[Hag92] E. Hagersten. *Toward Scalable Cache Only Memory Architectures.* PhD thesis, Royal Institute of Technology, Stockholm/ Swedish Institute of Computer Science, 1992.

[HHW90] E. Hagersten, S. Haridi, and D.H.D. Warren. The Cache-Coherence Protocol of the Data Diffusion Machine. In M. Dubois and S. Thakkar, editors, *Cache and Interconnect Architectures in Multiprocessors.* Kluwer Academic Publisher, Norwell, Mass, 1990.

[HLH91] E. Hagersten, A. Landin, and S. Haridi. Multiprocessor Consistency and Synchronization Through Transient Cache States. In M. Dubois and S. Thakkar, editors, *Scalable Shared-Memory Multiprocessors.* Kluwer Academic Publisher, Norwell, Mass, June 1991.

[HLH92] E. Hagersten, A. Landin, and S. Haridi. DDM – A Cache-Only Memory Architecture. *IEEE Computer*, 25(9):44–54, Sept. 1992.

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

[HS89] M. Hill and A.J. Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[Lar90] J. Larus. Abstract Execution: A Technique for Efficient Tracing Programs. Tech Report, Computer Science Department, University of Wisconsin at Madison, 1990.

[Len91] D. Lenoski. *The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor.* PhD thesis, Stanford University, 1991.

[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. Laudon, 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.

[LRW91] M.S. Lam, E.E. Rothberg, and M.E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the 4th Annual Architectural Support for Programming Languages and Operating Systems*, pages 63–74, 1991.

[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. Singh, W-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. Stanford University, Report, April 1991.

[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.

This article was processed using the LaTeX macro package with LLNCS style

resulted in a poor speedup for the DDM as well as for other architectures. This poor locality was tamed by restructuring its distribution of work from being static to being dynamic. The restructuring also increased MP3D's communication locality. Communication locality was also enhanced for Cholesky by adding hierarchical knowledge to its dynamical scheduler. Finally, the importance of the attraction memory was shown to increase with a larger data set.

The conclusion is that a COMA can successfully execute shared-memory applications written with a different architecture in mind. Further improvements can be obtained by small modification to the applications to better suit the unique properties of a COMA.

8 Acknowledgements

Part of the DDM work has been performed under the ESPRIT project 2741 PEPMA. We thank the many colleagues involved in or associated with the project. SICS is sponsored by Asea Brown Boveri AB, NobelTech Systems AB, Ericsson AB, IBM Svenska AB, Televerket (Swedish Telecom), Försvarets Materielverk FMV (Defense Material Administration), and the Swedish National Board for Industrial and Technical Development (Nutek).

References

[And91] P. Andersson. Performance Evaluation of Different Topologies for the Data Diffusion Machine. Final work for Undergraduate Studies, KTH, November 1991.

[BFKR92] H. Burkhardt, S. Frank, B. Knobe, and J. Rothnie. Overview of the KSR1 Computer System. Technical Report KSR-TR-9202001, Kendall Square Research, Boston, 1992.

[CGM90] D.R. Cheriton, H.A. Goosen, and P. Machanick. Restructuring Parallel Simulation to Improve Cache Behavior in Shared-Memory Multiprocessor: A First Experience. Computer Science Department, Stanford, Internal paper, 1990.

[CKA91] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the 4th Annual Architectural Support for Programming Languages and Operating Systems*, 1991.

[DGH90] H. Davis, S. Goldschmidt, and J. Hennessy. Tango: A Multiprocessor Simulation and Tracing System. Tech. Report No CSL-TR-90-439, Stanford University, 1990.

[Goo83] J. R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 124–131, 1983.

[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 422–431, 1988.

cache-miss rates as the problem set is increased for the studied applications.

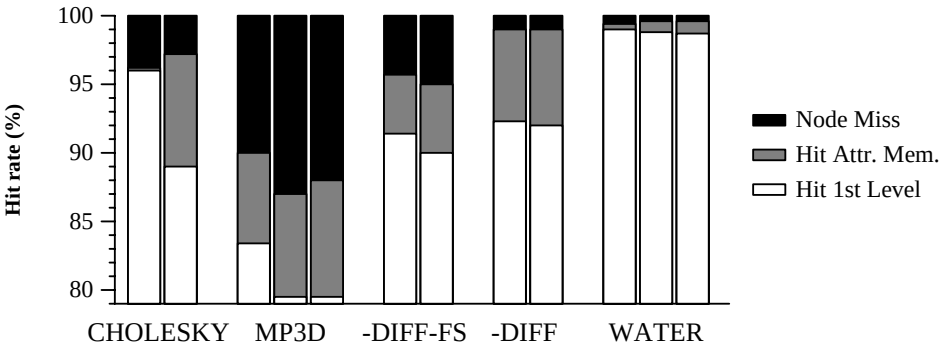


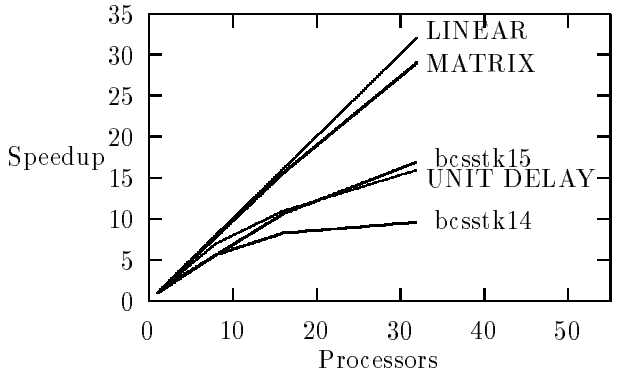
Fig. 8. The effects of the large attraction memories increase with the size of the data set. Here, the behavior of a small data set (to the left) is compared to larger data sets (to the right) for some applications. For the MP3D applications, the number of particles is increased while the number of space cells is held constant.

