ISRN SICS-R--91/19--SE

DDM - a Cache-Only Memory Architecture

by
Erik Hagersten, Anders Landin and Seif Haridi

SICS research report
R91:19
ISSN 0283-3638

DDM – A CACHE-ONLY MEMORY ARCHITECTURE

Erik Hagersten, Anders Landin and Seif Haridi

SICS Research Report R91:19 November 1991.*

Abstract

The long latencies introduced by remote accesses in a large multiprocessor can be hidden by caching. Caching also decreases the network load.

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 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), and its simulated performance for large applications. 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.

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

1 COMPARISON TO OTHER ARCHITECTURES

Existing architectures with shared memory are typically computers with one common bus connecting the processors to the shared memory, such as computers manufactured by Sequent and Encore, or with distributed shared memory, such as the BBN Butterfly and the IBM RP3.

Systems based on a single bus suffer from bus saturation and typically have some tens of processors, each one with a local cache. The contents of the caches are kept coherent by a cache-coherence protocol, in which each cache snoops the traffic on the common bus and prevents any inconsistencies from occurring [Ste90]. The architecture

*A revised version of R90017B. Swedish Institute of Computer Science; Box 1263 ; 164 28 KISTA ; SWEDEN. Email: {hag,landin,seif}@sics.se

provides a uniform access time to the whole shared memory, and is consequently called uniform memory architecture (UMA).

In architectures with distributed shared memory, 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. NUMAs often have networks other than a single bus, and the network delay might vary to different nodes. The earlier NUMAs did not have coherent caches, and left the coherence problem to the programmer. Research activities today are striving toward coherent NUMAs with directory-based cache-coherence protocols, e.g. Alewife [CKA91] and Dash [LLG+90]. Programs are often optimized for NUMAs by statically partitioning the work and data. Given a partitioning where the processors make most of their accesses to their part of the shared memory, a better scalability than for UMAs can be achieved.

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, the partitioning of data between the memories does not have to be static, since all distributed memories are organized like large (second-level) caches. The task of such a memory is twofold. Besides being a large (second-level) cache for the processor, it may also contain some data from the shared address space that the processor never has accessed, i.e., it is a cache and a virtual part of the shared memory at the same time. We call this intermediate form of memory *Attraction Memory*, (AM). A coherence protocol will attract the data used by a processor to its attraction memory. The coherence unit, comparable to a cache-line, which is moved around by the protocol is called an *item*. On a memory reference, a virtual address is translated into an item identifier. The item identifier space is logically the same as the physical address space of typical machines, but there is no permanent mapping between an item identifier and a physical memory location. Instead, an item identifier corresponds to a location in an attraction memory, whose tag matches the item identifier. Actually there are cases where multiple blocks could match.

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. Running an optimized NUMA program on a COMA architecture would result in a NUMA-like behavior, since the work spaces of the different processors would migrate to their attraction memories. However, an UMA version of the same program would give a similar behavior, since the data is attracted to the using processor regardless of the address. A COMA will also adapt to and perform well for programs with a more dynamic, or semi-dynamic scheduling. The work space migrates according to its usage throughout the computation. Programs can be optimized for a COMA to take this property into account in order to create a better locality.

COMA allows for a dynamic use of data without duplicating too much memory, compared to an architecture where a cached datum also occupies space in the shared memory. In order not to increase the memory cost, the attraction memories should be implemented with ordinary memory components. The COMA approach therefore should be viewed as a second-level, or higher level, cache technique in today's technology. The overhead required for accessing a large attraction memory compared to a large memory

as well as the increased amount of memory for implementation are surprisingly small. Figure 1 compares COMA to other shared-memory architectures.

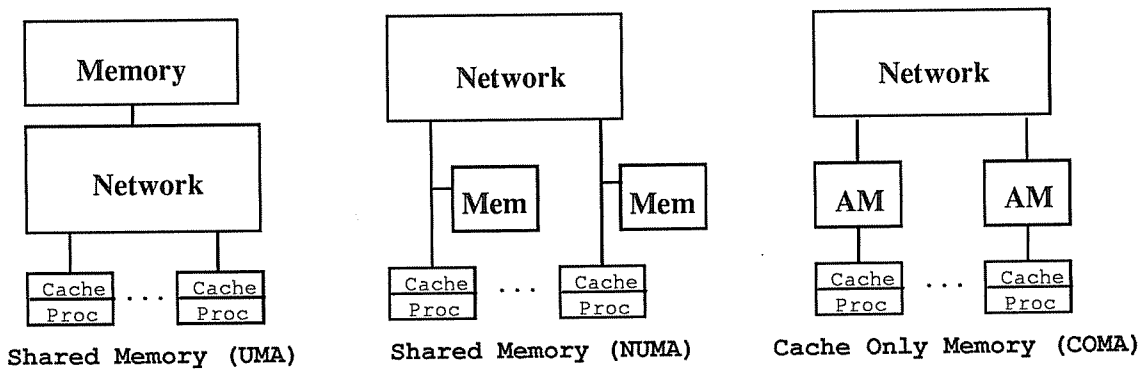

Figure 1: Shared-memory architectures compared to the COMA.

