ISBN 91-7170-103-6 ISRN SICS/D--8--SE

Toward Scalable Cache Only Memory Architectures

Erik Hagersten

A Dissertation submitted
for the Degree of Doctor of Technology
Department of Telecommunication and Computer Systems
The Royal Institute of Technology
Stockholm, Sweden

First Edition October 1992
Second Edition July 1993

The Royal Institute of Technology (KTH)
Department of Telecommunication
and Computer Systems
S-100 44 Stockholm, Sweden

SICS
Swedish Institute
of Computer Science
Box 1263
S-164 28 Kista, Sweden

TRITA-TCS-LPS-9213
ISSN 0284-4397

SICS Dissertation Series 08
ISSN 1101-1355

To KARIN,
ELSA,
and
CARL VICTOR

Toward Scalable Cache Only Memory Architectures

Erik Hagersten

Abstract

HIGH PERFORMANCE at a low cost is the common goal of most new computers. Even if the speed of microprocessors seems to double every year, there are, and will always be, important applications demanding even better performance. There are two ways of meeting this demand: a single processor designed with exclusive technology, or several processors in cooperation. The shared-memory model allows the processors to share inputs and results, and is regarded as the most general model for cooperation.

Our approach has been to take today's microprocessors as a starting point and to add the means for successful cooperation via shared memory. There exists, however, no physical shared memory in our proposal. Instead, all memory is divided among the processors and organized in such a way that data can be duplicated, moved freely, and allowed to reside in any memory. This behavior of data is not visible to the programmer, who sees the popular shared memory abstraction. It is, however, beneficial to performance and adapts well to different application behaviors.

We have introduced a new class of architectures based on the above model comprised of caches and processors connected by a network—*Cache-Only Memory Architectures* (COMA)—and an implementation proposal thereof—the *Data Diffusion Machine* (DDM). The large caches are managed by a *cache-coherence protocol* which makes sure the many copies of a datum have the same value. The protocol will also find a datum that is not in the caches of the requesting processor.

COMAs were shown to have superior performance over alternative shared memory architectures in a quantitative analytical performance study. The implementation proposal for the DDM has been simulated with good performance results. Two optimization techniques have also been proposed; *hardware-based prefetching*, and *adaptive write-update protocols*, both of which further improved performance. Several techniques for improving the *transaction frequency*, which is identified as the sparse resource, have also been proposed.

Descriptors

Multiprocessors, shared-memory architectures, cache-only memory architectures, hierarchical architectures, prefetching, write-update protocols, fat trees, split buses, analytical models, cache coherence.

Acknowledgments

Without the support and help of many people, this work would not have been possible. My advisor, Professor Seif Haridi, has supported me and is a constant flow of inspiration. He initiated the DDM architecture together with Professor David H.D. Warren, University of Bristol. Without their initiative and help this work would never have taken off.

I would also like to thank my parents Anna-Britta and Karl-Erik for always encouraging me to continue my studies.

The single most important event that brought me into computer-science research was my year as a visiting research engineer at Professor Arvind's Dataflow group at M.I.T. The many bright and interesting people eagerly striving toward the future in computers made for an exciting and enriching stay. I would especially like to thank Greg Papadopoulos for taking care of the Swede.

The biggest help, however, has been the daily work in the DDM group at SICS. Anders Landin joined the DDM project, doubling its size, in 1989. Anders has been active not only in DDM implementation and consistency-level issues, his main research areas; he is always willing to help improve my work.

In January 1992 the size of the DDM group increased to eight people. Thorbjörn Granlund, Mats Grindal, Peter Magnusson, Ashley Saulsbury, Bengt Werner, and Akio Yamamoto, have not only increased the development pace of the prototype, but also taken some work off my shoulders which enabled the completion of this work. I would like to thank the DDM group as a whole for inspiring my work, for commenting on crude drafts of this dissertation, and for a good time.

Other valuable sources of comments on earlier drafts have been Professor Per Stenström and Fredrik Dahlgren of Lund University.

Three students from the Royal Institute of Technology contributed to this work during their final three months of practical training at SICS: Mikael Löfgren implemented the DDM simulator, Pär Andersson ported the SPLASH programs, and Mats Grindal modeled the prefetcher in the simulator.

Since 1989, some of the DDM work has been part of the ESPRIT project 2741 PEPMA. I thank the many colleagues involved in or associated with the project.

Finally, I thank my wife Karin for all her love, support, and help. She not only accepted my long days (and nights and mornings) of writing, she also helped taking the pidgin out of my English. Thank you, we did this together.

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).

Contents

Preface

The first edition of this book was identical to the dissertation submitted for a Ph.D. in Computer Science at the Royal Institute of Technology, Stockholm, Sweden.

The procedure for getting a PhD in Sweden is first to publish the dissertation, and secondly to defend it publicly, receiving questions and feedback (i.e. criticism) from the external examiner, the evaluation committee, and the audience. The drawback of such a procedure is that you normally do not have a chance to include any of the constructive feedback in your dissertation.[1]

In this second edition, I have the opportunity to actually include some of the comments from the defense as well as other general comments. It also gives me a chance to express my appreciation for having Professor James R. Goodman, University of Wisconsin, as my external evaluator, and for having Professors Michel Dubois, USC, Per Stenström, LTH, and Stefan Arnborg, KTH, on the evaluation committee.

The second edition differs from the first in two major areas. The several different implementation possibilities of attraction memories have finally been explained with the addition of a recent paper "Simple COMA" by Erik Hagersten, Ashley Saulsbury and Anders Landin. Instead of squeezing it into the existing text, the paper is added as Appendix C, with numerous references to it. The second major difference is an extension to Chapter 4, "A Quantitative Performance Study," which compares the COMA and NUMA approaches more clearly. The second edition also corrects typographical errors and adds some minor clarifications.

Kista, Sweden July 1993.

Erik Hagersten

[1] The advantage is that people are tempted to be nice to you, since their advice will not have any impact on the final result anyhow.

Prologue

"What do you do for living?" At a party, you have a few fractions of a second to evaluate the asker and return an appropriate answer. You buy yourself some time by simply replying: "I work with computers." Then you assess the facial expression and categorize the asker as computer enemy, computer user, or computer freak. For the first category, you can end with: "...and we are working on a computer that outdoes all others in terms of size and ugliness. It actually eats small children."

Computer users can be given a more friendly answer: "I am working on a way of making computers both faster and cheaper. Today you can buy microprocessors by the kilo, while large and fast computers are very expensive. Fast computers require exclusive technology and the development cost per machine is tremendous, since they sell in small quantities and soon lose their edge in this fast-moving field. Microprocessors, on the other hand, sell in large enough quantities to be inexpensive, often have special development programs for your assistance, and might even have ready-to-use programs for your customers. If only a couple of hundred microprocessors could work together on the same problem! Then we would have a computer that was both faster and cheaper than today's large and fast computers.

"Building such a multiprocessor is very much like organizing a large company. First you have to establish a way for the microprocessors to communicate with each other. Then you must divide the work among the processors. The distribution must avoid bottlenecks. You should also try to divide the work evenly between the processors, so that they all complete their subtasks at the same time. If the processors can share common knowledge, like input data and partial results, the distribution of work is much easier."

If the computer user at the party still seems somewhat interested after this introduction, he might very well have to listen to the story about the telemarketing company.

"Ten people (processors) are organized in a sales force. The vice president of sales has divided potential customers into ten equally large groups, divided the groups among his salespeople, and asked them to start to sell the product. Common knowledge is that telephone directories covering all of Sweden exist. Even if there are only about eight million people in Sweden, the set of telephone directories is still about two meters wide. The most efficient scheme would be to give each salesperson his or her own complete set of telephone directories. However, the president of the company has decided this is too expensive (would require too much memory). Instead, all salespeople share one common set of telephone directories (the shared memory), creating a long line of eager salespeople at the directories (contention). Ms Saleswhiz soon found her directory search time could be shortened by organizing her customers in a small private book (a cache). Part of each name was used to find the specific page of the book to search for each name (a multiway associative cache).

Only when that name was not found on that page (cache miss) was a lookup in the telephone directory needed. Finally, when the number was found, Ms Saleswhiz updated her private book. If the page was already full, an outdated customer had to be erased (replaced). This was the customer who had not been called for the longest time (least recently used algorithm–LRU).

"Soon, all the salespeople had their own small address books (caches), which shortened the line to the telephone directories, increasing sales, and the vice president hired another thirty telemarketing people. Now the lines to the telephone directories grew long again. The vice president of sales bought larger address books for the salespeople where they kept a larger amount of names. They still kept their small address books where they looked first (first-level caches), looking next in the larger and slower address books (second-level caches).

"Once more the line to the telephone directories was gone, and the vice president hired another 100 salespeople, resulting in new lines to the directories. The vice president proposed buying even larger address books for his people, but the president complained that the size needed this time would be far too expensive. Instead, he divided the telephone directories among salespeople so that each had one part (nonuniform memory architecture–NUMA). Things improved, but the vice president found he could only distribute customers among salespeople in one way–geographically. He could not, as before, group customers by the products they were interested in. It was also difficult for one salesperson to help another who was behind in sales (dynamic scheduling). As a last straw, concentrated sales campaigns in one region created a long line to that directory (hot spot). The company became difficult to manage efficiently and went bankrupt.

"Ms Saleswhiz read this dissertation, however, and started her own company. She equipped all salespeople with huge address books equal in size to one section of the telephone directory combined with the larger address book each salesperson had had before (attraction memory), i.e., the same amount of paper per person. She then made sure all the telephone numbers existed in at least one of the huge address books. She financed the huge address books by selling all the telephone directories (cache-only memory architecture). Now, on the rare occasions a salesperson missed in the huge address book, algorithms described in the dissertation were used to find the telephone number in someone else's huge address book. This led to a more general organization, which was easier to manage and worked more efficiently. Ms Saleswhiz's company is still in business."

After the word "algorithm," the computer user at the party normally looks less interested in computers, and the subject is quickly changed to "what is your favorite editor."

A computer freak at the party has set himself up for a long monologue, a summary of which can be found in the remainder of this dissertation.

