MOVING THE SHARED MEMORY CLOSER TO THE PROCESSORS – DDM

Erik Hagersten, Anders Landin and Seif Haridi
SICS Research Report R90:17B May 1991.[*]

Abstract

Multiprocessors with shared memory are considered more general and easier to program than message-passing machines. The scalability is, however, in favor of the latter. There are a number of proposals showing how the poor scalability of shared memory multiprocessors can be improved by the introduction of private caches attached to the processors. These caches are kept consistent with each other by cache-coherence protocols.

In this paper we introduce a new class of architectures called Cache Only Memory Architectures (COMA). These architectures provide the programming paradigm of the shared-memory architectures, but are believed to be more scalable. COMAs have no physically shared memory; instead, the caches attached to the processors contain **all** the memory in the system, and their size is therefore large. A datum is allowed to be in any or many of the caches, and will automatically be moved to where it is needed by a cache-coherence protocol, which also ensures that the last copy of a datum is never lost. The location of a datum in the machine is completely decoupled from its address.

We also introduce one example of COMA: the Data Diffusion Machine (DDM). The DDM is based on a hierarchical network structure, with processor/memory pairs at its tips. Remote accesses generally cause only a limited amount of traffic over a limited part of the machine.

The architecture is scalable in that there can be any number of levels in the hierarchy, and that the root bus of the hierarchy can be implemented by several buses, increasing the bandwidth.

Keywords: Multiprocessor, hierarchical architecture, hierarchical buses, multilevel cache, shared memory, split-transaction bus, cache coherence.

[*]Swedish Institute of Computer Science; Box 1263 ; 164 28 KISTA ; SWEDEN. Email: {hag,landin,seif}@sics.se

1 CACHE-ONLY MEMORY ARCHITECTURES

Multiprocessor architectures are divided into two classes: shared-memory architectures and message-passing architectures.

Existing architectures with shared memory are typically computers with one common shared memory, such as computers manufactured by Sequent and Encore, or with distributed shared memory, such as the BBN Butterfly and the IBM RP3. In the latter, known as non-uniform memory architectures (NUMA), each processor node contains a portion of the shared memory; consequently access times to different parts of the shared address space can vary. Communication and synchronization in architectures with shared memory are done implicitly through the common shared address space and by the use of synchronization primitives.

Machines with shared memory typically have only a limited number of processors. Especially systems based on a single bus suffer from bus saturation. Their poor scalability can benefit from large caches local to the processors. The contents of the caches are kept coherent by a cache-coherence protocol. Each cache snoops the traffic on the common bus and prevents any inconsistencies from occurring [Ste90].

Message-passing architectures also have distributed memory, but lack the common shared address space. Instead, each processor has its own address space. Examples of such architectures are the Hypercube architectures by Intel and architectures based on the Transputer by Inmos. Communication and synchronization among the processors is handled by explicit messages. So, memory access and communication with other processors are viewed as completely separate functions.

Message-passing machines are said to scale better and consequently might have more processors. Message-passing architectures, however, burden the software to a much greater extent in that it has to statically distribute the execution and memory usage among the processors. Task migration and dynamic job allocation are very costly. Thus, the hardware scalability of a message-passing architecture is only useful to the extent that the software can keep communication to a minimum.

This is one of the reasons why shared-memory architectures are considered more general and easier to program. Efficient programs for NUMA architectures suffer software limitations similar to those of message-passing architectures, in that processors need to work as much as possible in "their" portion of the shared memory in order to get the shortest access time and to keep communication to a minimum.

In cache-only memory architectures (COMA), the memory organization is similar to that of NUMA in that each processor holds a portion of the address space. However, all memory in the COMA, called attraction memory (AM), is organized like large (second-level) caches. While each NUMA memory holds a static portion of the address space, the content of each AM changes dynamically throughout the computation. A cache-coherence protocol allows copies of a datum to exist in multiple AMs and maintains the coherence among the different copies. When replacing one datum in an AM with a more recently accessed one, the protocol makes sure that the last copy of the datum in the system is not lost.

COMA provides a programming model identical to that of shared-memory architectures, but does not require static distribution of execution and memory usage in order to run efficiently. COMA has advantages for moderately sized machines, as well as for large machines, in that it provides the largest possible (second-level) caches for a given total amount of memory, since **all** the memory is used in the AMs. The attraction memories not only dynamically distribute the address space over the machine, but also cut down the need for communication among processors, since most accesses are local to the attraction memory of the processor. The overhead required for accessing a large cache compared to a large memory and the increased amount of memory for implementation are surprisingly small. Figure 1 compares COMA to other shared-memory architectures.

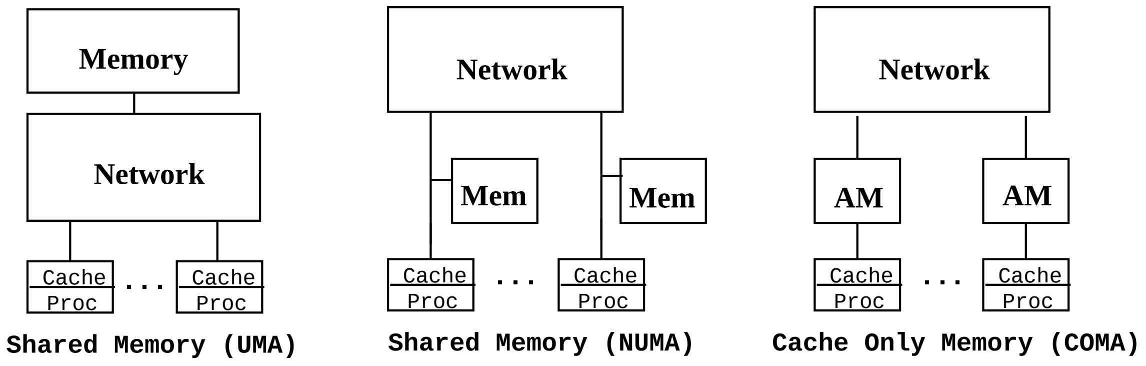

Figure 1: Shared-memory architectures compared to the COMA.

This paper describes the basic ideas behind a new architecture of the COMA class. The architecture, called the Data Diffusion Machine (DDM), relies on a hierarchical network structure. The paper first introduces the key ideas behind the DDM by describing a small machine and its protocol. It continues with a description of a large machine with hundreds of processors, followed by a more detailed study of some issues essential to machines of this class, how to increase the network bandwidth, and how to implement virtual memory and secondary storage. The paper ends with a brief introduction to the ongoing prototype project and the simulated performance figures.

2 CACHE-COHERENCE STRATEGIES

The problem of maintaining coherence among read-write data shared by different caches has been studied extensively over the last years. Coherence can either be kept by software or hardware. It is our belief that the cache line of a COMA must be small in order to prevent performance degradation by false sharing. Using a software-based protocol therefore introduces extensive overhead. Hardware-based schemes maintain coherence without involving software and can therefore be implemented more efficiently. Examples of hardware-based protocols are snooping-cache protocols and directory-based protocols.

Snooping-cache protocols have a distributed implementation. Each cache is responsible for snooping traffic on the bus and taking necessary actions if an incoherence is about to occur.

An example of such a protocol is the *write-once protocol* introduced by Goodman and reviewed by Stenström [Ste90]. In that protocol, shown in Figure 2, each cache line can be in one of the four states INVALID, VALID, RESERVED, or DIRTY. Many caches might have the same cache line in the state VALID at the same time, and may read it locally. When writing to a cache line in VALID, the line changes state to RESERVED, and a write is sent on the common bus to the common memory. All other caches with lines in VALID snoop the write and invalidate their copies. At this point there is only one cached copy of the cache line containing the newly written value. The common memory now also contains the new value. If a cache already has the cache line in RESERVED, it can perform a write locally without any transactions on the common bus. Its value will now differ from that in the memory, and its state is therefore changed to DIRTY. Any read requests from other caches to that cache line must now be intercepted, in order to provide the new value, marked by "intercept" in the figure. The figure also shows the optimization "Nread.inv", which allows a cache to read a private copy of a cache line when in state INVALID.