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. The paper ends with a brief overview to the ongoing prototype project and its 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. The cache-coherence protocol for a COMA can adopt existing techniques used in other cache-coherence protocols extended with the functionality for finding a datum on a cache read miss and for handling replacement. Coherence can either be kept by software or hardware. It is our belief that hardware-coherence is needed in a COMA for efficiency reasons, since the item must be small in order to prevent performance degradation by false sharing, i.e. two processors accessing different parts of the same item might conflict with each other even though they do not share any data. For instance, we have measured a speedup of 50% when false sharing was removed from an application [HALH91]. 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 discussed 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.

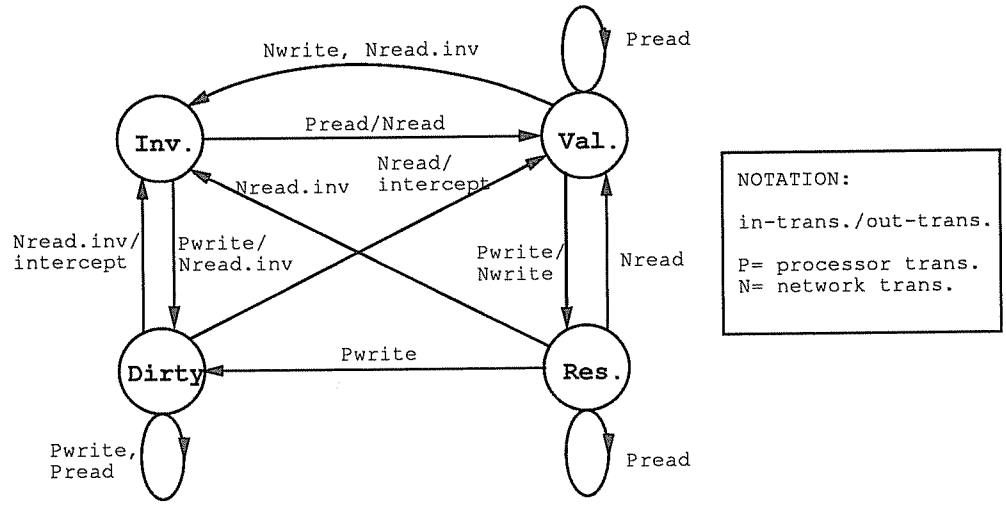

Figure 2: The write-once protocol.

Snooping caches, as described above, rely on broadcasting and are not suited for general interconnection networks: unrestricted broadcasting would drastically reduce the available bandwidth, thereby obviating the advantage of general networks. 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 which other caches have copies of the cache lines. Writing can now be performed with direct messages between all caches with copies.

3 A MINIMAL COMA

We will introduce the COMA architecture by first looking at the smallest instance of our architecture, the Data Diffusion Machine (DDM) [WH88]. 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. The architecture of the nodes in the single-bus DDM is shown in figure 3.

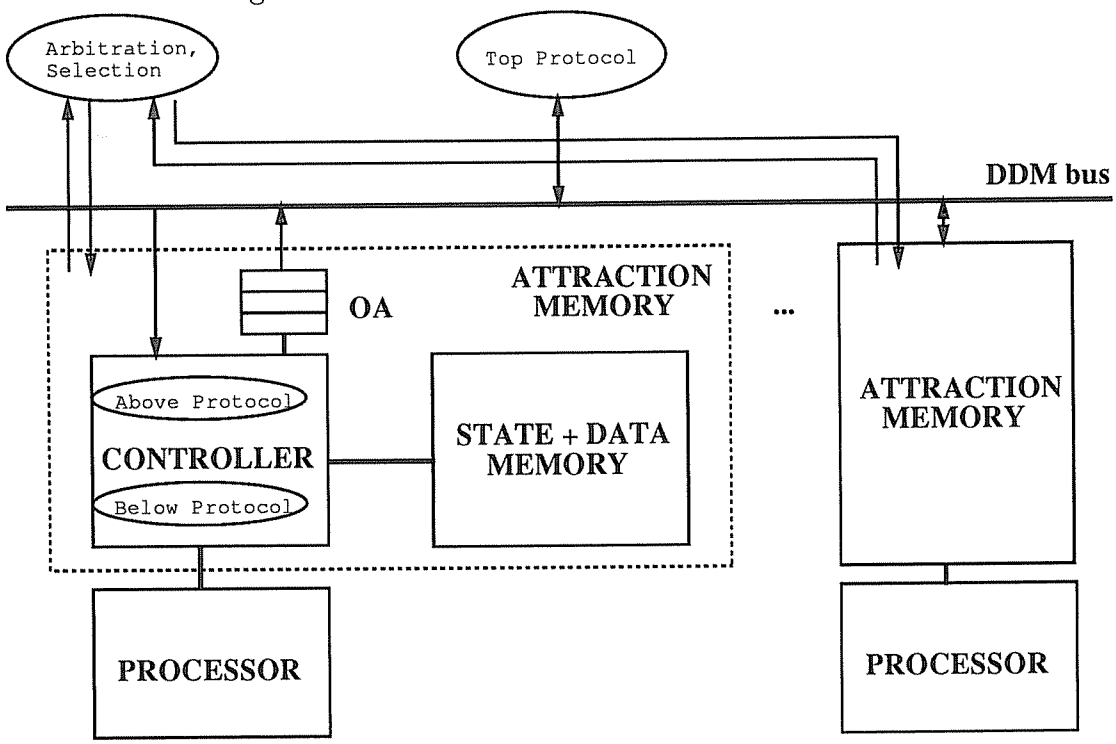

Figure 3: The architecture of a single-bus DDM. Below the attraction memories are the processors. Located on top of the bus are the arbitration, and selection.