1

Introduction

The execution speed of a computer has always been a scarce resource. Even though technology improvements have doubled the speed of a computer almost every second year, computers still seem to be just a bit too slow for the really interesting applications. The idea of building one fast computer out of several slower ones is not at all new. The goal is to design such a multiprocessor with performance comparable to the sum of its processor elements. One of the earlier attempts, the ENIAC project, dates back to the 1940s. Numerous multiprocessors have been built since then, of which only a fraction will be mentioned in this work. After all these years and all the effort put into multiprocessors, the long-awaited success is still not available. There are three negative attributes most multiprocessors have in common: they are hard to program, they do not achieve the expected performance, and they take a long time to develop. The tremendous speed improvements of uniprocessors are obstacles to multiprocessors. Both the software and hardware of multiprocessors are complicated to develop, and they tend to have a much longer development time than for uniprocessors. When a multiprocessor is ready for the market, uniprocessors have taken yet another giant performance step, and can offer performance comparable to the multiprocessor's. This work cannot present the final solution to these problems, but, hopefully, it introduces solutions that bring us one step closer to useful multiprocessors.

This dissertation describes a parallel computer designed according to a new paradigm—a cache-only memory architecture. Along with the description of the core ideas of this architecture, two new general strategies for performance improvement are presented. Neither the architecture nor the performance-improving strategies are visible to the programmer/compiler. Nor are special features expected from the processor.

A computer designed along these lines is expected to provide superior performance in most applications. Yet it assumes only the functionality found in today's microprocessor and compiler designs, allowing for rapid development based on known technology and compatibility with existing programs, compilers, and operating systems.

1.1 Scope

The main component of this work is the introduction of a new class of architectures, one possible implementation, and finally, suggested improvements for such an architecture. Although some of the solutions described are derived from related work, the introduction of a new class of architecture has created many new challenges. The search space for such work is enormous. The lack of previous architectural work of this kind at the Swedish Institute of Computer Science (SICS) and the need to create infrastructure like simulation and evaluation tools further increased the search space.

Nevertheless, we felt it was necessary to create a complete solution, including prototype implementation, in order to fully understand all the issues involved in building this new class of architectures. To achieve this in a finite time, we limited our scope to the critical areas.

The techniques presented are, or at least could be, included in the ongoing prototype project implementing a first version the architecture. Aspects not critical to initial implementation are presented in limited detail. All critical aspects are covered to the point of a detailed performance simulation.

1.1.1 Programming Model

Many programming models have been proposed for multiprocessors. Two common ones are message-passing and shared-memory models, which differ in the memory model provided to the programmer.

Both models are general and not tied to any specific programming language. Other models, like the first Dataflow architectures and reduction machines, integrate language primitives and synchronization with the memory system and are considered

less general. We were interested in general-purpose[1] multiprocessors and adopted the most common paradigm for such architectures: the shared-memory paradigm. Adapting to the mainstream made it easier to port existing applications and operating systems, resulting in a shorter development time—one of the critical factors for multiprocessors. The coherent shared-memory paradigm is often regarded as the most general programming model.

The use of existing programs from the public-domain reservoir eliminated much work. Although the programs used were originally written for multiprocessors with uniform memory access and small caches, they adapted well to this new class of architecture.

Modifications to application programs were tested to fully explore the advantages of the architecture, but programming issues are minor here. Rather, a choice was made to spare the programmer/compiler from bothering with underlying architectural properties. Programming a parallel computer is hard enough as is. Any advances in compiler technology are welcome but not prerequisite for the architecture to perform well.

1.1.2 Processors

Designing processors from scratch would be another major addition to search space and development time. The use of commercial processors in the design, however, incorporates the tremendous advances in microprocessor development into the multiprocessor with reasonable design effort. Designing a tailor-made processor for this class of architectures is tempting, of course, but would not have yielded maximum performance. The work presented here is based on existing commercial microprocessors. Architectural modifications to the processor caches that would be beneficial to this architecture are nonetheless proposed.

1.1.3 Memory System

The memory system has been the core activity of this work. Shared-memory systems were originally designed as such, i.e., several computers physically sharing a single memory. As the speed gap between processors and memories widened, the single memory became the bottleneck of the system. This bottleneck was removed by the introduction of local caches between the processors and the single physical memory. The single memory was still a scarce resource, leading to the next evolutionary step, architectures with local caches and with memory physically distributed between the processors. This work represents yet another step in that evolution by introducing a

[1] General purpose should here be interpreted as suiting a wide variety of the traditional languages and common problems.

system with only caches, yet with a shared-memory view provided to the processors. The system is endowed with the largest caches possible by using all the memory in the system for implementing them. The large caches not only hide network latency, but also reduce network traffic.

Despite using the largest possible caches, only some of the network latencies are hidden. Latency effects are reduced to an acceptable level for many applications by employing large caches. Other applications continue to suffer long delays: latency occurs the first time a datum is touched, and some misses caused by the write-invalidate protocol cannot be avoided. Two other latency-hiding techniques, which hide most of the remaining latencies, are described and evaluated. Both techniques are hardware supported and completely transparent to the programmer (or compiler).

A fast simulator is essential to a study of this kind. The simulator can be used to indicate whether the different ideas work properly and forces a complete description of the architecture. It also provides an inside view of what is really going on in the architecture, which is helpful for understanding the effects of the ideas and gathering useful statistics. Debugging both protocols and some architectural ideas have also been important issues, since some of the ideas presented here were later used in prototype development.

1.2 DDM Background

The proposed architecture, the Data Diffusion Machine, was initiated by Seif Haridi and David H.D. Warren [WH88]. The essentials of their proposal included a hierarchical architecture similar to the one described here, a hierarchical coherence protocol, and a strategy for increasing bandwidth.

The original proposal did not contain any solutions to the replacement and deadlock problems, nor was the protocol integrated with the bandwidth-increasing technique.

1.3 Contributions

In summary, this work includes novel research activities in these areas:

- Identifying and describing a new class of architectures: cache-only memory architectures
- A large quantitative analytic study of ten different architectures
- An operational COMA proposal including replacement issues and memory management

- An implementation proposal thereof
- A performance characterization of the proposed implementation by simulation
- An adaptive write-update protocol adjusting to the behavior of the applications
- A proposal for bandwidth-improving techniques in hierarchies while preserving the order among transactions
- A dynamic hardware-based prefetching scheme
- A simple and efficient simulation technique allowing for detailed studies of architectural ideas

1.4 Thesis Road Map

Each chapter is self-contained to some extent. To avoid redundancy, references to other chapters are often included rather than text. Each chapter starts with an abstract and ends with a short summary. There are three distinct parts of three chapters each.

The first part is a general discussion of important issues for efficient multiprocessors. Its first chapter is about scalability issues, the second an introduction to cache-only memory architectures and their properties, and the third a quantitative performance study comparing cache-only memory architectures to alternatives.

The second part describes the prototype architecture currently being built at SICS. The first chapter introduces architectural ideas, the second describes the prototype implementation, and the third presents simulated performance results.

The third part describes general techniques for improving the performance of any parallel computer: the hardware prefetcher (ROT), the adaptive write-update protocol (subscribe), and several ways of improving bandwidth in a hierarchical network while preserving the order among transactions. These techniques are especially interesting in the context of this work, since their improvements come from removing the kind of misses remaining in a cache-only memory architecture.

We conclude by a summary part presenting some related work, followed by a short summary of the dissertation, and end by a short conclusion.

Part I

JUSTIFYING COMA

2

Toward Scalability

SCALABILITY is one of the more misused terms in computer science. Many architectures are claimed to be scalable whereas most definitions proffered by the research community state an unreachable property. Were scalability regarded as a "yes or no" property, the answer would be "no" for all architectures. The research community has been asked to abandon or define scalability [Hil90]. We prefer to see scalability as an abstract goal rather than defining it to fit an acceptable parallel architecture.

This chapter discusses different ways of identifying a good architecture. We identify parallel efficiency *as a reasonable demand to put on a parallel architecture.*

We also identify some techniques for achieving parallel efficiency. These techniques are described briefly in this chapter and some of them are dealt with in more depth in later chapters.

2.1 What is Scalability?

Scalability issues are mostly of interest to academia. Multiprocessors can have memory, cost, bandwidth, and performance scalability. Memory scalability refers to a behavior of the shared memory. If a single processor of the architecture can make use of the whole shared memory, and thus avoid unnecessary disk accesses, the memory could be described as scalable. This is of great importance not only for the single user case, but also if processors have differing memory requirements during execution.

The cost of building an architecture should be linear to the number of processors. In practice, overhead per processor should be kept low and reasonably constant. The multiprocessor must be cheaper than a uniprocessor with comparable performance in order to be attractive on the market—one of the main ideas of parallel processing.

Scalable bandwidth is yet another property that is difficult to achieve. In order to avoid contention with a large number of processors, available bandwidth must scale with the number of processors. Infinite scalability is impossible in this respect, but enough bandwidth can be provided for a fixed number of processors.

In the context of multiprocessors scalable normally refers to a measure of increased performance when more processors are added, i.e., *scalable speedup*. An intuitive interpretation of scalable is *linear speedup* for up to an infinite number of processors. Speedup measures how much faster a program executes on n processors than the same program executing on one processor. Linear speedup implies a speedup equal to n. The definition of speedup is often coupled to a specific application.

$$speedup_{architecture}(n) = \frac{execution\,time_{architecture}(1)}{execution\,time_{architecture}(n)}$$

$$scalability: \; speedup(n) = n$$

In this discussion we will assume a constant set of programs for which the formulas apply. Defining speedup also requires a definition of problem size, which can either be held constant or increased according to the number of processors; this will be discussed later. No architecture has a linear speedup for an infinite number of processors. Linear speedup is hard to achieve even for a small number of processors. One begins to wonder if the term scalable should have such a pragmatic interpretation. The term is often applied to architectures that perform relatively well when executing an application on a large number of processors.

Scalability has become a property to strive for, but is never met. Striving for the unreachable could be depressing—and misleading. Given these problems, there must be a better way to judge the performance of parallel architectures.

2.1.1 Efficiency

A simple definition of efficiency is:

$$efficiency(n) = \frac{speedup(n)}{n} = \frac{execution\,time(1)}{n * execution\,time(n)},$$

or, how close to linear speedup did we get? Ideally, an efficiency of 1 should be the goal but, as pointed out by Hill [Hil90] and others, comparing with the unreachable does not make much sense.

Not all parallel programs have enough parallelism to utilize a large number of processors [Amd67]. Furthermore, the load balancing might be far from perfect, leaving processors idle at some barrier or waiting to achieve locks. A third source of limited speedup are deficiencies in the operating system, e.g., accesses to common resources might be serialized. The definition of efficiency of an architecture should not suffer from deficiencies outside the architecture.