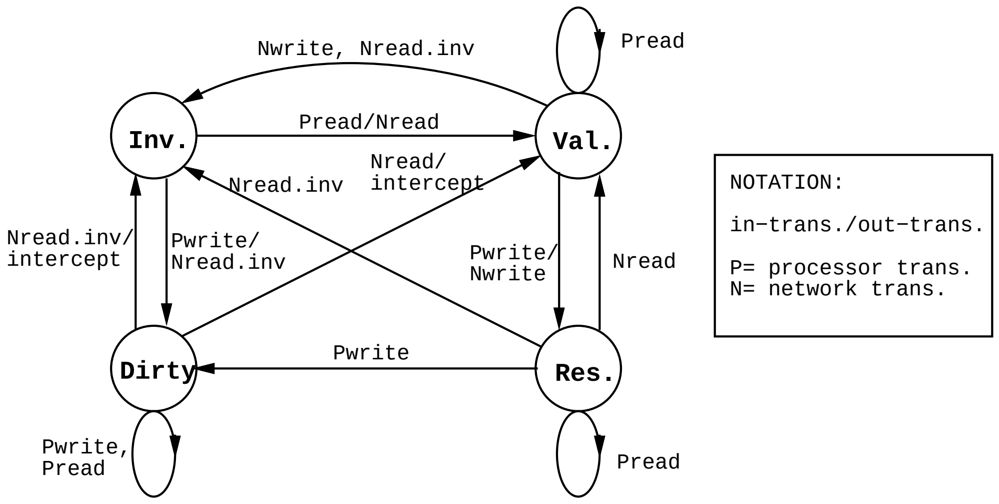

Figure 2: The write-once protocol.

Snooping caches, as described above, rely on broadcasting and are not suited for general interconnection networks: broadcasting would reduce the bandwidth available to that of a single bus. Instead, directory-based schemes send messages directly between nodes [Ste90]. A read request is sent to main memory, without any snooping. The main memory knows if the cache line is cached, in which cache or caches, and whether or not it has been modified. If the line has been modified, the read request is passed on to the cache with a copy, which provides a copy for the requesting cache. The caches might also keep information about what other caches have copies of the cache lines. Writing can now be performed with direct messages between all caches with copies.

Neither of the above schemes looks attractive for the implementation of a COMA. The snooping-cache implementation has poor scalability caused by its broadcast nature, and the directory-based scheme initially presents a read request to a shared "directory" location. This does not conform to the COMA idea, which has nothing but caches.

Instead, we developed a new protocol, similar in many ways to the snooping-cache protocol, limiting broadcast requirements to a smaller subsystem and adding support for replacement.

3 A MINIMAL COMA

We will introduce the COMA architecture by looking at the smallest instance of our architecture, the Data Diffusion Machine (DDM). The minimal DDM, as presented, can be a COMA on its own or a subsystem of a larger COMA.

The attraction memories of the minimal DDM are connected by a single bus. The distribution and coherence of data among the attraction memories is controlled by the snooping protocol *memory above*, and the interface between the processor and the attraction memory is defined by the protocol *memory below*. A cache line of an attraction memory, here called an *item*, is viewed by the protocol as one unit. The attraction memory stores one small state field per item.

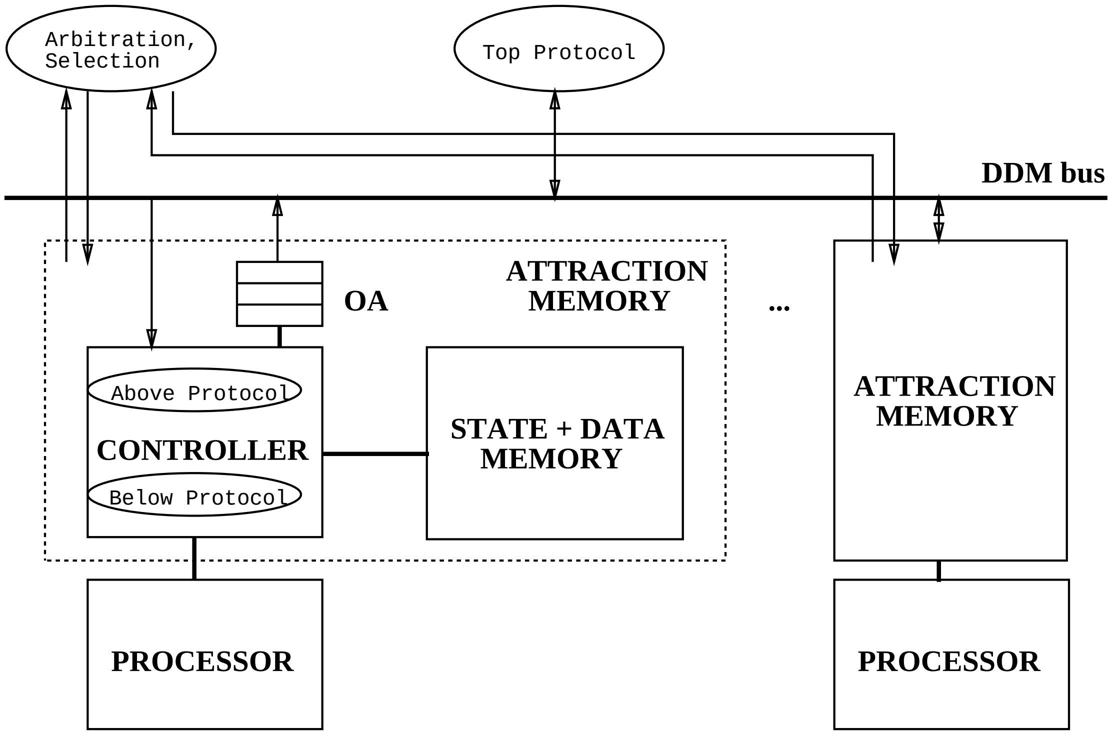

Figure 3: The architecture of a single-bus DDM. Below the attraction memories are the processors. Located on top of the bus are the *top protocol*, arbitration, and selection.

The architecture of the nodes in the single-bus DDM is shown in figure 3. The attraction memory has a small output buffer, *output buffer above* (OA), for transactions waiting to be sent on the bus above. The OA can be limited to a depth of three, and deadlock can still be avoided by the use of a special arbitration algorithm.

3.1 The Behavior of the Bus

The DDM uses an asynchronous split-transaction bus, where the bus is released between a requesting transaction and its reply, e.g., between a read request and its data reply. The delay between the request and its reply can be of arbitrary length, and there might be a large number of outstanding requests. The reply transaction will eventually appear on the bus as a different transaction.

Unlike other buses, the DDM bus has a selection mechanism, where many nodes can express interest in a transaction on the bus, but only one is selected to service it. There are selections of different priorities, used to make sure that each transaction on the bus does not produce more than one new transaction for the bus, a requirement necessary for deadlock avoidance.

3.2 The Protocol of the Single-Bus DDM

The DDM coherence protocol is a write-invalidate protocol; i.e., in order to keep data consistent, all copies of the item but the one to be updated are erased on a write. The small item size in combination with the large "cache" size gives it an advantage over the write-broadcast approach, where, on a write, the new value is broadcast to all "caches" with a shared copy of the item [EK89]. The protocol also handles the attraction of data (read) and replacement when a set in an attraction memory gets full. The snooping protocol defines a new state, a new transaction to send, and a selection priority as a function of the transaction appearing on the bus and the present state of the item in the attraction memory.

PROTOCOL: old state $\times$ transaction $\rightarrow$new state $\times$ new transaction $\times$ sel. prio.