6 Related Work

The hierarchical DDM and its protocol have several similarities with the architectures proposed by Wilson [Wil86], and Goodman [GW88]. The problem of maintaining coherence in hierarchical systems is addressed in the Wilson. The Goodman proposal has a network built by a grid of buses. A node snoops two buses and has two assignments in that architecture. First it is a processing node and secondly it acts as a directory. The DDM is different in its use of transient states in the protocol, its lack of physically shared memory, and its storage of state information but no data in the network (higher level caches). It is also different in that it handles replacement in such a way that at least one copy of all the data in the caches is always guaranteed, and thus the shared memory is not needed. Recently, the commercial machine KSR1 was released [BFKR92]. It is similar to the DDM being a hierarchical COMA, but has a much larger item size and longer remote access delay than the DDM prototype.

7 Conclusion

A detailed execution-driven simulator of the DDM has been developed to study its behavior when executing real applications. Programs from the SPLASH suite, developed for a UMA architecture, with the largest bearable problem size were used to evaluate the DDM. Good speedup was reported for two of the three studied SPLASH applications, demonstrating a COMA's ability to adapt to static as well as dynamic scheduling. The poor locality of the third application, MP3D,



Application	Cholesky:bcsstk15 (large matrix)			bcsstk14 (small)	bcsstk14-H (hier.sched.)	MATRIX 500x500
Topology	1×1	8×2	2×8×2	2×8×2	2×8×2	8×4
Hit rate Dcaches (%)	87	88	89	96	96	92
Hit rate in AM (%)	100	81	74	6	24	98
Node miss rate (%)	-	2.3	2.8	3.8	3.2	0.16
Busy rate:M bus (%)	27	63	60	70	60	55
Busy rate:DDM bus (%)	-	57	66	80	70	4
Busy rate:Top bus (%)	-	-	49	70	41	-
Speedup/#Processors	1/1	10.6/16	17/32	9.6/32	11/32	29.1/32

Fig. 7. Statistics for matrix programs. The unit delay is for bcsstk14.

The blocking algorithm is yet another example of part of the working set being attracted and worked on locally, resulting in increased speedup and low communication. The algorithm has a block size larger than the data cache, resulting in extensive use of the AM. The work space is about 3 Mbytes. It shows a speedup close to ideal on a DDM (Figure 7), generating extremely little communication. An even more optimal design would be to do the blocking in two levels, with very large blocks kept in the AMs, and smaller blocks read to the data caches.

5.2 What About a Larger Problem Size?

We have used the largest problem sizes that our patience could bear, i.e., a simulation time of about one day per run. Still, the problem size was far from realistic in many cases. We tried to compensate for this by exploring what would happen to the architecture when the data set was increased. The nature of a COMA, with large resources for "second-level caches," makes the DDM less sensitive to the small-cache effect. Actually, it is first when a realistic problem size is used that a COMA really starts to pay off. Figure 8 shows the trends of

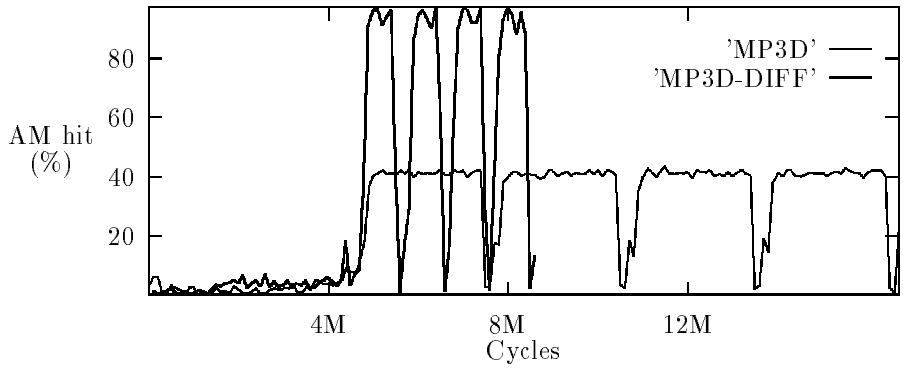


Fig. 6. Dynamic behavior of the hit rate in one attraction memory over time for the original MP3D and the modified version MP3D-DIFF on the topology 2x8x2. The move phases of the last four simulation steps can easily be identified by their higher hit rates. The first step includes the cold-start effect and takes longer time. The execution time of the last four steps represents the steady-state behavior of the simulation. The improvement of MP3D-DIFF by about three times comes partly from an increased hit rate in the processor caches from 86 percent to 92 percent (not shown here).

report good behavior for the small matrix because of a hit rate of 96 percent in the Dcache, a node miss rate of around 4 percent. However, simulating the larger matrix, usually neglected, would have resulted in an 11 percent node miss rate without a second-level cache instead of the achievable 2.8 percent.