Deficiencies orthogonal to the architecture can be compensated for by defining architectural efficiency in relation to some ideal architecture running the same application:

$$efficiency_{architecture}(n) = \frac{execution\,time_{theoretical\,model}(n)}{execution\,time_{architecture}(n)}.$$

A PRAM [FW78] could be such a theoretical model. The PRAM models represent a range of memory models with different access restrictions. In their base model, all memory accesses take one unit of delay. A more practical way to estimate a theoretical minimum execution time for an architecture is to run the application in a simulator where all memory references take one cycle. An efficient parallel unit-delay simulator for parallel architecture (UDS) can be made detailed enough to execute an application on top of a real operating system, running only 30 times slower than on one target processor element [Mag93]. It is therefore practical to make such estimates for large programs running on hundreds of processor elements, given that enough memory is available. Estimates for thousands of processors would require a parallel implementation of the UDS. If the same processor architecture, compiler, and operating system are used in the UDS as in the target parallel architecture, a practical lower bound for the execution time can be calculated.

Having identified the minimum execution time of an ideal architecture, given the application, compiler, and operating system, a definition of efficiency for the *architecture* can be produced:

$$efficiency_{architecture}(n) = \frac{execution\,time_{UDS}(n)}{execution\,time_{architecture}(n)}.$$

Such a definition of architectural efficiency contains some of the deficiency found in uniprocessors caused by their relatively slow accesses to primary memory. The architectural efficiency of a uniprocessor, i.e., processor utilization, is often in the range of 0.7–0.8. Architectural efficiency can therefore never be greater than the processor utilization of its nodes. It would be cruel to ask the designer of a parallel architecture for a higher processor utilization than that found in uniprocessors. A measure of how efficient the parallel part of the architecture is has to compensate for this deficiency. We will make use of *algorithmic speedup*, like the ones presented for the SPLASH suite [SWG91] and defined as:

$$speedup_{algorithmic}(n) = \frac{execution\,time_{UDS}(1)}{execution\,time_{UDS}(n)}.$$

This allows us to define parallel efficiency, a measure of how far from perfect our parallel strategy is, as:

$$\begin{aligned} parallel\ efficiency(n) &= \frac{speedup_{architecture}(n)}{speedup_{algorithmic}(n)} = \qquad (2.1)\\ &= \frac{execution\,time_{UDS}(n) * execution\,time_{architecture}(1)}{execution\,time_{UDS}(1) * execution\,time_{architecture}(n)} = \\ &= \frac{efficiency_{architecture}(n)}{efficiency_{architecture}(1).} \end{aligned}$$

Parallel efficiency is similar to, but slightly more practical than, a definition of scalability by Nussbaum and Agarwal [NA91], which is related to the EREW PRAM model.

A good parallel design should strive toward a parallel efficiency of 1. Parallel efficiency has the advantage of compensating for the latencies inside the node, also found in a uniprocessor. As can be seen in Equation 2.1, the architectural efficiency of n processors is divided by the architectural efficiency of one processor. Parallel efficiency also compensates for the effects of the instruction-set architecture and compiler used in the UDS. Thus, any UDS will do for an estimate, and a UDS for the target architecture is not necessary. Even better is if the algorithmic speedup for the application is already available, as is the case with the SPLASH suite [SWG91].

There are pitfalls, however; not only should the achieved parallel efficiency of an architecture be in focus, but also the performance of a single node[1] of the architecture. High parallel efficiency is more easily obtained for designs with slow processors or with a long latency to local memory, since the relative cost of a remote access is related to the performance of a single node.

[1] A unit containing a processor and memory.

If the penalty for remote accesses is also paid when a single processor is executing, then this would violate the concept of parallel efficiency—a measurement of the overhead for executing on a parallel architecture rather than a uniprocessor. Examples of such architectures are those with one common shared memory and those with distributed memory such that the data set of the execution does not fit in the local memory of a single node. Yet another example of parallel efficiency making little sense is when problem size is held constant and fairly small. When the number of processors increases, the size of the data set per processor decreases, with a positive effect in the caches of the architecture as a result. A similar positive effect can also be observed if the local memories are too small for hosting the data set. As the data set per processor decreases, all of a processor's data might fit in its local memory. Since this positive effect is not explored in the UDS, the architecture under study gets an advantage. Actually, a speedup larger than n can be achieved—*superlinear speedup*.

What can be regarded as satisfactory efficiency also depends on the application. As we will see later, satisfactory efficiency is more easily obtained in some applications than in others. The efficiency of an architecture can be divided into two parts:

$$\mathit{efficiency}_{architecture}(n) \quad = \quad \mathit{efficiency}_{latency}(n) * \mathit{efficiency}_{contention}(n) \qquad (2.2)$$

The latency component comes from all the latencies in the architecture—logic latency, communication latency, and memory latency—assuming no contention in the system. This efficiency is estimated with simpler simulation models and simple analytical models. The second component comes from the isolated contention delay, i.e., the time spent waiting for occupied resources, such as buses and memories. This part is harder to model unless a very detailed simulation is used. It heavily depends on the dynamic behavior of the machine, including the correct modeling of hot spots in time and space.

If the effects of the latency and contention are isolated and measured as suggested above, a third efficiency component might be necessary, adjusting for the changed behavior in the load balancing of the machine caused by side effects when moving from simulated delays to real delays. This component is hard to predict. It is debatable whether it should be included in the efficiency of the architecture, sorted under algorithmic deficiencies, or called pure luck. Applications for which this effect is large are not well suited for evaluating architectures.

The communication portion of $\mathit{efficiency}_{latency}$ increases with the number of processors, while memory latency and logic latency are more or less constant. In large systems, the communication part is the dominant source of latency. Mapping an architecture in 3-D space implies that the communication distance between processors increases by $\sqrt[3]{n}$. There is also the limitation of fan-in, fan-out introducing a logarithmic factor. Therefore, an efficiency larger than $O([n^{\frac{1}{3}} \log n]^{-1})$ should not

be expected for large systems. The physical limitations found in different topologies have been studied extensively by Nussbaum and Agarwal [NA91].

The $efficiency_{contention}$ will decrease if not enough bandwidth is provided by the network. A saturated network will limit achievable efficiency. However, unlike latency, there is no point in pushing available bandwidth too far; i.e., changing the network load from 10 percent down to 5 percent by doubling the available bandwidth in the network does not have the same effect as cutting its latency in half.

2.2 Latency Hiding and Reducing Techniques

Making a remote reference for each access to global data is costly in a large multiprocessor. To achieve reasonable efficiency, latency-hiding techniques are used to "hide" network latency during execution. Some latency-hiding techniques involve a smarter way of writing the program or smarter compilers that generate code to hide latency during execution. Other techniques are transparent to the program and result in more efficient execution without putting demands on the compiler or the programmer. We call these two classes of latency-hiding techniques software- and hardware-oriented solutions. They can coexist in a system, and each has its advantages. This work focuses on hardware-oriented solutions.

Several techniques have been proposed to reduce and/or hide latency caused by remote accesses [GHG+91].

- Caching is by far the most popular technique. A cache hit not only reduces latency, it also reduces the load on the network, since no remote access is performed. Introducing caches to a multiprocessor should be combined with a cache-coherence protocol to keep the many copies of a datum coherent, without bothering the programmer.
- Prefetching techniques hide latency in a network by stimulating an access (e.g., retrieval of a datum) before it is needed by the processor. Prefetching can be controlled by hardware or software.
- Multiple-context techniques hide latency by context-switching to a new task whenever a remote access is performed. For algorithms with good parallelism and algorithmic speedup, this method can be very efficient.
- Weaker consistency models allow for global operations by a processor to be pipelined—yet another example of hiding the delay of the network.
- Write-update protocols maintain coherence by remotely updating all shared copies on a write. For some applications, this is an efficient way of hiding latency.

2.3 Effects of Caches

Examples of how caches improve performance can be found in all classes of computers. Although caching has been in commercial use since 1968 (IBM 360/85), different strategies for its implementation are still being researched and evaluated.

A cache is a relatively small and fast memory containing a subset of the data available elsewhere in slower storage. The contents of the cache can either be controlled by software, e.g., the contents of the primary memory of a workstation, or by hardware, e.g., the small cache in a microprocessor. The replacement algorithms normally favor storing the most recently accessed data in cache.

A hardware-controlled cache is normally organized as set-associative memory, where one part of the accessed address is used to determine in which set the accessed datum might reside. This set may contain many alternative locations, or ways, where the datum can be stored. Each location has an address tag containing the remaining part of the address. Each location tag in the set is compared with the accessed address in order to determine if the accessed datum is in the cache—a cache hit.

A cache provides a shorter access time if the data is soon requested again. This is referred to as *temporal locality*. The data unit brought into the cache, *cache line*, is often larger than the word size of the processor. This is beneficial if other words close to the accessed one will be accessed soon, called *spatial locality*.

As technology moves on and more powerful processors with small and fast on-chip caches become available, larger second-level caches become useful to narrow the speed/size gap between the small on-chip caches and the slow main memory. The size and access time of a second-level cache lies in between those of the first-level cache and the slower memory.

Caching is probably the most commonly used latency-reducing method in a multiprocessor, where copies of recently accessed data may be held in the local cache of a processor. Caching not only hides the latency of the network, but also reduces network traffic. Taking load off the network also results in less contention and, subsequently, cuts access time to remote memory.

Caches in a multiprocessor allow several copies of the same datum to occur simultaneously—the datum is replicated. Thus, several processors sharing the datum gain a short access time, but having several copies of a datum also introduces a coherence problem, i.e., there might be several copies of the datum, but there should only be one value. Yet another, but related, problem is the consistency level, or memory access ordering model, stating what assumptions a programmer can make about the ordering of memory events. Both consistency and coherence are often dealt with by a special cache-coherence protocol integrated with the cache implementation.

In this work we focus on exploring the effects of very large second-level caches. All the memory resources of our architecture, the Data Diffusion Machine (DDM) [WH88, HLH92], are invested in the implementation of large caches local to processors. Therefore, it has no main memory. The sum of the large caches instead form the common shared memory. This results in caches of the largest possible size, but also introduces two new problems not solved in traditional cache-coherence protocols: on a read-miss, the datum must be searched for in some other cache and on replacement, care must be taken so that the last copy of a datum is not lost.

