figure 4.12: MPbus Composition.

4.6.6 MPbus

The MPbus is one of the most complex entities in the DASH simulation. It is responsible for displaying a large amount of state information detailing the on-going operation of the snoopy-bus protocol as well as carrying out the conventional tasks of bus arbitration, address and data transfer.

In the early prototype DASH simulations the MPbus was represented by a single icon and attempted to deal with all of the above functionality. This proved to be a very complex entity with much scope for error in programming. Another disadvantage of this 'single entity' approach was that it presented a 'black box' view of the bus to the user.

It was decided that the bus would be better implemented as a series of entities, each responsible for some part of the bus functionality. It seemed logical that these divisions should follow the pipelined operation of the bus. This would give the user of the simulation a better insight into the bus mechanisms used within DASH as well acting as a demonstration of the pipelining principle. The final MPbus entity is composed of three lower level entities (shown in Figure 4.12)

Once again we can see the allocation of free and normal ports and the fact that user definable parameters can be used to alter the timing characteristics of the pipeline stages.

A minimum bus transaction has a latency of seven bus cycles[7]. The composition of these transaction cycles is:

1. Arbitration (1 cycle)
2. Address transfer (3 cycles)
3. Data Transfer (4 cycles)

We note at this point that the MPbus classifies transactions as being one of three kinds – a cache transaction, a DMA transaction or an I/O transaction. Given the limited time available for this project only the cache-based transactions were modelled.

Next we will identify the tasks performed by the three MPbus sub-entities.

MPbus Arbitration Entity

Arbitration of MPbus control is performed on a fair, round-robin basis and is performed in a single bus cycle. The method used to implement this system in the DASH simulation model had the following attributes:

- The use of 'zero-time' polling of bus masters (see section 4.4).
- The use of a protocol based on an overloaded `t_p1_struct` packet format.

The arbitration protocol used is as follows:

(a). Poll Masters : Upon the start of a bus arbitration cycle the arbiter polls all bus masters (4 secondary caches and PCPU/RC) to see if any require service. This is done by calling `poll_all()` which sends a packet (in zero time) to all masters containing a `read/write` field of value `G` (Get poll response).

(b). Masters Receive Poll : On receipt of the poll packet masters wishing to claim the bus return a packet containing a `read/write` field value of `Y` (or `N` if they do not require bus control this cycle) via function `do_mp_g()`.

[7]A detailed set of timing diagrams for the MPbus can be found in [E.L92]

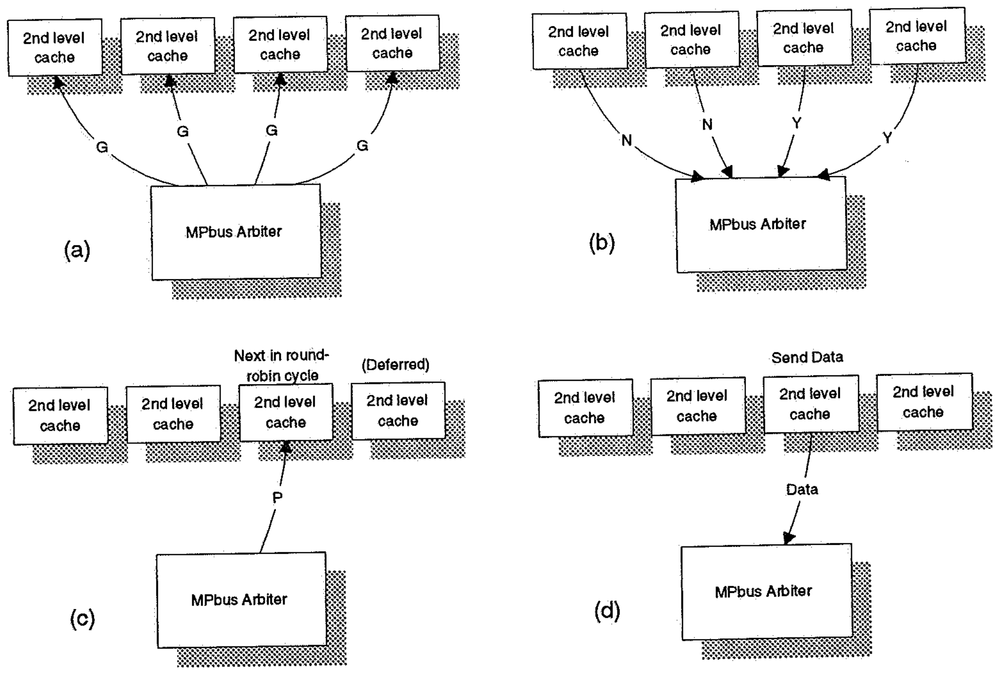


Figure 4.13: MPbus Arbitration Protocol.