Cholesky-H The scheduler part of Cholesky has been modified so that each cluster also has its own task queue, and task migration is hierarchical. Initially, all tasks reside in one global task queue. All processors retrieve jobs from the global queue and put newly created jobs in their local cluster queues. When the global queue is empty, the processors start retrieving tasks from their cluster queues. When the cluster queue is empty, a processor first looks for jobs in its binary brother cluster.[1] Secondly, the two binary cousins are checked for tasks, etc. Not only are tasks kept local to a bus this way, but the probability of retrieving a job related to one the clusters previously worked on is higher. The most notable difference between bcsstk14 and bcsstk14-H in Figure 7 is that the traffic on the top bus has decreased, even though the execution speed is about 10 percent faster. The reported speedup is relative to the execution of the unmodified program on a single processor.

Matrix is a program multiplying two 500-by-500 matrices using a blocking algorithm[LRW91]. The blocking algorithm is interesting, since it tries to make the most effective use of caches. Once a portion of a matrix (a block) has been read to a cache, it is used many times before being replaced with a new block.

[1] Calculated by toggling the least significant bit of the processor ID.

Rewriting the same code for a NUMA involves adding one extra layer of indirection in accesses to the particle data and explicitly copying particle states between the local memories. The move phase now shows an improved speedup. The move phase that accounted for 93 percent of execution time on a uniprocessor now occupies around 50 percent of execution time on 32 processors. Improving speedup above 32 processors means optimizing the other phases, since they now are the dominant part of the execution.

MP3D-DIFF somewhat improves the communication locality of the application. Adjacent space cells are handled by adjacent processors. This improves the locality in the diffusion of particles. As the number of processors increase while the number of space cells is constant, a negative effect on the node miss rate can be expected, since the number of space cells per processor decreases, with increased particle diffusion as a result.

The steady-state execution speed of the modified MP3D-DIFF is about three times that of the original MP3D on 32 processors. The number of remote accesses is decreased to about 10 percent of the original number. An earlier version of MP3D-DIFF had each particle represented by 44 bytes, resulting in a fair amount of false sharing, so that two processors wrote to different parts of the same cache line and therefore appeared to share data, resulting in conflicting writes. The false sharing disappeared when each particle instead was made 48 bytes to better suit our 16-bytes cache line. The effect of false sharing can be studied as MP3D-DIFF-FS in Figure 5, where all the different runs are compared. Figure 6 compares the hit rates in the AM of the MP3D and MP3D-DIFF.

Work on improving the cache behavior for MP3D has also been reported by Cheriton et al. [CGM90]. In that study, machines with small caches were used. Such machines are not practical when applying this method to real-sized problems.

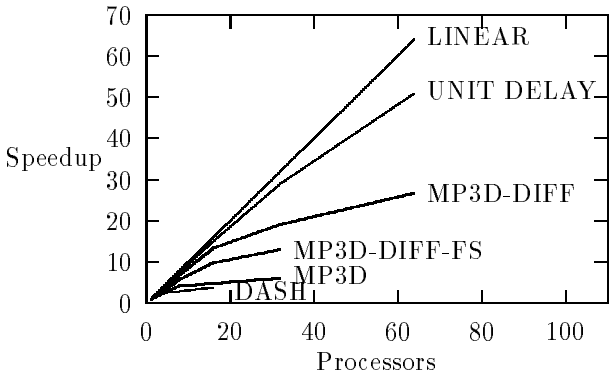
Reported speedups for MP3D-DIFF and MP3D-DIFF-FS are relative to the execution of the original MP3D on a single processor.

Cholesky factorizes a sparse positive definite matrix. The matrix is divided into supernodes that are put in a global task queue to be picked up by any processor. Locks are used for the task queue and for modifications in the matrix. We have used two input matrices as input to the program. The large matrix bcsstk15 occupies 800 kbytes unfactored and 7.7 Mbytes factored. Bcsstk 15 has a speedup of about 17 using 32 processors and seems to have potential for more speedup on larger DDMs (Figure 7). The smaller matrix bcsstk14, which yields a worse speedup, has been reported for the unit delay. Its input matrix occupies 420 kbytes unfactored and 1.4 Mbytes factored. Its speedup on 32 processors is 9.6.

From the numbers in Figure 7 it is interesting to note that the larger matrix not only has a better speedup, but also produces less traffic. It is divided into larger supernodes than the smaller matrix, resulting in more local execution per communication unit.

This application really highlights the danger of drawing general conclusions based on a small data set. Any architecture with small first-level caches would

resulting in poor scalability on the DDM, as well as for other architectures. MP3D is normally run for many simulation steps. To avoid cold-start effects in our tables, we present the steady-state behavior of the last four simulation steps.