Part II is devoted to the structure and performance of the DDM. We introduce a new class of architectures based on the above idea in Chapter 3—Cache-Only Memory Architectures (COMA). That chapter also argues why COMA can be regarded as a more general class of architectures than the alternatives. Chapter 4 is a quantitative performance comparison of different COMA implementations and other architectures showing a clear advantage for COMA.

2.3.1 Effects of Cache Misses in Multiprocessors

Even if the largest possible caches are used, misses still remain. In some of the applications studied, as many as 99.5 percent of data references are satisfied by the large cache, while for others only 90 percent can be completed locally in the cache, and 10 percent miss. Is this node hit rate high enough to make a large multiprocessor competitive?

Here we assume a 100 percent hit rate for instructions.[2] Processor elements perform one instruction per cycle. The processors have an access time of one cycle to their caches, and average $Latency_{rem}$ to remote memory. $Penalty_{latency}$ is the extra time spent doing remote accesses normalized to the ideal execution time without any penalty. It can be calculated by multiplying the cache miss fraction by the cache miss latency and dividing by the average number of cycles per data references. The $efficiency_{latency}$ can be obtained by simply adding one and inverting.

$$
\begin{aligned}
penalty_{latency} &= \frac{node\ miss\ rate * Latency_{rem}}{instructions\ per\ data\ reference} \\
efficiency_{latency} &= \frac{1}{penalty_{latency} + 1}
\end{aligned}
$$

The studied classes of architectures and applications have approximately four instructions per data reference and a $Latency_{rem}$ in the range of 100 cycles. Figure 2.1 shows the simplified relationship between efficiency and miss rate for different

[2] This is a realistic estimation for the programs studied here (SPLASH et al.).

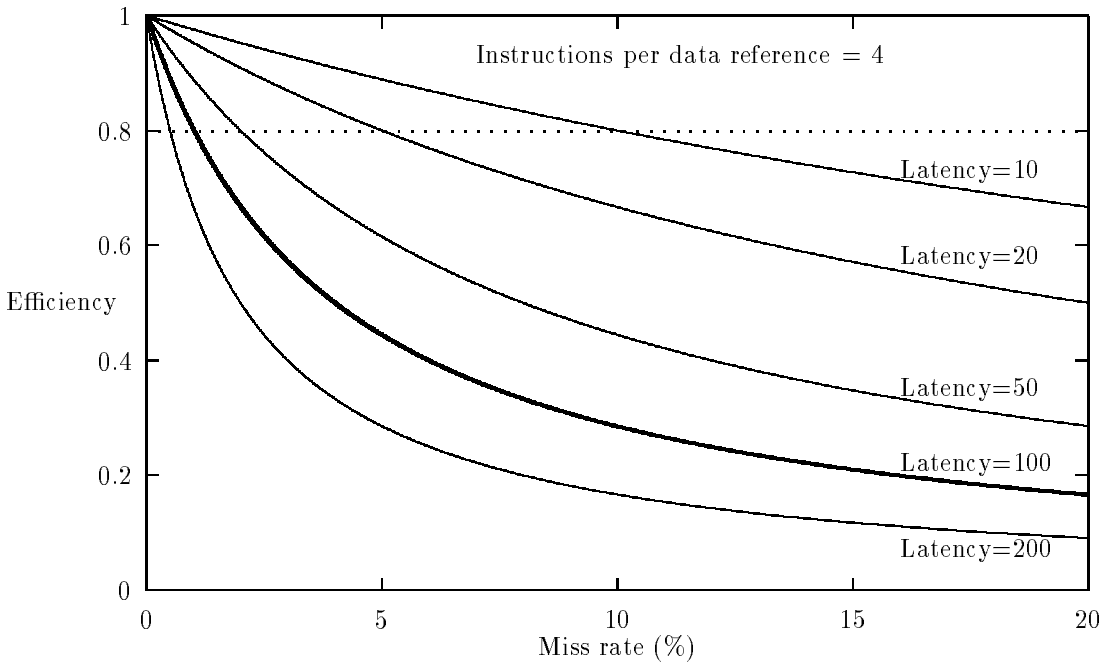


Figure 2.1: The relationship between efficiency and miss rate shows the importance of low miss rates for systems with long latencies.

remote-access latencies. It shows that a latency of 100 cycles and a miss rate of 10 percent result in an efficiency in the range of 0.3, while a miss rate of 0.5 percent brings the efficiency up to almost 0.9.

Without having stated any absolute number for what efficiency is acceptable, one could believe that 0.9 is allright, while 0.3 is too low. As a comparison, a miss rate of 10 percent on a uniprocessor with a 10-cycle delay to main memory results in an efficiency of 0.8 according to this model. A goal could be to achieve comparable performance for the parallel architecture and for the uniprocessor. An efficiency of 0.8 at a latency of 100 cycles corresponds to a 1 percent miss rate. As a rule of thumb, the effect of a 1 percent miss rate can be accepted in a multiprocessor. So, large caches alone do not make an attractive architecture for all applications. This is justified later by more detailed simulations. The effect of the misses must therefore be subdued.

2.4 Removing Remaining Misses

We define a *cache miss* as being an access to a cache that makes the processor stall. The cause for stalling the processor could be either a data read miss, a data write miss, or an instruction miss.

- A *read miss* is caused by a read access to a cache line not present in the cache.
- A *write miss* could be caused by a write access to a cache line not present in the cache, or a write access to a present cache line in a state that delays the write in order to achieve some *consistency level*.
- An *instruction miss* means that the processor tries to execute an instruction that is not currently in the (instruction) cache.

In the calculation above, we assumed that all misses result in the same remote delay. For real systems this is generally not the case.

To further increase performance, the reasons for the misses must be understood and avoided. Hill and Smith [HS89] classify the cache misses of uniprocessors into three categories.

- *capacity misses* are caused by a limited cache size (the cache is too small),
- *conflict misses* by too many active blocks mapping to a fraction of the sets (not enough associativity), and
- *compulsory misses* by the cache line being touched for the first time.

The first two categories of misses can be avoided by increasing the size of the caches, and the second category can be decreased by more associativity (more ways). The compulsory misses cannot be decreased without any prefetching and are the dominant cause of misses for uniprocessors with large caches [HS89].

In multiprocessors cache coherence is often maintained by a cache-coherence protocol of the write-invalidate type. In other words, on a write to a shared datum, all other old copies are destroyed. Cache misses in multiprocessors with write-invalidate protocols have another two categories:

- *invalidation misses*, caused by the invalidation of shared data performed by another processor, and
- *shared misses,* which delay writes since there are other copies of the datum in the system.[3]

[3]This assumes a sequentially consistent protocol.

These two types of misses are often referred to as *coherence misses* and are caused by two or more processors sharing a read/write datum. Because of large cache lines, adjacent data, not directly shared, will also be invalidated/shared, resulting in *false sharing*. False sharing increases with cache-line size [EK89]. We have measured invalidation misses to be as high as 4 percent of data accesses. In the same application, share misses also amounted to 4 percent, not surprisingly, since one can imagine situations where the two misses are paired. False sharing has been measured at 4 percent of all data accesses. Large caches will cut down on the number of capacity and conflict misses, but will not affect the other types of misses, which must be reduced by other methods.

2.4.1 Prefetching

Misses can be removed by issuing a prerequest for the data prior to the point in time where they are actually needed. If the prerequested data arrive in cache before they are asked for, the miss rate in the cache will decrease. If the address of a future data access is known, or can be calculated, a special prefetch instruction can be inserted into the code. The prefetch instruction initiates the fetching of data, but does not block the processor. Prefetching can either bring the data to a register or to the cache. Bringing the data to a register results in the highest performance, but requires far more registers than prefetching to cache. A prefetched datum residing in a register will also be excluded from cache-coherence actions. In this work, we only cover techniques that prefetch data to cache.

Most data misses can be removed by issuing extra prefetch instructions for all data accesses. This is clearly not practical, since it would produce a large overhead in extra instructions executed and hence required bandwidth. Instead, the programmer or compiler can try to determine which accesses will miss the cache, or at least, which accesses have the potential of missing. The prefetch distance, i.e., how far ahead of use data should be prefetched, can be determined by program analysis. Properly used, this technique is very beneficial for removing misses [KL91, GHG+91]. However, if unnecessary prefetches are generated, the system might suffer from the extra contention generated. Sometimes, extra address calculation is needed for prefetching, adding unnecessary work.

Programming a parallel processor is difficult. The programmer should not be required to insert special instructions in order to achieve reasonable speed. Nor can today's compiler technology be trusted to always perform this task. Instead, we propose a hardware-based prefetch technique, hidden from the compiler/programmer.

A simple algorithm implemented in hardware detects access patterns and predicts what accesses will be made in the future. The algorithm will also detect how far ahead of use data must be retrieved in order to hide network latency. We call this

technique Right-On-Time Prefetching (ROT). The operations of ROT are transparent to the programmer.

Prefetching can also remove share misses, if the type of access, like read or write, is also monitored, and the prefetcher prepares for a forthcoming write by issuing a dummy write. The dummy write brings the cache line into the right state, prior to executing a write. Like the software-based techniques, care must be taken to avoid overhead resulting in lower, rather than higher, performance. The ROT technique cuts the miss rate in large caches for a studied multiprocessor application from 4 percent down to 2 percent. Using ROT does not rule out the use of software-based techniques. If software-based latency-hiding techniques are provided by the programmer, only latencies not covered by the software will be handled by ROT. ROT is described in Chapter 8.

2.4.2 Relaxing the Memory Consistency Model

Shared misses can also be removed by relaxing the memory consistency model, thereby removing the need to stall the processor on writes to a shared cache line.

Lamport states a requirement of the memory system of a multiprocessor called *sequential consistency,* which executes programs "as if the operations of all the processors were executed in some sequential order" [Lam79]. Sequential consistency requires a global interleaving of the writes in the system observable by all processors. Sequential consistency can be viewed as a specification of a memory's behavior. Hardware designs fulfilling this requirement can correctly execute programs written with this assumption of behavior. Sequential consistency is not the only specification of such behavior. Actually, there are very few programs that really require global order between all writes.

Processor consistency [Goo89] relaxes the requirement of global order between all writes, and instead states that the writes from each processor should be observed by all processors in program order. This requirement is enough for most existing programs. Implementing a memory system that supports processor consistency can be made much more efficient than a sequentially consistent one. This is especially true for systems with hierarchical networks, as discussed in more detail later, where the processors do not have to stall on writes to shared data.

