ISRN SICS-R--91/17--SE

A Performance Study of the DDM - a Cache-Only Memory Architecture

by

Erik Hagersten, Pär Andersson, Anders Landin and Seif Haridi

SICS research report
R91:17
ISSN 0283-3638

A Performance Study of the DDM –a Cache-Only Memory Architecture

Erik Hagersten, Pär Andersson, Anders Landin and Seif Haridi
SICS Research Report R91:17 November 1991.*

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, resulting 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. It might exist in multiple caches. Even though no shared memory exists, the architecture still provides the shared memory view to a programmer.

Simulation results from large programs running on 64 processors indicate that the COMA adapts well to existing programs for shared memory. They also show that an application with a poor locality can benefit by adopting to the COMA principle of no home for data, resulting in a reduced execution time of a factor three.

In a COMA, a large majority of the misses are invalidation misses, or share misses caused by write-once/read-many behavior, or a producer-consumer relation, i.e. would benefit from write broadcast. A new protocol is proposed that behaves like a write-invalidate protocol by default for all data. A reader can detect its need for a write-broadcast behavior for a datum, which it enables by sending a subscribe request for the datum to the writer.

1 INTRODUCTION

Large-scale shared-memory multiprocessors suffer from a large remote delay in the network. The delay can be expected to increase with the number of processors in a system. Such behavior has a negative impact on the goal of making multiprocessors that scale well with the number of processors.

Several techniques have been proposed to reduce and/or hide latency due to remote accesses [GHG+91]. Prefetching techniques hide the latency in a network by stimulating an access (e.g. retrieval of a datum) before it is requested by the processor. Multiple context techniques hide the latency by context-switching to a new task whenever a remote access

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

is performed. Looser consistency models allows for global operations by a processor to be pipelined, yet another example of hiding the delay of the network. Caching is probably the most commonly used latency-hiding method, where copies of recently accessed data may be held in the local cache of a processor. It provides a shorter access time if the data is requested (soon) again. In contrast to the other techniques, caching not only hides the latency of the network, but also reduces the network traffic. Taking some of the load off the network also results in less contention and, subsequently, shorter access time.

In this study we focus on exploring the effects of large second-level caches [SL88]. One specialized form of caching, *data diffusion*, is also studied. In data diffusion, a datum has no home and might be moved to live in any or many of the large associative memories local to the processors. This implies that the whole address space, and not just a small portion of it, may be moved according to its usage.

The results presented are achieved by large simulations with a detailed, execution-driven simulator. The architecture simulated is the Data Diffusion Machine prototype, that is currently being implemented at SICS. The simulator has been parameterized by data from the ongoing hardware implementation. We study three unchanged applications from the Stanford SPLASH selection of engineering programs [SWG91]. The programs studied are the MP3D, a "wind tunnel" simulator; Cholesky, a factorization program for sparse matrices; and Water, a simulator of water molecules. We also present numbers from a matrix multiplication application that uses blocking [LRW91]. The MP3D is also run in a modified version, which explores the data diffusion capability of our system.

Our results show a good performance for the studied applications. The studied architecture seems to adapt well to both dynamically and statically scheduled programs, even though they were written with a completely different architecture in mind. It seems to perform especially well for larger working set. An effect of the large caches. Systems with up to 64 processors have been simulated for up to 2 CPU minutes.

The next section gives an overview of the Cache-Only Memory Architectures followed by a section focusing on one instance of COMA, the Data Diffusion Machine. Section 4 presents the simulation technique used followed by the a section reporting the results. Finally we end by suggesting a new mechanism that would remove most of the misses for a COMA running the studied applications.

2 CACHE-ONLY MEMORY ARCHITECTURES

A Cache-Only Memory Architecture, (COMA), is characterized by its lack of any physically shared memory [HLH]. Instead processors host a large (set-) associative memory. The task of such a memory is twofold. It is a large (second-level) cache for the processor, but 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 AM. Even though no shared memory exists, the processors, and subsequently

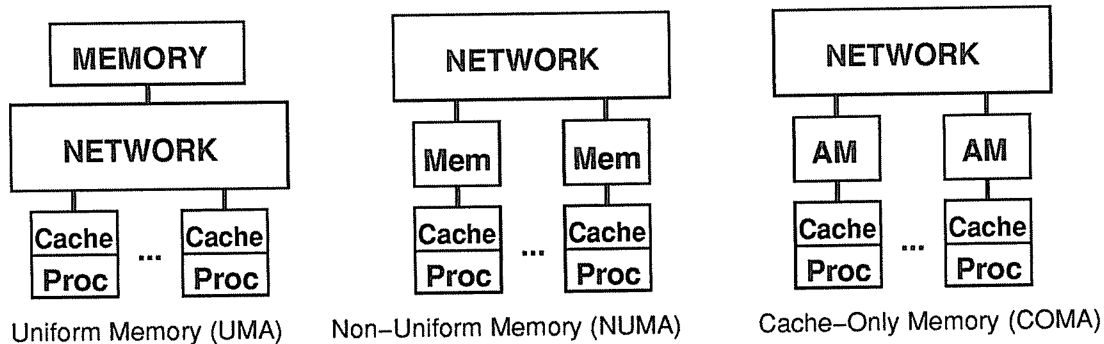


Figure 1: Comparing COMA to more conventional architectures

the programmer, are still provided with a coherent shared-memory view of the system.