(c). Arbiter Receives Votes : The arbiter counts the number of **Yes** votes returned. If no masters require service the arbitration delay (1 cycle) is made before starting to poll again. However if masters require service the arbiter selects which master to grant access to via the function `grant_bus()` (a round-robin method of allocation applies). This function call sends a permission packet (one with a `read/write` field value of `P`) to the selected master.

(d). Master Receives Permission : Upon receipt of the permission packet the master sends the address request to the arbiter using the standard `t_p1_struct` packet format. The arbitration cycle starts again.

A typical arbitration cycle adhering to the description given above appears in Figure 4.13.

Next we shall consider another major role played by the MPbus arbiter entity – that of co-ordinator for the snoopy-bus protocol. As the snoopy bus protocol is executed by all snooping caches in the cluster there is a need for a central entity to collate snoop data and present it in a meaningful format to the user of the simulation. Although this is unrealistic in terms of the real architecture's operation it offers the advantage of a central control point for the snoopy protocol. Also it is possible to implement the protocol in such a way that the user of the simulation would perceive the protocol to be running as in the original architecture. As long as this is possible we can justify the use of the bus arbiter as a co-ordinating entity.

The snoopy-bus protocol used in the DASH simulation is identical to that of the actual architecture. The protocol is the MESI Illinois protocol as outlined in [PP84]. We shall examine the protocol in two sections. Firstly we will consider the coherency algorithm itself, then the simulation based communication protocols which support its operation.

The flowcharts of Figure 4.14 detail the general strategy for reads and writes in the DASH system. We note that a cache line may be in one of four states:

1. **Invalid**: Block does not contain valid data.

2. **Exclusive-Unmodified**: (Excl-Unmod) No other cache has this block. Therefore the data in the block is consistent with main memory.

3. **Shared-Unmodified**: (Shared-Unmod) Some other caches *may* have this block. Data in this block is once again consistent with main memory.

4. **Exclusive-Modified**: (Excl-Mod) No other cache has this block. Data in the block has been modified locally and is therefore inconsistent with main memory.

From Figure 4.14 we note:

- The use of cache-to-cache transfers. Although cache-to-cache transfers do nothing to reduce the latency of local memory they do allow sharing of data from remote clusters between processor caches. This means the set of local secondary level processor caches act as a cluster cache for remote memory.

- In section 3.2 we noted that snoopy-bus protocols can be classified by their use of either a `write-broadcast` or `write-invalidate` approach to the handling of `stale` data within the system. The MESI Illinois protocol uses a `write-invalidate` scheme.

In the DASH simulation the flowchart algorithms of Figure 4.14 are implemented in the locality of the secondary processor caches as in the real system. However the bus arbiter entity is also involved in the coherency mechanism. The protocol[8] supporting the MESI snooping protocol in the DASH simulation is as follows:

[8]This protocol is once again executed in 'zero-time'.