Weaker forms of consistency, e.g., *weak ordering* [DSB86] and *release consistency* [GLL+90], remove all restrictions on the write order except around certain synchronization points put in by the programmer or by the compiler. This also implies that the delays for writes to shared data can be removed. Furthermore, writes to nonexistent data and read misses can be avoided by dynamic scheduling of instructions [GGH92]. The effects of weaker consistency models have been studied by Gharachorloo et al. [GGH91], and Zucker and Baer [ZB92].

Consistency models are discussed briefly in Chapters 9 and 10.

2.4.3 Write Update Protocol

Most cache-coherent shared-memory multiprocessors use a write-invalidate protocol. Using this strategy, a write to a shared cache line results in invalidations of all other cache lines through erase messages in the network before[4] the write can be performed locally in the cache. This causes *invalidation misses*; in other words, the requested cache lines would have been in the local cache if they had not been invalidated by the write of another processor.

Invalidation misses can be avoided by a write-update strategy. On a write, all other copies are updated instead of being invalidated. This strategy has been implemented in some multiprocessors, such as Firefly [TSS88]. Though superior for many applications, this strategy has been fatal for others [EK89]. None of the updated copies might ever get used, but the writer still keeps updating all copies, resulting in a high network load and no miss reduction. Attempts have been made to design adaptive single-bus protocols, which switch back and forth between the two strategies [KMRS86].

This dissertation suggests a general, adaptive method which implements the write-invalidate strategy by default but switches to write update on a per-cache line basis. It combines the write-update strategy with weaker forms of consistency and thus removes both invalidation misses and share misses. This protocol improved the node miss rate from 11 percent to 0.3 percent in one of the studied applications. The protocol is described in Chapter 9.

2.5 Context Switching

Another way of hiding latency is to change context on a cache miss and work on a different task until the requested data is retrieved. This technique is used in the Alewife project [CKA91], using a modified SPARC architecture with low overhead for a context switch. A more extreme use of the technique is in Dataflow [AI83], where a context switch is made for each instruction. A similar technique was also used in the HEP processor [Smi78], which had a limited number of contexts interleaved on the instruction level. The instruction frequency for each context matched access time to memory, effectively hiding its latency. Modern Dataflow architectures [NPA92] propose a less aggressive context-switching method with an increased granularity between switching points.

[4] If sequential consistency is supported.

Context switching can hide the effect of latency and some of the extra delay caused by contention. One obvious overhead with this technique is the extra time spent performing context switches. This overhead can be made very small if the contexts are interleaved, and/or if disjoint register sets are used. This method of context switching requires more active contexts than there are processors. The number of contexts per processor needed depends on the number of instructions between misses, and also on remote access latency. The Alewife project estimates a need for three to four times as many contexts as there are processors.

The overhead in running an application using c interleaved contexts per processor adds one factor to the efficiency in Equation 2.2.

$$efficiency_{multiple\ context}(n, c) = \frac{speedup_{algorithmic}(n * c)}{c * speedup_{algorithmic}(n)}$$

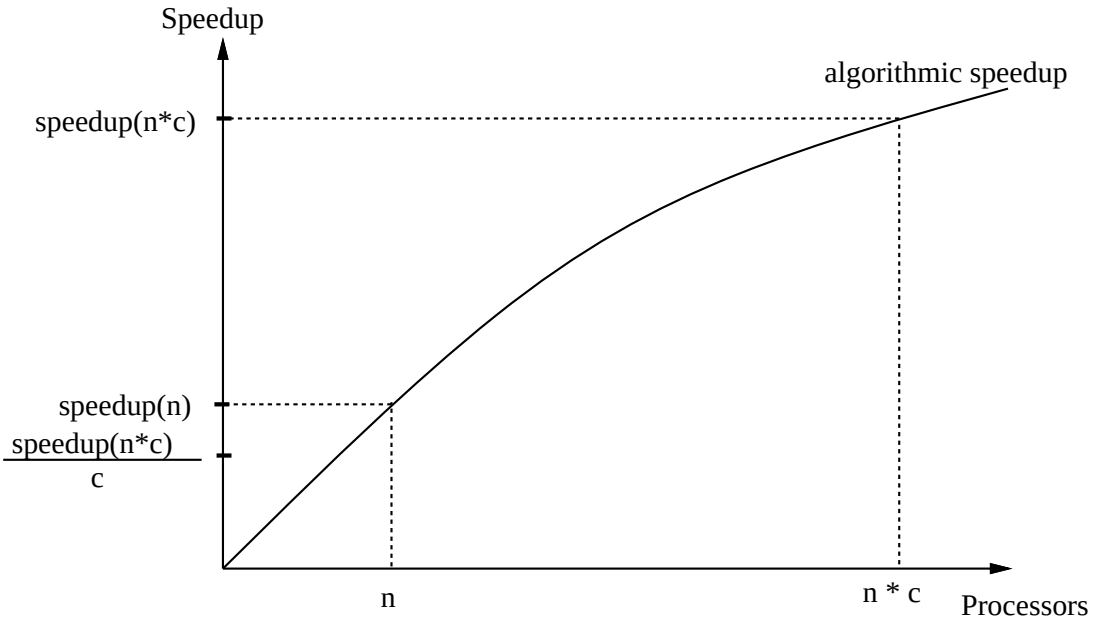


Figure 2.2: Illustration of the overhead of running c contexts on n processors.

The overhead is illustrated in Figure 2.2; $c * n$ contexts together achieve a speedup of $speedup(n*c)$. However, since each of the interleaved contexts runs c times slower than a single processor, the achieved speed is $speedup(c * n)/c$. This limitation is severe for some algorithms with a declining increase in their algorithmic speedup. Another drawback of context switching can be found in small processor caches. If the active contexts of a processor have large disjoint data sets, they interfere with each other in the small processor caches. The importance of this problem also varies with different applications. For algorithms not suffering from these limitations, context switching is a very effective latency-hiding technique, since the $efficiency_{latency}$ is

close to 1. These different constraints make it difficult to compare this strategy to the other latency-hiding techniques.

Context switching as a latency-hiding technique is not discussed further in this dissertation.

2.6 Networks

The processors of a parallel architecture are connected by a network. The network allows for the processors to share data. The network carries requested remote data. In a coherent multiprocessor, the network also carries the traffic that maintains coherence between different copies of the same data. The requirement of the network in a multiprocessor can be summarized as *low latency* and *enough bandwidth*.

Most speed-improving methods increase the burden on the network since computation, with its constant communication, is completed in a shorter time. However, one way of limiting the bandwidth requirement of the network has been discussed: caching. For the parallel programs studied in Chapter 4, architectures with infinite caches have a 2–40 times lower communication need than architectures with small caches, and a 10–200 times lower communication need than architectures without caches.

Yet another way of limiting the bandwidth requirement is to explore the locality of communication in such a way that only a limited part of the network is utilized for communication between two closely located processors.

Finally, efficiency can be improved by building up muscles—the bandwidth. In a mesh network, the bisectional bandwidth, i.e., the bandwidth available for communication between the two halves of the system, increases by the square root of the number of processors [Len91]. This is not true for the hierarchical network we are using in our Data Diffusion Machine. Its topmost component in the hierarchy easily could become the system's bottleneck since its available bandwidth does not increase with the number of processors in a natural way. That does not mean available bandwidth cannot be increased, however. We can widen the bottleneck by building a fast ring, time-slotted into address domains [BD91], and/or by building *fat trees* [Lei85] where the topmost component is implemented as several parallel components divided among address domains. The topmost component of a hierarchy may be built from any kind of interconnection network—a *heterogeneous network*. Several strategies for utilizing these methods will be discussed in Chapter 10.

2.7 Synchronization

Throughout the computation, the different parallel processes need to synchronize with each other. Small-scale multiprocessors have successfully used software busy-wait locks and barriers to implement the synchronization needs. The busy waiting can be performed as local reads to the cache as long as the value of the lock stays the same. On large-scale multiprocessors, the busy-wait locks produce heavy communication bursts. The reason for this is that each processor busy-waits on a value in the same cache line. As soon as the value changes, all processors will see the change and draw the same conclusion, e.g., when a lock is changed from locked to unlocked, all processors will see the free lock and try to grab it. The communication produced is $O(n^2)$.

A queue-based lock structures the lock requesters as a linked list. A processor busy-waits on a flag changed by the predecessor in the list. The queue-based locks have two obvious advantages over the simple test and test&set lock: requesters will obtain the lock in a first-come-first-serve manner and it will not produce a communication burst each time the lock is released. Software implementations of queue-based locks can be made efficient [MCS91], but add a small overhead for the cases of no contention. Queue-based locks can also be integrated in the cache-coherence protocol, further reducing the overhead [GVW89].

Wake-up implementations of locks instead actively wake up a processor (or process) when an awaited event occurs. Examples of this can be found in the synchronization in the HEP processor [Smi78], and in many implementations of data-flow. These kinds of synchronization can solve data-dependencies at run-time, which allows for a more aggressive scheduling and exploration of a more fine-grained parallelism. This kind of synchronization can also be implemented in cached systems [HLH91].

Synchronization is not covered in this work.

⋆ ⋆ ★ ⋆ ⋆

In this chapter we identified scalability as a goal rather than a yes-or-no proposition. Parallel efficiency was identified as a measurement of parallel architectures.

As a rule of thumb, a large multiprocessor can accept the negative effects of a one-percent node miss rate and still achieve reasonable parallel efficiency. For some applications this can be achieved with large caches, while for others latency-hiding techniques are needed. We briefly discussed some possible latency-hiding techniques.

Finally we noted that architectures with (infinitely) large caches have a network traffic need 2–40 times lower than those of architectures with small caches and 10–200 times lower than those of architectures without caches.

3

Cache Only Memory Architectures

LONG LATENCIES introduced by remote accesses in a large multiprocessor can be reduced by caching. Caching also decreases the network load.

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

This chapter discusses different ways of implementing a COMA. We also identify four properties of a COMA that makes it more general than a Non-Uniform Memory Architecture (NUMA), in the sense that a COMA has a behavior that suits a larger class of programs.

3.1 Introduction

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

Systems based on a single bus suffer from bus saturation and typically only have a few dozen processors—each with a local cache. The contents of the caches are kept coherent with a cache-coherence protocol, whereby 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 consequently called uniform memory architecture (UMA).