The COMA is reminiscent of a non-uniform memory architecture (NUMA), like the DASH [LLG+90] and Alewife [CKA91], in that all the shared memory is physically divided between the processors. In a NUMA, however, different portions of the data are statically allocated to each memory, while in a COMA, a datum has no home and might be moved by the protocol to reside in any or many AMs according to its usage. Figure 1 compares a COMA to other shared memory architectures.

Programs are often optimized for NUMAs by statically partitioning the work and data so that a processor makes most of its accesses to the part of the memory that has been statically allocated to it. Running such an optimized NUMA program on a COMA architecture would result in a NUMA-like behavior, since the work spaces of the different processors would migrate to their attraction memories. However, running the unmodified version would perform as well, 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. A program optimized for a COMA could take this property into account. NUMAs can have caches to allow for additional copies of a datum outside its allocated home memory, but can never change its home location, in contrast to a COMA.

Multiprocessors with multiple caches might contain several copies of the same datum. Inconsistencies might occur if writes are not performed with care. Often, cache coherence is maintained by a cache-coherence protocol implemented in hardware. The strategy used is either write-invalidate, where all other copies but the one updated are destroyed, or, write-update, where the other copies instead are updated. Both approaches have their advantages [EK89].

A COMA coherence protocol can adapt the techniques used in other coherence protocols, but must be extended to search for and retrieve a datum on a read miss. The protocol must also make sure that the last copy of a datum is not lost upon replacement[HHW90].

The address space of a COMA corresponds to the physical addresses of conventional architectures. There is however nothing very physical about them nor are they addresses, since they do not tell where a datum resides. We still guarantee room for every datum by making the

shared address space slightly smaller than the sum of the AMs. How much smaller depends on what degree of sharing should be guaranteed by the system, i.e. how many additional copies of data should be allowed. This can be changed dynamically by the operating system.

3 Data Diffusion Machine, a Hierarchical COMA

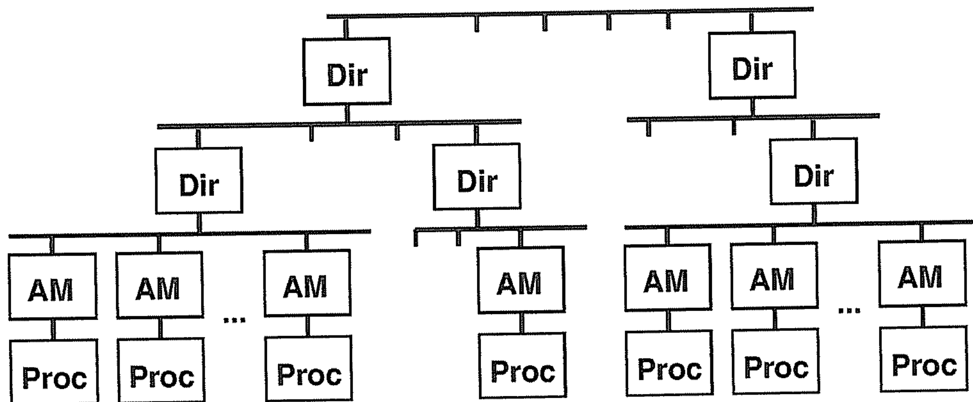


Figure 2: The Data Diffusion Machine with its hierarchical topology.

The Data Diffusion Machine (DDM) [WH88] is a COMA architecture with a hierarchical network topology and set-associative AMs at its tips storing data-value, address-tag and state for all data. Between each level in the hierarchy are set-associative state memories, directories, storing state information for all data in their subsystem, but not their values.

In this performance study we have used a hierarchical write-invalidate cache-coherence protocol which uses the directories to make the coherence traffic as local as possible [HHW90]. The state of a datum in the directory indicates if the datum resides in the subsystem of the directory, and if so, whether other copies might exist outside the subsystem (stable states: *exclusive, shared and invalid*). A directory therefore can judge if a coherence transaction needs to be propagated upwards or downwards and serves as a filter keeping the traffic as local as possible. Finding a datum as locally as possible not only limits the search time, but also minimizes the traffic in the network.

The DDM protocol has a hierarchical search algorithm that uses the state information in the directories to find a datum. A read request climbs up the hierarchy as far as necessary to find a directory marked as having a copy of the datum in its subsystem. On its way down in the subsystem, the request is guided by the directories to one copy of the datum[1]. The search takes at most $2L-1$ bus transactions for L levels in the hierarchy. The read request marks its search path with transient states, used by the reply to find its way to the requesting node. The reply changes the transient states to appropriate stable states. The reply is returned in

[1]A selection mechanism makes sure that only one AM receives the request

at most $2L - 1$ bus transactions. The transient states will also detect and handle combining effects, like combining reads, shown to be useful for large multiprocessors [P+85, G+83].