An item can be in one of the following states, where subsystem refers to the attraction memory:
I Invalid. This subsystem does not contain the item.
E Exclusive. This subsystem and no other contains the item.
S Shared. This subsystem and possibly other subsystems contain the item.
R Reading. This subsystem is waiting for a data value after having issued a read.
W Waiting. This subsystem is waiting to become exclusive after having issued an erase.
RW Reading and Waiting. This subsystem is waiting for a data value, later to become exclusive. A combination of states R and W.

The first three states, I, E, and S, correspond to the states INVALID, RESERVED, and VALID in Goodman's write-once protocol. The state DIRTY in that protocol, with the meaning: this is the only cached copy and its value differs from that in the memory, has no correspondence in a COMA with no shared memory. New states in the protocol are the *transient states* R, W, and RW. The need for the transient states is created by the nature of the split-transaction bus and the need to remember outstanding requests.

The bus carries the following transactions:

e, erase. Erase all your copies of this item.
x, exclusive. Now there is only one copy of the item in the system.
r, read. Request to read a copy of the item.
d, data. Carries the data in reply to an earlier read.
i, inject. Carries the only copy of an item and is looking for a subsystem to move into, caused by a replacement.
o, out. Carries the data on its way out of the subsystem, caused by a replacement. It will terminate when another copy of the item is found.

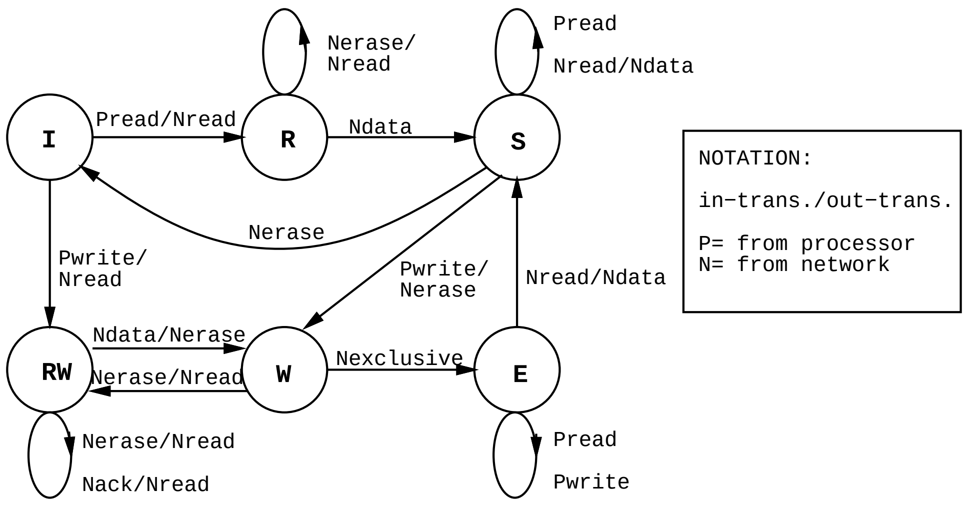

Figure 4: A simplified representation of the attraction memory protocol.

A processor writing an item in state E or reading an item in state E or S will proceed without interruption. A read attempt of an item in state I will result in a *read* request and a new state R as shown in figure 4. The selection mechanism of the bus will select one attraction memory to service the request, eventually putting *data* on the bus. The requesting attraction memory, now in state R, will grab the *data* transaction, change state to S, and continue.

Processors are only allowed to write to items in state E. If the item is in S, all other copies have to be erased, and an acknowledge received, before the writing is allowed. The AM sends an *erase* transaction and waits for the acknowledge transaction *exclusive* in the new state, W. Many simultaneous attempts to write the same item will result in many attraction memories in state W, all with an outstanding *erase* transaction in their output buffers. The first erase to reach the bus[1] is the winner of the write race. All other *transactions* bound for the same item are removed from the small OA buffers. Therefore, the buffers also have to snoop transactions. The losing attraction memories in state W change state to RW while sending a *read* request on the bus. Eventually the *top protocol* sitting on top of the bus replies with an *exclusive* acknowledge, telling the only attraction memory left in state W that it may now proceed.

Writing to an item in state I results in a *read* request and a new state RW. Upon the *data* reply, the state changes to W and an *erase* request is sent.

[1]The attraction memories send transactions one at a time

3.3 Replacement

Like ordinary caches, the attraction memory will run out of space, forcing some items to leave room for more recently accessed ones. If the set where an item is supposed to reside is full, one item in the set[2] is selected to be replaced. Replacing an item in state S generates an *out* transaction. The space used by the item can now be reclaimed. The *out* transaction is either terminated by a subsystem in states S, R, W, or RW, or converted to an inject transaction by the top protocol. An *inject* transaction can also be produced by replacing an item in state E. The *inject* transaction is the last copy of an item trying to find a new home in a new attraction memory. In the single bus implementation it will do so firstly by choosing an empty space, and secondly by replacing an item in state S. If the DDM address space used is smaller than the sum of the attraction-memory sizes, it can be guaranteed a space.

The details of the protocols *top, memory above*, and *memory below* are defined by the state transition tables in Table 1 in the appendix.

3.4 Conclusion

What has been presented so far is a cache-coherence single-bus multiprocessor without any physically shared memory. Instead, the resources are used to build huge second-level caches, called attraction memories, minimizing the number of accesses to the only shared resource left: the shared bus. Data can reside in any or many of the attraction memories. Data will automatically be moved where needed.

4 THE HIERARCHICAL DDM

The single-bus DDM as described can become a subsystem of a large hierarchical DDM by replacing the top with a directory, which interfaces between the bus described and a higher level bus in a hierarchy as shown in Figure 5. The directory is a set-associative status memory, which keeps information for all the items in the attraction memories below it, but contains no data. The directory can answer the questions: "Is this item below me?" and "Does this item exist outside my subsystem?"

From the bus above, its snooping protocol *directory above* behaves very much like the *memory above* protocol. From the bus below, the *directory below* protocol behaves like the *top protocol* for items the exclusive state. This makes operations to items local to a bus identical to those of the single-bus DDM. Only transactions that cannot be completed inside its subsystem or transactions from above that need to be serviced by its subsystem are passed through the directory. In that sense, the directory can be viewed as a filter.

The directory as shown in figure 6 has a small output buffer above it (OA) to store transactions waiting to be sent on the higher bus. Transactions for the lower bus are stored in the buffer *output below* (OB), and transactions from the lower bus are stored in the buffer *input below* (IB). A directory reads from IB when it has the time and space

[2]The oldest item in state S, of which there might be other copies, may be selected, for example.

Figure 5: The hierarchical DDM, here with three levels.

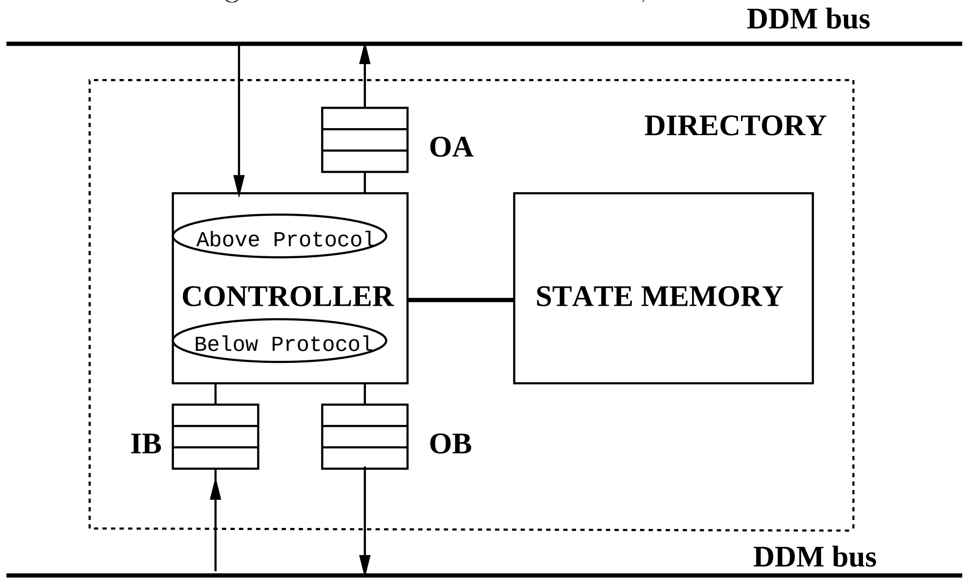

Figure 6: The architecture of a directory.

to do a lookup in its status memory. This is not part of the atomic snooping action of the bus.

