DDM — A Cache-Only Memory Architecture

Erik Hagersten, Anders Landin, and Seif Haridi
Swedish Institute of Computer Science

A new architecture has the programming paradigm of shared-memory architectures but no physically shared memory. Caches attached to the processors contain all the system memory.

Multiprocessors providing a shared memory view to the programmer are typically implemented as such — with a shared memory. We introduce an architecture with large caches to reduce latency and network load. Because all system memory resides in the caches, a minimum number of network accesses are needed. Still, it presents a shared-memory view to the programmer.

Single bus. Shared-memory systems based on a single bus have some tens of processors, each one with a local cache, and typically suffer from bus saturation. A cache-coherence protocol in each cache snoops the traffic on the common bus and prevents inconsistencies in cache contents.[1] Computers manufactured by Sequent and Encore use this kind of architecture. Because it provides a uniform access time to the whole shared memory, it is called a uniform memory architecture (UMA). The contention for the common memory and the common bus limits the scalability of UMAs.

Distributed. Computers such as the BBN Butterfly and the IBM RP3 use an architecture with distributed shared memory, known as a nonuniform memory architecture (NUMA). Each processor node contains a portion of the shared memory, so 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 can vary to different nodes. The earlier NUMAs did not have coherent caches and left the problem of maintaining coherence to the programmer. Today, researchers are striving toward coherent NUMAs with directory-based cache-coherence protocols.[2] By statically partitioning the work and data, programmers can optimize programs for NUMAs. A partitioning that enables processors to make most of their accesses to their part of the shared memory achieves a better scalability than is possible in UMAs.

Cache-only. In a cache-only memory architecture (COMA), the memory organization is similar to that of a NUMA in that each processor holds a portion of the address space. However, the partitioning of data among 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 cache for the processor, it may also contain some data from the shared address space that the processor never has accessed — in other words, it is a cache and a virtual part of

0018-9162/92/0900-0044$03.00 © 1992 IEEE

the shared memory. We call this intermediate form of memory *attraction memory*. A coherence protocol attracts the data used by a processor to its attraction memory. Comparable to a cache line, the coherence unit 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 locations of different attraction memories could match.

A COMA provides a programming model identical to that of shared-memory architectures, but it does not require static distribution of execution and memory usage to run efficiently. Running an optimized NUMA program on a COMA results in a NUMA-like behavior, since the work spaces of the different processors migrate to their attraction memories. However, a UMA version of the same program would have a similar behavior, because the data are attracted to the using processor regardless of the address. A COMA also adapts to and performs well for programs with a more dynamic or semidynamic 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 to achieve a better locality.

A COMA allows for dynamic data use without duplicating much memory, compared with an architecture in which a cached datum also occupies space in the shared memory. To avoid increasing the memory cost, the attraction memories should be implemented with ordinary memory components. Therefore, we view the COMA approach as a second-level, or higher level, cache technique. The accessing time to the attraction memory of a COMA is comparable to that to the memory of a cache-coherent NUMA. Figure 1 compares COMAs to other shared-memory architectures.

A new COMA. This article describes the basic ideas behind a new COMA. The architecture, called the Data Diffusion Machine (DDM),[3] relies on a hierarchical network structure. We introduce the key ideas behind DDM by describing a small machine and its protocol. We also describe a large machine with hundreds of processors, overview the ongoing prototype project, and provide simulated performance figures.

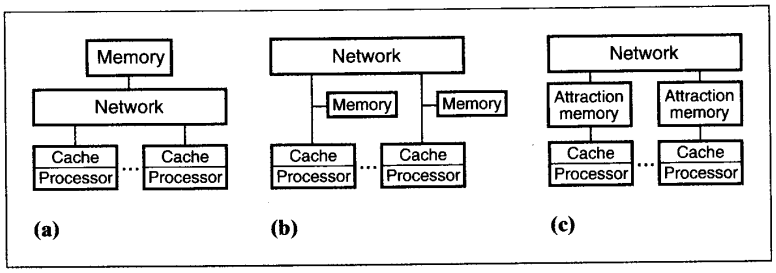

Figure 1. Shared-memory architectures compared with COMAs: (a) uniform memory architecture (UMA), (b) nonuniform memory architecture (NUMA), and (c) cache-only memory architecture (COMA).

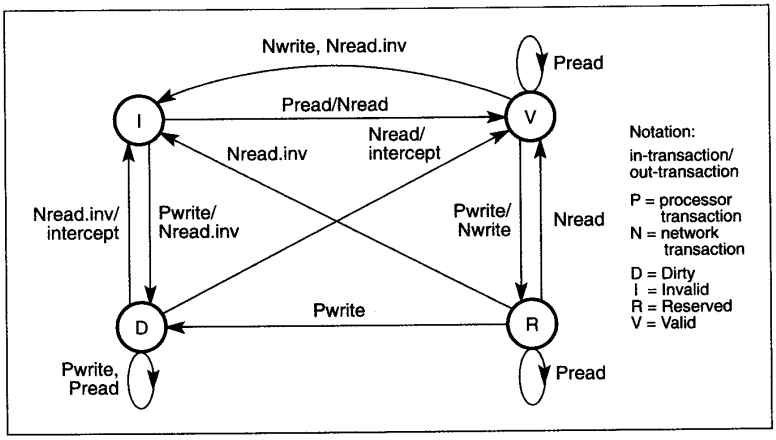

Figure 2. An example of a protocol similar to the write-once protocol.