In architectures with distributed shared memory, known as nonuniform memory architectures (NUMAs), 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, with a varying network delay to different nodes. Earlier NUMAs had no coherent caches and left the coherence problem to the programmer. Research activities today strive toward coherent NUMAs with directory-based cache-coherence protocols, for example DASH [LLG+90] and Alewife [CKA91]. Each data quantum—or cache line—in a NUMA has a dedicated *home node*. The home node stores its data values, and, in the case of a cache-coherent NUMA, some *directory information* for deducing the locations of additional cache-line copies. In this text, NUMA refers to this kind of architecture.

Programs can be optimized for NUMAs by partitioning the work and data. Indeed, the operating system may play an active role in this partitioning. Where processors are made to make the most of their accesses to their own part of the shared memory, clearly scalability superior to that of UMA can be achieved.

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. The partitioning of data among the memories need not be static, however, 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, 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 attracts data used by a processor to the processor's attraction memory. The coherence unit, which is moved around by the protocol, is called an *item*—comparable to a cache line. Figure 3.1 compares COMA to more traditional architectures.

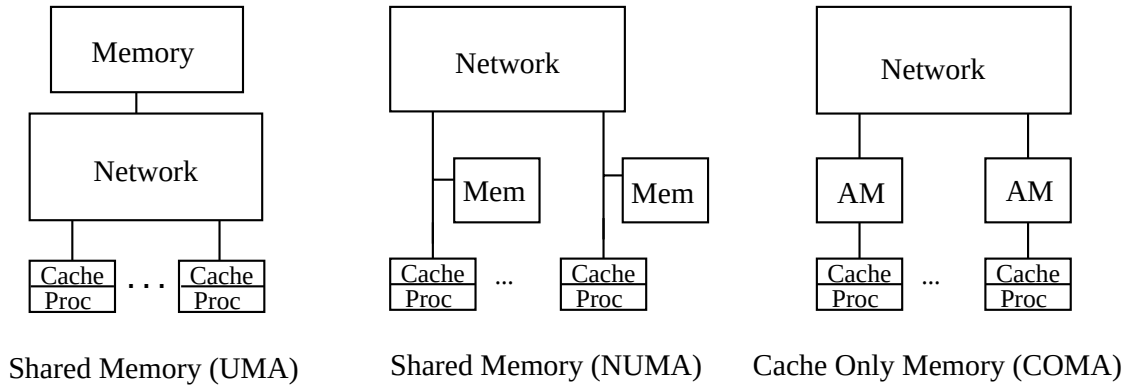


Figure 3.1: Comparing COMA to more conventional architectures.

When memory is referenced, a virtual address is translated into an *item identifier*. The *item identifier space* is logically the same as the physical address space of other machines, but there is no permanent mapping between an item identifier and a physical location. The physical location of an item can be in any attraction memory, and the item can exist in many attraction memories simultaneously.

3.2 COMA Implementation Issues

At first sight a COMA looks expensive and difficult to implement. The programming model of such an architecture also seems unfamiliar. However, a closer look at the implementation of a COMA, and its programming model, shows that it differs only slightly from more traditional architectures.

3.2.1 Cache-Coherence Protocols

Recent years have seen extensive study of the problem of maintaining coherence among read-write data shared by different caches—for example directory-based and snooping-based techniques [Ste90].

Coherence can be kept by either software or hardware. We believe hardware coherence in a COMA is more efficient. Each item must be small to prevent performance degradation by false sharing. In other words, two processors accessing different parts of the same item might clash although they share no data. For instance, we measured a speedup of 50 percent when false sharing was removed from the SPLASH application MP3D-DIFF.

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

is about to occur. An example of such a protocol is the *write-once protocol* introduced by Goodman [Goo83]. In that protocol, shown in Figure 3.2, each cache line can be in one of four states: Invalid, Valid, Reserved, or Dirty. Many caches might have the same cache line in the Valid state at the same time, and may read it locally. When writing to a cache line in the Valid state, 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. Now the common memory also contains the new value. When a cache already has the cache line in the Reserved state, it can perform a write locally without any transactions on the common bus. Its value will now differ from that in the memory, and its state is therefore changed to Dirty. Any read requests from other caches to that cache line must now be intercepted, in order to provide the new value, marked by "intercept" in the figure.

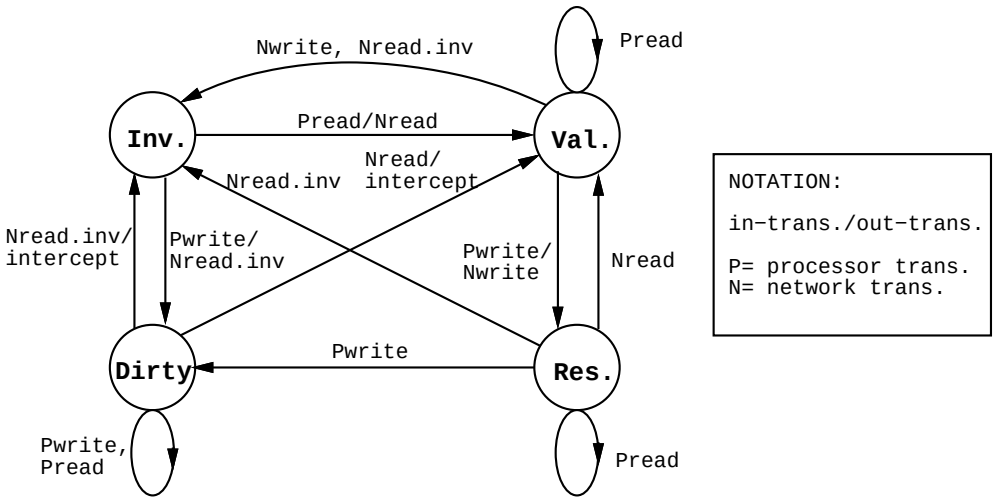


Figure 3.2: The write-once protocol.

Snooping caches, as described above, rely on broadcasting and are not suited for general interconnection networks: unrestricted broadcasting would drastically reduce the available bandwidth, thereby obviating the advantage of general networks. Instead, directory-based schemes send messages directly between nodes [CF78]. This is the technique most often found in cache-coherent NUMAs, where each of the distributed directories contain data for their own part of the shared address space. The directory is the *home* for that part. The home node also keeps directory information for each of its data quanta[1] of their address space. Directory-based cache-coherence protocols differ as to how they store directory information.

Fully mapped directories [LLG+90] store one *presence bit* for each of the nodes in the system, indicating which nodes have cached copies of each quantum. The *LimitLess*

[1] A quantum is of the same size as a cache line.

scheme [CKA91], in contrast, contains a limited set of pointers and can keep track of a limited amount of sharing per data quantum. When the number of copies exceeds the number of pointers, it emulates a fully mapped directory scheme in software. The new standard IEEE P1596—*Scalable Coherence Interface*, SCI [JLGS90], keeps a pointer per quantum in the home, pointing to a doubly linked list of the caches with shared copies of the quantum. Yet another approach, the Stanford Distributed Directory protocol (SDD) [TD90] stores a pointer in the home referencing a singly linked list. Each of these schemes exhibits different performance and memory overheads. The scheme with the highest memory overhead, but also the best performance, is the fully mapped directory scheme.

In the fully mapped directory scheme, a read request is sent directly to the home, which replies with data and sets the bit corresponding to the requesting node. An extra bit, contained in the directory, may indicate that the datum is Dirty in another remote node. In this case the read request is passed on to the cache with the Dirty copy, which provides a copy of the datum for the requesting node and also sends an update message to the home. Upon a write to a shared datum, a write request is sent to the home, which then sends out invalidation messages to all caches with shared copies. The caches each respond with an acknowledge message to the requesting node. In order to achieve *sequential consistency*, all acknowledges must be received before the write can be performed

3.2.2 Implementing a COMA

In this edition, a new chapter has been added in Appendix C. It presents several different attraction memory implementations, their complexity and performance. This is a good introduction to the extra functionality needed in a COMA architecture.

The cache-coherence protocol for a COMA can adopt the techniques of other cache-coherence protocols and add functionality for finding an item on a cache read miss and for handling replacement [HLH92]. The search for the item compensates for the lack of a home for data in a COMA. The replacement strategy must make sure that the last copy of an item is not lost.

A protocol for a directory-based COMA built on a general network could have part of the directory information statically distributed in NUMA fashion. The data would be allowed to move freely, while the directory information of each item has a home. The directory home could also be a synchronization point for replacement and assure that the last copy of an item is not lost. The protocol could be similar to those of NUMAs, but extended with the functionality of safe replacement and finding a requested item.

Figure 3.3.a shows a COMA implementation on a general network. A read request first goes to the home node of the datum, where its directory information always

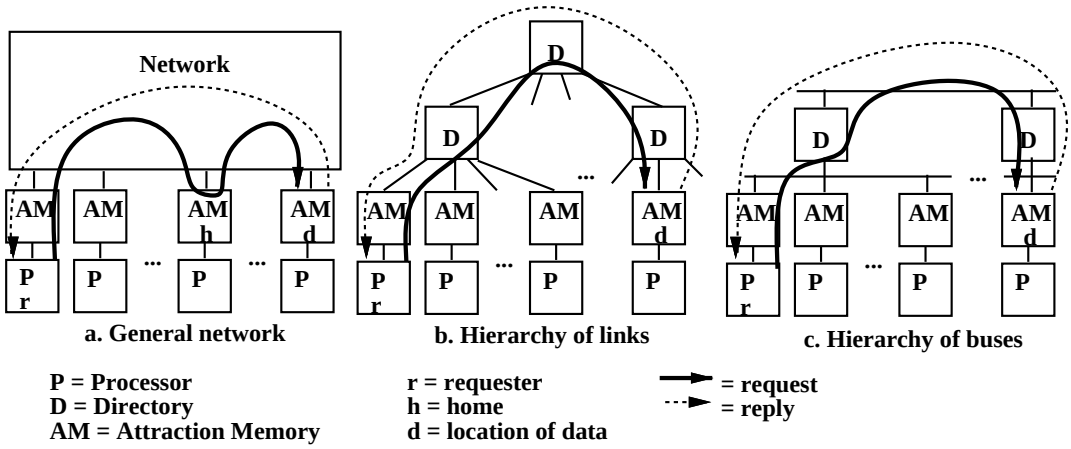


Figure 3.3: Examples of how a read request can be serviced on COMAs implemented with different topologies.

resides, to find out in which node a valid copy exists. One extra hop to that node is often needed before the data value can be returned to the requesting node by a third hop. A software-based implementation of this type, called *distributed virtual shared memory* (DVSM), has been proposed by Li and Hudak [LH89], where the retrieval of and coherence among data are maintained by software. The items in DVSM are page-sized, which enables the validity check of accesses and translation to local physical memory to be performed by the MMU at runtime. A shared item (page) resident in the local memory is tagged with read permission, and an exclusive item is tagged with read and write permission. In DVSM terms this is called the *fixed distributed manager algorithm*. Another proposal for such a protocol, targeted for hardware, has been proposed by Gupta et al. [GJS92]. A COMA based on the proposals above cannot explore the communication locality and combining to the same extent as a hierarchical COMA, since the home of a datum does not move. Yet another solution has been proposed, where the home has the ability to move and where sometimes the third hop can be avoided [HL91]. There also exists a similar DVSM proposal called *dynamic distributed manager* [LH89]. A hierarchical COMA relies on a hierarchical search algorithm to locate a requested item. A hierarchy can also explore locality in communication in a natural way.