4.1 Multilevel Read

If a read request cannot be satisfied by the subsystems connected to the bus, the next higher directory retransmits the *read* request on the next higher bus. The directory also changes the item's state to reading (R), marking the outstanding request. Eventually, the request reaches a level in the hierarchy where a directory, containing a copy of the item, is selected to answer the request. The selected directory changes its state to answering (A), marking an outstanding request from above, and retransmits the *read* request on its lower bus. The new states, R and A in the directories, mark the request's path through the hierarchy, shown in Figure 7, like rolling out a red thread when walking in a maze [HomBC]. When the request finally reaches an attraction memory with a copy of the item, its *data* reply simply follows the red thread back to the requesting node, changing all the states along the path to shared (S). Table 2 in the appendix shows the detailed actions step by step for a multilevel read. Often many processors try to read

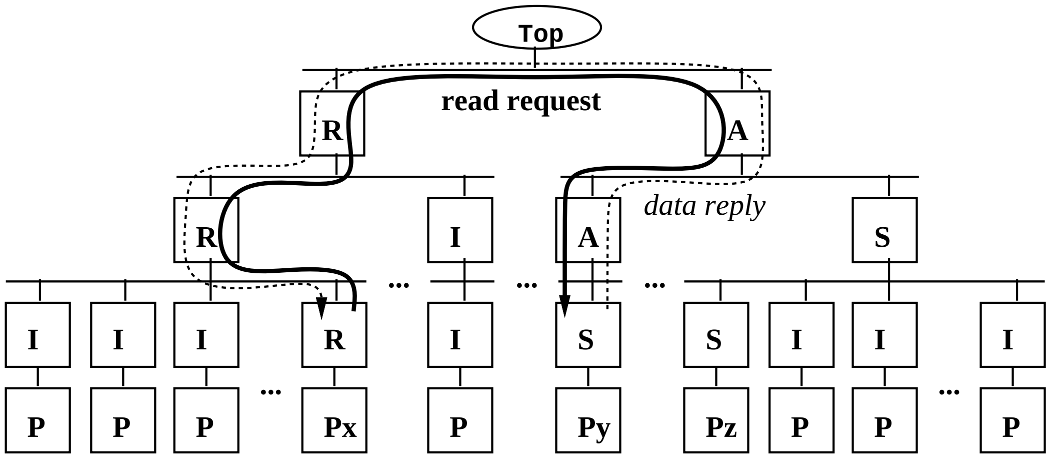

Figure 7: A read request from processor Px has found its way to a copy of the item in the AM of processor Py. Its path is marked with states reading and answering (R and A), which will guide the data reply back to Px.

the same item, creating the "hot-spot phenomenon." Combined reads and broadcasts are simple to implement in the DDM. If a *read* request finds the red read thread rolled out for the requested item (state R or A), it simply terminates and waits for the *data* reply that eventually will follow that path on its way back.

4.2 Multilevel Write

An *erase* from below to a directory with the item in state exclusive (E), results in an *exclusive* acknowledge being sent below. An *erase* that cannot get its acknowledge from the directory will work its way up the hierarchy, changing the states of the directories to waiting (W), marking the outstanding request. All subsystems of a bus carrying an *erase* transaction will get their copies erased. The propagation of the *erase* ends when a directory in state exclusive (E) is reached (or the top), and the acknowledge is sent back along the path marked with state W, changing the states to exclusive (E).

A write race between any two processors in the hierarchical DDM has a solution similar to that of a single-bus DDM. The two *erase* requests are propagated up the hierarchy. The first *erase* transaction to reach the lowest bus common to both processors is the winner, as shown in Figure 8. The losing attraction memory will generate a new write action automatically. Table 3 in appendix shows a write race in detail.

4.3 Replacement in the Hierarchical DDM

Replacement of a shared item in the hierarchical DDM will result in an *out* transaction propagating up the hierarchy and terminating when a subsystem in any of states S, R,

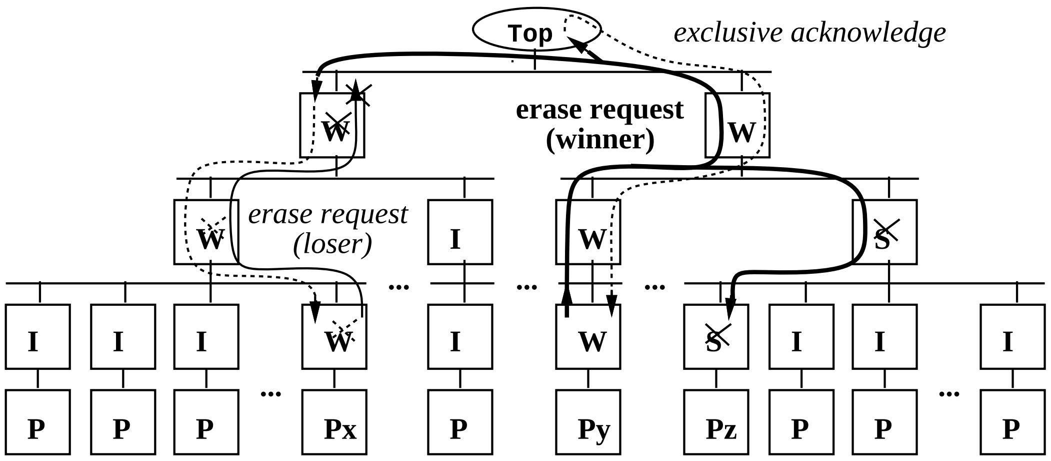

Figure 8: A write race between two processors, Px and Py, is resolved when the request originating from Py reaches the top bus (the lowest bus common to both processors.) The top can now send the acknowledge, *exclusive*, which follows the path marked with Ws back to the winning processor Py. The states W will be changed to E by the *exclusive* acknowledge

.

W, or A is found. If the last copy of an item marked with state S is replaced, an *out* that fails to terminate will reach a directory in state E.

Replacing an item in state E generates an *inject* transaction, trying to find an empty space in a neighboring AM. Nonterminated *out* transactions and *inject* transactions failing to find empty spaces will force themselves into an AM by replacing an item in state S as described for the single-bus DDM. A replacement might take place in order to give space to the homeless item. In order to avoid chain reactions and livelocks, all items have a preferred bus to which they turn whenever they are refused space elsewhere, called *go home*.

The item space is equally divided among all directories at the same level, so that each item has one preferred subsystem. A directory can detect if a transaction is within their portion of the item space and can guide the *go home* to its home bus. The home item space of a bus is slightly smaller than the sum of the sizes of the attraction memories connected to that bus, guaranteeing each *go home* a space, if necessary by throwing shared and/or foreign items out.

The preferred location, as described, is different from the memory location of NUMAs in that the notion of a home is only used after failing to find space elsewhere. When the item is not there, its place can be used by other items.

The details of the directory protocols are defined by the state transition diagram in Table 4 in the appendix.

4.4 Replacement in a Directory

Other hierarchical bus projects have identified a problem called the full-inclusion property, making such architectures impractical.

The memory size (size here meaning number of items) of the directories increases higher up in the hierarchy. However, in order to guarantee space in a directory for all items in its subsystem, it is not enough to just make the size of the directory equal to the number of items in the attraction memories below. The degree of associativity (number of ways) in a directory should also be equal to the product of the number of attraction memories in its subsystem and their degree of associativity. In the higher directories of big systems, the associativity would range in the hundreds. Even if implementable, such memories would be expensive and slow.