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, making sure that at most one node is selected to service a request. This guarantees that each transaction on the bus does not produce more than one new transaction for the bus, a requirement necessary for deadlock avoidance.

3.1 The Protocol of the Single-Bus DDM

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, described in detail in [HHW90]. The write coherence part of the protocol is of write-invalidate type; i.e., in order to keep data coherent, all copies of the item but the one to be updated are erased on a write. In a COMA with a small item size, the alternative approach, write-broadcast, could also be attractive where, on a write, the new value is broadcast to all "caches" with a shared copy of the item [HALH91].

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 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

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.

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. 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. Acknowledge to an erase request.
r, read. Request to read a copy of the item.
d, data. Carries the data in reply to an earlier read request.
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.

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 attraction memory 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

Figure 4: A simplified representation of the attraction memory protocol not including replacement.

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 is the winner of the write race. All other *transactions* bound for the same item are removed from the small output buffers, OA. Therefore, the buffers also have to snoop transactions. The OA can be limited to a depth of three, and deadlock can still be avoided by the use of a special arbitration algorithm. The losing attraction memories in state W change state to RW while one of them puts a *read* request in its output buffer. 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.

3.2 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[1] 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. If an *out* transaction sees an attraction memory in either of states S, R, W, or RW it does nothing, otherwise it is 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 (state I), and secondly by replacing an item in state S, i.e. it will decrease the amount of sharing. If the item identifier space, that corresponds to the physical address space of conventional architectures, is not made larger than the sum of the attraction memory sizes, it is possible to device a simple scheme that guarantees a physical location for each item.

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

Often, only a portion of the physical address space is used in a computer. This is especially true for operating systems with an eager reclaiming of unused work space like Mach [Ras86]. In the DDM, the unused item space may be used to increase the degree of sharing, if the unused items are purged. The operating system might even change the degree of sharing dynamically.

3.3 Conclusion

What has been presented so far is a cache-coherent single bus multiprocessor without 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 above 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 of the same type in a hierarchy as shown in Figure 5. The

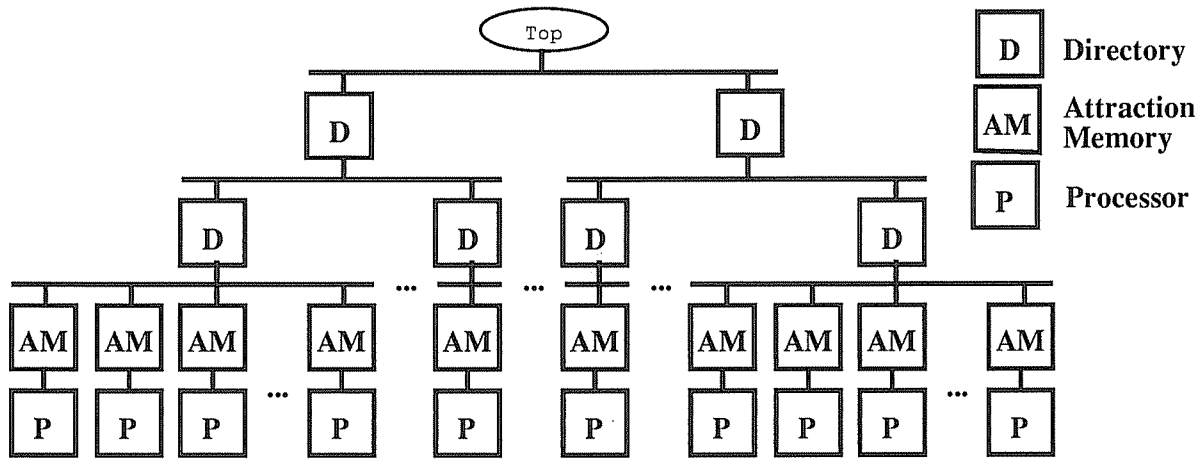

Figure 5: The hierarchical DDM, here with three levels.

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 in the exclusive state. This makes operations on 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

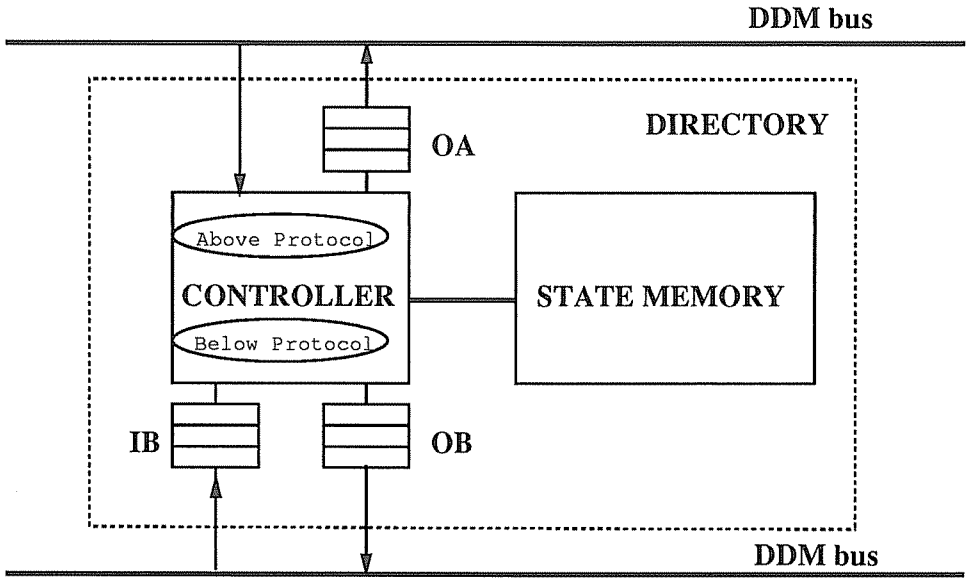

Figure 6: The architecture of a directory.