Cache-coherence strategies

The problem of maintaining coherence among read-write data shared by different caches has been studied extensively. Either software or hardware can maintain coherence. We believe hardware coherence is needed in a COMA for efficiency, since the item must be small to prevent performance degradation by false sharing. (In false sharing, two processors accessing different parts of the same item conflict with each other, even though they do not share any data.) We measured a speedup of 50 percent when false sharing was removed from the wind tunnel application, MP3D-Diff, reported in the "Simulated performance" section. Hardware-based schemes maintain coherence without involving software and can 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 actions to avoid an incoherence. An example of such a protocol is the write-once protocol introduced by Goodman and discussed by Stenström.[1] As Figure 2 shows, in that protocol, each cache line can be in one of four states: Invalid, Valid, Reserved, or Dirty. Many caches can 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 now differs 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 to provide the new value (marked by "intercept" in Figure 2).

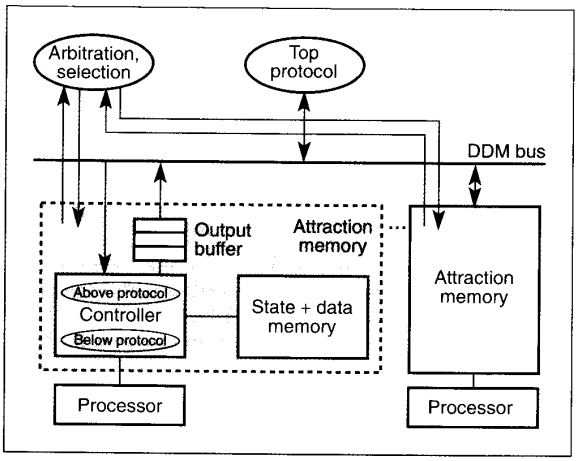

Figure 3. The architecture of a single-bus DDM. Below the attraction memories are the processors. On top of the bus are arbitration and selection.

Snooping caches 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.[1] A read request is sent to main memory, without any snooping. The main memory knows if the cache line is cached — and in which cache or caches — and whether 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. On a write to a shared cache line, a write request sent to the main memory causes invalidation messages to all caches with copies to be sent. The caches respond with acknowledge messages. To achieve sequential consistency, all acknowledgments must be received before the write is performed.

The cache-coherence protocol for a COMA can adopt techniques used in other cache-coherence protocols and extend them with the functionality for finding a datum on a cache read miss and for handling replacement. A directory-based protocol could have a part of the directory information, the *directory home*, statically distributed in a NUMA fashion, while the data would be allowed to move freely. Retrieving the data on a read miss would then require one extra indirect access to the directory home to find where the item currently resides. The access time, including this extra indirection, would be identical to that required for reading a dirty cache line not in a NUMA's home node. The directory home can also make sure that the last copy of an item is not lost.

Instead of the above strategy, DDM is based on a hierarchical snooping bus architecture and uses a hierarchical search algorithm for finding an item. The directory information in DDM is dynamically distributed in the hierarchy.

A minimal COMA

We introduce DDM by looking at the smallest instance of the architecture, which could be a COMA on its own or a subsystem of a larger COMA. A single bus connects the attraction memories of the minimal DDM. The distribution and coherence of data among the attraction memories are controlled by the snooping protocol *memory above*, and the interface between the processor and the attraction memory is defined by the protocol *memory below*. The protocol views a cache line of an attraction memory, here called an item, as one unit. The attraction memory stores one small state field per item. Figure 3 shows the node architecture in the single-bus DDM.

DDM uses an asynchronous split-transaction bus: The bus is released between a requesting transaction and its reply, for example, 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 to make 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.

Single-bus DDM protocol. 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.[4] The write coherence part of the protocol is the write-invalidate type: To keep data coherent, all copies of the item except the one to be updated are erased on a write. In a COMA with a small item size, the alternative approach, write update, could also be attractive: On a write, the new value is multicast to all "caches" with a shared copy of the item.

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 and 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 × transaction → new state × new transaction

An item can be in one of seven states (the subsystem is the attraction memory):

- *Invalid.* This subsystem does not contain the item.
- *Exclusive.* This subsystem and no other contains the item.
- *Shared.* This subsystem and possibly other subsystems contain the item.
- *Reading.* This subsystem is waiting for a data value after having issued a read.

- *Waiting.* This subsystem is waiting to become Exclusive after having issued an erase.
- *Reading-and-Waiting.* This subsystem is waiting for a data value, later to become Exclusive.
- *Answering.* This subsystem has promised to answer a read request.

The first three states — Invalid, Exclusive, and Shared — correspond to the states Invalid, Reserved, and Valid in Goodman's write-once protocol. The state Dirty in that protocol — with the meaning that 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 Reading, Waiting, Reading-and-Waiting, and Answering. Transient states are required because of the split-transaction bus and the need to remember outstanding requests.