Write-invalidate is implemented in a general network by the writing node sending out an erase request and waiting for write acknowledges to be received from each individual AM with a copy of the datum. In a hierarchical network the topmost node of the subsystem in which all the copies of the item reside may send the acknowledge. The acknowledge might be received by the writing AM even before all other AMs have received the erase request. Still, sequential consistency is provided [LHH91]. This not only reduces the traffic in the network, but also shortens the waiting time for the acknowledge.

Although most memory accesses tend to be localized in the machine, the higher level in the hierarchy may nevertheless demand a higher bandwidth than the lower systems, which creates a bottleneck. However, the top part of the hierarchy can be widened to become a fat tree or be replaced by a general network, to form a heterogeneous network.

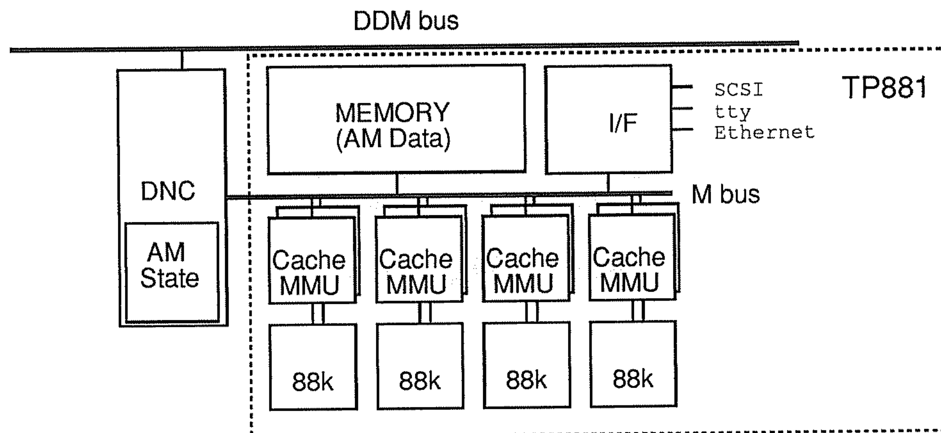


Figure 3: The implementation of a DDM node consisting of up to four processors sharing one attraction memory

A prototype of the DDM is near its completion at SICS, where each AM here hosts a cluster of processors. The hardware implementation of the AM cluster is based on a commercial computer system by Tadpole Technology, UK. Each such system has between one and four Motorola 88100 20 MHz processors, each one with two 88200 processor 16 kbytes caches, 8 or 32 Mbytes DRAM, and interfaces for SCSI-bus, Ethernet, and terminals, as shown in figure 3.

A DDM Node Controller (DNC) board is being developed at SICS that interfaces the processor node to the first level DDM bus. The DNC handles memory accesses between the processor and the main memory of the node, the behavior of which is changed into a set-associative attraction memory. The copy-back protocol of the processor caches has been adapted to the DDM protocol. The processor caches have a cache-line size of 16 bytes. This is also the coherence unit of the attraction memories.

The standard DRAM memory has been turned into a two-way set-associative AM with

a small overhead in the access time. A read-line hit to the AM takes 8 cycles compared to 7 cycles in the original design. A write-line hit is performed in 12 cycles compared to 10 cycles for the original system. A remote read to a node on the same DDM-bus takes in best case 65 cycles. Most of those cycles are spent making Mbus transactions (a total of four accesses). Read accesses climbing one step up and down the hierarchy adds about 20 extra cycles. Calculations of real access times, however, also need to add for extra arbitration time caused by busy buses, and will vary for different network loads. Since the design of high-speed buses is not part of this research, we make a straight forward design of a DDM bus with a 80 Mbytes/s raw bandwidth, and about 40 Mbytes/s effective bandwidth. Other structures might be possible for improving the bandwidth, like a ring-based bus time-slotted into different address domains [BD91].

