Simulating the DASH Architecture in HASE*

L. M. Williams and R. N. Ibbett
Computer Systems Group
Department of Computer Science
University of Edinburgh, Edinburgh, EH9 3JZ, UK
lmw@dcs.ed.ac.uk, rni@dcs.ed.ac.uk

Abstract

HASE is a Hierarchical computer Architecture design and Simulation Environment which allows for the rapid development and exploration of computer architectures at multiple levels of abstraction, encompassing both hardware and software. The user interacts with HASE via an X-Windows/Motif graphical interface, and one of the main forms of output is an animation of the design window.

The DASH architecture was designed to prove the feasibility of building a scaleable high performance machine with multiple coherent caches and a single address space. The HASE simulation therefore concentrates on implementing the cache coherency protocols, and the animator has been used to check that the simulation conforms to the architecture. Future work will involve performance checks of the simulator, and thence possible architectural enhancements.

1. Introduction

HASE [5] is a Hierarchical computer Architecture design and Simulation Environment which allows for the rapid development and exploration of computer architectures at multiple levels of abstraction, encompassing both hardware and software. It is intended for use both as a research tool, allowing "what if?" experiments to be performed on computer architectures, and as a teaching/demonstration tool. The simulation described here was inspired by Hennessey's video on the DASH architecture [2], and was therefore designed as a demonstration system, though the model has potential to allow research experiments to be performed.

*The HASE project is supported by the UK EPSRC

2. HASE

The ideas for HASE grew from a simulator built for an MC88000 system [11], written in occam and run on a Meiko Computing Surface at the Edinburgh Parallel Computing Centre. However, since the components of a computer can be treated very naturally as objects, HASE itself is based on the object oriented simulation language Sim++ [6] and an object oriented database management system, ObjectStore [9].

The environment includes a design editor and object libraries appropriate to each level of abstraction in the hierarchy, plus instrumentation facilities to assist in the validation of the model. The system can thus be set up to return event traces and statistics which provide information about, for example, synchronisation, communication and memory latencies, cache hit/miss ratios.

The user interacts with HASE through an X-Windows/Motif graphical interface. Many complex systems of interacting components can be more easily understood as pictures rather than words, and in computer architectures the dynamic behaviour of systems is frequently of interest. HASE therefore allows users to view the results of simulation runs though animation of the design window.

HASE is being developed specifically for use in the ALAMO project (Algorithms, Architectures & Models of Computation: Simulation Experiments in Parallel Systems Design), but is also been used in a number of other projects. The ALAMO project itself involves an investigation of the use of the H-PRAM model of computation [4] as a bridging model for parallel computation. Algorithmic skeletons are written in an H-PRAM notation, compiled on to simulation models of parallel architectures created in HASE, and the performance metrics of various hardware architectures investigated. The goal is to determine the properties of cost-effective systems based on scalable architectures to provide efficient support of the H-PRAM model.

HASE is also being used to evaluate the performance of a variety of multiprocessor interconnection networks. This involves setting up a simulation testbed containing simple processor models which can generate network activity corresponding to that found in standard benchmarks used to evaluate real parallel systems, and instantiating different network models in the testbed.

An example of a different type of project is the On-Line Teaching System for Computer Architecture. This project has produced a demonstration to aid students in the understanding of computer architecture. The demonstration involves playing back a pre-run simulation of the DLX architecture [3] which both animates the diagram of the architecture and displays a sequence of text windows which explain what is happening in the simulation. The simulation deals with hazards, multicycle operations, scoreboarding, *etc.*

3. The DASH Architecture

The DASH architecture [7, 8] was built in the Computer Systems Laboratory at Stanford University. The main motivation underlying its inception was a desire to prove the feasibility of building a scalable high performance machine with multiple coherent caches and a single address space. The intention was to produce a parallel architecture offering both ease of programmability (facilitated by the single-address space) and very high performance (by using hundreds to thousands of high performance (low-cost) processors).

The DASH hardware is organised hierarchically as shown in Figure 1. At the bottom of the hierarchy there is a set of processing nodes, grouped together in *clusters* of four and connected together via a common bus (the lower level interconnection mechanism). These buses are in turn connected together by a (dual) interconnection network. The 4D/340 network interface connects to two independent worm-hole routed meshes, one used for outgoing memory requests and the other for inbound replies.

DASH clusters are based upon a modified version of the Silicon Graphics POWER Station 4D/340 [1], a block diagram for which is given in figure 2. The major components shown are:

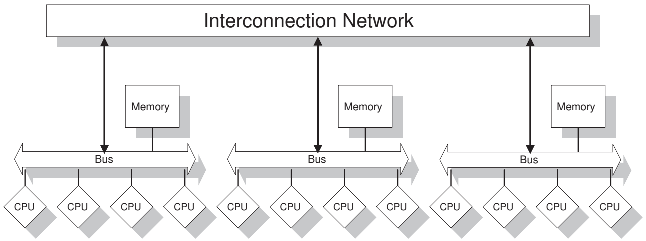

Figure 1. DASH Prototype Interconnection Hierarchy.