The bus carries the following transactions:

- *Erase.* Erase all copies of this item.
- *Exclusive.* Acknowledge an erase request.
- *Read.* Read a copy of the item.
- *Data.* Carry the data in reply to an earlier read request.
- *Inject.* Carry the only copy of an item and look for a subsystem to move into — caused by a replacement.
- *Out.* Carry the item 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 Exclusive state or reading an item in Exclusive or Shared state proceeds without interruption. As Figure 4 shows, a read attempt of an item in Invalid will result in a Read request and a new state, Reading. The bus selection mechanism will select one attraction memory to service the request, eventually putting a Data transaction on the bus. The requesting attraction memory, now in Reading, will grab the Data transaction, change to Shared, and continue.

Processors are allowed to write only to items in Exclusive state. If the item is in Shared, all other copies have to be erased and an acknowledgment received before the writing is allowed. The attraction memory sends an Erase transaction and waits for the Exclusive acknowledgment in the new state, Waiting. Many simultaneous attempts to write the same item will result in many attraction memories in Waiting, all with an outstanding Erase transaction in their output buffers. The first Erase to reach the bus is the winner of the write race.

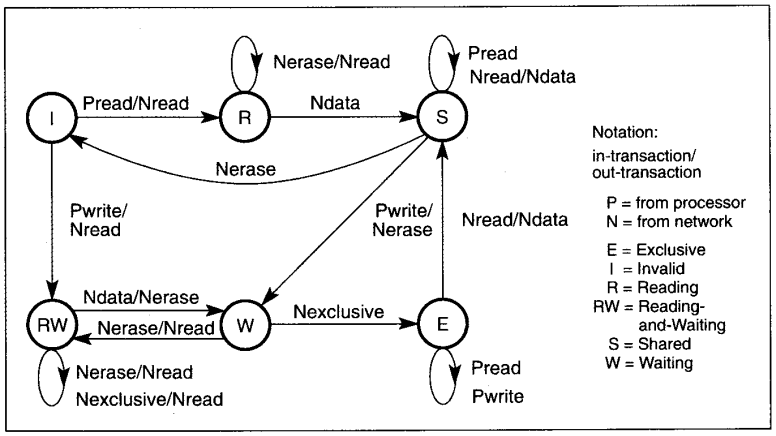

Figure 4. A simplified representation of the attraction memory protocol not including replacement.

All other transactions bound for the same item are removed from the small output buffers. Therefore, the buffers also have to snoop transactions. The output buffers can be limited to a depth of three, and deadlock can still be avoided with a special arbitration algorithm. The losing attraction memories in Waiting change state to Reading-and-Waiting, while one of them puts a read request in its output buffer. Eventually the top protocol of the bus replies with an Exclusive acknowledgment, telling the only attraction memory left in Waiting that it may now proceed. Writing to an item in the Invalid state results in a Read request and a new state, Reading-and-Waiting. Upon the Data reply, the state changes to Waiting and an Erase request is sent.

Replacement. Like ordinary caches, the attraction memory will run out of space, forcing some items to make room for more recently accessed ones. If the set where an item is supposed to reside is full, one item in the set is selected to be replaced. For example, the oldest item in Shared, of which there might be other copies, may be selected. Replacing an item in Shared generates an Out transaction. The space used by the item can now be reclaimed. If an Out transaction sees an attraction memory in Shared, Reading, Waiting, or Reading-and-Waiting, 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 Exclusive. 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 first by choosing an empty space (Invalid state), and second by replacing an item in Shared state — in other words, it will decrease the amount of sharing. If the item identifier space, which 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 devise a simple scheme that guarantees a physical location for each item.

Often a program uses only a portion of a computer's physical address space. This is especially true of operating systems with a facility for eager reclaiming of unused work space. In DDM, the unused item space can be used to increase the degree of sharing by purging the unused items. The operating system might even change the degree of sharing dynamically.

The hierarchical DDM

So far, we have presented a cache-coherent single-bus multiprocessor without physically shared memory. Instead, the resources form huge second-level caches called attraction memories, minimizing the number of accesses to the

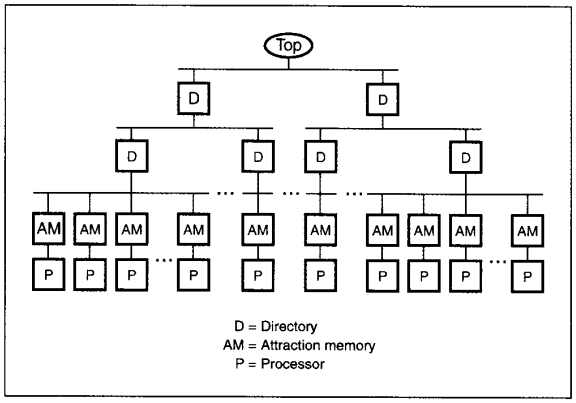

Figure 5. A hierarchical DDM with three levels.

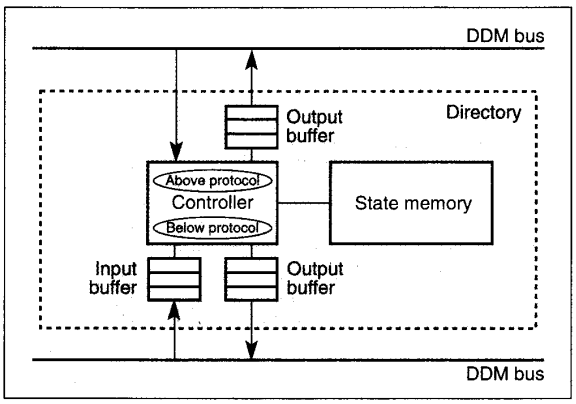

Figure 6. The architecture of a directory.

only shared resource left: the shared bus. Data can reside in any or many of the attraction memories. Data are automatically moved where needed.