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

The hierarchical DDM and its protocol have several similarities with the proposed architectures by Wilson [Wil86], Vernon [VJS88] and Goodman [GW88]. The DDM is however different in its use of transient states in the protocol, its lack of physically shared memory and that only state-information and no data is stored in the network (higher level caches).

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 the state of the item to answering (A), marking an outstanding request from above, and retransmits the *read* request on its lower bus. The transient 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].

A flow control mechanism in the protocol prevents deadlock if too many processors try to roll out a red thread to the same set in a directory. 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). Often many processors try to read the same item, creating the "hot-spot

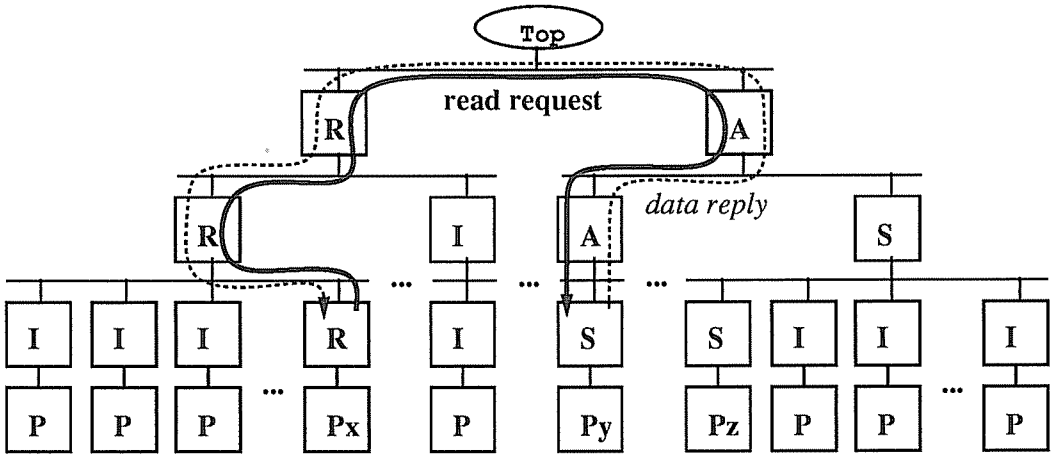

Figure 7: A read request from processor Px has found its way to a copy of the item in the attraction memory of processor Py. Its path is marked with states reading and answering (R and A), which will guide the data reply back to Px.

phenomenon [P+85]." 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 (in state RW) will restart a new write action automatically upon the reception of the erase.

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, W, or A is found. If the last copy of an item marked with state S is replaced, an

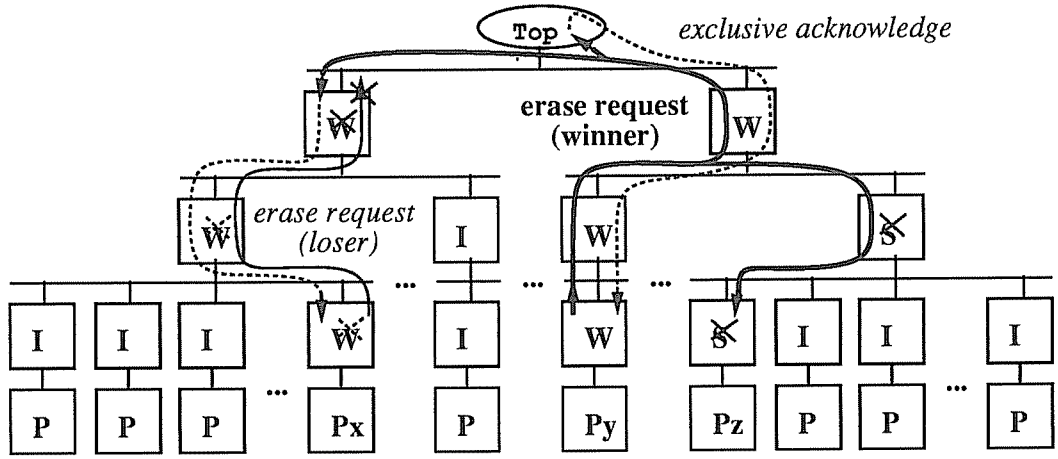

Figure 8: A write race between two processors, Px and Py, is resolved when the request originating from Py reaches the top bus (the lowest bus common to both processors.) The top can now send the acknowledge, *exclusive*, which follows the path marked with Ws back to the winning processor Py. The states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

out that fails to terminate will reach a directory in state E, and turned into an *inject*. Replacing an item in state E generates an *inject* transaction, trying to find an empty space in a neighboring attraction memory. *Inject* transactions will first try to find an empty space in the attraction memories of the local DDM bus, like for the single-bus DDM. Unlike the single-bus DDM, an inject failing to find an empty space on the local DDM bus will turn to a special bus, its home bus, determined by the item identifier. On the home bus, the *inject* will force itself into an attraction memory, possibly by throwing a foreigner and/or shared item out. The item home space is equally divided between the bottom-most buses, and therefore space is guaranteed on the home bus.