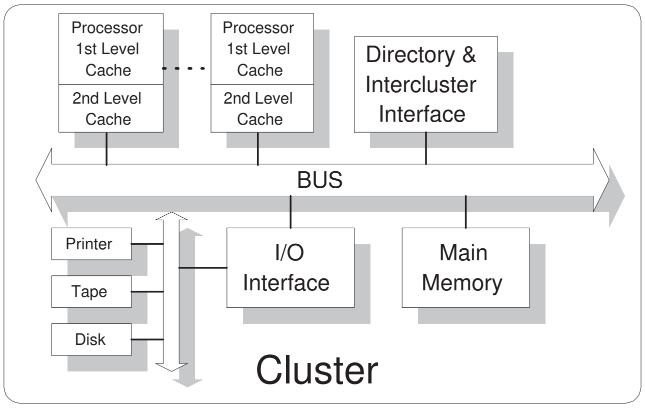

Figure 2. The SGI 4D/340 Hardware Configuration.

- **Four MIPS R3000 processors** each running at 33MHz (Figure 3). Each processor has two levels of cache memory. The first level has a 64 KByte instruction cache and 64 KByte write-through data cache; the second is a 256 KByte write-back cache. Both caches are direct mapped with 16-byte cache lines. The first level caches match the processor speed (33MHz) whilst the second level cache matches the bus speed (16MHz).
- **The MPbus**, common to all four processors and utilising a snoopy-based cache coherency protocol. The MPbus is pipelined but does not support split transactions.
- **An I/O interface** for general purpose device handling.
- **Memory** shared between the processors and forming part of the global address space.

3.1. Cache Coherence Protocols

The DASH architecture uses a two-level cache coherence protocol, a snoopy bus protocol at the intracluster level and a directory based system at the intercluster level. Each cache holding a copy of some physical memory block also contains information regarding the block's usage, throughout the system. A block may be **Invalid**, **Exclusive-Unmodified**,

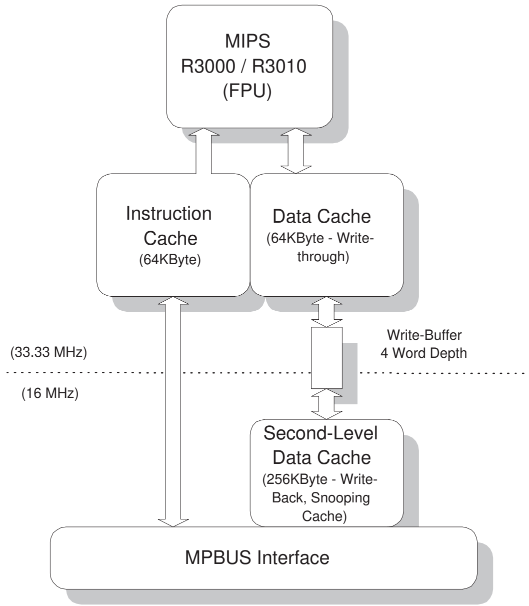

Figure 3. The MIPS R3000 Cache Design.

Shared-Unmodified or **Exclusive-Modified**. The cache controller in each node monitors the bus, examining memory requests from all other caches. Depending on the nature of these requests, the snooping caches update their contents according to a MESI[1] (Illinois variant) cache coherency protocol [10, 12], which uses a *write-invalidate* update scheme. The MESI protocol allows cache-to-cache transfers; thus if a local processor requests data from a remote cluster, and this data is held in an unmodified state in another local cache, the transfer of data can be made locally rather than involving a remote memory request (along with its associated delay). In effect the local caches form a composite cache for remote memory locations.

Snoopy based protocols do not scale well to large numbers of processors, however, because all caches must be connected via a common bus. Any message placed onto a bus takes the form of a broadcast, and all processors must share the finite bus bandwidth, thus limiting the scalability of the system. The DASH architecture overcomes this limitation by interconnecting a number of buses via a mesh interconnection network, but has to use a directory based cache coherency scheme at this higher level.

Directory-based protocols revolve around the use of a single directory which keeps information regarding the status of every block in main memory. This information is maintained via the use of presence bits. For every block there is one presence bit for every cache in the system; this bit is set if the corresponding cache currently holds a copy of the block. In addition to the presence bits, each directory entry also contains state bits which reflect whether or not main memory is consistent with cache memory values for a given block.

[1]The MESI acronym is derived form the possible states of a cache line (*i.e.* Modified, Exclusive, Shareable or Invalid.)

Obviously the directory itself can form a central bottleneck within the system. However the directory can be distributed across the interconnection network allowing different requests to go to different sections of the directory, thus reducing contention. The status of any single memory block is found at one unique location within the distributed directory. For this reason a distributed directory mechanism scales better than one based on bus-snooping for large numbers of processors. One disadvantage of a directory based system is the amount of storage required for the directory (one entry per memory block). In a snoopy scheme storage is only required for those blocks present in the cache.