To make the single-bus DDM a subsystem of a large hierarchical DDM, we replace the top with a directory, which interfaces between the bus and a higher level bus of the same type. Figure 5 shows the hierarchy.

The directory is a set-associative state memory that keeps information for all the items in the attraction memories below it, but contains no data. The directory can answer these questions: "Is this item below me?" and "Does this item exist outside my subsystem?" From the bus above, the directory's snooping protocol *directory above* behaves very much like the *memory above* protocol. From the bus below, its *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. The directory passes through only transactions from below that cannot be completed inside its subsystem or transactions from above that need to be serviced by its subsystem. In that sense, the directory acts as a filter.

As Figure 6 shows, the directory has a small output buffer above it to store transactions waiting to be sent on the higher bus. Transactions for the lower bus are stored in another output buffer below, and transactions from the lower bus are stored in an input buffer. A directory reads from the input buffer 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 architectures proposed by Wilson[5] and Goodman and Woest.[6] DDM is, however, different in its use of transient states in the protocol, its lack of physically shared memory, and its network (higher level caches) that stores only state information and no data.

Multilevel read. If the subsystems connected to the bus cannot satisfy a read request, the next higher directory retransmits the request on the next higher bus. The directory also changes the item's state to Reading, 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 item's state to Answering, marking an outstanding request from above, and retransmits the Read request on its lower bus. As Figure 7 shows, the transient states Reading and Answering in the directories mark the request's path through the hierarchy, like an unwound red read thread that shows the way through a maze, appearing in red in Figure 7.

A flow-control mechanism in the protocol prevents deadlock if too many processors try to unwind a read 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 read thread back to the requesting node, changing all the states along the path to Shared.

Combined reads and broadcasts are simple to implement in DDM. If a Read request finds the read thread unwound for the requested item (Reading or Answering state), it simply terminates and waits for the Data reply that eventually will follow that path on its way back.

Multilevel write. An Erase from below to a directory with the item in Exclusive state results in an Exclusive acknowledgment being sent below. An Erase that cannot get its acknowledgment from the directory will work its way up the hierarchy, changing the directories' states to Waiting to mark the outstanding request. All subsystems of a bus carrying an Erase transaction will get their copies erased. The propagation of the Erase ends when it reaches a directory in Exclusive (or the top), and the acknowledgment is sent back along the path marked Waiting, changing the states to Exclusive.

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 Reading-and-Waiting) will restart a new write action automatically upon receipt of the erase.

Replacement in the hierarchical DDM. Replacement of a Shared item in the hierarchical DDM results in an Out transaction propagating up the hierarchy and terminating when it finds a subsystem in any of the following states: Shared, Reading, Waiting, or Answering. If the last copy of an item marked Shared is replaced, an Out transaction

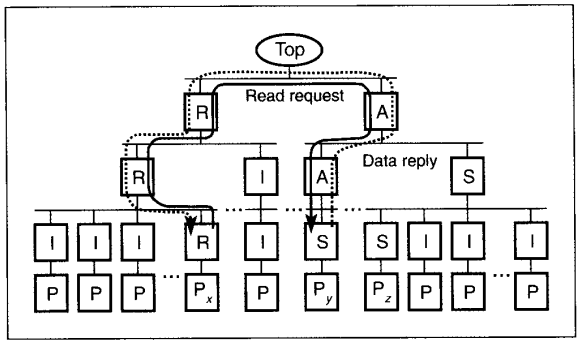

Figure 7. A read request from processor P_x has found its way to a copy of the item in the attraction memory of processor P_y. Its path is marked with states Reading (R) and Answering (A), which will guide the data reply back to P_x. (I indicates processors in the Invalid state, S processors in the Shared state.)

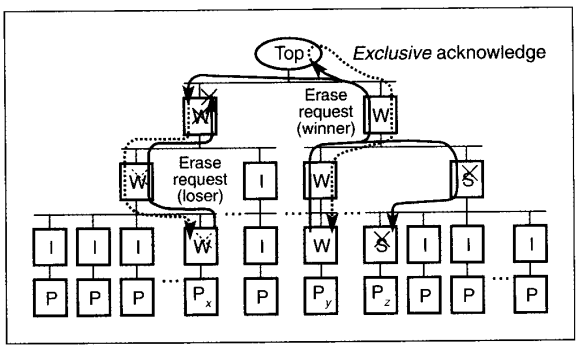

Figure 8. A write race between two processors P_x and P_y is resolved when the request originating from P_y reaches the top bus (the lowest bus common to both processors). The top can now send the acknowledgment, Exclusive, which follows the path marked with W's (processors in the Waiting state) back to the winning processor P_y. The Waiting states are changed to Exclusive by the acknowledgment. The Erase transaction will erase the data in P_x and P_z, forcing P_x to redo its write attempt.

that fails to terminate will reach a directory in Exclusive and turn into an Inject transaction. Replacing an item in Exclusive generates an Inject transaction that tries to find an empty space in a neighboring attraction memory. Inject transactions first try to find an empty space in the attraction memories of the local DDM bus, as in the single-bus DDM. Unlike in a 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 out a foreigner or a Shared item. The item home space is equally divided between the bottommost buses, and therefore space is guaranteed on the home bus.