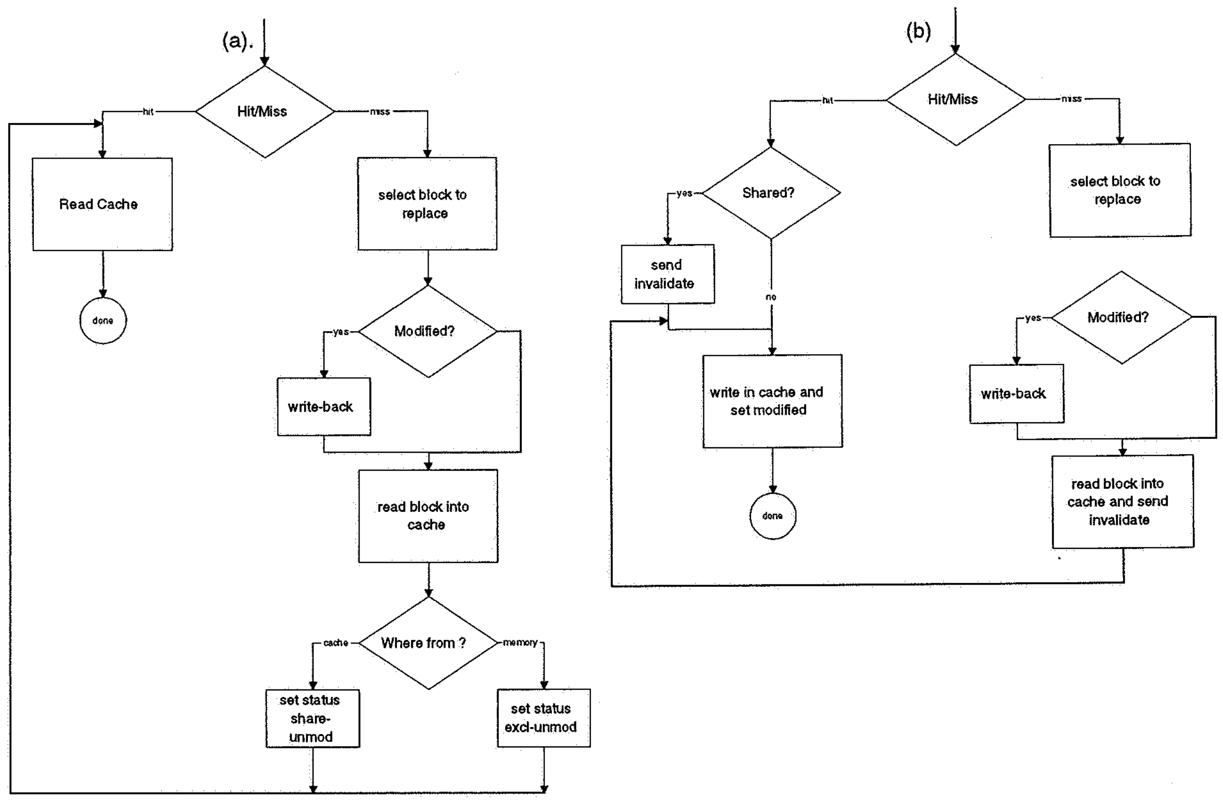


Figure 4.14: MESI Illinois Protocol (a).Read, (b).Write

(a). Reception of Read/Write Request : The processor which was granted the bus at the last arbitration phase transmits its address request to the bus arbiter entity. This transmission is of a standard `t_p1_struct` packet.

(b). Broadcast of Request : The arbiter receives the read/write request and immediately broadcasts the message back on the bus (the broadcast is implemented as four separate messages, one per processor, sent in zero time). This special message is an overloaded `t_p1_struct` type packet containing an 'r' or 'w' in the instruction/data field and a 'S' in the read/write field. The 'S' indicates a snoop-probe message.

(c). Execute MESI algorithm : On receipt of the snoop-probe the secondary caches look up the appropriate line in their caches and execute the MESI algorithm. This will adjust the shared/modified bits according to the address in the received packet. The results of this snoop are then transmitted back to the bus arbiter[9].

(d). Collate Results The arbiter gathers the response packets and generates a text based description of the snoop outcome which is displayed via a 'snoop panel' on

[9]There are four possible response packets/ For the sake of brevity these are not given here but details regarding the packet format of this response are given in Appendix B.

the arbiter's icon (Figure 4.15 below). The panel has four information sources:

1. The display on the left-hand side of the panel changes with respect to the bus master granted access to the bus (an arrow indicates the current master).
2. The information box at the top of the panel displays whether an access is a read, write or write-back.
3. The small information box at the bottom of the display indicates where the request's result will come from. This can be either cluster memory or another cache (because of the MESI protocol's use of cache-to-cache transfers).
4. The most complex part of the snoop information panel is the four line display in the centre of the icon. These lines give a summary of the snoop activity that occurs in each of the second levels caches. For example if a cache holds an exclusive-unmodified copy of a piece of data and another processor then reads the same item, the display line for this cache would read `SU-(EU)`. This code means the new cache line bit settings (after the snoop activity) reflect a shared-unmodified state. The value in brackets represents the 'before' status of the cache line (i.e. exclusive-unmodified).

   The panel is also used to display the message `INVALIDATION SENT TO ALL CACHES` when a write-invalidate action occurs.

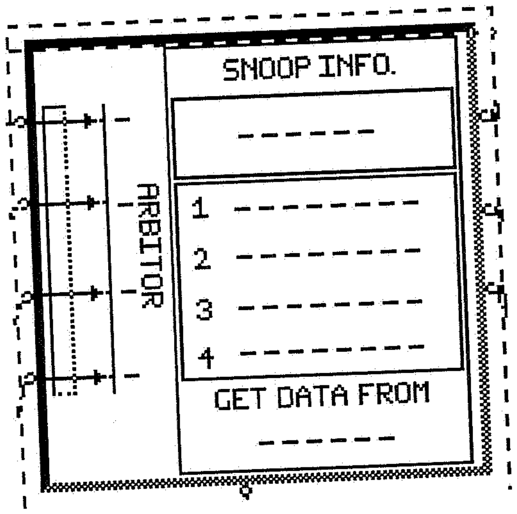


Figure 4.15: Snoopy-Bus Protocol Information Panel.