The DASH prototype employs a custom-built directory controller and network interface to connect the 4D/340s to the interconnection network. The directory controller forms part of the distributed directory. As the MPbus does not support split transactions, a bus retry mechanism is also used to allow remote memory accesses to take place whilst local memory accesses are being processed. This is because of the relatively long time taken for a remote memory access to be satisfied.

Figure 4 summarises the cache coherency methods used within the DASH hardware hierarchy.

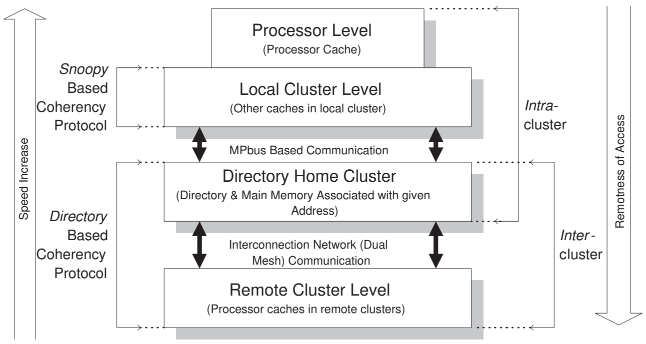

Figure 4. DASH Prototype Cache Coherency Mechanisms.

4. Simulation Design & Implementation

The DASH architecture is modelled in HASE as a three-level entity hierarchy as shown in Figures 5 and 6. Figure 5 shows a cluster viewed at the highest level of abstraction and then expanded to show its internal

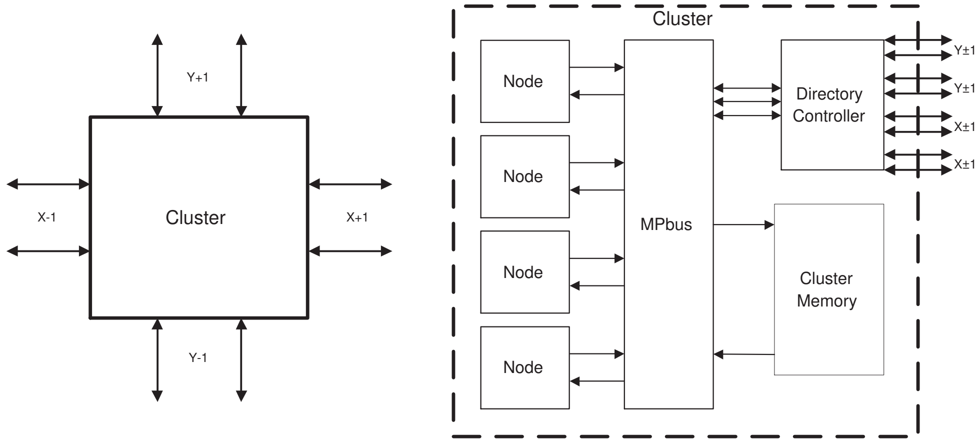

Figure 5. High and Medium Levels of Architectural Abstraction.

structure at the medium level of abstraction (a dotted line surrounding a group of entities indicates that it is made up of a collection of entities which have been grouped together to form a single icon at a higher level of abstraction). In Figure 6 the cluster is further expanded to show the internal structure of the processing nodes, bus and directory-related logic. Figure 12 is an actual HASE display showing all three levels.

A processing node consists simply of the two data caches and a MIPS address generation box, *i.e.* rather than attempt to simulate the MIPS instruction set and run programs to generate cache addresses, the processor simply emits a sequence of addresses (with read/write status) held in a notional memory.

4.1. Processor Caches

The primary cache is direct-mapped and operates a write-through policy. Aside from these DASH-dictated attributes, the demonstration cache was designed to allow the user to redefine 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.

The primary cache 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.

The data structure central to the operation of this entity is a HASE memory array which represents the cache memory contents via a C++ based array of structs. This structure specifies storage for valid, modified and shared bits as well as the cache entry tag and stored values.

This cache line format is shared with the secondary cache unit (described below); the only difference in use is that the primary cache need never use the shared 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.

Throughout the simulation the state of the cache (a value from an enumerated type) 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.

The secondary level processor cache is identical in terms of its caching operation to the primary cache. 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. The operation of the snooping mechanism is explained below alongside the operation of the MPbus arbiter and its associated communication protocols.

4.2. The Node Entity

The node entity consists of the MIPS address generation unit and its associated primary and secondary level caches. The node entity provides an abstraction from the processor cache level, shown as a single entity on the screen, and from which addresses which have 'missed' at both cache levels (or values being sent to memory as a consequence of a write-back) can be observed being issued across the MPbus during animation (Section 5).