The preferred location in DDM is different from memory location in NUMAs in that an item seeks a home only at replacement after failing to find space elsewhere. When the item is not in its home place, other items can use its place. The home also differs from the NUMA approach in being a bus: Any attraction memory on that bus will do. The details of the directory protocols are available elsewhere.[4]

Replacement in a directory. Baer and Wang studied the multilevel inclusion property,[7] which has 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. In other words, 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 system. However, the effects of the multilevel inclusion property are limited in DDM. It stores only state information in its directories and does not replicate data in the higher levels. Yet another way to limit the effect is to use "imperfect directories" with smaller sets (lower number of ways) than what is required for multilevel inclusion and to give the directories the ability to perform replacement, that is, to move all copies of an item out of their subsystem. We can keep the probability of replacement at a reasonable level by increasing the associativity moderately higher up in the hierarchy. A higher degree of sharing also helps to keep that probability low. A shared item occupies space in many attraction memories, but only one space in the directories above them. The implementation of directory replacement requires one extra state and two extra transactions.[4]

Other protocols. Our protocol gives the programmer a *sequentially consistent* system. It fulfills the strongest memory access model, but performance is degraded because the processor has to wait for the acknowledgment before it can perform the write. However, the acknowledgment is sent by the topmost node of the subsystem in which all copies of the item reside, instead of by each individual attraction memory with a copy. This not only reduces the remote delay, it also cuts down the number of system transactions. The writer might actually receive the acknowledgment before all copies are erased. Nevertheless, sequential consistency can be guaranteed.[8] The hierarchical structure can also efficiently support looser forms of consistency providing higher performance. We have designed a processor-consistent protocol[8] and a protocol combining processor consistency with an adaptive write update strategy.

Increasing the bandwidth

Although most memory accesses tend to be localized in the machine, the hierarchy's higher levels may nevertheless demand a higher bandwidth than the lower systems, creating a bottleneck. To take the load off the higher levels, we can use a smaller branch factor at the

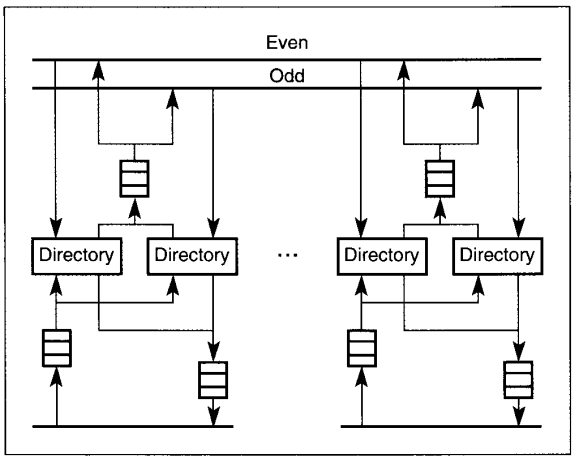

Figure 9. Increasing the bandwidth of a bus by splitting buses.

top of the hierarchy than lower down. This solution, however, increases the levels in the hierarchy, resulting in a longer remote access delay and an increased memory overhead. Instead, we can widen the higher levels of the hierarchy to produce a fat tree.[9] We split a directory into two directories half the original directory's size. The two directories deal with different address domains (even and odd). The communication with other directories is also split, which doubles the bandwidth. We can perform a split any number of times and at any level of the hierarchy. Figure 9 shows that regardless of the number of splits, the architecture is still hierarchical to each specific address.

Yet another solution is a heterogeneous network: We use the hierarchy with its advantages as far as possible and tie several hierarchies 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.

Related activities

At the Swedish Institute of Computer Science, we are developing an operating system for the DDM prototype. This work is based on the Mach operating system from Carnegie Mellon University, which we modified to support DDM efficiently. Related activities involve a hardware prefetching scheme that dynamically prefetches items to the attraction memory; this is especially useful when a process is started or migrated. We are also experimenting with alternative protocols.[1]

A DDM emulator is currently under development at the University of Bristol.[2] The emulator runs on the Meiko transputer platform and models an architecture with a tree-shaped link-based structure, with transputers as directories. The transputers' four links permit 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 emulator's DDM protocol has a different representation suited for a link-based architecture structured like a tree, rather than for a bus-based architecture. The implementation has certain similarities to directory-based systems.

References

1. E. Hagersten, *Towards a Scalable Cache-Only Memory Architecture*, PhD thesis, SICS Dissertation Series 08, Swedish Institute of Computer Science, Kista, Sweden, 1992.

2. S. Raina and D.H.D. Warren, "Traffic Patterns in a Scalable Multiprocessor Through Transputer Emulation," *Proc. Hawaii Int'l Conf. System Sciences*, Vol. I, IEEE-CS Press, Los Alamitos, Calif., Order No. 2420, 1992, pp. 267-276.

The DDM prototype project

A prototype DDM design is near completion at the Swedish Institute of Computer Science. The hardware implementation of the processor and attraction memory is based on the system TP881V by Tadpole Technology, UK. Each such system has up to four Motorola MC88100 20-MHz processors, each one with two MC88200 16-Kbyte caches and memory management units; 8 or 32 Mbytes of DRAM; and interfaces for the SCSI bus, Ethernet, and terminals, all connected by the Motorola Mbus as shown in Figure 10.