4 SIMULATION TECHNIQUE

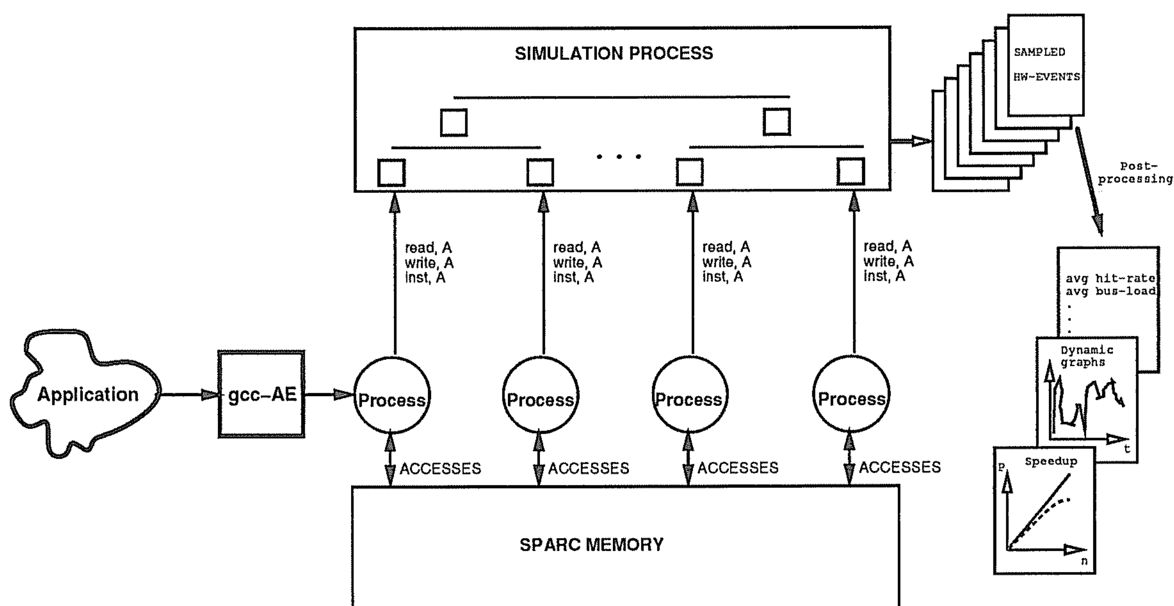


Figure 4: The structure of execution-driven simulation

Inspired by the Tango simulator at Stanford [DGH91], 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 UNIX. 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 has been the full address trace of instructions and data for this study. 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 4. 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.

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 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 64 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. 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 RESULTS

In this study, three programs from Stanford Parallel Applications for Shared Memory [SWG91] (SPLASH), and one matrix multiplication program are used. The SPLASH programs represent applications used in an engineering computing environment. The applications used are written in C and use the synchronization primitives provided by the Argonne National Laboratory (ANL) macro package. The programs are developed for a single-bus architecture, assuming a uniform memory access time. One of them, MP3D, has been studied in two versions, where one has been changed to make better use of the data diffusion ability of a COMA.

Simulation results produced by a small program with a small working set restricts drawing conclusions. This is especially true if the working set is small enough to fit in the first-level caches of the processors. We simulate at most 64 processors, with a total of 1 Mbyte of data caches. Most of the simulated applications have a larger working set. The nature of a COMA, with large resources for "second-level caches," should make the DDM less sensitive to the small-cache effect. Actually, a better hit rate in the data caches results in a poor hit rate in the AMs, and vice versa.

In this study we present numbers for DDMs with only three or fewer hierarchy levels,

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

Many different protocols for the DDM have been designed [HLH91, LHH91]. In this study we use the simplest protocol, supporting sequential consistency [HHW90].

For the configuration 1×1 and 4×1, the DDM network has not been simulated. Instead a 100% hit rate in the AM is assumed. The hit-rates for instructions in the processor caches and the AMs are close to 100% for all configurations and applications. The reported numbers for Dcache and AM hits are for data only. The remote access rates, defined as the ratio of accesses missing in both the Dcache and the AM, are also for data only, The speedups presented in graphs are self-relative compared to the execution time for 1×1 . In the graphs the linear ($Speedup = \#processors$) are shown. For the SPLASH applications, the unit delay speedups from a simulation where all memory accesses take one cycle, reported by Sing et. al. [SWG91], are also shown.

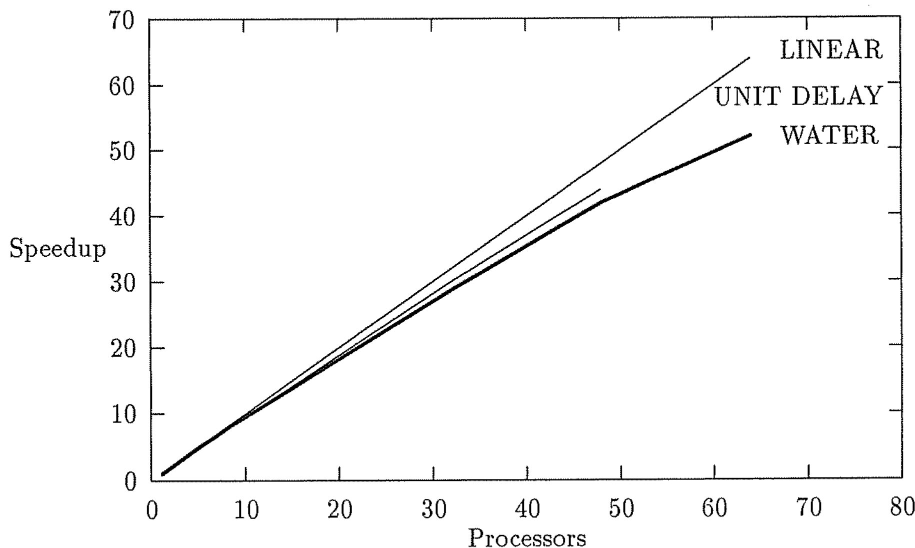


Figure 5: The speedup for WATER with 192 molecules running two time steps. The unit delay is reported for 288 molecules, and does not include cold-start effects.