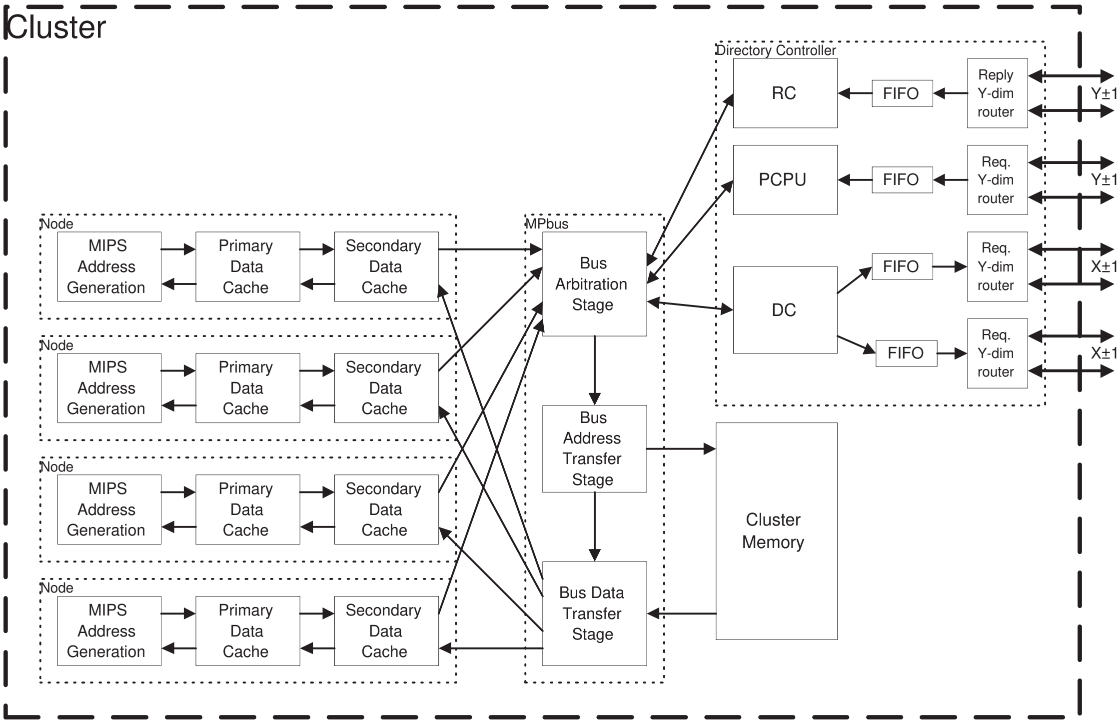

Figure 6. Lowest Level of Architectural Abstraction.

4.3. Inter-entity Communication

Entities in HASE communicate by sending to, and receiving from, ports which are connected by links. When defining a simulation in HASE it is necessary to associate a packet type with a link. The packet format is designed to reflect the communication task in hand.

For example, a node-level packet is the most basic of the three formats used in the DASH simulation. It is designed to carry address requests from the MIPS address generator through the processor caches and on to the MPbus. The packet contains three fields representing the requested address, the read/write classification (also used to pass a variety of control/polling information between the secondary level cache and MPbus arbiter) and the instruction/data classification.

Two other packet formats are used within the DASH simulation; these can generally be classified as supporting intra-cluster level transactions (between the 4 processors of the cluster) and inter-cluster level transactions (where the processors involved in the transaction are from different DASH clusters).

4.4. The MPbus

The MPbus is one of the most complex entities in the 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.

The bus is implemented as a set of entities, each responsible for some part of the bus functionality. These divisions follow the pipelined operation of the bus. This not only gives the user of the simulation a better insight into the bus mechanisms used within DASH but also acts as a demonstration of the pipelining principle. The MPbus entity is thus composed of three lower level entities, as shown in Figure 7.

A bus transaction has a latency of at least seven bus cycles[2] (one cycle for arbitration, three for address transfer and four for data transfer). The MPbus classifies each transaction as belonging to one of three types: a cache transaction, a DMA transaction or an I/O transaction. The present DASH simulation model only models cache-based transactions.

4.4.1. MPbus Arbitration Entity

Arbitration of the MPbus is performed on a fair, round-robin basis, in a single bus cycle. The protocol is shown in Figure 8.