Figure 3.3 shows two possible hierarchical implementations, 3.3.c based on buses [HLH92] and 3.3.b based on links [RW91]. State memories between each level in the hierarchy, *directories*, contain enough information to guide a request on its way to a copy of the requested datum [WH88, HHW90]. The search time is proportional to the number of levels in the system, which are $\log_b n$ where b is the branching factor and n is the number of processors in the system. The topmost bus (or node) of a hierarchy can easily become the bottleneck of the system. A fat tree [Lei85] could be the solution to that problem, where the address domain is split

into several distinct parts, each with its own top bus. Chapter 10 presents several alternative solutions along these lines.

3.2.3 Software

Although no shared memory exists, the processors in a COMA—and subsequently the programmer—are still given a coherent shared-memory view of the system. Actually, the programming model presented by a COMA architecture is more general than that presented by the NUMA in-so-far as it is less sensitive to some optimizations.

Programs may be 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 is attached to its node [BSF+91]. Running a thus optimized NUMA program on a COMA architecture would result in NUMA-like behavior, since the work spaces of the different processors would migrate to their own attraction memories. However, running an unoptimized version of the program on a COMA would also perform well, since data are attracted to the processors which use them regardless of the memory addresses of the data.

Another positive property is the COMA's migration behavior, which adapts to dynamic scheduling in a general way: the work space migrates throughout the computation according to its usage. A program optimized for a COMA could take this into account. This property also simplifies the task of process migration for the operating system. A process is migrated by moving its current program counter to a new processor. The process's active working set is then attracted to the AM of the new processor by the protocol.

To a single user, the sum of the AMs looks like a single shared memory. Like a cache-coherent NUMA, a single program can use all the available memory as its primary memory, thus avoiding unnecessary disk accesses—the property of scalable memory. The AM of the node where the program is run acts as a huge second-level cache.

Even if most of the underlying functionality of a COMA is hidden from the programmer, some optimizations, or lack thereof, will have an impact on performance. Although the replication entity is small, a COMA still suffers from true sharing and false sharing, as shown by our study. One of the studied applications has been rewritten to explore the ability to migrate small data entities, resulting in a miss frequency reduction of one order of magnitude. The hierarchical COMA could also benefit by an understanding of the hierarchy by the application scheduler, and thus keep communication local to a branch of its tree. Such rewriting can be seen in two applications studied in Chapter 7.

3.3 Comparing COMA to Other Architectures

The COMA is reminiscent of a nonuniform memory architecture (NUMA), like the DASH [LLG+90] and Alewife [CKA91], in that all the shared memory is physically divided among the processors. In a NUMA, however, different portions of the data are semi-statically allocated to each memory unit according to the semi static mapping of the pages to local node memory.

In a COMA, an item has no home and might be moved by the protocol to reside in any or many AMs according to its usage.

Compared to a NUMA, the COMA suits a larger class of applications. The differences between a COMA and a NUMA can be divided into four somewhat related areas:

- **Fine-grain Migration** – In a COMA, the only copy of an item will be moved close to where it is being used. Migration is also possible in a NUMA through page migration, but with a larger grain.
- **Massive Replication** – More than one copy can exist in the system. This property can be found in a NUMA, but to a much smaller extent.
- **Dynamic Adjustment of Address Space** – The ratio between replication and memory size is adjustable dynamically, so that either applications that benefit by massive replication or applications that require a large address space can perform well.
- **Layering of Remote Accesses** – Attraction memories add another memory system layer to fill the speed-size gap for remote accesses, making multiprocessors with long remote accesses less sensitive to hit rates in small caches.

3.3.1 Migration

In a NUMA, a virtual page is physically allocated to a physical page with its location fixed to one of the nodes. For some applications, allocating the page to the node which first accessed it is a good strategy. For other applications, where a master processor initiates the memory before the parallel part of the execution starts, this strategy would create a memory bottleneck at the master's node. Regardless of the strategy used to first locate a page, its usage might be changed by a dynamic scheduling strategy in the application itself or a process-migration strategy in the operating system. When implementing an operating system on a NUMA, effort has been devoted to algorithms that move a page to the new node when a process is migrated [BSF+91]. Other suggestions rely on specialized hardware to monitor the access-pattern behavior of a page and help the operating system decide where a

page should reside [SJG92a]. This strategy will also help migration in dynamically scheduled algorithms, given a coarse data-grain size. The proposed hardware support for detection and migration is far from simple, however, and the migration of entities smaller than a page is not possible.

The migration of data is natural to the basic COMA. Data in quanta of items are moved close to their usage in the system, i.e., if there is only one copy of an item, its location is decided by its usage. Moving larger entities, like pages, can have a performance advantage over moving several smaller adjacent items. In Chapter 8, a simple hardware-based prefetcher will be shown to improve COMA performance in this respect.

3.3.2 Replication

The same datum might exist simultaneously as several copies in a multiprocessor with caches. In some applications, large sets of data are requested by many (all) processors. Allowing many (all) processors to have their own copy of a popular data set can have a large impact on the performance. Replication is the amount of duplicated copies of items in the system. NUMA can support replication of read-only data, like instructions, by its virtual memory system in that every processor can have its own copy of a page with code in its local memory.

COMA supports massive replication of data because additional copies are allowed in other AMs. This property is similar to that of remote-node caching in a NUMA. The local caches of a NUMA can store copies from outside the node's home. There might be several copies of a cache line cached in different nodes at the same time. This replication is limited by the size of the caches in a NUMA. NUMA caches are also used to compensate for the lack of fine-grain migration support; i.e., instead of migrating the only instance of an item to where it is currently being used, one copy stays in the home node, while another copy is moved to the cache of the remote node. The replication ability of a NUMA can be increased by introducing large second-level caches. However, the memory invested in the second-level cache of one node cannot be used by the other nodes, i.e., memory scalability is violated.

The potential for replication is much greater in a COMA than a NUMA. Replication in a COMA will keep increasing as long as there is space in the AMs. When a steady state is reached and the AMs are full, further replication of a datum will force replacement to occur, decreasing the replication of another datum. So, replication in a COMA is predetermined by how much larger the sum of the AM sizes is compared to the current size of the item space.

3.3.3 Variable Item Space

One motivation for shared memory is the scalable-memory property that increases the generality of the architecture; i.e., a single program run on one of the processors may use the whole shared memory and thus avoid unnecessary page swapping. A COMA has yet another property that increases its generality: variable address space. Some programs can benefit by massive replication. Other applications do not benefit at all by massive replication, but require a large physical address space. The single program example above illustrates such a case. Multiprogramming—several simultaneous users—is another example where a large memory space is preferable to massive replication. In contrast, database and knowledge-based applications are examples that benefit from extensive replication. Replication is supported in a NUMA by its second-level caches. The ratio between replication of read/write data and physical address space is predefined by the size of the NUMA caches and that of the physical memory. A COMA has the potential for massive replication of read/write data. In order to provide massive replication with a limited amount of AM memory, the number of different items in the system must be limited, i.e., the item space must be cut down. In a COMA, this adjustment of the item space can dynamically be controlled by the operating system.

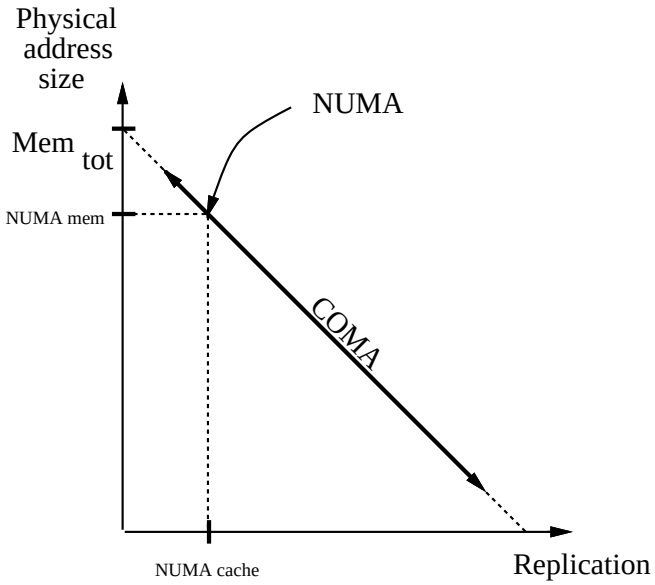


Figure 3.4: A NUMA has a statically fixed relationship between the size of the caches (replication) and the physical memory, while a COMA dynamically can change its working point to suit the application.

As stated before, a COMA will increase replication until space runs out in the AMs. At this point, the amount of replication is determined by the size of the item space presently mapped by the operating system. A large, mapped item space results in a lower amount of replication, and vice versa, as shown in Figure 3.4.

The item space is divided into pages of traditional-architecture size, and the item pages ("physical pages") not in use are kept on a *free list*. When a new item page is needed, a page from the free list can be used. So far, pages in a COMA are handled similarly to pages in a traditional architecture. A page-reclaiming algorithm can be used to move pages to the free list.

The operating system of a COMA can also decide to purge all the items on an item page and move the page to a *purge list*. Item pages on the purge list do not occupy any physical locations in the AMs, thus permitting a higher degree of replication for other item pages belonging to the same set in the AMs. The pages on the purge list should be equally distributed over the mapping of addresses into physical AM memory to effectively support extra sharing. The purged pages require extra bookkeeping.