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. 192 molecules have been simulated for two simulation steps. The working set is only 320 kbytes. The execution time of this application is $O(n^2)$ to the number of molecules, therefore it is difficult to simulate a real-sized working set. Running this program on a SPARC

Topology	1×1	1×4	8×4	$2 \times 8 \times 4$	64×1
Hit rate Dcaches (%)	99	99	99	99	99
Hit rate in AM (%)	100	100	50	44	12
Remote access rate (%)	0	0	0.5	0.6	0.9
Busy rate:Mbus (%)	2	9	21	31	32
Busy rate:DDMbus (%)	-	-	24	39	80
Busy rate:TOPbus(%)	-	-	-	25	-
Speedup/#Processors	1/1	3.95/4	28.7/32	52.1/64	(39.5/64)

Figure 6: Statistics for WATER, 192 molecules.

station takes in the order of 15 CPU seconds. The small working set results in an extremely good hit rate in the data cache. Misses in the data cache are mostly caused by invalidation misses, and are hard for the AM to do anything about. The speedup for WATER shown in figure 5 shows an almost ideal speedup. The deviation from unit delay is mostly due to the small working set (the 64 processors handle only three molecules each), and the fact that we include the cold-start effect for filling the caches and AMs in our numbers. Some statistics is presented in Figure 6. Note the difference in AM hit rate between 64×1 and $2 \times 8 \times 4$. Obviously, the four processors sharing a cluster also share some data.

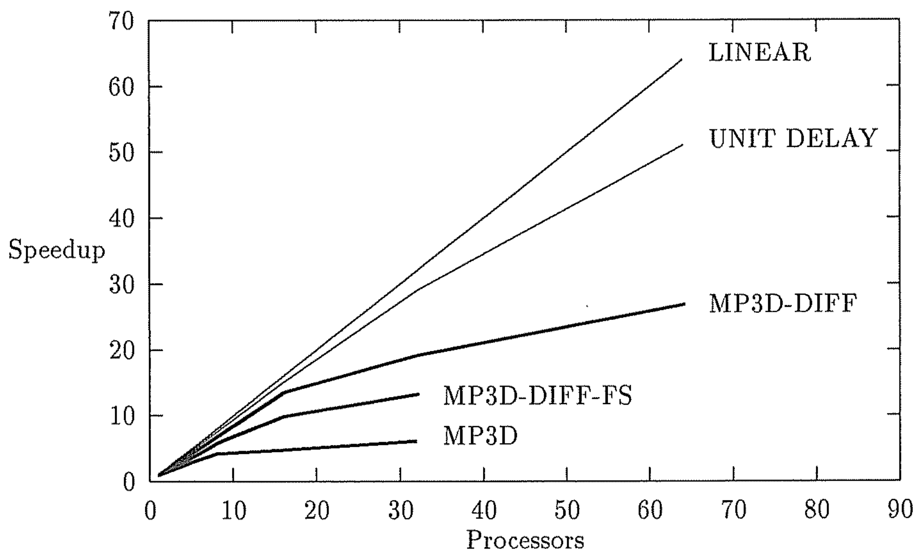


Figure 7: The speedup for MP3D with 75000 particles at steady, i.e., the execution time of steps two through five. The unit delay curve is for 3000 particles.

MP3D simulates the pressure and temperature around an object flying at high speed through the upper atmosphere. The primary data objects are particles (i.e., air molecules)

Application	MP3D-DIFF			MP3D	
Topology	1×1	$2 \times 8 \times 2$	$4 \times 8 \times 2$	1×1	$2 \times 8 \times 2$
Hit rate Dcaches (%)	89	92	93	80	86
Hit rate in AM (%)	100	88	76	100	40
Remote access rate (%)	0	1	1.7	0	8.4
Busy rate:Mbus (%)	33	54	53	40	86
Busy rate:DDMbus (%)	-	24	29	-	88
Busy rate:TOPbus(%)	-	13	36	-	66
Speedup/#Processors	1/1	19/32	27/64	1/1	6/32

Figure 8: Statistics for MP3D-DIFF and MP3D simulating 75.000 particles.

moving around in a 3-dimensional "wind-tunnel," represented by space-cell objects. The simulation is performed in discrete time steps, where each molecule is moved according to its velocity and possible collision with other molecules, the flying object, and the boundaries. This "move-phase" accounts for 93% of the execution time profiled on a single DECstation 3100 [SWG91]. 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. We simulate 75.000 particles and 14x24x7 space cells, resulting in a total work space of about 4 Mbytes. The execution on a SPARCstation takes in the order of 5 CPU seconds.

MP3D is normally run with the whole memory filled with data objects, i.e. mostly particles. The algorithm is parallelized by statically dividing the particles between the processors, so that each processor moves the same particles each time. The algorithm has poor locality, especially in the move phase, resulting in a poor scalability on the DDM, as for other architectures. MP3D is normally run for many simulation steps. To avoid the cold-start effect in our measurements, we present the steady-state behavior of the last four simulation steps.