[2]Detailed timing diagrams for the MPbus can be found in [7]

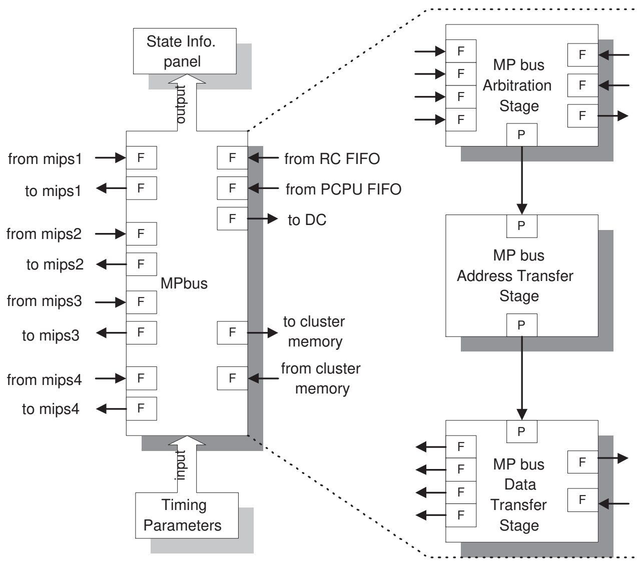

Figure 7. MPbus Composition.

a. **Poll Masters**: At the start of a bus arbitration cycle the arbiter polls all bus masters to see if any require service.

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).

c. **Arbiter Receives Votes**: The arbiter counts the number of **Y**es votes returned. If no masters require service the arbitration delay (1 cycle) is simulated before starting to poll again. However if one or more masters requires service the arbiter selects and grants access to one of them via a function `grant_bus()` (a round-robin method of allocation applies).

d. **Master Receives Permission**: On receipt of the permission packet the master sends the address request to the arbiter. The arbitration cycle now starts again.

The simulation actually performs all of the above polling phases in zero simulation time units. Only when the poll outcome is known is simulation time incremented to reflect the arbitration delay. This use of 'zero time' polling proved to be a useful mechanism throughout the creation of the DASH simulation.

Another major role played by the MPbus arbiter entity is that of co-ordinator for the snoopy-bus protocol (collating snoop data and presenting it in a meaningful format to the user).

The snoopy-bus protocol used in the DASH simulation is identical to that of the actual architecture, the MESI Illinois protocol as outlined in [10]. The flowcharts of Figure 9 detail the general strategy for reads and writes in the DASH system.

A cache line may be in one of four states:

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

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

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

d. **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.

Figure 9 shows the use of cache-to-cache transfers within the protocol. 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 that the set of local secondary level processor caches act as a composite cache for remote memory.

In the HASE simulation of DASH the flowcharts in Figure 9 are implemented as part of the secondary processor caches as in the real system. However the bus arbiter entity is also involved in the coherency mechanism. The protocol supporting the MESI snooping protocol in the DASH simulation (also executed in 'zero-time') is as follows:

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.

b. **Broadcast of Request**: The arbiter receives the read/write request and immediately broadcasts the message back on the bus (in the simulation the broadcast is implemented as four separate messages, one per processor).

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 updates the shared/modified bits according to the address in the received packet. The results of this snooping operation are then transmitted back to the bus arbiter.

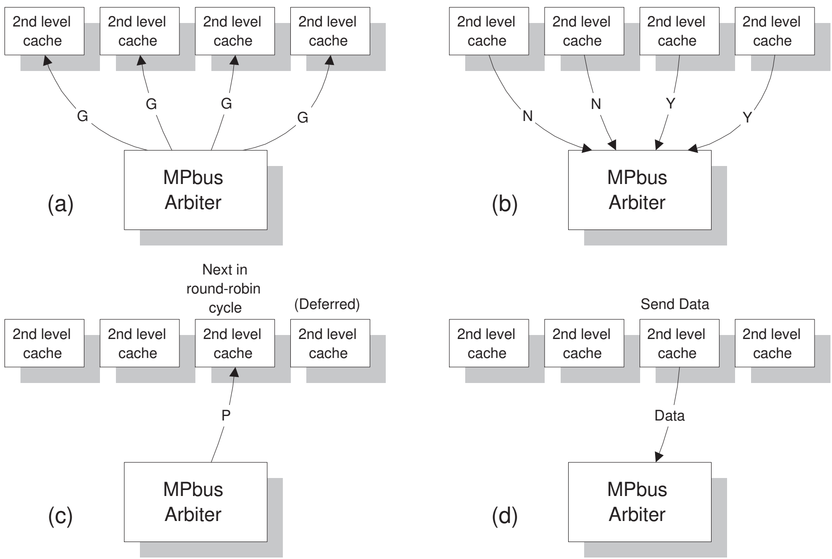

Figure 8. MPbus Arbitration Protocol.

d. **Collate Results** The arbiter gathers the response packets and generates a textual description of the snoop outcome which is displayed via a 'snoop panel' on the arbiter's icon (*c.f.* Figure 12). This panel has four information sources which provide details regarding the bus master most recently granted bus access (an arrow indicates the current master), current access classification (read, write or write-back), result source (either cluster memory or another cache) and a four-line display which gives a summary of the snoop activity in each of the second level caches.

The stages in the snooping operation can be seen in Figure 10.

The protocol must also deal with cache line invalidations when the bus arbiter receives a write request. The address to be written is broadcast with an invalidation signal to all caches *except* the one making the write. Caches receiving the invalidate packet check their contents for a match of address with the requested invalidation. If a match is found the caches reset the appropriate valid bit (Figure 11).