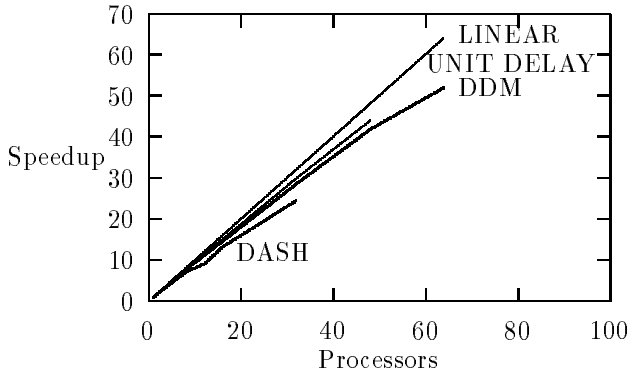


Application	MP3D		–DIFF-FS	–DIFF	
Topology	1×1	2×8×2	2×8×2	2×8×2	4×8×2
Hit rate Dcaches (%)	80	86	90	92	93
Hit rate in AM (%)	100	40	53	88	76
Node miss rate (%)	-	8.4	5.0	1	1.7
Busy rate: M Bus (%)	40	86	76	54	53
Busy rate:DDM bus (%)	-	88	83	24	29
Busy rate:Top bus(%)	-	66	60	13	36
Speedup/#Processors	1/1	6/32	13/32	19/32	27/64

Fig. 5. Speedup for MP3D with 75000 particles at steady state, i.e., the execution time of steps two through five. The unit-delay curve is for 3000 particles.

MP3D-DIFF is a modified version of the program, where a better hit rate is achieved [And91]. The distribution of particles over processors is based here on their current location in space [SWG91]; in other words, all particles in the same space cells are handled by the same processor. The update of both a particle's state and its space-cell state is now local to one processor. When a particle is moved across a processor border, its data is handled by a new processor; i.e., the particle data diffuses to the attraction memory of the new processor. This modification involves adding some 30 extra lines of code.

The move phase of MP3D is now optimized, since most operations are local to the processors. Only the diffusion of particles accross cells generates traffic. Efficiently supporting such a diffusion of the major data structure requires a COMA architecture. In a COMA, the particle data that occupy the major part of the physical memory are allowed to move freely among attraction memories.



Data Set	192 molecules				384 mols.	
Topology	1×1	8×4	2×8×4	64×1	2×8×4	4 × 8 × 4
Hit rate Dcaches (%)	99	99	99	99	98.9	98.9
Hit rate in AM (%)	100	50	44	12	65	58
Node miss rate (%)	-	0.5	0.6	0.9	0.4	0.5
Busy rate:M bus (%)	2	21	31	32	26	37
Busy rate:DDM bus (%)	-	24	39	80	30	40
Busy rate:Top bus(%)	-	-	25	-	20	53
Speedup/#Processors	1/1	28.7/32	52/64	(39.5/64)	53/64	95/128

Fig. 4. The speedup for WATER with 384 molecules running two time steps. The unit delay is reported for 288 molecules and does not include cold-start effects. DASH simulates 512 molecules.

MP3D simulates the pressure and temperature around an object flying at high speed through the upper atmosphere. The primary data objects are particles (air molecules) moving around in a 3-dimensional "wind tunnel," represented by space-cell objects. The simulation is performed in discrete time steps, in which each molecule is moved according to its velocity and possible collision with other molecules, the flying object, and the boundaries. The algorithm is parallelized by statically dividing the particles among processors such that each processor moves the same particles each time.

Moving a particle involves updating the state of the particle and the state of the space cell where the molecule currently resides; in other words, all processors write to all space cells, resulting in poor locality. Between each move phase, some administrative phases are performed, like moving or removing particles from the entrance of the wind tunnel and calculating collision probabilities for each space cell. Simulating 75,000 particles and 14x24x7 space cells results in a total work space of about 4 Mbytes.

MP3D is normally run with the whole memory filled with data objects, mostly particles. The algorithm has poor locality, especially in its "move phase",

in the graphs are self-relative, i.e., compared to the execution time for 1×1. A hit is defined as a read or write that can be completed without stalling the processor. The hit rates for instructions in the processor caches and the AMs are close to 100 percent for all configurations and applications. The numbers reported for the *data cache* (Dcache) and AM hits are for data only. The node miss rate, defined as the ratio of accesses missing in both the Dcache and the AM, is also for data only.

We present our results in graphs where speedup is a function of the number of processors. For comparison, we also show the linear speedup ($Speedup = \#Processors$) and the algorithmic speedups (UNIT DELAY) reported by Singh et al. [SWG91], i.e., the maximum speedup on an ideal architecture.

The architecture modeled in this study differs slightly from the first DDM prototype. The processor caches in the simulator are 16 kbytes, compared to 64 kbytes in the prototype. This partly compensates for the small problem size in the simulation. The attraction memory modeled is two-way set-associative using the last-accessed-memory technique [Hag92], while the prototype implements a true two-way set-associative or direct mapped attraction memory. The associative state memory is modeled as if it was implemented by dual-ported memory rather than interleaved between the two buses. We do not model contention for writes back to the associative state memory.

For comparison, we also show the speedup for the DASH prototype [Len91] for cases where the numbers reported are for comparative problem sizes. Although these numbers are from real—not simulated—prototype hardware the problem size is about the same as for our simulations. The DASH prototype is built from clusters of four 33 MHz MIPS R3000 processors. Each processor has write-through 64 kbytes instruction and data caches and a unified second-level cache of 256 kbytes. The DASH prototype implements release consistency.

5.1 Application Performance

Water is an N-body molecular dynamics application that evaluates forces and potentials in a system of water molecules in the liquid state. It has a static scheduler and uses barriers for synchronization. Water is simulated running two time steps and 192/384 molecules.