We are developing a DDM node controller board to host a single-ported state memory. As Figure 10 shows, it will interface the TP881V node with the first-level DDM bus. The node controller snoops accesses between the processor caches and the memory of the TP881V according to the memory-below protocol, and also snoops the DDM bus according to the memory-above protocol. We have integrated the copy-back protocol of multiple processor caches into the protocol mechanisms. The node controller thus changes the memory's behavior into that of an attraction memory. Read accesses to the attraction memory take eight cycles per cache line, which is one more than in the original TP881V system. Write accesses to the attraction memory take 12 cycles compared with 10 cycles for the original system. A read/write mix of 3/1 to the attraction memory results in an access time to the attraction memory on the average 16 percent slower than that to the original TP881V memory.

As Table 1 shows, a remote read to a node on the same DDM bus takes 55 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 45 extra cycles. Write accesses to shared state take at best 40 cycles for one level and 50 cycles for two levels.

The DDM bus is pipelined in four phases: transaction code, snoop, selection, and data. We designed our initial

bus conservatively, since pushing the bus speed is not a primary goal of this research. The prototype DDM bus operates at 20 MHz, with a 32-bit data bus and a 32-bit address bus. It provides a moderate bandwidth of about 80 Mbytes per second, which is enough for connecting up to eight nodes — that is, 32 processors. Still, the bandwidth has not been the limiting factor in our simulation studies. We can increase bus bandwidth many times by using other structures. The slotted ring bus proposed by Barosso and Dubois[10] has a bandwidth one order of magnitude higher.

For translations to item identifiers, DDM uses the normal procedures for translating virtual addresses to physical addresses, as implemented in standard memory management units. This means that an operating system has knowledge of physical pages.

Any attraction memory node can have a connected disk. Upon a page-in, the node first attracts all the data of an item page as being temporarily locked to its attraction memory. If the items of that page were not present in the machine earlier, they are "born" at this time through the protocol. Then the node copies (by direct memory access) the page from the disk to the attraction memory, unlocking the data at the same time. Page-out reverses the process, copying a dirty page back to the disk. The operating system can purge the items of unused pages for more sharing.

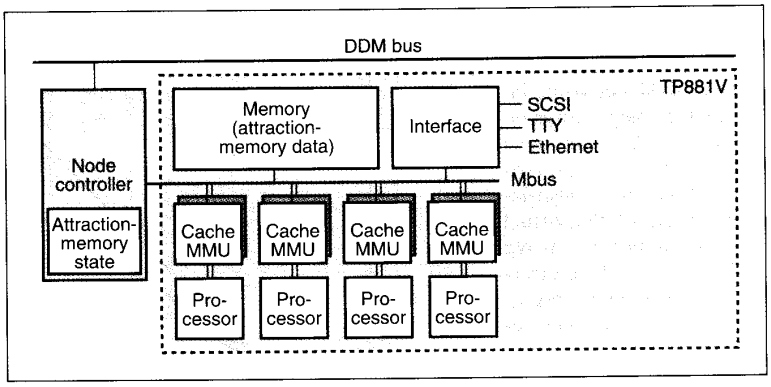

Figure 10. A node of the DDM prototype consisting of four processors sharing one attraction memory.

Memory overhead

It might seem that an implementation of DDM would require far more memory than alternative architectures. Extra memory is 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 in only one copy (worst case). We assume an item size of 16 bytes — the cache line size of the Motorola MC88200.

A 32-processor DDM — that is, 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 we said earlier, the item space is not larger than the sum of the sizes of the attraction memories, so the size of each attraction memory is one eighth of the item space. Because each set in the attraction memory is divided two ways, 16 items can reside in the same set. In addition to the four bits needed to tell items apart, each item needs four bits of state. Thus, an item size of 128 bits gives an overhead of (4+4)/128 = 6 percent.

Table 1. Remote delay in a two-level DDM (best cases).

CPU Access	State in Attraction Memory	Delay, One Level (cycles)	Delay, Two Levels (cycles)
Read	Invalid	55	100
Write	Shared	40	50
Write	Invalid	80	130

Adding another layer with eight eight-way set-associative directories brings the maximum number of processors 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 calculation similar to the one 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 percent. A larger item size would, of course, decrease these overheads.

To minimize the memory overhead, we can use a different interpretation of the implicit state for different parts of the item space. In our initial implementation of DDM, the absence of an entry in a directory is interpreted as Invalid. The replacement algorithm introduces 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 Exclusive for items in its home subsystem, and as Invalid for items from outside. This would drastically reduce a directory's size. The technique would be practical only to a limited extent. Too small directories restrict the number of items moving out of their subsystems and thus limit sharing and migration, resulting in drawbacks similar to those of NUMAs.

Item space is slightly smaller than the sum of the attraction memories because of sharing in the system. This introduces a memory overhead not taken into account in the above calculations. However, in a COMA a "cached" item occupies only one space, while in other shared-memory architectures it requires two spaces: one in the cache and one in the shared memory.