The snoop stages previously outlined can be seen in Figure 4.16.

The description given so far covers the most common cases of snoopy-action. However we must still examine the less frequent operation of an invalidation. This can be sum-

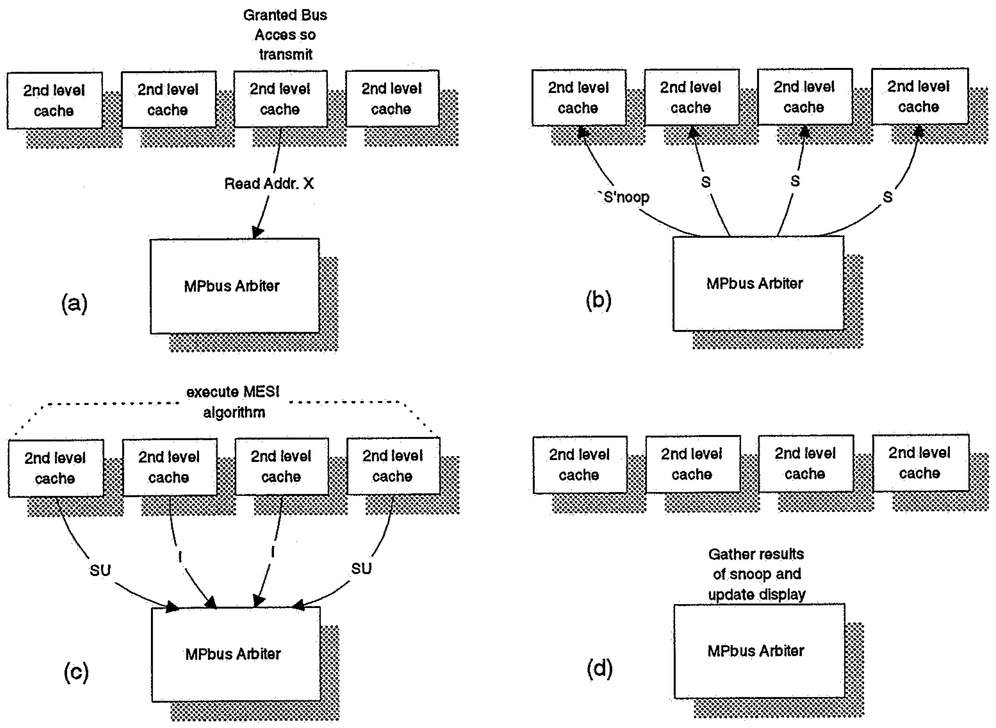


Figure 4.16: Zero Time Protocol for MESI Snooping Algorithm.

marised as follows:

1. The bus arbiter receives a write request.

2. The address to be written is broadcast with an invalidation signal to all caches `except` the one making the write.

3. Caches receiving the invalidate packet check their contents for a match of address with the requested invalidation. If a match is found the caches appropriate valid bit is reset.

The invalidation mechanism is illustrated in Figure 4.17

The final function performed by the MPbus arbiter entity is that of simulation shutdown. During normal operation the bus continually polls masters checking for bus access requests. This polling continues even when each of the processors has exhausted its input trace file the arbiter will continue this cycle. To avoid the arbiter generating these poll events forever a global simulation variable is used to control shutdown. After each processor in the system exhausts its input file a variable `shut_down` is incremented. At the start of each bus arbitration cycle a check is made to see if the value of `shut_down` is equal to the total number of processors in the simulation. If this test returns true then