The working set is only 320/640 kbytes. The execution time of this application is $O(n^2)$ to the number of molecules, so simulating a real-sized working set is difficult. The small working set results in an extremely good hit rate in the data cache. Misses in the data cache are caused mostly by invalidation misses [HS89], which the AM can do nothing about. The speedup shown for WATER in Figure 4 is almost ideal. Some statistics are presented in Figure 4. Note the difference in the AM hit rate between 64×1 and $2 \times 8 \times 4$. The processors in a cluster share data in their common AM, resulting in an increased hit rate for the four processors. Note, too, the decreased node miss rate when the data set is doubled to 384 molecules. Running this application with real-sized working sets will continue to provide impressive hit rates for large attraction memories.

addresses. On a TLB fault, all necessary transactions from the MMU are sent to the DDM network. New translations from virtual pages to physical pages are created on demand from a randomized free list to make the behavior more realistic. No penalty is added for "reading from disk" since we assume that all pages are already in the machine when the simulation starts. The DDM initially has empty caches and AMs. The first read request for each datum is sent to a special AM, which makes all necessary transactions for returning the value.

The simulation model is instrumented with counters of hardware events, periodically sampled into a large statistics file. The technique has been used to simulate up to 128 processors running programs of up to 2 CPU minutes simulated time. The simulation currently runs the programs 2000 times slower than execution on a single SPARC station. The number of simulated processors has a small effect on the slow down if the application simulated has an ideal speedup, which allows for large machines running large applications to be studied.

5 Simulated Performance of the DDM Prototype

The SPLASH [SWG91] programs represent applications used in an engineering computing environment. They are written in C and use the synchronization primitives provided by the Argonne National Laboratory (ANL) macro package. They were developed for the Encore Multimax, a UMA architecture with small caches tied by a single bus to a single shared memory. The original versions of the programs are used.

Three programs from Stanford Parallel Applications for Shared Memory (SPLASH), MP3D, Water, and Cholesky are reported here. We have identified MP3D as the toughest one for a COMA which makes it interesting to study [Hag92], while Cholesky and Water, appeared midway among the SPLASH applications. They are interesting since they represent two different program behaviors. Water is statically scheduled with barrier synchronization, and Cholesky is dynamically scheduled and uses a task queue as its means of synchronization. MP3D was also studied in two rewritten versions to make better use of the data diffusion ability of a COMA. A modified Cholesky using a hierarchical scheduler was also simulated exploring the hierarchical property of the DDM. Finally, a matrix multiplication program was studied.

The results we present are for DDMs with only two or fewer hierarchical levels and clusters of processors at the leaves, classified by their branch factor from top to bottom $T \times I \times C$, or $T \times C$, where:

T is the branch factor at the top DDM bus,
I is the branch factor at the intermediate DDM bus, and,
C stands for number of processor in one cluster, sharing an M bus.

Many different protocols for the DDM have been designed [HLH91, LHH91]. Here, we use the simplest protocol providing sequential consistency [HHW90].

For configurations 1×1 and 4×1, the DDM network has not been simulated. Instead, a 100 percent hit rate in the AM is assumed. The speedups presented

4 Simulation Technique

Inspired by the Tango simulator at Stanford [DGH90], we have developed an efficient execution-driven simulation method that models the parallel applications as if they were running on a real physical implementation of the architecture.

The parallel applications are developed in, or ported to, C to run on a SUN SPARC station as multiple processes sharing memory under SUN-OS. A modified gcc compiler, Abstract Execution (AE) [Lar90], is used to produce processes that not only execute the programs, but also produce a stream of information when doing so. The level of detail in the information stream is selectable, and for this study has been the full address trace of both instructions and data. AE was originally made to produce trace files from uni-processor execution. In our system the streams of information from the different processes are sent to different inputs of a simulation process, modeling the target architecture, as shown in figure 3. The streams serve as models of the processors. The execution speed of each process is determined by how fast the information in its stream is consumed by the simulated architecture, stalling the application process if necessary, making the relative execution speed between the processes that of their execution on the target architecture. Synchronization between the application processes is performed by ordinary shared-memory primitives, i.e., locks and barriers, in the shared memory.