The DDM uses directories with smaller sets, called imperfect directories, and endows the directory with the ability to perform replacement. The probability of replacement can be kept at a reasonable level by increasing the associativity moderately higher up in the hierarchy. A higher degree of sharing will also help to keep that probability low. A shared item occupies space in many attraction memories, but only one space in the directories above them.

Directory replacement is implemented by an extension to the existing protocol, which requires one extra state and two extra transactions [HHW90].

4.5 Other Protocols

The described protocol provides a *sequentially consistent* [LHH91] system to the programmer. While fulfilling the strongest programming model, performance is degraded by waiting for the acknowledge before the write can be performed. Note though that the acknowledge is sent by the topmost node of the subsystem in which all the copies of the item reside, instead of by each individual AM. This not only reduces the remote delay, but also cuts down the number of transactions in the system.

A more efficient protocol has been proposed that performs the write without waiting for an acknowledge, allowing for many outstanding writes at the same time. A newly written, but not yet acknowledged, item is marked "write pending", while the processor continues its execution. Its new value will not be revealed before the acknowledge is received. This protocol, called *fast write* [HLH91], results in a better performance, while supporting the looser programming model *processor consistency.*[3] Other proposals for architectures allowing multiple outstanding writes can only support models of even looser consistency.

5 INCREASING THE BANDWIDTH

The system described so far has two apparent bottlenecks:

- The size of the directories grows the higher up one gets in the hierarchy. The biggest directories are found right underneath the top bus. A practical limit to how big these directories can be made limits the size of the system.
- Although most memory accesses tend to be localized in the machine, the higher level buses may nevertheless demand a higher bandwidth than the bus can provide,

[3]It also provides causal correctness.

which creates a bottleneck. Snooping in the big directories makes the top bus slower rather than faster.

A way of taking the load off the higher buses is to have a smaller branch factor at the top of the hierarchy than lower down [VJS88]. This solution, however, makes the higher directories bigger rather than smaller.

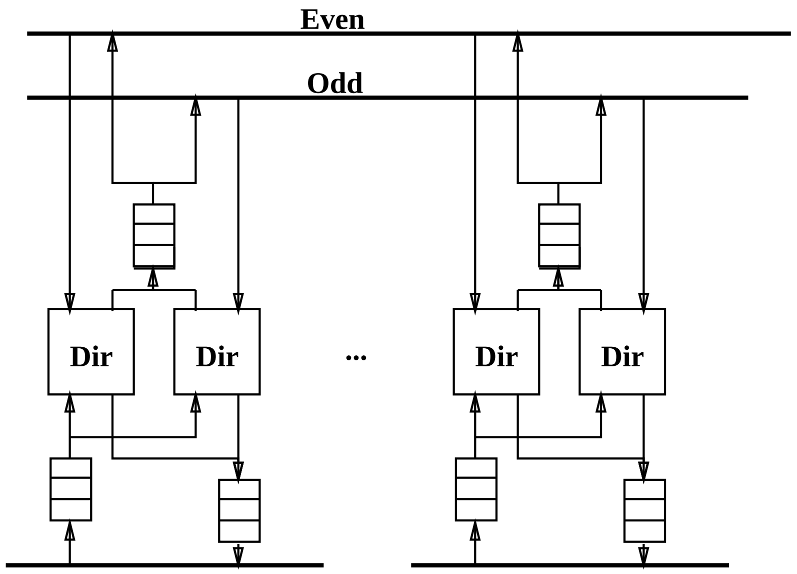

Figure 9: Increasing the bandwidth of a bus by splitting buses.

Instead, both problems are solved with the same solution: splitting the higher buses as shown in Figure 9. The directory is split into two directories half the size. The directories deal with different address domains (even and odd). The number of buses above is also doubled, each bus dealing with its own address domain. Repeated splits will make a bus as wide as possible, and directories as small as needed. Splitting is possible at any level. Regardless of the number of splits, the architecture is still hierarchical to each specific address. Transactions on the respective buses can be prevented from overtaking each other, which leaves us with a network that is said to be race-free. This can be achieved by selectively restricting the transactions allowed to be transferred in parallel [LHH91]. One example of where the splitting can be used is a ring-based bus time-slotted into different address domains.

Nonrace-free networks would require the write acknowledges to be produced by each individual AM, rather that by the topmost node of the subsystem in which all the copies of the item reside. It would also prevent the implementation of the *fast write* protocol.

6 THE MEMORY SYSTEM

An architecture containing only caches has an interesting property in that it can have a larger address space than there is physical memory in the system, given that the whole address space is never used at the same time. Applications with sparse addresses and garbage-collecting applications could benefit from such a behavior. However, it also requires a replacement strategy different from the one described, and introduces the need for some kind of secondary storage (called item reservoir) for situations when replaced items fail to find an empty space.

Described below is a more conventional way of handling the memory system, better suited for general-purpose programs and operating systems.

6.1 Virtual and Physical Memory

Virtual memory is traditionally used to allow for multiple address spaces. Two different processes, possibly located in the same processor, may use the same virtual address for accessing their own private address spaces. They must therefore access different parts of the physical memory. The processes might also access different virtual addresses when referring to the same physical address. This behavior of virtual addresses makes them hard to use in snooping caches. Instead, the translation from virtual to physical addresses is often performed by a memory management unit (MMU), located between the processor and the cache. The MMU maps pages of virtual memory to pages of physical memory. The mapping is done by the operating system when the page is first touched. During execution, the page might get unmapped (paged out) and remapped (paged in) to a new physical page. The operating system keeps a free list of physical pages not in use.

The DDM handles the virtual memory in exactly the same way, allowing for multiple address spaces. An MMU identical to the one above is located between the processor and the attraction memory. The MMU maps pages of virtual addresses to pages of *item identifiers*, which are the addresses of the items described in earlier examples. The operating system maps, unmaps, and remaps pages using a free list of item pages not in use. A page of item identifiers differs from a physical page in that it does not occupy consecutive physical space in a shared memory. Item identifiers have unique names and might reside anywhere in the machine. An item page on the free list will not occupy any memory in the machine. When allocating a new item page, none of the items on that page will occupy any space before they are used. They will get "born" when first used (written to). The latency of the birth can be hidden if a protocol allowing for multiple outstanding writes is used.

6.2 Connecting Disks

Many existing I/O devices, like disks, expect a memory with a contiguous address space. Each AM can host a number of those devices, such as one disk for each AM.

When the operating system decides to send a page to secondary storage, it first invalidates its entries in the MMUs, and then gives an order to a selected node to send the page to its secondary storage. The selected node first gets all the items into the new state "exclusive-immune" to make them immune to replacement. Secondly, it starts the DMA to disk. Each item read by DMA will automatically be invalidated.

Trying to access a page that has been paged out reverses the process. The operating system orders the node holding the page on its disk to write the page to a selected item page. The node first allocates space in its AM by putting the items of the selected item page in the state "exclusive-immune". Then it starts the DMA transfer from the disk, changing states to exclusive. Finally, the virtual page is mapped to the item page by the operating system.

7 THE DDM PROTOTYPE PROJECT

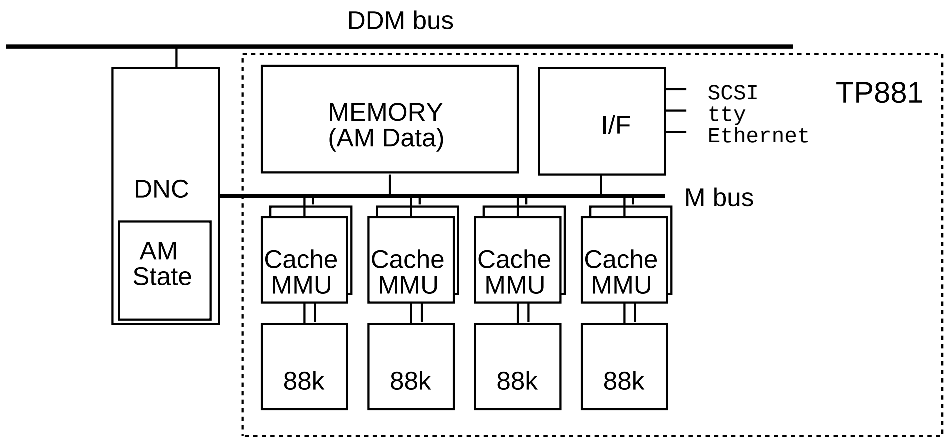

Figure 10: The implementation of a DDM node consisting of four processors sharing one attraction memory.