MP3D-DIFF is a modified version of MP3D with some 30 lines added. The modification is along the lines suggested in [SWG91] by Cheriton et. al. Instead of statically assigning a portion of the particles to each processor, MP3D-DIFF assigns one portion of the space to each processor. During the move phase, a processor handles all the particles that are currently in its space. When a particle moves across a processor boundary, it is handled by another processor and its data structure is attracted (diffused) to its AM. The move phase is now optimized, since most operations are local to the processors. Only the diffusion of particles generates traffic. In order to efficiently support such a diffusion of the major data structure, a COMA architecture is needed. The move phase of the application now shows a good speedup and accounts for around 50% of the execution time on 32 processors. 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% of the original number. In order to obtain a better speedup above 32 processors, the rest of the application should be modified in a similar manner, since it is now a dominant part of the execution. An earlier version

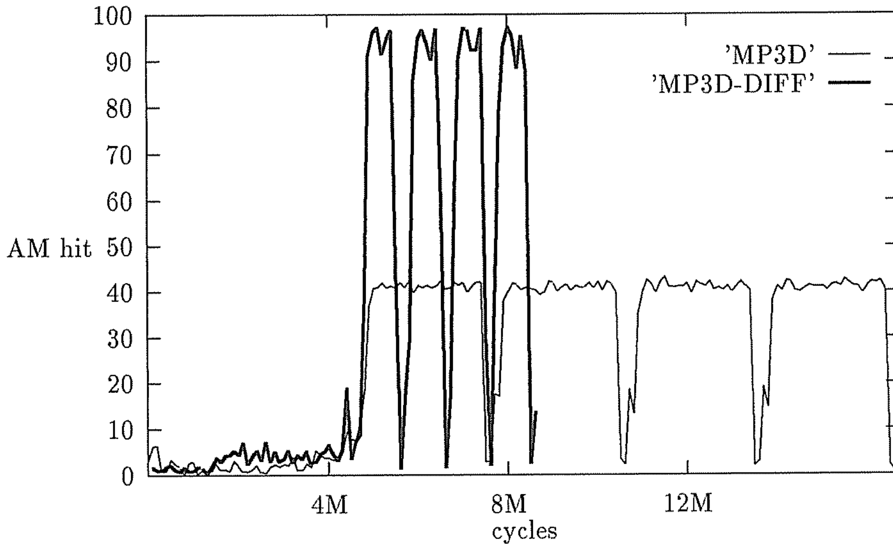
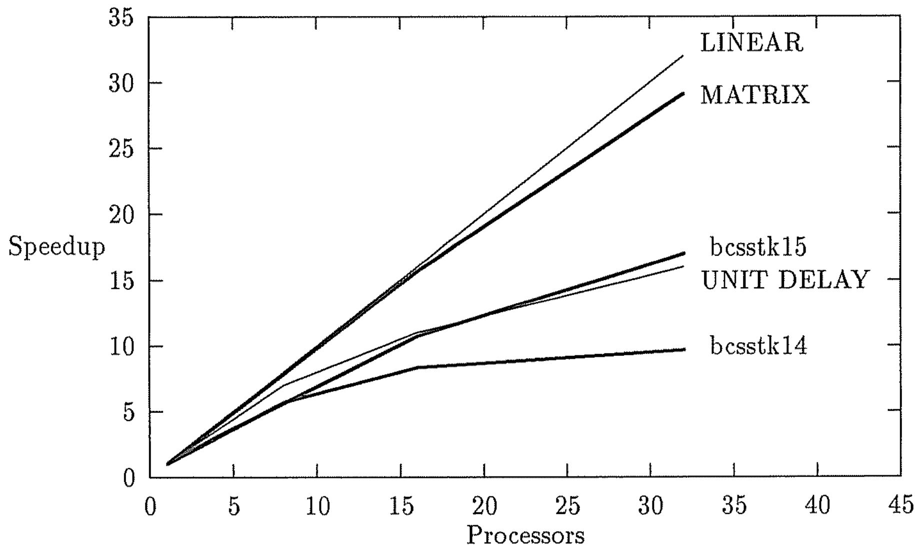


Figure 9: The dynamic behavior of the AM hit rate over time for the original MP3D and the modified version MP3D-DIFF on the topology 2x8x2. The move-phases of the last four simulation steps can easily be identified by their higher hit rate. The first step includes the cold-start effect, and takes a longer 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 also comes from an increased hit rate in the processor caches from 86% to 92 % (not shown here).

of MP3D-DIFF had each particle represented by 44 bytes, resulting in a fair amount of false sharing, i.e., two processors write to different parts of the same cache line and therefore appear to share data, resulting in conflicting writes. The false sharing disappeared when each particle instead was made 48 bytes to better suit our 16 byte cache line. The effect of false sharing can be studied as MP3D-DIFF-FS in Figure 7, where all the different runs are compared. Figure 8 shows the data for MP3D-DIFF and MP3D. Figure 9 compares the hit rates in AM between MP3D and MP3D-DIFF.

Cholesky factorises a sparsely positive definite matrix. The matrix is divided into supernodes that are put in a global task queue to be picked up by any worker. 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, which occupies 800 kbytes unfactored, and 7,7 Mbytes factored, takes about 75 seconds to run on a SPARCstation. Bcsstk 15 has a speedup of about 17 using 32 processors and seems to have potential for more speedup on larger DDMs (Figure 10). 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.



Application	Cholesky: bcsstk15			bcsstk14	MATRIX
Topology	1×1	8×2	$2 \times 8 \times 2$	$2 \times 8 \times 2$	8×4
Hit rate Dcaches (%)	87	88	89	96	92
Hit rate in AM (%)	100	81	74	6	98
Remote access rate (%)	0	2.3	2.8	3.8	0.16
Busy rate:Mbus (%)	27	63	60	70	55
Busy rate:DDMbus (%)	-	57	66	80	4
Busy rate:TOPbus (%)	-	-	49	70	-
Speedup/#Processors	1/1	10.6/16	17/32	9.6/32	29.1/32

Figure 10: Statistics for matrix programs. The unit delay is for bcsstk14.