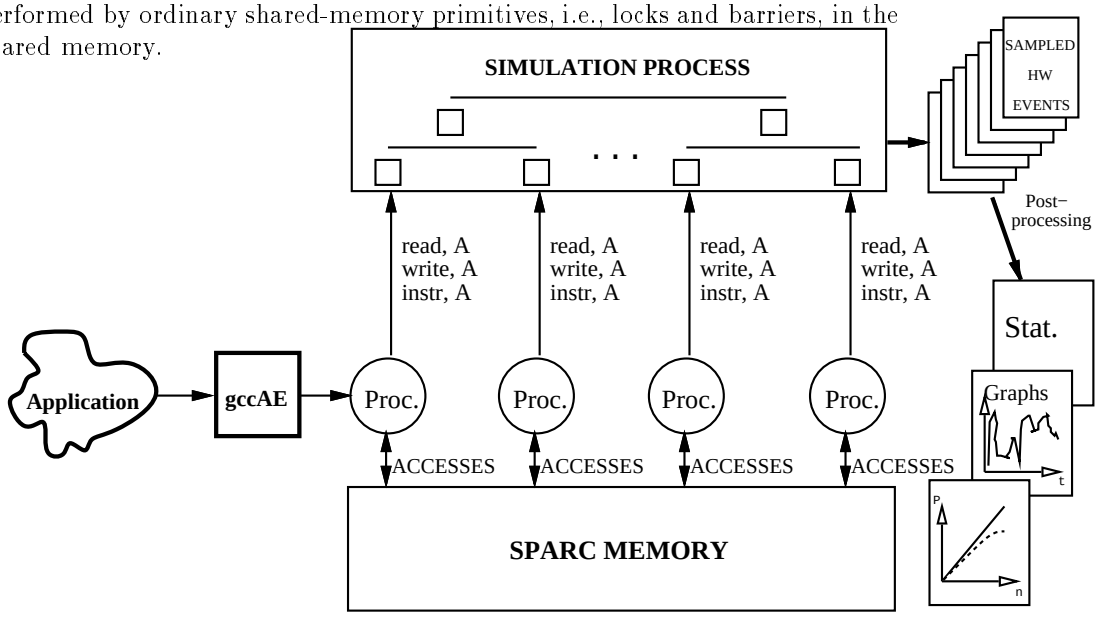


Fig. 3. The structure of execution-driven simulation.

Our simulation model is parameterized with data from our ongoing prototype project, and accurately describes its behavior. Part of the virtual memory system has been modeled, and MMUs make the translation from virtual to physical

lacked the possibility of connecting yet another master board to the M bus. Tadpole Technology, U.K., had a design TP881V that suited us, marked with the dashed line in Figure 2. The TP881V consists of two VME-sized cards: a processor module card with four processors, their eight caches and 32 Mbytes DRAM, and a base module card with SCSI and Ethernet interfaces. The extra functionality required by a DDM node fits on one card designed at SICS. We can fit six DDM nodes with a total of 24 processors and 192 Mbytes attraction memory, power, and directory into a VME rack with 21 slots. The directories are yet not designed, but could be implemented using the described DDM design with small modifications. With an integration higher than the one used in our prototype, eight processor clusters will fit in a box rather than six. Up to a two-level DDM can rely on buses. The second-level split buses connecting eight clusters are 30 cm long. A DDM of three levels must rely on point-to-point connections at its top. Each point-to-point link is about 60 cm long.

3.4 Performance

We decided on a conservative bus design initially, since pushing bus speed is not a primary research goal. The DDM bus in the prototype operates at 20 MHz, with a 32-bit data bus and a 16-bit address bus. It uses drivers developed for the Future bus for all parallel signals, such as address and data. A new transaction starts every fourth cycle, i.e., a transaction frequency of 5 Mtransactions/s. It provides a moderate bandwidth of about 80 Mbytes/s.

A specified latency of a memory system is not necessarily equal to the number of cycles a processor has to stall. The scoreboard mechanism of the MC88100 makes the processor stall only if the register is read before the value of the load has arrived. Similarly, a slow write will not necessarily stall the processor. Here we specify the latency as the number of cycles elapsing between the issue of the load, and when the register can be read.

Read accesses from the CMMU to the attraction memory take seven cycles per cache line, and write accesses to the attraction memory take eleven cycles. To these numbers must be added an extra latency of four cycles for going through the processor caches. The best case latencies at no contention for different accesses can be found in the table below. Latencies for remote accesses are represented by two number: latency in a one level DDM / latency in a two-level DDM if the transaction has to go to the top bus.

Latency in the DDM Prototype [Processor Cycles]

CPU access	Cache hit	Cache miss, AM in state:		
		Exclusive	Shared	Invalid
read	2	11	11	60/115
write	2	15	35/60	70/145

item identifier space. The higher order bits of the item identifiers are stored as address tags in the associative state memory (ASM).

A direct-mapped AM has a specific item always mapped to the very same location, so there is no need to compare tags before we know in which set an item should reside if it is there. We can assume that a transaction will succeed and start a *read line* before approval from the MBP is received. A CMMU that has already read three words can be forced to restart before reading the fourth, and last, word of a cache line. The MBP can therefore wait until the very last cycle before deciding whether to force a retry or not. This allows for state lookup and data transfer to overlap. In most situations, the delay of accessing the ASM will be completely hidden, adding no extra latency to the functionality of the AM, i.e., no wait states are inserted by the MBP.

The MBP compares the address tag bits stored in the ASM to the higher order bits of the item identifier on the M bus and the state stored in the ASM is checked; e.g., a *read* request to a present item in the Shared state is approved, resulting in no actions. If the transaction was not approved, e.g., a *read* request to state Invalid, the MBP:

1. asserts the retry signal, forcing the CMMU to release the M bus,
2. sets the address tag bits in the ASM to the higher bits of the item identifier,
3. changes the item's state to Reading, and,
4. puts a *read* request in the output above buffer.

When the *data* reply eventually comes back, the memory above protocol (MAP):

1. puts the data part of the transaction in the DI,
2. puts a *write line* transaction in the OB containing the item identifier, and,
3. changes the item's state to Shared.

The output below buffer has the highest priority on the M bus and gets the M bus next. It writes the contents of the DI to the item's location in the attraction memory. When the CMMU repeats its request again, it will not be interrupted by the memory below protocol. It can be noted that this method turns the M bus into a split-transaction bus, i.e., it is released between the original request and its completion. A write transaction on the M bus to an item in an inappropriate state is intercepted by a *retry* in a similar way by the memory below protocol, and necessary actions are taken. Therefore, from the viewpoint of the M bus, the rest of the DDM looks like yet another CMMU, only slower and noisier.