The preferred location, as described, is different from the memory location of NUMAs in that the notion of a home is only used at replacement after failing to find space elsewhere. When the item is not there, its place can be used by other items. The home also differ from NUMAs in it being a bus, i.e. any attraction memory on that bus will do. The details of the directory protocols can be found elsewhere [HHW90].

4.4 Replacement in a Directory

Baer and Wang have studied the multi-level inclusion property [BW88] with the following implications for our system: a directory at level $i+1$ has to be a superset of the directories, or attraction memories, at level i, i.e. the size of a directory and its associativity (number of ways) must be B_i times that of the underlying level i, where B_i is the branch factor of the underlying level i, and size means the number of items.
SIZE: $Dir_{i+1} = B_i * Dir_i$
ASSOCIATIVITY: $Dir_{i+1} = B_i * Dir_i$

Even if implementable, higher level memories would become expensive and slow

if those properties were fulfilled for a large hierarchical systems. The effects of the multi-level inclusion property are however limited in the DDM that only stores state information in its directories, and not replicate data in the higher levels. Yet another way to limit the effect is to use directories with smaller sets (less number of ways) than what is required for multi-level inclusion, called imperfect directories, and to endow the directories 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 in the DDM 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* [Lam79] system to the programmer. While fulfilling the strongest memory access 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 attraction memory with a copy. This not only reduces the remote delay, but also cuts down the number of transactions in the system. The writer might actually receive the acknowledge before all copies are erased. Still sequential consistency can be guaranteed [LHH91]. Looser forms of consistency providing a higher performance, can also be supported in an efficient way by the hierarchical structure [LHH91]. Yet another protocol which is write invalidate by default but changes strategy to write broadcast on a per-item basis has been proposed [HALH91]

5 INCREASING THE BANDWIDTH

Although most memory accesses tend to be localized in the machine, the higher level in the hierarchy may nevertheless demand a higher bandwidth than the lower systems, which creates a bottleneck. A way of taking the load off the higher levels is to have a smaller branch factor at the top of the hierarchy than lower down [VJS88]. This solution, however, increases the levels in the hierarchy, resulting in a longer remote access delay and an increased memory overhead. The higher levels of the hierarchy can instead be widened to become a fat tree [Lei85]. A directory can be split into two directories of half the size. The two directories deal with different address domains (even and odd). The communication with other directories is also split, which doubles the bandwidth. A split may be performed any number of times, and may be applied to any level of the hierarchy. Regardless of the number of splits, the architecture is still hierarchical to each specific address, as shown in Figure 9. Yet another solution is to use a heterogeneous network, i.e. the hierarchy with its advantages is used as far as possible and several hierarchies are tied together at their tops by a general network with a directory-based protocol. This scheme requires some changes in the protocol to achieve the same consistency model.

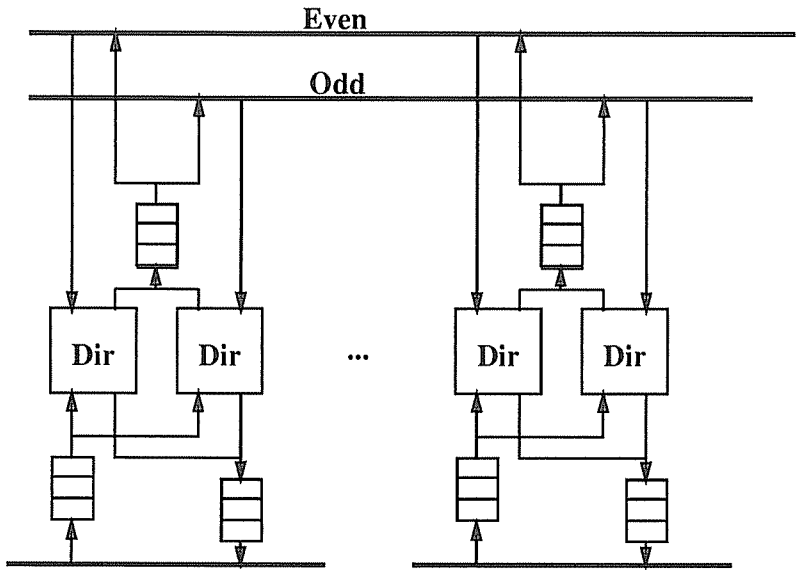

Figure 9: Increasing the bandwidth of a bus by splitting buses.