4.4.2. MPbus Address Transfer Entity

This entity is responsible for simulating the 4 bus-cycle delay in the address transfer stage of the actual architecture and for routing on-going requests to either cluster memory or another cache.

When the arbiter has finished its cycle of operations (*i.e.* it has simulated the processing delay of 1 bus cycle) it needs to send the requested address/status to the address transfer unit. This must only be done if the address transfer unit is idle. A simple handshake protocol is used to test the status of the address transfer entity. After the handshake phase the address transfer unit decides whither the address request is bound, advances simulation time by the appropriate delay and finally transmits the request packet.

4.4.3. MPbus Data Transfer Entity

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, the destination node is identified and the data forwarded appropriately. The data transfer cycle now restarts.

4.5. 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 a memory unit cycle consists of

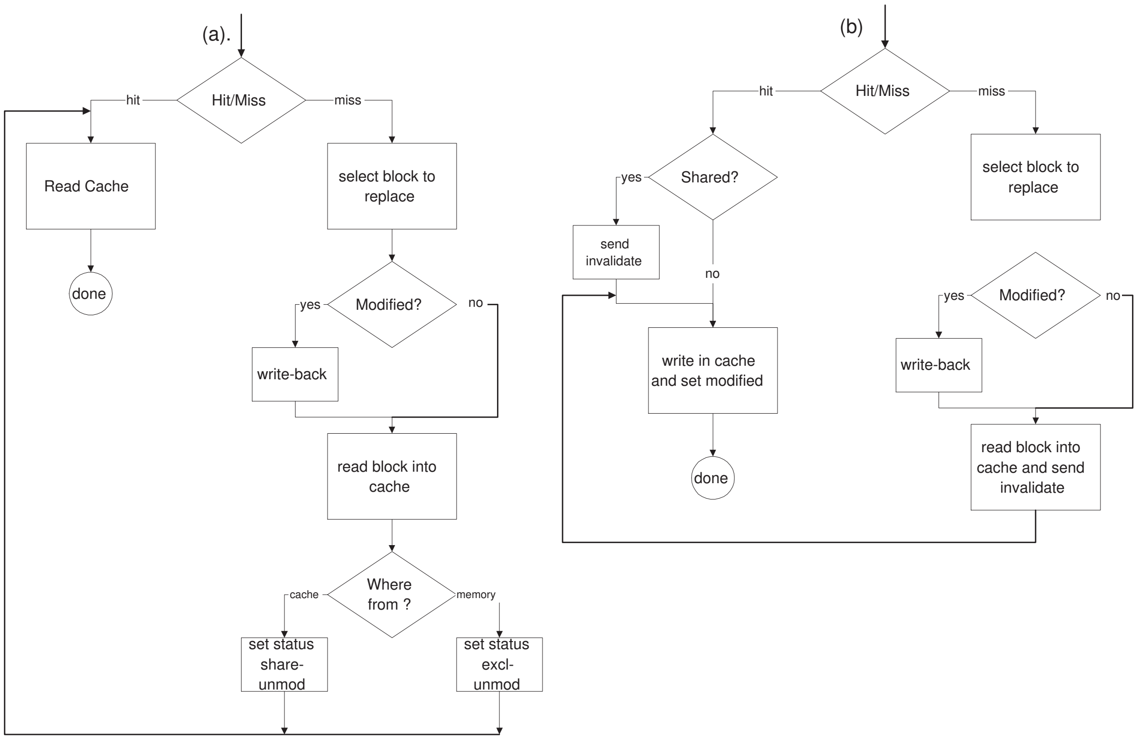

Figure 9. MESI Illinois Protocol (a) Read, (b) Write

receiving an in-bound request, displaying read/write information on-screen and finally transmitting the result packet back onto the MPbus.

5. Running a Simulation

Once the architectural design is complete, the user issues commands via the HASE pull-down menus (*c.f.* Figure 12) to initiate compilation. In order to drive the DASH simulator the user provides input trace files for each MIPS entity consisting of sequences of address, read/write and instruction/data triples. The appropriate file name is included in the definition of each entity and during compilation, HASE effectively loads these traces into the processors. For the DASH simulator, input traces were manually generated to test each feature of the protocols, and the animator used to check for correct operation.

During the simulation run HASE generates an output trace file which details all events occuring within the DASH simulation model. The HASE system then allows the user to examine the simulation results via various forms of output. The most powerful of these is an on-screen animation of the simulated architecture.

The Animator uses the event sequence held in the event trace file to provide the user with a visual display of activity in the system. It allows the data flowing between simulation entities to be visualised in a variety of ways, *e.g.* through moving icons showing individual communication transactions taking place over links joining entities, or changes to the contents of a cache memory unit when address requests are issued or results received. The important benefit of the Animator is that it lets the user check that the model produces correct results.