3.3 Building on Tadpole TP881V

In an attempt to save development effort and time, we searched for commercially available board systems implementing most of the desired functionality. We evaluated all known board systems based on the 88000 family. Most systems

associative state memory (ASM), containing the state and the address tag for each item in the node, as shown in Figure 2.

The MBP checks each transaction on the M bus for validity. If it is a *read* of an Invalid item, for example, the MBP asserts the retry signal. The retry signal makes the current CMMU bus master stop and release the bus, while the MBP initiates necessary actions for retrieving the requested item. The arbitration between the CMMUs is round-robin, allowing other CMMUs to access the memory while the data is being retrieved. The DDM node also hosts the *output above* FIFO (OA) for transactions bound for the DDM bus. The OA contains the transaction code and the item identifier of the transaction, but no data. The MAP can access the M bus by putting an M bus transaction in the *output below* FIFO (OB). The OB only contains address and transaction code. Transactions on the M bus from the OB have the data FIFOs *data in* (DI) or *data out* (DO) as an implicit source or destination. Data is retrieved from the node's data memory and put in the DO by a *read line* from the OB. Data is written from DI to the node's memory by putting a *write line* from the OB.

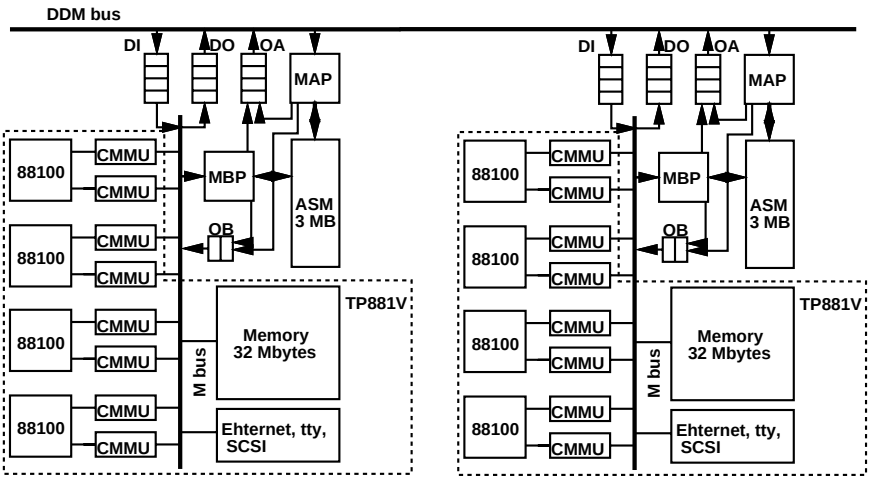


Fig. 2. DDM implementation consisting of two DDM nodes based on the 88000 family.

3.2 Implementing the Attraction Memory and its Protocol

There are several ways an AM and its protocol can be implemented based on the functionality of the retry signal.

Here, we describe a direct-mapped implementation of the AM, i.e., a one-way set-associative implementation. The location of data in the node's memory is determined by looking at the lower bits of its item identifier. The address space of the memory is mapped over and over again sequentially to cover the whole

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. Transient states R and A in the directories mark the request's path through the hierarchy like unrolling a red thread while walking in a maze [HomBC]. When the request finally reaches an attraction memory with a copy of the item, its *data* reply simply follows the red thread back to the requesting node, changing all the states along the path to Shared (S).

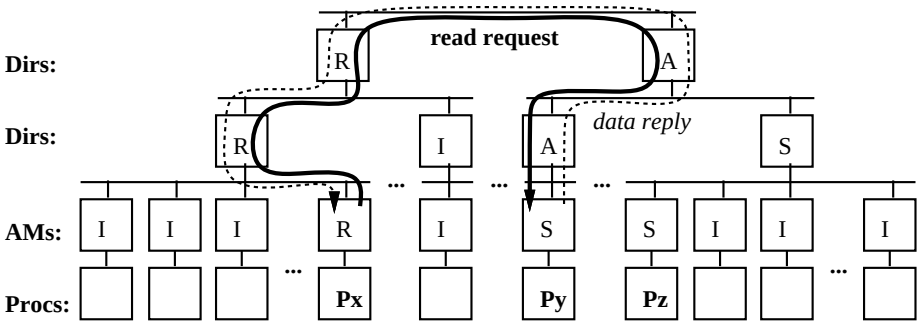


Fig. 1. A read request from processor Px has found its way to a copy of the item in the attraction memory of processor Py. Its path is marked with states Reading and Answering (R and A), which will guide the data reply back to processor Px.

3 DDM Prototype Overview

There are two projects currently working on DDM architectures. A link-based DDM based on Transputers is being developed at the University of Bristol [RW91]. This simulation study models a second, bus-based, DDM currently being built at the Swedish Institute of Computer Science based on Motorola MC88100 processors [HLH92]. The MC88100 has a combined cache and memory management unit (CMMU) chip MC88204 interfaced to a memory bus called M bus. The CMMU use a copy-back cache-coherence protocol similar to the write-once protocol [Goo83]. Each CMMU snoops all transactions on the M bus, and may stop a master by asserting a *retry* signal on the bus. The master immediately stops, backs off the bus, and turns into a slave with the need to arbitrate for the bus again. The CMMU that received the snoop hit is granted the bus and can update the memory during the next cycle.