From the numbers in Table 10 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. Note also the large hit rate in the data cache for the bcsstk14, which is an example of how important it is to simulate real-sized problems.

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

The blocking algorithm is yet another example where a part of the working set can be 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 program takes more than three minutes to run on a SPARC station. The work space is

about 3 Mbytes. It shows a speedup close to ideal on a DDM (figure 10), 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.

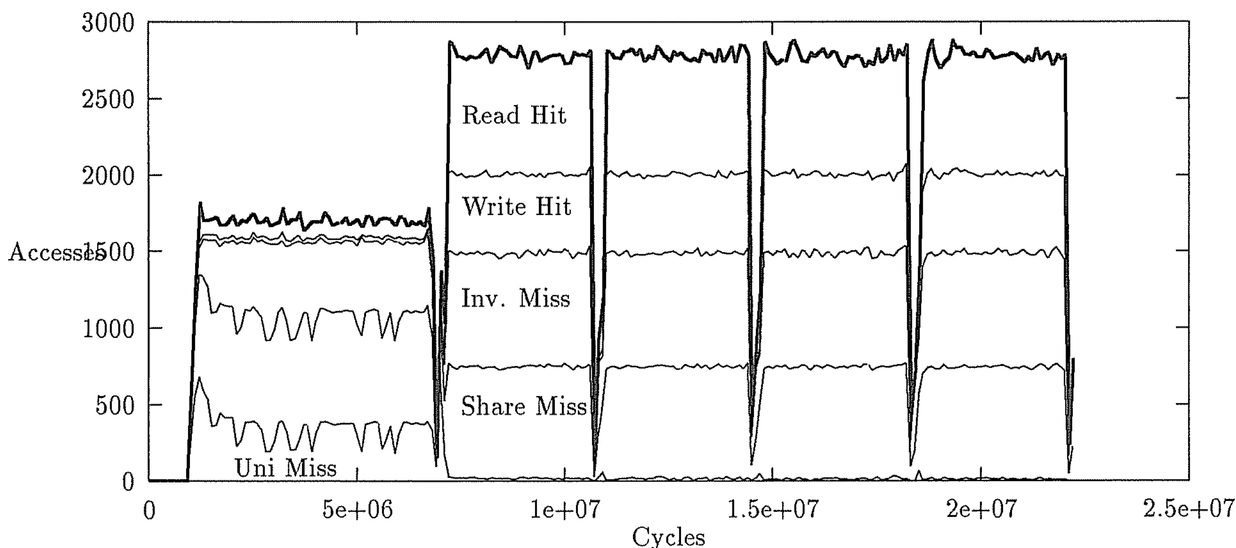


Figure 11: Number of accesses to the AM / 100000 cycles (bold line) for MP3D running on a 8×2 DDM. The accesses are divided into different categories.

6 REMOVING SOME OF THE MISSES

Hill and Smith classify cache-misses into three categories, capacity misses (the cache is not large enough), conflict misses (there is not enough associativity), and compulsory misses (the datum is being touched for the first time) [HS89]. They also note that compulsory misses are the major source of misses for uni processors with large caches. We refer to the sum of these three categories as "uni-misses".

When running statically scheduled programs that run in steps, like WATER and MP3D, compulsory misses disappears after the first step. If the caches are large, the other categories of uni-misses are also rare. Instead, the the major source of misses[2] are *invalidation misses*, i.e. the datum a processor reads used to be in the cache but has been invalidated by a writer, and *share misses*, i.e. the datum a processor writes is shared by others and needs to invalidate other copies before the write is performed. In figure 11 we present the total number of accesses for MP3D divided into different categories. The uni-misses almost disappear after the first step of the simulation.

[2]A miss here means an access by the processor in the DDM that causes the write invalidate protocol to stall the processor in order to maintain sequential consistency