Manipulation of the animation of the architecture is handled through an on-screen Animation Controller where time, speed and message display are handled as well as the standard 'tape' functions of PLAY, REWIND and STOP.

Using this facility, the operation of the DASH cache coherency protocols has been thoroughly checked. To test each aspect of the protocols, appropriate series of read and write requests were loaded into the processors, the simulation run, and the results observed on screen during play back. In each case the results conform to the specifications given in the original papers.

An alternative post-simulation analysis tool available in HASE is the timing diagram, which displays the state of entities with respect to simulation time. The enumerated parameters of each entity are treated as the state, and different colours/patterns are alloc-

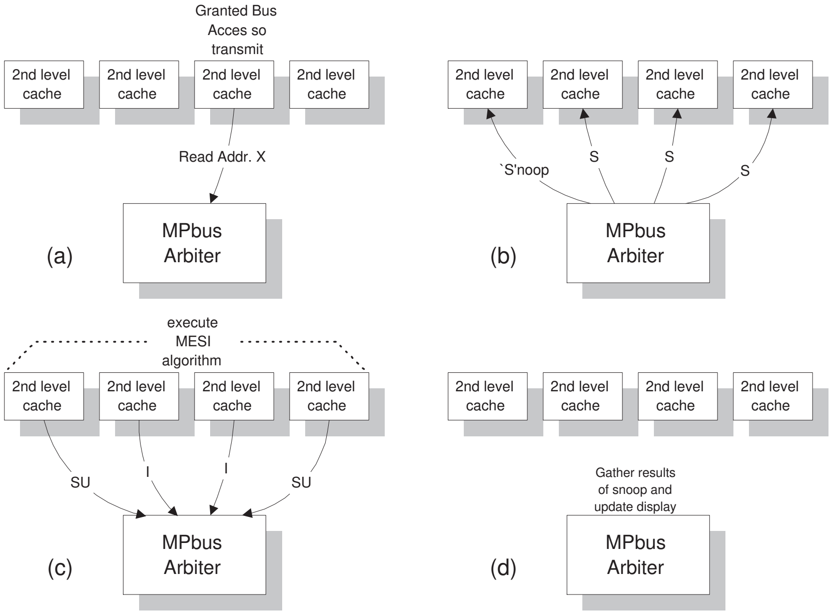

Figure 10. Zero Time Protocol for MESI Snooping Algorithm.

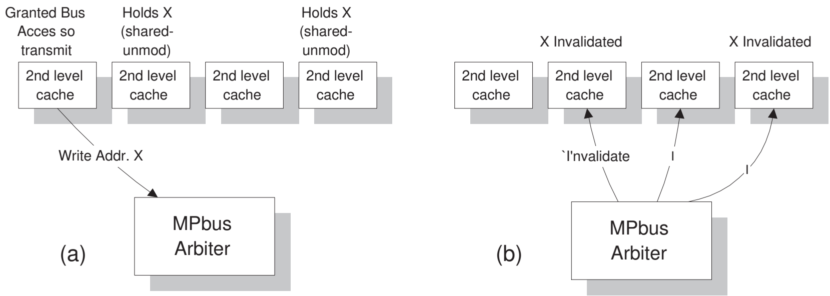

Figure 11. Invalidation Mechanism Used in DASH Simulation.

ated for each different state. Time measurements may be taken with two measuring bars and marked regions may be expanded to show finer detail. These facilties will be of use in the DASH simulator once realistic time delays have been incorporated into the model.

The output trace file can also be processed to allow statistics to be derived which measure system operation (*e.g.* cache hit/miss rates).

6. Conclusions

HASE has proved an effective tool for the creation of a simulation model of a real architecture, and the animator has allowed visual verification of many operational features of the original DASH architecture. These features include the write-back/write-though policies found in the different levels of processor cache and the MESI Illinois snoopy-bus protocol used to maintain intra-cluster cache coherency. This visual insight into the operation of the architecture is complemented by HASE timing diagrams which let the user see how an entity's state changes with respect to a given period of simulation time.

The existing HASE simulation model of DASH also has potential for use as a research tool to investigate possible performance enhancements to the original system. This requires the development of an accurate mapping between HASE simulation time units and actual DASH timing characteristics, and the use of more realistic address traces. It is already possible, for example, to convert Dinero trace files to a HASE compatible format in order to provide large scale input for more realistic simulations aimed at monitoring performance of the DASH model over a large number of simulation events.