3.1 Interfacing the DDM Node and the M bus

A DDM node contains the *memory below protocol* (MBP), and the *memory above protocol* (MAP) state machines, implementing the DDM protocol, and an

However, a NUMA 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 better locality.

2.1 The Data Diffusion Machine

The Data Diffusion Machine (DDM) [WH88] is a hierarchical COMA with its directory information distributed in the network. Between each level in the hierarchy sit state memories, called *Directories* (as shown in Figure 1). The directory is a set-associative status memory, which keeps information for all the items in the attraction memories below it, but contains no data. The directories can answer questions such as "Is this item below me?" or "Does this item exist outside my subsystem?" They guide *read requests* to a copy of the data and keep coherence traffic as local as possible. The lowest level bus in the hierarchy connects several DDM nodes. A DDM node contains one attraction memory and one or more processors.

The coherence protocol of the DDM [HHW90] attracts requested data to the attraction memories, controls the coherence among different copies of the same data, and makes sure that the last copy of a data is not lost on replacement.

The hierarchy as described here has a single top bus which easily could become the bottleneck of the system. This bottleneck can be widened by using several top buses, each one responsible for a specific address domain, e.g., even and odd. The topmost directories are also split into different address domains, and thus the transaction frequency is increased [Hag92].

The memory overhead, i.e., the extra memory required to implement the attraction memories and the directories, is surprisingly low. It has been calculated to 5 percent for a 32-processor DDM and 16 percent for a 256-processor DDM [HLH92].

2.2 A Protocol Example: Multilevel Read

Figure 1 shows an example of a multilevel read. Originally, the item studied existed in state shared (S) in the attraction memories of processors Py and Pz. The directories above them also had the item in state shared. The directory common to Py and Pz right underneath the top bus had the item in state exclusive (E), since its subsystem contained all existing copies of the item. All other directories and attraction memories had the item in state invalid (I).

At this point, a read request by processor Px cannot be fulfilled by its local local attraction memory, which puts the requested item in state Reading (R) and transmits the *read* request on the DDM bus. The *read* request cannot be satisfied by the subsystems connected to the bus, and the next higher directory retransmits the *read* request onto the next higher bus. The directory also changes the item's state to Reading (R), marking the outstanding request. Eventually,

2 Background

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, SUN 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 therefore typically have only 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 provides a uniform access time to the whole shared memory, and is therefore 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 to different nodes might vary. The earlier NUMAs did not have coherent caches, and left the problem of coherence to the programmer. Research activities today are striving toward coherent NUMAs with directory-based cache-coherence protocols, e.g. Dash [LLG+90] and Alewife [CKA91]. Programs can be optimized for NUMAs by statically partitioning the work and data. Given a partitioning where the processors make the 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 (COMAs), the memory organization is similar to that of NUMA in that each processor holds a portion of the shared memory space. However, the partitioning of data between the memories is not static, since all distributed memories are organized as 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, is moved around by the protocol and 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 conventional 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 address tag matches the item identifier. Actually there are cases where multiple attraction memories could have matching items.

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 local attraction memories.

Simulating the Data Diffusion Machine

Erik Hagersten, Mats Grindal, Anders Landin, Ashley Saulsbury,
Bengt Werner, and Seif Haridi

Swedish Institute of Computer Science; Box 1263 ; 164 28 KISTA ; SWEDEN.

Abstract. Large-scale multiprocessors suffer from long latencies for remote accesses. Caching is by far the most popular technique for hiding such delays. Caching not only hides the delay, but also decreases the network load. Cache-Only Memory Architectures (COMA), have no physically shared memory. Instead, all the memory resources are invested in caches, enabling in caches of the largest possible size. A datum has no home, and is moved by a protocol between the caches according to its usage. Furthermore, it might exist in multiple caches. Even though no shared memory exists in the traditional sense, the architecture provides a shared memory view to a processor, and hence also to the programmer. The simulation results of large programs running on up to 128 processors indicate that the COMA adapts well to existing shared memory programs. They also show that an application with a poor locality can benefit by adopting the COMA principle of no fixed home for data, resulting in a reduction of execution time by a factor three.

1 Introduction

Simulation is a core technology for research in the computer architecture field. It is important to evaluate architectural ideas using large realistic programs and problem sizes. This study presents a simulation study of one implementation of the Data Diffusion Machine (DDM) [HLH92]. The DDM is a shared memory multiprocessor, but its organization is quite different from other architectures in that its memory system comprises of only caches.

In this study, we wanted to see how well the DDM adapted to existing shared memory applications written with a completely different architecture in mind. We developed an execution-driven simulation environment, which can run parallel programs written in C. The applications used in this study come from the Stanford Parallel Applications for Shared Memory (SPLASH) [SWG91]. A detailed architecture model describes the prototype DDM, currently being implemented at SICS. It allows us to study the behavior of the prototype and to collect statistics one cannot gather in a real implementation. However, the simulation model has a slowdown of approximately 2000 times, which limits the practical problem size of the studied programs. To compensate for this, we varied the data set for the application, and found that the benefits from the DDM architecture increased with the size of the data set.

The simulation of the DDM shows encouraging behavior for the studied programs. In this paper we present our simulation method, the performance of the DDM, and some internal dynamic statistics.