A prototype design of the DDM is near completion at SICS. To minimize the work, the hardware implementation of the processor/attraction memory is based on the system TP881V by Tadpole Technology, U.K. Each such system has four Motorola 88100 20 MHz processors, eight 88200 processor caches, between 8 and 32 Mbyte DRAM, and interfaces for the SCSI-bus, Ethernet, and terminals, all connected by the Motorola Mbus.

A DDM Node Controller (DNC) board, hosting a four-way set-associative state memory, is being developed, interfacing the TP881 node and the first level DDM bus as shown in Figure 10. The DNC snoops accesses between the processor caches and the memory of the TP881 according to the protocol *memory below*,[4] and also snoops the DDM bus according to the protocol *memory above*. The DNC thus changes the behavior of the memory into a four-way set-associative attraction memory. Read accesses to the attraction memory take seven cycles per cache line, which is identical to the read accesses of the original TP881 system. Write accesses to the attraction memory take twelve cycles compared to ten cycles for the original system. A read/write mix of 3/1 to the attraction memory results in the access time to the attraction memory being on the average 6 percent slower than that to the original TP881 memory.

A remote read to a node on the same DDM-bus takes 65 cycles at best, most of which are spent making Mbus transactions (a total of four accesses). Read accesses climbing one step up and down the hierarchy add about 20 extra cycles.

The DDM bus is pipelined in three phases: transaction code, selection, and data. We have decided to make an initial conservative bus design, since pushing the bus speed is

[4]Extended to handle multiple 88200 caches

not a primary goal of this research. The DDM bus of the prototype operates at 20 MHz, with a 32-bit data bus and a 32-bit address bus. It provides a moderate bandwidth of about 80 Mbyte/s which is enough for connecting up to eight nodes, i.e., 32 processors.

A new, ring-based, bus is also being designed. It explores the splitting of buses and directories into different address domains by time-slotting. It is targeted for a bandwidth of 0.5 Gbyte/s using off-the-shelf components.

We expect to have a small DDM prototype of 16–32 processors running in 1991. The prototype should gradually be increased to include hundreds of processors.

8 MEMORY OVERHEAD

At first sight, it might be tempting to believe that an implementation of the DDM would require far more memory than alternative architectures. Extra memory will be required for storing state bits and address keys for the set-associative attraction memories, as well as for the directories. We have calculated the extra bits needed if all items reside only in one copy (worst case). An item size of 128 bits is assumed.[5]

A 32-processor DDM, i.e., a one-level DDM with a maximum of eight four-way set-associative attraction memories, needs five bits of address tag per item, regardless of the attraction memory size. As stated before, the item space is slightly smaller than the sum of the sizes of the attraction memories, i.e., the size of each attraction memory is 1/8 of the item space. Each set in the attraction memory is divided four ways, i.e., there are 32 items that could reside in the same set. The five bits are needed to tell them apart. Each item also needs four bits of state. An item size of 128 bits gives an overhead of (5+4)/128 = 7 percent.

By adding another layer with eight 8-way set-associative directories, the maximum number of processors comes to 256. The size of the directories is the sum of the sizes of the attraction memories in their subsystems. A directory entry consists of six bits for the address tag and four bits of state per item, using a similar calculation as above. The overhead in the attraction memories is larger than in the previous example, because of the larger item space: eight bits of address tag and four bits of state. The total overhead per item is (6+4+8+4)/128 = 17 percent. A larger item sized would, of course, decrease these overheads.

The fact that the item space is slightly smaller than the sum of the attraction memories will also introduce a memory overhead, which has not been taken into account in the above calculations.

The absence of any entry in a directory has previously been interpreted as state invalid, since this is the state in which most of the items in the item space will reside. The replacement algorithm introduced a notion of a home bus for an item. If an item is most often found in its home bus and nowhere else, the absence of any entry in a directory could instead be interpreted as state exclusive, for items in its home subsystem, and as state invalid for items from outside. This would drastically cut down the size of a directory. The technique is only practical to a limited extent, however, since too

[5]This is the cache line size of the Motorola 88200.

small directories prevent items from moving out of their subsystems, and thus prevent sharing and migration, with drawbacks similar to those of NUMAs as a result.

Note that in a COMA, a "cached" item occupies only one space, while other shared-memory architectures require two spaces, one in the cache and one in the shared memory.

9 SIMULATED PERFORMANCE

The DDM is a general-purpose multiprocessor with a shared-memory view. As such, the architecture can be used for most of today's shared-memory applications. SICS has a tradition of implementing parallel Prolog systems for shared-memory multiprocessors. Since we have the insight in these systems, they have been part of our first evaluations. We can see no reason why the DDM could not perform well for other parallel applications, and are in the process of evaluating more commonly used benchmarks.

The DDM simulator models the architecture of the prototype and has a protocol similar to the one described in this paper. Four DDM nodes are connected by a DDM bus. Each node contains 8 Mbyte of four-way attraction memory and eight four-way, 16 kbyte processor caches for instruction and data. The item size is 16 bytes. The processor caches are fed with references generated by execution taking place in application processes running in parallel with the simulator process. All virtual addresses touched by an application process, both data and instructions, are detected. The translation to item identifiers (physical addresses) takes place in the modeled MMUs, performing all their necessary memory operations in the DDM. The execution-driven simulator runs at about 50 000 CPU-cycles per second on a SUN SPARC station.

In this study, we have used the OR-parallel Prolog system MUSE [AK90] optimized for Sequent Symmetry. It can be viewed as a large parallel application written in C, taking Prolog programs as inputs. The execution of parallel Prolog tends to be very irregular compared to traditional computing, having replaced regular structures like arrays with irregular structures like stacks. Load balancing is done dynamically and the parallelism is often fine-grained. The MUSE required no modifications in order to run in our simulator.

We have run benchmarks commonly used for evaluating parallel Prolog systems. The DDM showed a high processor utilization due to the high hit-rates in the attraction memories. The highest hit-rates is recorded for the coarse-grained queens(40) that took 70 Mcycles to run on a 16 processors machine. The 300 kitems, i.e., 5 Mbytes, touched during the execution were accessed with a average hit-rate of 96 percent in the attraction memories.

A more realistic benchmark was a natural language system from Unisys Paoli Research Center called Pundit. Pundit is interesting in that it is one of the largest Prolog applications and has a lot of OR-parallelism. We ran the benchmark phrase "starting air regulating valve failed." About 125 kitems, i.e., 2 Mbyte, were touched during the execution, resulting in an average hit rate of 91 percent in the attraction memories, and 98.5 percent in the processor caches. Each reference by the processor took on the average less than 1.3 cycles. The high hit-rate also resulted in a low DDM-bus utilization that ranged between 15–30 percent. The hit rate of one attraction memory and one processor over time is shown in Figure 11.