6 THE DDM PROTOTYPE PROJECT

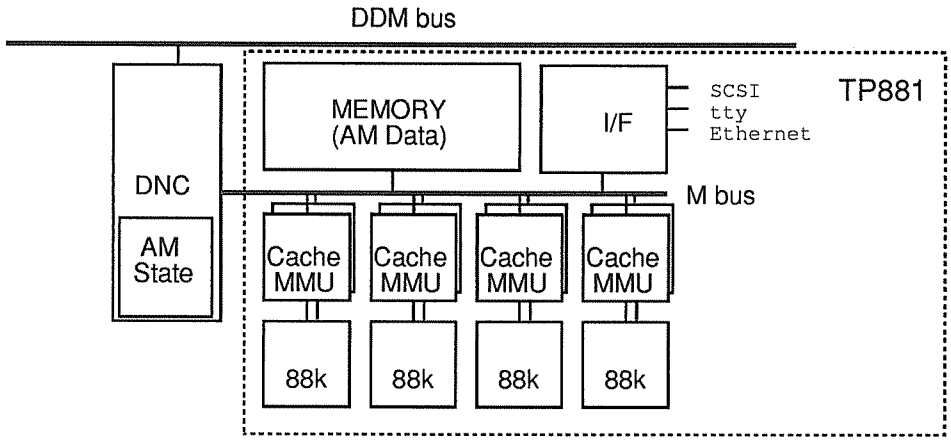

Figure 10: The implementation of a DDM node consisting of four processors sharing one attraction memory.

A prototype design of the DDM is near completion at SICS. The hardware implementation of the processor/attraction memory is based on the system TP881V by Tadpole Technology, U.K. Each such system has up to four Motorola 88100 20 MHz processors, each one with two 88200 16 kbyte cache/MMU, 8 or 32 Mbyte of DRAM, and interfaces for the SCSI-bus, Ethernet, and terminals, all connected by the Motorola Mbus.

A DDM Node Controller (DNC) board, hosting a two-way set-associative single-ported 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*, and also snoops the DDM bus according to the protocol *memory above*. The protocol used has

been modified to integrate the copy-back protocol of multiple processor caches into the protocol mechanisms described. The DNC thus changes the behavior of the memory into a two-way set-associative attraction memory. Read accesses to the attraction memory take eight cycles per cache line, which is one more than in the original system 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 16 % 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. Write accesses to state S takes at best 30 cycles for one level and 45 cycles for two levels.

CPU-access	State in AM	Delay, one level (cycles)	Delay, two levels (cycles)
read	I	65	85
write	S	30	45
write	I	80	125

The DDM bus is pipelined in four phases: transaction code, snoop, selection, and data. We have decided to make an initial conservative bus design, since pushing the bus speed is 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. Still, the bandwidth has not been the limiting factor in our simulation studies. The bandwidth of a bus can be increased many times by using other structures. The slotted ring bus proposed by Barosso and Dubois [BD91] has a one order of magnitude higher bandwidth.

The DDM uses the normal procedures for translating virtual addresses to physical addresses, as implemented in standard MMUs, for translations to item identifiers. This implies that the knowledge of physical pages appears to an operating system.

Any attraction memory node can have a disk connected. Upon a page-in, the node first attracts all the data of an item page ("physical page") as being temporarily locked to its attraction memory. If the items of that page was not present in the machine earlier, they will get born at this time by the protocol. Secondly it copies (by DMA) the page from the disk to the attraction memory, unlocking the data at the same time. Page-out reverses the process, possibly by copying a dirty page back to the disk. The operating system might decide to purge the items of some unused pages, in favor for more sharing.

7 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, i.e. the cache line size of the Motorola 88200.

A 32-processor DDM, i.e., a one-level DDM with a maximum of eight two-way set-associative attraction memories, needs four bits of address tag per item, regardless of the attraction memory size. As stated before, the item space is not larger 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 two ways, i.e., there are 16 items that could reside in the same set. The four 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 (4+4)/128 = 6%.

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: seven bits of address tag and four bits of state. The total overhead per item is (6+4+7+4)/128 = 16%. A larger item sized would, of course, decrease these overheads.

An optimization to minimize the memory overhead involves a different interpretation of the implicit state for different parts of the item space. The absence of an entry in a directory has previously been interpreted as state invalid. 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 an 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 small directories restrict the number of items moving out of their subsystems, and thus limits sharing and migration, with drawbacks similar to those of NUMAs as a result.

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

8 SIMULATED PERFORMANCE

In this study, we have used an execution-driven simulation environment allowing for large programs running on many processors to be studied at a reasonable time. The DDM simulation model is parameterized with data from our ongoing prototype project, and accurately describes its behavior including the compromises introduced by taking an existing commercial product as a starting point. The model also describes parts of the virtual memory handling system. Attraction memories are 2-way 1 Mbyte in this study. A protocol similar to the one described in this paper has been used, providing sequential consistency.

APPLICATION	Water		MP3D	-DIFF	Cholesky		Matrix	MUSE
INPUT DATA	192 mols.	384 mols.	75.000 part.	75.000 part.	m14 (small)	m15 (large)	500 x 500	Pundit
COLDSTART INCL?	yes	yes	no	no	yes	yes	yes	no
DDM TOPOLOGY	2x8x4	2x8x4	2x8x2	2x8x2	2x8x2	2x8x2	8x4	4x4
HIT RATES(data)								
Dcache (%)	99.0	98.9	86	92	96	89	92	98.5
Attr. Mem. (%)	44	65	40	88	6	74	98	91
RAR (data) (%)	0.6	0.4	8.4	1.0	3.8	2.8	0.16	0.20
BUS UTILIZATION								
Mbus (%)	31	26	86	54	70	60	55	-
Lower DDMbus (%)	39	30	88	24	80	66	-	-
Top DDMbus (%)	25	20	66	13	70	49	4	-
Speedup/#Proc.	52/64	-/64	6/32	19/32	10/32	17/32	29/32	-/16

Table 1: Statistics from DDM simulations. Hit rate statistics are for data only, except MUSE, where a unified I+D cache was used. RAR is the remote access rate, i.e., the ratio of the data accesses issued by a processor that creates remote coherence traffic. Note that an increased working set results in less load on the busses for Water and Cholesky.