Simulated performance

We used an execution-driven simulation environment that lets us study large programs running on many processors in a reasonable amount of time. We parameterized the DDM simulation

model with data from our ongoing prototype project. The model accurately describes DDM 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. We used two-way 1-Mbyte attraction memories and a protocol similar to the one described here, providing sequential consistency.

For a representation of applications from engineering and symbolic computing, we studied parallel execution of the Stanford Parallel Applications for Shared Memory (Splash),[11] the OR-parallel Prolog system Muse, and a matrix multiplication program. All programs were originally written for UMA architectures (Sequent Symmetry or Encore Multimax computers) and use static or dynamic scheduler algorithms. They adapt well to a COMA without any changes. All programs take on the order of one CPU minute to run sequentially, without any simulations, on a Sun Sparcstation. The speedups reported in Figure 11 and Table 2 are relative to the execution of a single DDM node with one processor, assuming a 100 percent hit rate in the attraction memory.

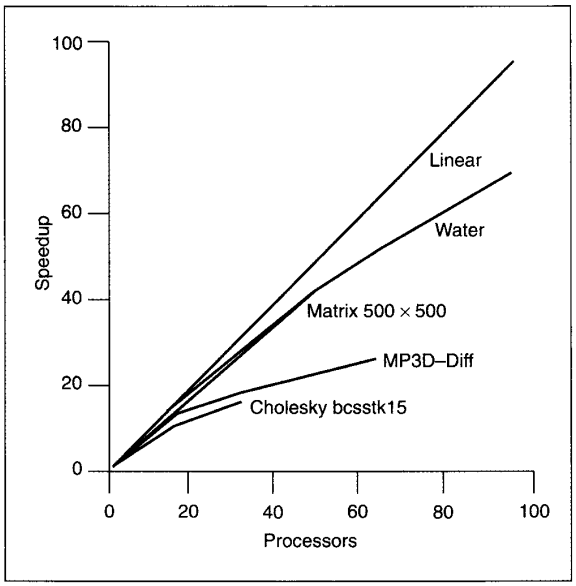

Figure 11. Speedup curves for some of the reported programs.

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. We used 192 molecules and a working set of 320 Kbytes. Each of the 96 processors in Figure 11 handles only two molecules. Most of the locality in the small working set can be exploited on the processor cache, and only about 44 percent of the transactions reaching the attraction memory will hit. A real-size working set would still have the same good locality and would benefit more from the large attraction memories to maintain the speedup. We tested this hypothesis with a single run with 384 molecules, as shown in Table 2.

The Splash-MP3D program is a wind tunnel simulator with which a good speedup is harder to achieve because of a high invalidation frequency resulting in a poor hit rate. The program is often run with the memory filled with data

Table 2. Statistics from DDM simulations. Hit rate statistics are for data only, except with Muse, where we used a unified I + D cache. The remote access rate is the percentage of the data accesses issued by a processor that create remote coherence traffic. An increased working set results in less load on the buses for Water and Cholesky.

	Water		MP3D	MP3D-Diff	Cholesky		Matrix	Muse
Input data	192 molecules	384 molecules	75,000 particles	75,000 particles	m14 (small)	m15 (large)	500×500	Pundit
Cold start included?	yes	yes	no	no	yes	yes	yes	no
DDM topology	2×8×4	2×8×4	2×8×2	2×8×2	2×8×2	2×8×2	8×4	4×4
Hit rate (data) percent								
D cache	99	99	86	92	96	89	92	98.5
Attraction memory	44	65	40	88	6	74	98	91
Remote access rate	0.6	0.4	8.4	1.0	3.8	2.8	0.16	0.20
Bus utilization percent								
Mbus	31	26	86	54	70	60	55	—
Lower DDM bus	39	30	88	24	80	66	—	—
Top DDM bus	25	20	66	13	70	49	4	—
Speedup per number of processors	52/64	53/64	6/32	19/32	10/32	17/32	29/32	—/16

structures representing particles, divided equally among the processors. The three-dimensional space is divided into space cells represented by data structures. MP3D runs in time phases and moves each particle once each phase. Moving a particle involves updating its state and also the state of the space cell where the molecule currently resides. All processors must write to all the space cells, resulting in a poor locality. In fact, 95 percent of the misses we found in DDM were due to this write-invalidate effect. We simulated 75,000 particles, a working set of 4 Mbytes.

MP3D-Diff is a rewritten version of the program that achieves a better hit rate. Particle distribution over processors is based on their current location in space. In other words, all particles in the same space cells are handled by the same processor. Updating of both particle state and space cell state is local to the processor. When a particle moves across a processor border, a new processor handles its data — the particle data diffuse to the new processor's attraction memory. The rewritten program has some 30 extra lines and requires a COMA to run well. In a COMA the particle data that occupy the major part of the physical memory are allowed to move freely among the attraction memories.

Splash-Cholesky factorizes a sparsely positive definite matrix. The matrix is divided into supernodes in a global task queue to be picked up by any worker — the scheduling is dynamic. We 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, which depends on the size of the input matrix. For comparison, Table 2 presents a run with the smaller matrix bcsstk14 (m14) of 420 Kbytes unfactored and 1.4 Mbytes factored.

The matrix multiplication program performs plain matrix multiplication on a 500×500 matrix using a blocked algorithm. The working set is about 3 Mbytes.