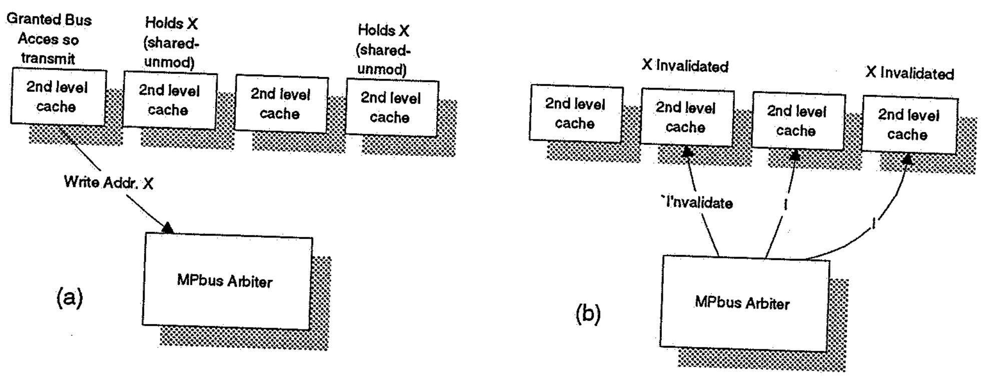


Figure 4.17: Invalidation Mechanism Used in DASH Simulation.

the entity exits its main body-code loop and no more simulation events are placed in the event queue by the bus arbiter.

MPbus Address Transfer Entity

The next section of the MPbus to be examined is the address transfer entity. This entity is responsible for simulating the 4 bus cycle delay present in the address transfer stage of the actual architecture.

The operation of the address transfer unit is described below:

Arbiter/Address Transfer Entity Handshake : When the arbiter has finished its operational cycle (i.e. made the processing delay of 1 bus cycle) it needs to send the requested address details to the address transfer unit. This should only be done if the address transfer unit is idle. A simple protocol to test the status of the address transfer entity is used as follows:

1. The bus arbiter sends a probe packet to the address transfer unit and blocks waiting for a response.
2. The address transfer unit (when ready to accept a data packet) examines its input port for a probe message. On finding a probe it sends a permission packet back to the arbiter.
3. The arbiter now sends its data packet and continues its operation.

This handshake protocol is shown in Figure 4.18.

Directing Data Packet : After the handshake phase the address transfer unit decides whither the address request is bound. This can be either the cluster memory unit or, in certain cases of the MESI protocol, another cache. The packet is directed

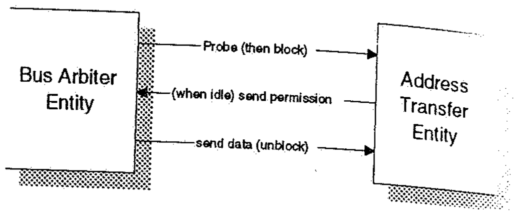


Figure 4.18: Handshake Protocol Used Between MPbus Components.

toward the appropriate output port. Following this the operational delay of the address transfer phase is simulated and the address transfer cycle then restarts.

MPbus Data Transfer Stage

The data transfer entity returns request results to the issuing processor after they have been processed. The same handshake protocol used between the arbiter and address transfer entities is used to co-ordinate incoming and outgoing packets. This entity takes its input from the cluster memory unit or address transfer unit (the latter indicating that the data has come via a cache-to-cache transfer). The operational delay of the data transfer phase is now simulated. Finally, on examining the input data packet's `p2_dev_id` field the destination node is identified and the data forwarded appropriately. The data transfer cycle now restarts.

4.6.7 Cluster Memory

The cluster memory is relatively simple in design. Because the simulation is only concerned with modelling the effects of read/writes throughout the system (and not the contents of memory locations) no actual storage needs to be modelled other than that present in the processor caches (and in these only addresses need be stored). Therefore the operation of the memory unit is as follows:

1. An incoming packet is taken from the in-bound port.
2. The packet is classified as read/write
3. The on-screen memory state information is updated.
4. The memory delay is simulated
5. The 'result' packet is forwarded to the data transfer entity.

4.6.8 RC Board

Due to reasons explained below (in section 4.7) the implementation of the inter-cluster communication logic is only partially completed. Therefore the discussion of the DC and RC board components will be limited to that of the design work undertaken and the implementation work completed so far.

The RC board consists of three components – the reply controller, the PCPU and the request/reply Y-dimension of the interconnection network.

The PCPU and reply controller each have a FIFO connecting them to the Y-bound inter-connection communication logic (see Figure 3.9 for details). These FIFOs have been implemented and are identical for all DC and RC board cases. They operate as a simple buffering mechanism and utilise the handshaking protocol used to connect the MPbus components together to connect them to neighbouring entities. The FIFOs are uni-directional and feature one input and one output port.

The PCPU is not yet fully implemented. As in the original DASH architecture the PCPU will closely imitate the behaviour of a processing node. As this is the case it is envisaged that a large proportion of the secondary processor cache Sim++ code could be reused when implementing this entity.