We have studied the parallel execution of the Stanford Parallel Applications for Shared Memory (SPLASH) [SWG91], the OR-parallel Prolog system MUSE and a matrix multiplication program, representing applications from engineering computing and symbolic computing. All programs were originally written for UMA architectures (Sequent Symmetry or Encore Multimax) and use static or dynamic scheduler algorithms. They adapt well for a COMA without any changes. The details of this study can be found elsewhere [HALH91]. All programs take in the order of one CPU minute to run sequentially, without any simulations, on a SUN SPARCstation. The speedups reported in Figure 11 and Table 1 are self-relative to the execution of a single DDM-node with one processor, assuming a 100% hit rate in the attraction memory.

The **SPLASH-Water** program simulates the movements of water molecules. Its execution time is $O(m^2)$, where m is the number of molecules. Therefore it is often simulated with a small working set, in this case 192 molecules and a working set of 320 kbyte, i.e. the 96 processors in Figure 11 each handles only two molecules. Most of the locality in the small working set can be explored of the processor cache, and only about 44 % of the transactions reaching the attraction memory will hit. A real-sized working set would still have the same good locality, and would benefit more from the large attraction memories in order to maintain the good speedup. Justified by a single run with 384 molecules in Table 1.

The **SPLASH-MP3D** program is a wind tunnel simulator where a good speedup is harder to achieve, due to a high invalidation frequency resulting a poor hit rate. The program is often run with the memory filled with data structures representing particles, divided equally between the processors. The 3D space is divided into space-

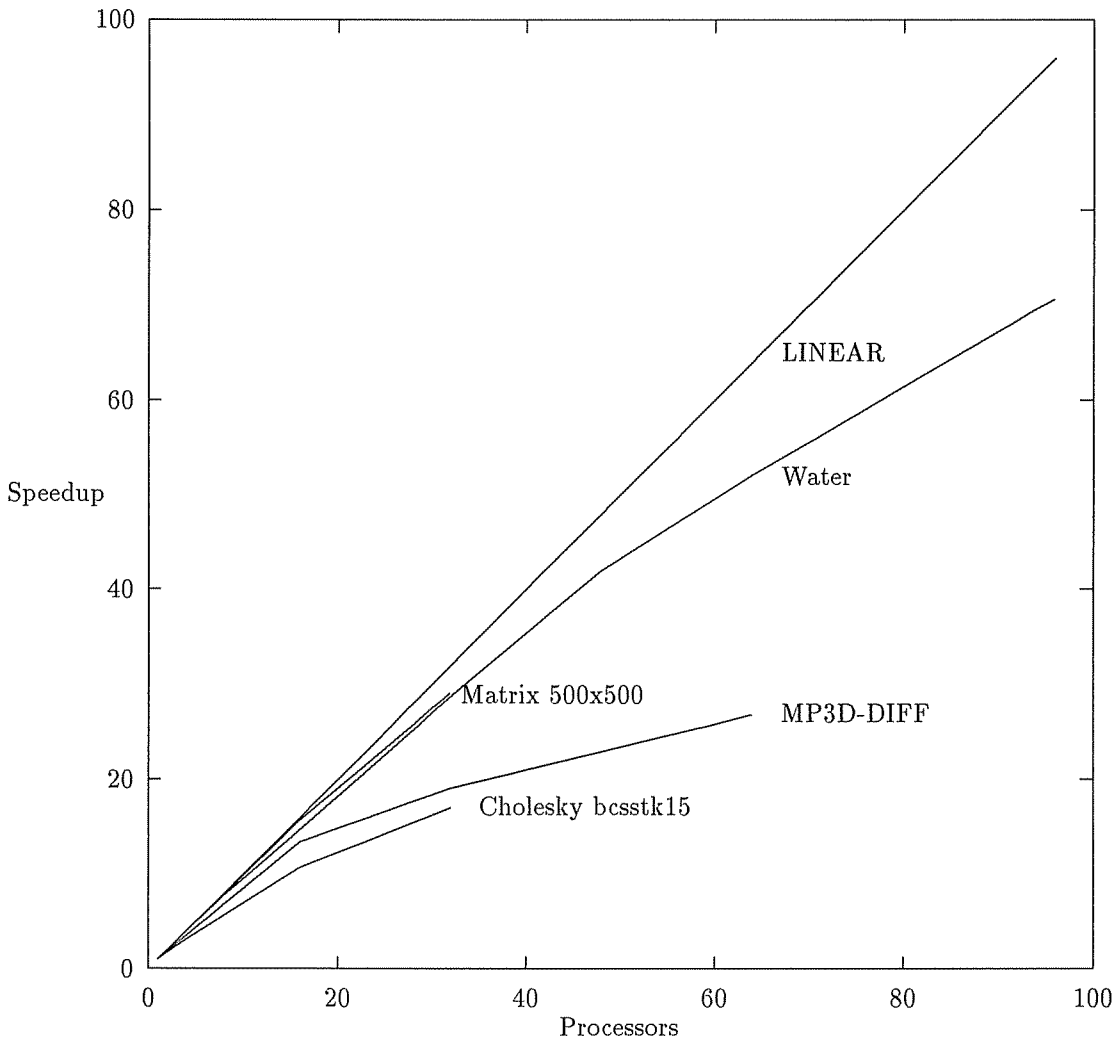

Figure 11: The speedup curves for some of the reported programs.