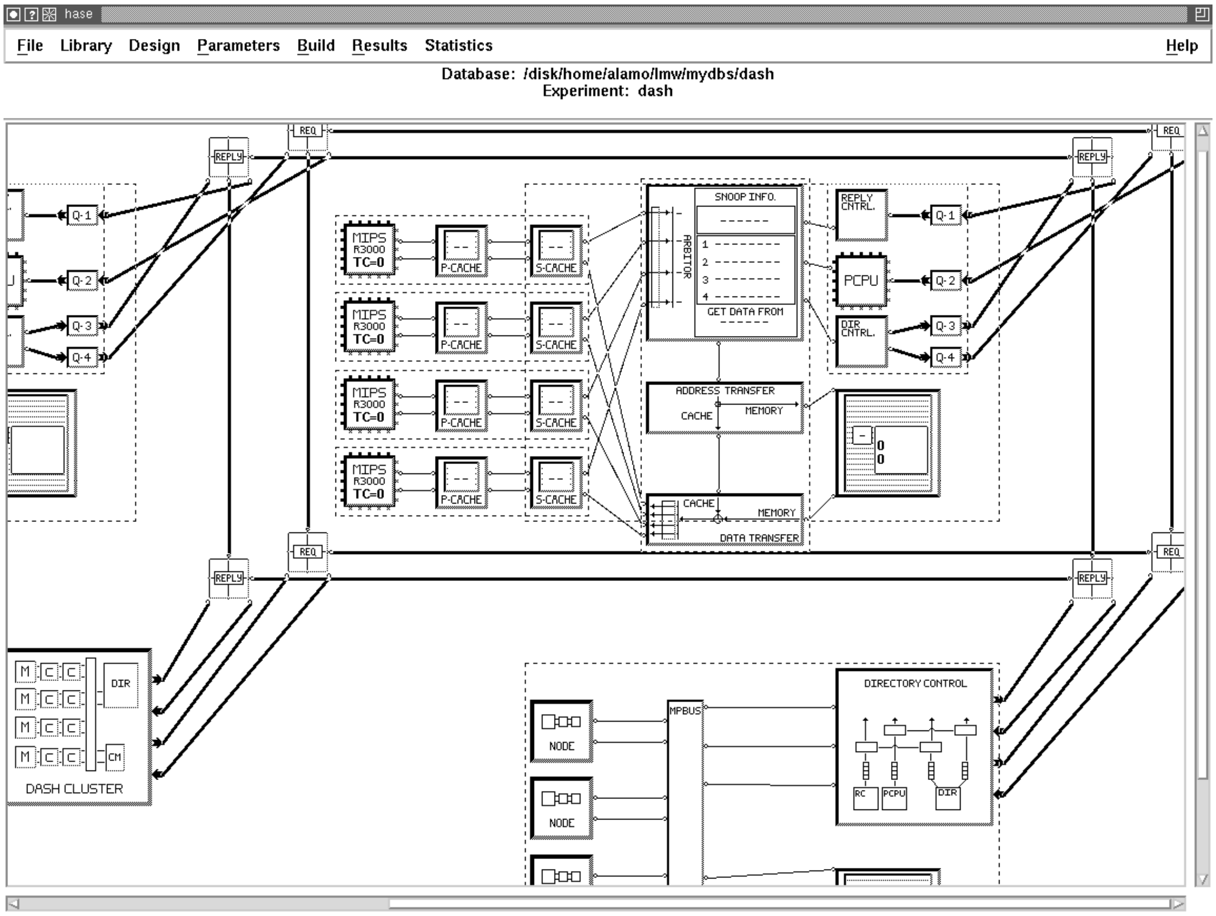

Figure 12. Typical User View of DASH Simulation.

References

[1] F. Baskett, T. Jermoluk, and D. Solomon. The 4d-mp graphics superworkstation: Computing + graphics = 40 mips + 40 mflops and 100000 lighted polygons per second. In *Proc. Compcon Spring 88*, pages 468–471, 1988.

[2] J. Hennessey. Scalable multiprocessors and the dash approach, distinguished lecture series vol iv. University Video Communications, Stanford, USA, 1992.

[3] J. Hennessy and D. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann, 1990.

[4] T. Heywood and S. Ranka. A practical hierarchical model of parallel computation i: The model. *J. Parallel and Distributed Computing*, 16:212–232, 1992.

[5] R. Ibbett, P. Heywood, and F. Howell. Hase: A flexible toolset for computer architects. *(to appear) The Computer Journal*, 1996.

[6] Jade Simulations International Corporation, Calgary, CanadaCalgary, Canada. *Sim++ User Manual*.

[7] D. Lenoski. *The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor, TR:CSL-TR-92-507*. PhD thesis, Stanford University, 1992.

[8] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The dash prototype: Implementation and performance. In *19th ISCA*, pages 92–103, May 1992.

[9] Object Design Incorporated, Burlington, MA. *ObjectStore Release 3.0 User Guide*, December 1993.

[10] M. Papamarcos and J. Patel. A low overhead coherence solution for multiprocessors with private cache memories. In *Proc. 11th ISCA*, pages 348–354, 1984.

[11] A. Robertson and R. Ibbett. Simulation of the mc88000 microprocessor system on a transputer network. In *Proc. EDMCC2, Springer-Verlag, Berlin*, 1991.

[12] P. Sweazy and A. Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. In *Proc. 13th ISCA*, pages 414–423, June 1986.