The reply controller is also similar in structure to a secondary processor cache in that it is required to snoop on the MPbus. Internally the reply controller currently has a HASE memory array designed to hold the information necessary to track remote requests. A counter is also featured for each table entry to act as a remote write-request invalidation counter. Apart from this internal data store and the entities communication links the reply controller is currently otherwise unimplemented.

The highest level of abstraction in the DASH simulation features four DASH clusters. This number of clusters was decided upon because of screen space limitations and the extra complexity of working with HASE templates (a HASE template allows an entity to be reproduced across a variable size interconnection network - see [PHV95] for details). Having a fixed size interconnection network made the task of implementing the inter-cluster routing mechanisms relatively simple.

After the final screen layout had been performed each routing entity (in the case of the RC board the request/reply logic appertaining to the Y-dimension of the interconnection network) had assigned to it a state variable `clus_id`. This value is unique for all clusters. It is envisaged that a general purpose routing handler could simply direct messages to other clusters by selecting a hard-coded transmission path via a `switch` statement according to the destination `clus_id` value in the packet to be routed.

4.6.9 DC Board

The DC board consists of three major components. These are the performance monitor, directory controller and request/reply logic for the X-dimension of the interconnection network. The first of these components, the performance monitor, is not considered in our simulation (its original purpose was to aid analysis of the DASH prototype).

The X-dimension interconnection network logic is in an identical state to that of the Y-dimension logic described above.

The directory controller entity is currently in much the same state as the reply controller having been defined in terms of the HASE interface but lacking any real substance in Sim++ functionality. All communication ports and links have been defined and an internal HASE memory array has been set up to represent the directory memory. The bus access required by the directory controller could be implemented by reusing the bus access code found in the secondary processor cache.

4.7 Completion Status

As noted in the description of the DC and RC boards the simulation is not yet fully implemented, this is largely due to a combination of time-constraints and problems encountered during the project lifetime.

One of the main problems was the underestimation of the time required to become familiar with HASE and Sim++. This was initially estimated to take two to three weeks of project development time however in actual fact the time taken was close to size weeks.

Other problems encountered included:

- Coding bugs in some of the HASE menus (e.g. it was possible to redefine primitive data types from within HASE which rendered the HASE generated entity source code useless). These problems were fixed by Pat Heywood of the HASE project.
- Being confronted with the somewhat cryptic error messages generated by ObjectStore. Often these could be trapped back to the HASE error handler routine whereby a fix could be initiated. However on several occasions experimental databases were rendered useless and a full rebuild of the data contained within them was required.

- Another problem encountered was that of server load. As the HASE system was running on a shared ObjectStore server there were certain periods during implementation that performance degraded very badly thus making any real progress difficult.

- An interesting problem encountered was that there were no previous 'large' simulations in existence that had been created using the new HASE interface. Because of this HASE project management was dealt with on an ad-hoc basis which often resulted in the reworking of an entities implementation once experience with the system had been gained.

The problems listed above all contributed to project timetable slippage, however the major problem encountered was that of ObjectStore database degradation. Over time ObjectStore databases would grow to a very large file size (this has recently been traced to a HASE routine responsible for garbage collection). This problem is currently being fixed by the HASE development team. Other problems as yet untraced resulted in corrupt data being entered into the simulation database. This rendered later simulation prototypes *useless*. The only way around this problem was to rebuild the database from scratch This is a very time consuming process and future releases of HASE need to address the current requirement for a mechanism independent of ObjectStore to allow databases to be reconstructed automatically.

We note that this database reconstruction was not a problem with older ADF based HASE simulations where the C++ ADF could be recompiled resulting in the generation of a new ObjectStore database.

Appendix B

DASH Simulation Communication Protocol Summary.

B.1 Packet Formats

This section illustrates the three packet formats used throughout the DASH simulation. Details regarding the design of these packet formats are given in section 4.6.5.

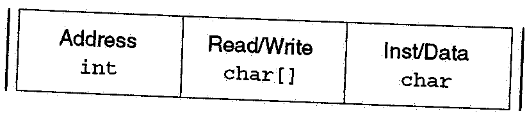


Figure B.1: Packet (P1) Structure Diagram.

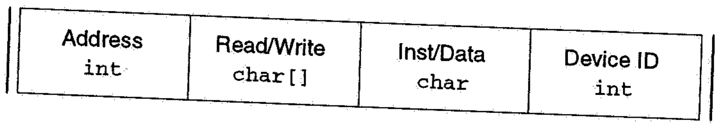


Figure B.2: Packet (P2) Structure Diagram.