cells represented by data structures. MP3D is run in time phases, where each particle is moved once each phase. Moving a particle involves updating the state of the particle and also the state of space-cell where the molecule currently resides, i.e., all processors write to all the space-cells resulting in a poor locality. 95% of the misses we explore in the DDM are due to this write-invalidate effect. We simulate 75.000 particles, i.e. a working set of 4 Mbyte.

MP3D-DIFF is a rewritten version of the program, where a better hit rate is achieved. The distribution of particle over processors is here based on their current location in space, i.e. all particles in the same space-cells are handled by the same processor. The update of the both the particle state and the space cell state are now local to the processor. When a particle is moved across a processor boarder, its data is handled by a new processor, i.e., the particle data diffuse to the attraction memory of the new processor. The rewriting added some 30 extra lines and requires a COMA

architecture, where data can live anywhere, to run well.

SPLASH-Cholesky factorises a sparsely positive definite matrix. The matrix is divided into supernodes that are put in a global task queue to be picked up by any worker, i.e. the scheduling is dynamic. We have used the large input matrix bcsstk15 (m15), which occupies 800 kbytes unfactored, and 7,7 Mbytes factored. The nature of the Cholesky algorithm limits the available parallelism, that depends on the size of the input matrix. As a comparison, a run with the smaller matrix bcsstk14 (m14) of 420 kbytes unfactored and 1.4 Mbytes factored is presented in Table 1.

Matrix performs plain matrix multiplication on a 500x500 matrix using a blocked algorithm. The working set is about 3 Mbyte.

MUSE is an OR-parallel Prolog system implemented in C at SICS. The large natural language system Pundit from Unisys Paoli Research Center is used as an input. An active working set of 2 Mbytes is touched during the execution. MUSE distributes work dynamically, and shows a good locality on a COMA. MUSE was run on an earlier version of the simulator, and some of the statistics are therefore not reported in Table 1.

9 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 [Ras86] 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 [RW91]. 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 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.

10 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. Simulation show that the COMA principle behaves well for programs originally written for UMA architectures, and that the slow busses of our prototype still allows for many processors to be connected. The overhead of COMA

explored in our hardware prototype is limited to 16% in the access time between the processor caches and the attraction memory, and a memory overhead of 6–16 % for 32–256 processors.

11 ACKNOWLEDGMENTS

SICS is a nonprofit research foundation sponsored by the Swedish National Board for Technical Development (NUTEK), Swedish Telecom, Ericsson Group, ASEA Brown Boveri, IBM Sweden, Nobel Tech System AB, and, the Swedish Defence Material Administration (FMV). Part of the work on the DDM is being carried out within the Esprit project 2741 PEPMA. We thank our many colleagues involved in or associated with the project. Especially David H.D. Warren, University of Bristol, who is a coinventor of the DDM. Mikael Löfgren, SICS, wrote the DDM simulator, basing his work on "Abstract Execution", (AE), which was provided to us by James Larus, University of Wisconsin.

References

[BD91] L. Barroso and M. Dubois. Cache Coherence on a Slotted Ring. In *Proceedings of the International Conference on Parallel Processing*, pages 230–237, 1991.

[BW88] J-L. Baer and W-H. Wang. On the Inclusion Properties for Multi-Level Cache Hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 73–88, 1988.

[CKA91] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the 4th Annual ASPLOS*, 1991.

[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.

[HALH91] E. Hagersten, P. Andersson, A. Landin, and S. Haridi. A Performance Study of the DDM – a Cache-Only Memory Architecture. Swedish Institute of Computer Science, Report R91:17. Submitted to ISCA92, 1991.

[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.

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

[Lam79] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[Lei85] C.E. Leiserson. Fat Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, pages 892–901, Oct. 1985.

[LHH91] A. Landin, E. Hagersten, and S. Haridi. Race-free Interconnection Networks and Multiprocessor Consistency. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991.

[LLG+90] D. Lenoski, J. Laundo, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, 1990.

[P+85] G.F. Pfister et al. The IBM Research Parallel Processor Prototype (RP3). In *Proceedings of the 1985 International Conference on Parallel Processing, Chigago*, 1985.

[Ras86] R Rashid. Threds of a new system (Mach). *UNIX Review*, 4(8):386–392, August 1986.

[RW91] S. Raina and D.H.D Warren. Traffic Patterns in a Scalable Multiprocessor through Transputer Emulation. In *International Hawaii Conference on System Science*, 1991.

[Ste90] P. Stenström. A Survey of Cache Coherence for Multiprocessors. *IEEE Computer*, 23(6), June 1990.

[SWG91] J.S. Sing, W.-D. Weber, and A Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. Stanford University, Report, April 1991.

[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.