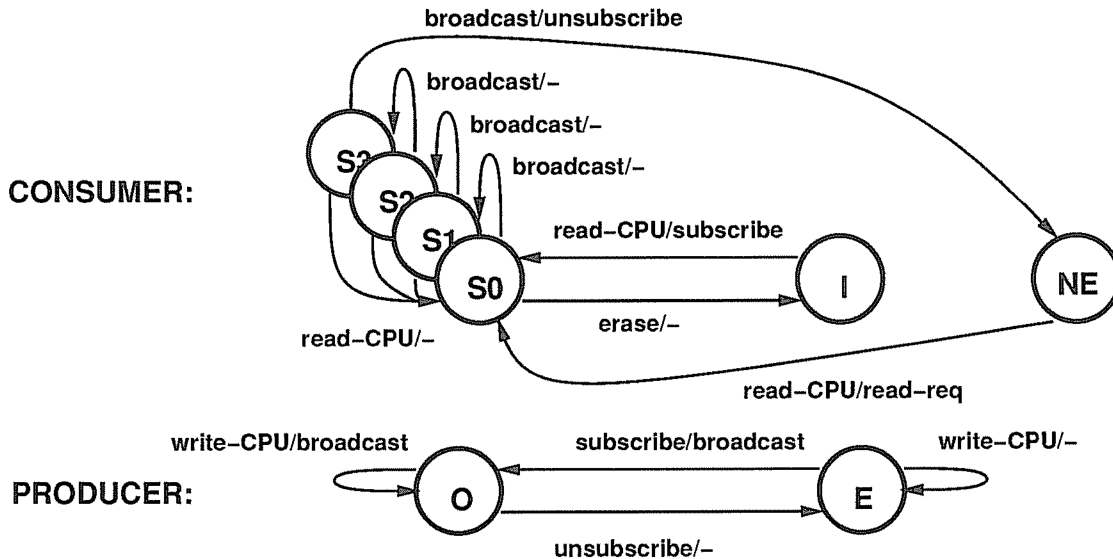


Figure 12: A fraction of the the subscribing protocol with states: NE=non-existent(with the wrong address tag), I=invalidated (with the right address tag), Sx=shared, E=exclusive, O=owner.

Write broadcast protocols have been proposed to eliminate the invalidation misses and share misses. However, write broadcast may also create unnecessary traffic for many applications that shades the positive effects of the better hit rate. This is especially true for large caches. Write broadcast protocols have been studied that turn the broadcasting off if nobody uses the broadcaster values (competitive snooping [KMRS86]). But the communication overhead is still too large [EK89].

Instead we propose a protocol that is a write-invalidate protocol by default. A read miss to a datum in state *invalidated*[3] tells us that the datum has been here but was invalidated by a writer. Since we are reading it again now, chances are high that this datum would benefit from write broadcast in the future. The AM sends out a subscribe request, telling the writer to start write-broadcasting this datum. The writer now becomes the owner of this datum with write permission and pipelines all writes to the requesting node(s). The write broadcast can be turned off using the competitive snooping technique: if the requesting node receives a number of write broadcasts, without reading the value, it invalidates the datum and sends an unsubscribe request to the writer, as shown in Figure 12. Note that reading a read miss in state *non-existent* does not turn on the broadcasting.

We believe that combining the subscribing protocol with pipelined writes for the owner[4]

[3] State invalidated implies that the datum had right address tag and the state bits explicitly in state invalid.

[4] And thereby relaxing the access order model to processor consistency.

would reduce the number of misses significantly without increasing the traffic for statically scheduled simulators.

7 CONCLUSIONS

Cache-only memory architectures have the largest possible caches, which enables the support of data diffusion, i.e. a datum has no home and will be moved to where it is used. We have shown that a COMA performs well for applications written with a completely different architecture in mind. It is also shown how an application with poor locality caused by invalidation misses, can be rewritten using the data diffusion to achieve a reduction in execution time by a factor three. We identify the major source of misses in a COMA to be invalidation misses and share misses. Finally we proposed a new mechanism for write-invalidate protocols, that would turn on write-broadcast on a per-cache-line basis, to avoid the misses.

8 ACKNOWLEDGMENTS

This work is supported in part by ESPRIT project 2471 (PEPMA), and the SICS frame program sponsored by the Swedish National Board for Technical Development (NUTEK), Swedish Telecom, LM Ericsson, ASEA Brown Boveri, IBM Sweden, Bofors Electronics, and, the Swedish Defence Material Administration (FMV). We are grateful to the people at Stanford, who made the SPLASH benchmarks available to us [SWG91], and to James Larus for providing us with the AE tracing tool [Lar90]. Mats Grindal modified the simulator and gathered the statistics for classifying the misses.

References

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

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

[DGH91] H. Davis, S. Goldschmidt, and J.L. Hennessy. Tango: a Multiprocessor Simulation and Tracing System. Tech. Report No CSL-TR-90-439, Stanford University, 1991.

[EK89] S.J. Eggers and R.H. Katz. Evaluating the Performance of Four Snooping Cache Coherency Protocols. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 2–15, 1989.

[G+83] A. Gottlieb et al. THE NYU Ultracomputer – Designing a MIMD Shared-Memory Parallel Machine. *IEEE Transactions on Computers*, pages 175–189, Feb. 1983.

[GHG+91] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991.

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

[HLH] E. Hagersten, A. Landin, and S. Haridi. Moving the Shared Memory Closer to the Processor – DDM. Swedish Institute of Computer Science, Report 1990. To appear in IEEE Computer.

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

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

[KMRS86] A.R. Karlin, M.S. Manasse, L. Rudolph, and D.D. Sleator. Competitive Snoopy Caching. In *Proceedings of the 27th Annual International Symposium on Foundation of Computer Science*, 1986.

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

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

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

[LRW91] M.S. Lamm, E.E. Rothberg, and E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the 4th Annual ASPLOS*, 1991.

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

[SL88] R.T. Short and H.M. Levy. A Simulation Study of Two-Level Caches. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 81–88, 1988.

[SWG91] J.S. Sing, 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–multiprocessor. In *International Conference on Fifth Generation Computer Systems 1988*. ICOT, 1988.