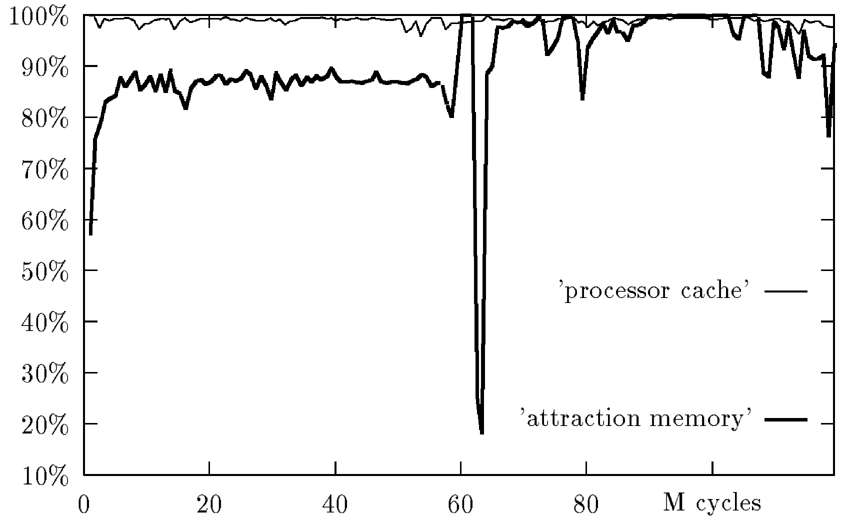

Figure 11: Simulated hit rate for an attraction memory and a processor cache running Pundit. The program is run a second time starting after 63 Mcycles.

10 RELATED ACTIVITIES

An operating system targeted for the DDM prototype is under development at SICS. This work is based on the Mach operating system from CMU that is modified to efficiently support the DDM. Other related activities at SICS involve a hardware prefetching scheme that dynamically prefetches items to the attraction memory, especially useful when a process is started or migrated. We are also experimenting with alternative protocols.

An emulator of the DDM is currently under development at the University of Bristol. The emulator runs on the Meiko Transputer platform. The modeled architecture has a tree-shaped link-based structure with Transputers as directories. Their four links allow for a branch factor of three at each level. The Transputers at the leaves execute the application. All references to global data are intercepted and handled in a DDM manner by software. The access to large Transputer facilities in the U.K. will allow for studying real-size problems running on hundreds of Transputers. The DDM protocol in the emulator has a different representation, which is suited for a link-based architecture structured like a tree, rather than a bus-based one. The implementation has certain similarities to directory-based systems.

11 CONCLUSION

We have introduced a new class of architectures, *cache-only memory architectures*, that allows for private caches of the largest size possible, since all data memory is used to implement the caches. The caches, which are kept coherent by a hardware protocol and have an extended functionality that handles replacement, are called *attraction memories*. A hierarchical bus structure has been described that ties a large number of attraction memories together and isolates the traffic generated by the hardware protocol to as small part of the machine as possible. The higher levels of the hierarchy can be split to overcome bandwidth bottlenecks, while still providing a race-free network.

The overhead of COMA explored in our hardware prototype is limited to 6 percent in the access time between the processor caches and the attraction memory, and a memory overhead of 7–17 percent for 32–256 processors.

12 RELATED WORK

The DDM has many similarities to Wilson's proposal [Wil86] for a hierarchical shared-memory architecture and certain similarities to the Wisconsin Multicube [GW88] and the TREEB architecture [VJS88]. However, all of these machines, unlike the DDM, depend on physically shared memory providing a "home" location for data. The Wisconsin Multicube can also be contrasted with the DDM in that certain requests need to be broadcast throughout the entire machine. Data moving closer to the processor accessing it can be found in the architecture of Mizrahi [MBLZ89]. The memory overhead of that architecture is much bigger than in the DDM, due to a low branch factor and full inclusion. It is also provided with a common shared memory and restricts writable objects to be cached to only one copy.

13 ACKNOWLEDGMENTS

SICS is a nonprofit research foundation sponsored by the Swedish National Board for Technical Development (STU), Swedish Telecom, LM Ericsson, ASEA Brown Boveri, IBM Sweden, Bofors Electronics, and, the Swedish Defence Material Administration (FMV). Work on the DDM is being carried out as part of the Esprit project 2741 PEPMA. We thank our many colleagues involved in or associated with the project. Especially we thank David H.D. Warren, University of Bristol, who is a coinventor of the DDM. Mikael Löfgren, SICS, wrote the DDM simulator and performed the simulations, basing his work on "Abstract Execution", (AE), which was provided to us by James Larus, University of Wisconsin. We are also grateful for the constructive criticism of the reviewers.

References

[AK90] K. A. M. Ali and R. Karlsson. The MUSE OR-parallel Prolog model and its performance. In *North American Conference on Logic Programming*. MIT Press, October 1990.

[EK89] S.J. Eggers and R.H. Katz. Evaluating the performance of four snooping cache coherency protocols. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 2–15, 1989.

[GW88] J.R. Goodman and P.J. Woest. The Wisconsin Multicube: a new large-scale cache-coherent multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture, Honolulu, Hawaii*, pages 442–431, 1988.

[HHW90] E. Hagersten, S. Haridi, and D.H.D. Warren. The cache-coherence protocol of the data diffusion machine. In M. Dubois and S. Thakkar, editors, *Cache and Interconnect Architectures in Multiprocessors*. Kluwer Academic Publisher, Norwell, Mass, 1990.

[HLH91] E. Hagersten, A. Landin, and S. Haridi. Multiprocessor consistency and synchronization through transient cache states. In M. Dubois and S. Thakkar, editors, *Scalable Shared-Memory Multiprocessors*. Kluwer Academic Publisher, Norwell, Mass, 1991.

[HomBC] Homer. *Odyssey*. 800 BC.

[LHH91] A. Landin, E. Hagersten, and S. Haridi. Race-free interconnection networks and multiprocessor consistency. In *To appear in Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991.

[MBLZ89] H. E. Mizrahi, J-L Baer, D.E. Lazowska, and J. Zahorjan. Introducing memory into the switch elements of multiprocessor interconnection networks. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 158–176, 1989.

[Ste90] P. Stenström. A survey of cache coherence for multiprocessors. *IEEE Computer*, 23(6), June 1990.

[VJS88] M.K. Vernon, R Jog, and G.S. Sohi. Performance analysis of hierarchical cache-consistent multiprocessors. In *Conference Proceedings of International Seminar on Performance of Distributed and Parallel Systems*, pages 111 – 126, 1988.

[WH88] D. H. D. Warren and S. Haridi. Data Diffusion Machine–a scalable shared virtual memory multiprocessor. In *International Conference on Fifth Generation Computer Systems 1988*. ICOT, 1988.

[Wil86] A. Wilson. Hierarchical cache/bus architecture for shared memory multiprocessor. Technical report ETR 86-006, Encore Computer Corporation, 1986.

Appendix

ATTRACTION MEMORY BELOW						
Trans-action	**State**					
	I	E	S	R	W	RW
read	R:r_A[1,2]	$\perp$	$\perp$	$\emptyset$	$\emptyset$	$\emptyset$
write [3]	RW:r_A[1,2]	$\perp$	W:e_A[2]	$\emptyset$	$\emptyset$	$\emptyset$
replace		I:i_A	I:o_A	$\emptyset$	$\emptyset$	$\emptyset$

TOP		
Trans-action	$a = 0 \rightarrow$	$a \geq 1 \rightarrow$
r	create item[4]	
e	x_B	x_B
d		
x		
o	i_B	
i	go home[4]	

ATTRACTION MEMORY ABOVE						
Trans-action	**State**					
	I	E	S	R	W	RW
r		*sel1*→S:d_A $\neg$*sel*→$\emptyset$	*sel1*→S:d_A $\neg$*sel*→S:d_A		*sel2*→ $\neg$*sel*→	
e		$\emptyset$	I:-	*sel1*→R:r_A $\neg$*sel*→	*sel1*→RW:r_A $\neg$*sel*→RW:-	*sel1*→RW:r_A $\neg$*sel*→
d		$\emptyset$		S:$\perp$		*sel1*→W:e_A $\neg$*sel*→
x		$\emptyset$	$\emptyset$	*sel1*→R:r_A $\neg$*sel*→	E:$\perp$	*sel1*→RW:r_A $\neg$*sel*→
o		$\emptyset$	*sel0*→ $\neg$*sel*→	*sel1*→S:$\perp$ $\neg$*sel*→S:-	*sel0*→	*sel2*→W:e_A $\neg$*sel*→
i	*seln*→[4] E:- $\neg$*sel*→	$\emptyset$	$\emptyset$	*sel1*→S:$\perp$ $\neg$*sel*→	$\emptyset$	*sel2*→W:e_A $\neg$*sel*→

$a = 0 \rightarrow$ No client tried to be selected.