Muse is an OR-parallel Prolog system implemented in C at the Swedish Institute of Computer Science. Its input is the large natural language system Pundit from Unisys Paoli Research Center. An active working set of 2 Mbytes is touched during the execution. Muse distributes work dynamically and shows a good locality on a COMA. Because we ran Muse on an earlier version of the

simulator, some of the statistics are not reported in Table 2.

Simulation shows that the COMA principle works well for programs originally written for UMA architectures and that the slow buses of our prototype can accommodate many processors. The overhead of the COMA explored in our hardware prototype is limited to 16 percent in the access time between the processor caches and the attraction memory. Memory overhead is 6 to 16 percent for 32 to 256 processors. ■

Acknowledgments

The Swedish Institute of Computer Science 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 Materiel Administration (FMV). Part of the work on 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 of the University of Bristol, who is a coinventor of DDM. Mikael Löfgren of the Swedish Institute of Computer Science wrote the DDM simulator, basing his work on "Abstract Execution," which was provided to us by James Larus of the University of Wisconsin.

References

1. P. Stenström, "A Survey of Cache Coherence for Multiprocessors," *Computer*, Vol. 23, No. 6, June 1990, pp. 12-24.
2. D. Lenoski et al., "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," *Proc. 17th Ann. Int'l Symp. Computer Architecture*, IEEE-CS Press, Los Alamitos, Calif., Order No. 2047, 1990, pp. 148-159.
3. D.H.D. Warren and S. Haridi, "Data Diffusion Machine—A Scalable Shared Virtual Memory Multiprocessor," *Int'l Conf. Fifth Generation Computer Systems*, ICOT, Ohmsha, Ltd., Tokyo, 1988, pp. 943-952.
4. E. Hagersten, S. Haridi, and D.H.D. Warren, "The Cache-Coherence Protocol of the Data Diffusion Machine," in *Cache and Interconnect Architectures in Multiprocessors*, M. Dubois and S. Thakkar, eds., Kluwer Academic, Norwell, Mass., 1990, pp. 165-188.
5. A. Wilson, "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessor," Tech. Report ETR 86-006, Encore Computer Corp., Marlborough, Mass., 1986.
6. J.R. Goodman and P.J. Woest, "The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor," *Proc. 15th Ann. Int'l Symp. Computer Architecture*, IEEE-CS Press, Los Alamitos, Calif., Order No. 861, 1988, pp. 422-431.
7. J.-L. Baer and W.-H. Wang, "On the Inclusion Properties for Multi-Level Cache Hierarchies," *Proc. 15th Ann. Int'l Symp. Computer Architecture*, IEEE-CS Press, Los Alamitos, Calif., Order No. 861, 1988, pp. 73-80.
8. A. Landin, E. Hagersten, and S. Haridi. "Race-Free Interconnection Networks and Multiprocessor Consistency," *Proc. 18th Ann. Int'l Symp. Computer Architecture*, IEEE-CS Press, Los Alamitos, Calif., Order No. 2146, 1991, pp. 106-115.
9. C.E. Leiserson, "Fat Trees: Universal Networks for Hardware-Efficient Supercomputing," *IEEE Trans. Computers*, Vol. 34, No. 10, Oct. 1985, pp. 892-901.
10. L. Barroso and M. Dubois, "Cache Coherence on a Slotted Ring," *Proc. Int'l Conf. Parallel Processing*, IEEE-CS Press, Los Alamitos, Calif., Order No. 2355, 1991, pp. 230-237.
11. J.S. Singh, W.-D. Weber, and A. Gupta, *Splash: Stanford Parallel Applications for Shared Memory*, Tech. Report, CSL-TR-91-469, Computer Systems Laboratory, Stanford Univ., Stanford, Calif., 1991.

Erik Hagersten has led the Data Diffusion Machine Project at the Swedish Institute of Computer Science since 1988. He is a coinventor of the Data Diffusion Machine. His research interests include computer architectures, parallel processing, and simulation methods. From 1982 to 1988, he worked at the Ericsson Computer Science Lab on new architectures and at Ericsson Telecom on high-performance fault-tolerant processors. From 1984 to 1985, he was a visiting research engineer in the Dataflow Group at MIT.

Hagersten received his MS in electrical engineering in 1982 from the Royal Institute of Technology, Stockholm, where he is currently finishing off his PhD degree.

Anders Landin is a research staff member at the Swedish Institute of Computer Science, where he has been working with the DDM project since 1989. His research interests include computer architecture, parallel processing, memory systems for shared-memory multiprocessors, and VLSI systems and simulation.

Landin received his MS in computer science and engineering from Lund University, Sweden, in 1989. He is a PhD student at the Royal Institute of Technology, Stockholm.

Seif Haridi is leader of the Logic Programming and Parallel Systems Lab at the Swedish Institute of Computer Science. He is also an adjunct professor at the Royal Institute of Technology, Stockholm. His research interests include combining parallel logic programming, concurrent objects, and constraints, and multiprocessor architectures suitable for such programming paradigms. He is a coinventor of DDM. Before joining the Swedish Institute of Computer Science, he was at the IBM T.J. Watson Research Center.

Haridi received his BS from Cairo University and his PhD from the Royal Institute of Technology.

Readers can contact the authors at the Swedish Institute of Computer Science, Box 1263, 164 28 Kista, Sweden; e-mail {hag,landin,seif}@sics.se.