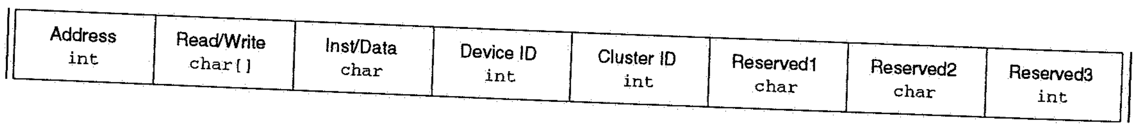


Figure B.3: Packet (P3) Structure Diagram.

B.2 Protocol Message Definitions

Table B.1 lists the field values used for signalling information in the various 'zero time' protocols used within the simulation.

Read/Write	Inst/Data	Description
r	unused	Address to read
w	unused	Address to write
u	unused	Update (write-Back)
G	unused	Probe to change bus master
P	unused	Permission to send granted
Y	unused	Positive Probe Response (ACK)
N	unused	Negative Probe Response (NAK)
S	R W	Snoop Probe (read) Snoop Probe (write)
I	unused	Snoop Response - Invalid
E	U M	Snoop Response - Excl-Unmod Snoop Response - Excl-Mod
S	U	Snoop Response - Shared-Unmod

Table B.1: Zero Time Protocol Field Codes.

B.3 Packet Scope Map

Figure B.4 offers a 'packet format' map detailing the use of the various packet types across current simulation entity links.

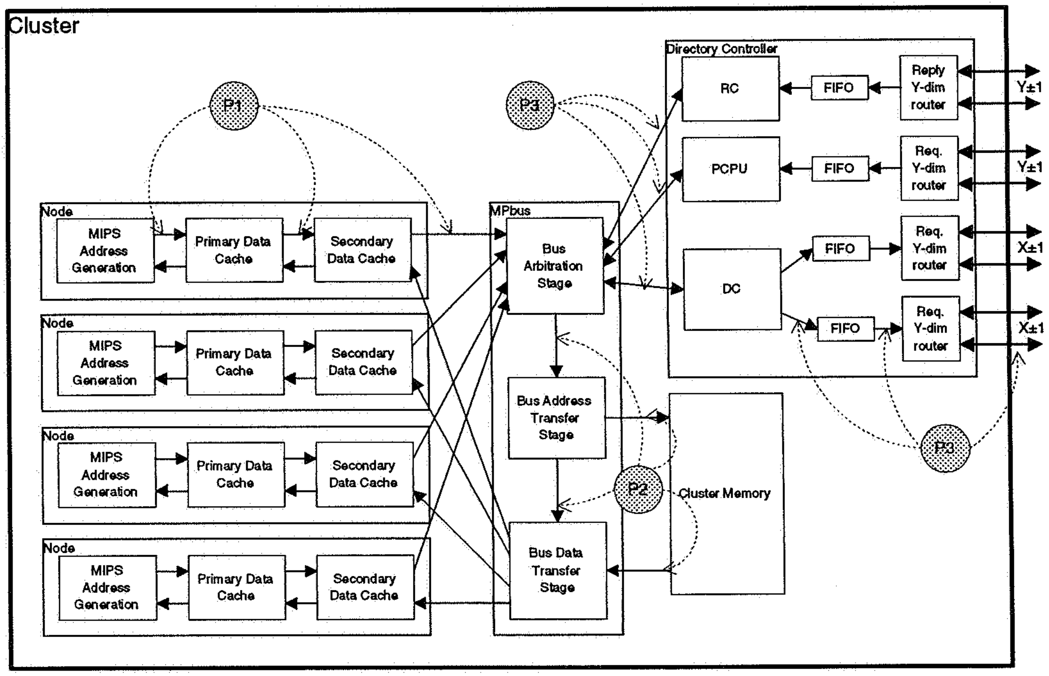


Figure B.4: Scope Map of Packet Use in Simulation.

The Stanford DASH Multiprocessor

Motivation.