$a \geq 1 \rightarrow$ At least one client tried to be selected.

$\emptyset$ This situation is impossible.

$\perp$ The processor may continue with its operation.

seln→ Selection of the nth priority succeeded. The priority increases with the number.

$\neg$*sel*→ The attempt to become selected failed.

[1] Preceded by a replace of the old item.

[2] The processor is suspended. The processor will be awakened by a $\perp$ for this very item.

[3] Read with intent to modify is treated like a write.

[4] Discussed further in Section 4.3.

Table 1: The protocol of the single-bus DDM. Each state has its own column and each transaction its own row. Actions have the format: guard→NEWSTATE:transaction-to-send$_{Index}$, where index A means to the bus above and index B means to the bus below. An empty square means no action; the rest of the symbols are explained in the table.

$\mathbf{E}$											
$r^2\ d^5$											
$\mathbf{E}\xrightarrow{3}\mathbf{A}\xrightarrow{5}\mathbf{S}$				$\mathbf{I}\xrightarrow{2}\mathbf{R}\xrightarrow{6}\mathbf{S}$				$\mathbf{I}$			
$r^3\ d^4$				$r^1\ d^6$							
$\mathbf{I}$	$\mathbf{S}\xrightarrow{4}\mathbf{S}$	$\mathbf{I}$	$\mathbf{S}$	$\mathbf{I}$	$\mathbf{I}\xrightarrow{1}\mathbf{R}\xrightarrow{7}\mathbf{S}$	$\mathbf{I}$	$\mathbf{I}$	$\mathbf{I}$	$\mathbf{I}$	$\mathbf{I}$	$\mathbf{I}$
P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12

r^1 Indicates a read transaction on the bus at phase 1.

$\mathrm{I}\xrightarrow{1}\mathrm{R}$ Indicates a state transition from state invalid to reading at phase 1.

The actions at the different phases are explained below.

[1] A read by P6 to an I item generates a *read* and changes the state to R.

[2] The directory detects a nonlocal action and repeats the *read* upward, changing its state to R.

[3] A directory in state E answers the request by changing its state to A, sending *read* below.

[4] One of the memories, P2, is selected to service the *read*. It stays in S and sends *data*.

[5] The directory in state A has promised to answer. It send *data* above and changes its state to S.

[6] The directory in state R is waiting for the *data*. It changes state to S and sends the *data* below.

[7] The attraction memory in state R is waiting for the *data*. It receives the *data* and changes state to S.

NOTE 1: Many subsystems on a bus may have an item in state S. Letting all of them reply with the *data* would produce unnecessary bus transactions; instead, one is selected in phase 4.

NOTE 2: After phase 3, the return path for data is marked with As and Rs.

Table 2: Multilevel read on a DDM with twelve processors (P1-P12). The table shows the states for one item, originally residing in state shared (S) in the attraction memories of processors P2 and P4. Their parent directories start with the item in state exclusive (E). The state transitions and transactions generated are indexed chronologically, showing a read attempt by processor P6.

$\mathbf{E}\xrightarrow{3}\mathbf{E}$ $e^2x^3r^5d^8$											
$\mathbf{S}\xrightarrow{3}\mathbf{I}$ e^3				$\mathbf{S}\xrightarrow{2}\mathbf{W}\xrightarrow{4}\mathbf{E}\xrightarrow{6}\mathbf{A}\xrightarrow{8}\mathbf{S}$ $e^1x^4r^6d^7$				$\mathbf{S}\xrightarrow{2}\mathbf{W}\xrightarrow{3}\mathbf{I}\xrightarrow{5}\mathbf{R}\xrightarrow{9}\mathbf{S}$ $e^1e^3r^4d^9e^{10}$			
I	$\mathbf{S}\xrightarrow{4}\mathbf{I}$	**I**	**I**	**I**	$\mathbf{S}\xrightarrow{1}\mathbf{W}\xrightarrow{5}\mathbf{E}\xrightarrow{7}\mathbf{S}$	**I**	**I**	**I**	$\mathbf{S}\xrightarrow{1}\mathbf{W}\xrightarrow{4}\mathbf{RW}\xrightarrow{10}\mathbf{W}$	**I**	**I**
P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12

^{0} Originally the item resides in state S in the AMs of P2, P6, and, P10.

^{1} P6 and P10 try to write to the shared item, generate *erase* (e) transactions, and change states to W.

^{2} The directories detect nonlocal *erase*s, change their states to W, and retransmit *erase*s above.

^{3} The *erase* originating in P6 is the winner and is carried on the top bus. All other directories change their states to I and retransmit the *erase* below.

^{4} P10 receives the bad news (*erase*). Instead of just invalidating it starts a read transaction.

^{5} P6 becomes the exclusive owner of the item and carries out the write.

$^{6-9}$ The *read* from P10 reaches P6, which changes state to S and sends *data* containing the new value. The *data* follows the path back to P10

^{10} The *data* reach P10, which changes state to W and once more sends an erase.

Table 3: Write race between processors P6 and P10. The erase request origin from processor P6 is the winner.

DIRECTORY BELOW						
Trans-	**States**					
action	I	E	S	R	W	A
r	R:r_A[1]					
e		E:x_B	W:e_A	$\emptyset$	$\emptyset$	W:e_A
d				$\emptyset$	$\emptyset$	S:d_A
o		$a = 0 \rightarrow$E:i_B	$a = 0 \rightarrow$I:o_A	$\emptyset$	$\emptyset$	$a = 0 \rightarrow$I:o_A $a \geq 1 \rightarrow$S:d_A
i[2]	$\emptyset$	$a = 0 \rightarrow$I:i_A	$\emptyset$	$\emptyset$	$\emptyset$	$a = 0 \rightarrow$I:o_A $a \geq 1 \rightarrow$A:r_B

DIRECTORY ABOVE						
Trans-	**States**					
action	I	E	S	R	W	A
r		*sel1*$\rightarrow$A:r_B $\neg$*sel*$\rightarrow\emptyset$	*sel1*$\rightarrow$A:r_B $\neg$*sel*$\rightarrow$		*sel2*$\rightarrow$ $\neg$*sel*$\rightarrow$	*sel2*$\rightarrow$ $\neg$*sel*$\rightarrow$
e		$\emptyset$	I:e_B	*sel1*$\rightarrow$R:r_A $\neg$*sel*$\rightarrow$	I:e_B	I:e_B
d		$\emptyset$		S:d_B		S:-
x		$\emptyset$	$\emptyset$	*sel1*$\rightarrow$R:r_A $\neg$*sel*$\rightarrow$	E:x_B	$\emptyset$
o		$\emptyset$	*sel0*$\rightarrow$	*sel1*$\rightarrow$S:d_B $\neg$*sel*$\rightarrow$S:d_B	*sel0*$\rightarrow$	*sel1*$\rightarrow$S:- $\neg$*sel*$\rightarrow$S:-
i	*seln*$\rightarrow$[3] E:i_B	$\emptyset$	$\emptyset$	*sel1*$\rightarrow$S:d_B $\neg$*sel*$\rightarrow$S:d_B	$\emptyset$	$\emptyset$

$a = 0 \rightarrow$ No client tried to be selected.

$a \geq 1 \rightarrow$ At least one client tried to be selected.

[1] If the corresponding set is full, an item x is chosen to be replaced.

[2] The transaction might be sent by the directory itself.

[3] Discussed further in Section 4.3.

Table 4: The protocol of directories of a hierarchical DDM.