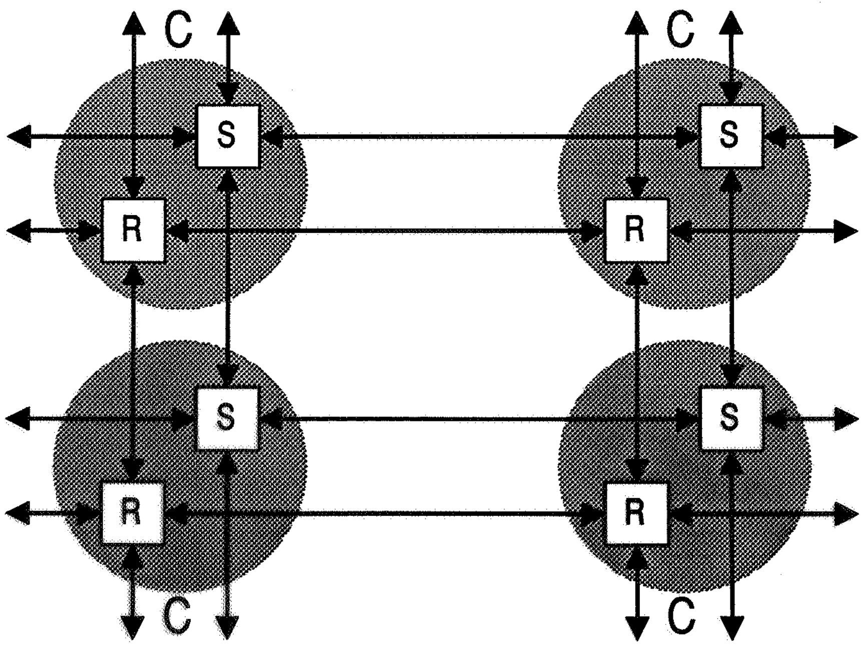
- Project started at a time when no large-scale shared memory multiprocessors with cache coherency existed.
- Single address space should facilitate ease of programmability.
- Desire to use hundreds of low-cost processors in coherent multiprocessor.
- The DASH architecture was therefore initiated as a feasibility study.
- Since its inception other architectures provide similar facilities
- For example:
    - The KSR-1
    - IEEE SCI (Scaleable coherent Interface)

Cache-Coherency Mechanisms.

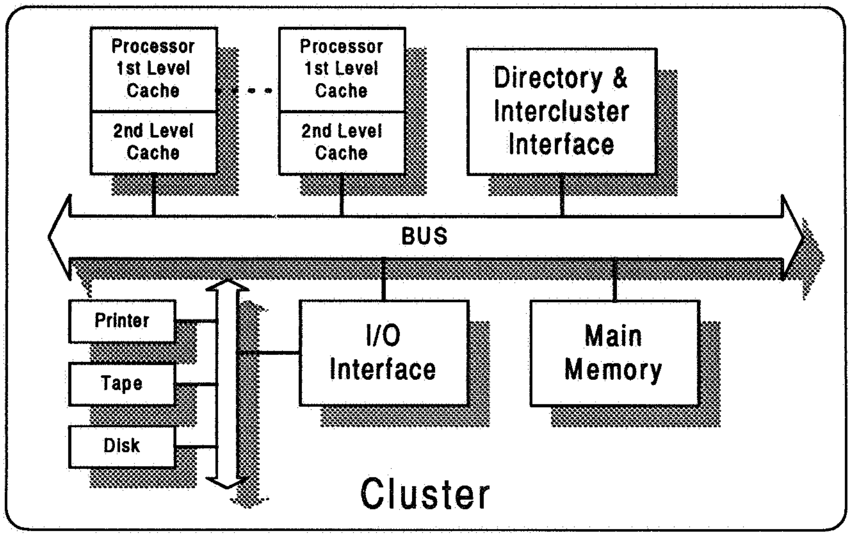
- In DASH there are two mechanisms employed for ensuring cache coherency:
    1. Snoopy-Bus
        - Used within processing 'clusters'
        - Snoopy is based on broadcast medium (bus) so scalability limited. - bus saturates after few processors added (limited bus/snooping bandwidth).
    2. Directory Based
        - In order to overcome the processor limitations enforced by the snoopy-bus based protocol a second coherency mechanism based around a directory (directory based mechanisms actually predate snoopy mechanisms and were to an extent 'rediscovered' for DASH).
        - A global directory is distributed between clusters.

Hardware Topology

- So, at most abstract level:-



- Set of processing clusters (each with several CPU's) connected double mesh interconnection network:
    - 'S' for sending remote memory requests.
    - 'R' for receiving requests.
- Cluster based on Silicon Graphics 4D/340 (itself a small scale multiprocessor), allowed for rapid development of cluster hardware (i.e. most of work already done).


Processor 1st Level Cache
2nd Level Cache
Processor 1st Level Cache
2nd Level Cache
Directory & Intercluster Interface
BUS
Printer
Tape
Disk
I/O Interface
Main Memory
Cluster

- Finally, there exists within this department a HASE based Simulation of the DASH architecture which helps demonstrate the coherency mechanisms used.