What size should the address space be for the application run? This is determined by two parameters: the frequency of pages in/out and the pattern of replacement traffic at steady-state execution, as shown in Table 3.1.

The possibility of a variable address space, as described above, is a completely new property not yet studied in an architecture. The algorithms described probably need some adjustment and tuning through experimentation on real machines. The cost of a page in/out and the cost of a replacement should be included in the strategy.

Page in/out frequency	**Replacement frequency**	
	Low	High
Low	OK	addr. space too large
High	addr. space too small	buy more memory

Table 3.1: Adjusting the address space to the application.

A NUMA with COMA-like behavior can be implemented by using half the available memory as "shared memory" and half the memory as remote-access caches. Such a system would have a maximum physical address space of half the maximum size of a COMA, and a maximum replication of half the maximum replication in a COMA, if the same amount of memory is used. If memory is considered as being cheap, this can be compensated for by adding more memory for the NUMA implementation.

3.3.4 Layering of Remote Accesses

Fast memory is relatively expensive. Normally, only a fraction of the memory system will be built with memories of the fastest kind. Layering the memory system into several layers of caches with increasing size and access times is a way of making uniprocessors fast, yet less sensitive to the hit rate of one specific layer. Upon a miss in a small one-cycle first-level cache, a second-level cache hit of eight-cycles

latency could prevent a twenty-cycle main-memory access. With a twenty-cycle latency to the main memory, a miss rate of five percent can be accepted and a reasonable efficiency can still be achieved according to the discussion in Chapter 2 (Figure 2.1). However, in a multiprocessor, the latency of a remote miss is in the hundred cycle range. The fraction of misses that can be tolerated is therefore in the one percent range. Relying on the second-level caches as the only layering of remote accesses might create too great a speed-size gap to the remote memory.

Another way of viewing the large attraction memories is as an extra layer in the large speed-size gap between the (second-level) cache and the remote shared memory space, which is slow and huge. One example where this picture becomes clear is the memory-scalability example—a single running processor element using the entire shared memory as its main memory. The sum of all the node memories acts as the huge, but very slow, main memory of that single processor. Even though the memory is scalable in that all of it can be used, the access time to it could still make such use unattractive. An acceptable hit rate in the local cache would be crucial to behavior. In a COMA, the large attraction memory of the running node acts as a second- (or third-) level cache, making execution less sensitive to an acceptable hit rate in small caches.

The layering might be just as important in a multiprocessor, but less obvious. The large second-level cache of a NUMA works well up to a certain data set size per processor. Increasing the data set above a certain critical size could result in a sudden and unexpected performance drop. A similar drop in performance could also be expected from a COMA, but for a much larger data set size, as will be seen in Chapter 7. The COMA is therefore a more general architecture.

Private data residing in the relatively slow local memory benefit from caching in a NUMA. The second-level caches of a NUMA can be used to layer accesses both to local and to remote memory. Layering private data and replicating remote data interfere with each other. A remote datum with the potential of avoiding a remote access of around 100 cycles might be replaced by a private datum with the potential of preventing a 10 cycle delay. The replacement strategy in that case is not obvious, and should include some cost function for replacement. For this reason, an optimal organization for a NUMA might be to have large (second-level) remote caches adjacent to the local memory, dealing only with replication. Such a second-level cache layers remote accesses in a similar manner to the attraction memory of a COMA.

A COMA may also use a second-level cache between its attraction memory and its first-level cache. The purpose of this cache is to fill the speed-size gap between the attraction memory and the processor cache, and not to fill the speed-size gap between the processor cache and the remote memory.

3.3.5 Drawbacks of a COMA

The massive migration and replication abilities of a COMA are beneficial to performance. However, one negative property of a COMA is its higher price of locating a requested datum on a read miss. Since no home for a datum exists, there is no obvious place where that datum can be found. COMA's search for the datum on a read miss is expected to add extra latency compared to the NUMA, even though there might be cases where a hierarchical COMA could explore the shorter latencies of communication locality. As pointed out earlier, only capacity and conflict misses are removed by a larger cache. The coherence misses remain. A COMA has to pay in search time for each coherence miss. On the other hand, it is rare that a NUMA finds a clean copy of a datum in its home on a coherence miss [SJG92a]. Thus, an additional hop in the network is needed to retrieve the Dirty copy. A NUMA with a three-hop miss time that is shorter than the COMA's miss time could outperform a COMA in applications where coherence misses are dominant. Comparing COMAs and NUMAs implemented on identical networks, COMAs pay for only one extra directory lookup for each coherence miss.[2] This is caused by the extra layering offered by the attraction memories. An extra memory layer adds latency to the accesses missing in its layer. A request is not sent out on the network before its miss in the attraction memory is determined.

Another drawback is the COMA's usage of extra memory components. The memory overhead of a COMA can be divided into two parts: the extra memory needed to implement the associativity of the AM and the extra memory used by the cache-coherence protocol. Compared to a NUMA, only the AM overhead should be counted, since the NUMA has the same requirement for implementing a cache-coherence protocol as the COMA. The extra bits needed per item in an AM are $\log_2(am * ways)$, where am is the maximum number of attraction memories, and $ways$ is their associativity. So, 64 AMs, each with 4 ways, need 8 extra bits. An item size of 16 bytes results in a 6 percent memory overhead.

At first glance, it looks like the access time to the attraction memories should be long. For each access to the attraction memory, the COMA has to make a directory lookup to determine if the item is present in the directory and in the right state. However, a similar directory lookup also has to be performed by the cache-coherent NUMA. An access to the local memory of the node has to be combined with a directory lookup to determine if the cache line is in the right state, e.g., a read of a cache line that is Dirty in another node is not allowed. This is identical to the attraction memory access. An attraction memory with more associativity will add a small extra latency for reads, and a longer latency for writes as discussed in more detail in Chapter 6.

[2] This is discussed in more detail in Chapter 4.

It is also tempting to assume that the implementation of a COMA implies a greater complexity. We have found no support for that assumption. Apart from the extra memory overhead accounted for, there is no extra hardware needed for a COMA implementation compared to a NUMA. Actually, the need for a second-level cache is smaller in a COMA than in a NUMA—a possible source of simplification. The attraction memory of a COMA also removes the need for the remote-access caches used in NUMAs.

⋆ ⋆ ★ ⋆ ⋆

	NUMA	**COMA**
Programming model	*Shared memory*	*Shared memory*
Program address	*Virtual*	*Virtual*
Coherence address	*Physical address*	*Item identifier*
Address binding	*Static*	*Dynamic*
To/from disk	*Page*	*Page*
Migration quanta	*Page*	*Item*
Replication	$\sum cache\ size$	$0 \leftrightarrow \sum memory$
(Physical) Address Space	$\sum memory$	$\sum memory \leftrightarrow AM\ size$
Who decides repl/addr space	*HW designer*	*OS + HW monitoring*

Cache-only memory architectures (COMA) can provide the effects of large caches at a small extra cost—all of the memory is organized like caches. We briefly described a few alternative implementations of COMA and identified four properties which make COMA more general than NUMA: fine-grain migration, massive replication, dynamic adjustment of address space, and layering of remote accesses. Finally, we discussed the drawbacks of COMA, all of which can be regarded as minor.

4

A Quantitative Performance Study

WHICH parallel architecture—NUMA or COMA—is preferable, and when? In Chapter 3 we gave reasons why COMA can be regarded as more general than NUMA. Here we compare the relative performances of some NUMA and COMA implementations.

Comparing different architectures is difficult—especially if they differ in many important respects. In this study we have favored evaluating a large design space over a single, detailed study. We chose a simple and easy-to-understand analytical model, based on the varying behaviors of capacity and coherence misses. A time-consuming simulation may have cut possible design space, but it would not have produced more accurate results, unless very detailed models were used.

We model ten possible implementations of COMA and NUMA executing eight different applications. The models allow for general studies of the effects of different technologies, topologies, and numbers of processors.

We also present models of three real architectures, one NUMA and two COMAs, and compare how their relative performances vary with the number of processors.

4.1 Analytical Study

This study is partly inspired by a paper comparing COMA and NUMA performance by Stenström et al. [SJG92a]. We felt that this study compared one instance of an unoptimized COMA, a hierarchy of links with a branching factor of four, to a completely different NUMA, built on a 2-D mesh. Furthermore, we could not recognize the latency equations used for the COMA, or the technology parameters chosen. During the ISCA19 conference, where that paper was presented, we performed a back-of-the-envelope calculation resulting in a completely different result. That method was slightly refined together with Per Stenström and has been expanded here to cover several configurations.

Analytical models often describe simplified models of some more complex system. This is also true of the model described here. Simplicity enables explanation and understanding of the study. With the help of computers, the effect of varying a parameter is determined in seconds rather than days. This study emphasizes variation rather than accuracy, in contrast to the detailed simulation study in Chapter 7. We vary the following parameters in our model:

- **Two architectures**—NUMA and COMA.
- **Four topologies**—a link-based hierarchy, a bus-based hierarchy, a two-dimensional mesh, and a three-dimensional mesh.
- **Branching factors**—their variations have an impact on the performance of hierarchical architectures.
- **Three implementation technologies**—today's, influenced by the technology used in the hierarchical DDM design and the mesh design of the DASH [Len91], and tomorrow's, exemplified by the technology used in the Tera-DASH proposal [Len91], and a future technology.
- **Latencies of different sized machines**—up to 1024 processors.
- **Three first-level cache sizes**—4 kbytes, 16 kbytes, and 64 kbytes.
- **Eight applications**—from the SPLASH suite [SWG91]:
  MP3D, particle-based wind tunnel simulator
  PTHOR, logic simulator
  Locus Route, standard cell router
  Water, N-body molecular dynamics simulator
  Cholesky, factorizing sparse matrices
  LU, LU decomposition program
  Barnes-Hut, N-body problem solver
  Ocean, Ocean basin simulator

4.1.1 The Simple Analytical Models

The time a processor performs useful work is often referred to as its busy time. In this study we assume that the busy time for all studied architectures is identical. Only the time the processors are idle waiting for some external action, like global communication, differs. We compare the performance of the different architectures by comparing how much idling they do while waiting for global communication or slower memories. One of the main factors slowing the execution of an application is the time the processors spend waiting for remote data read accesses. Our models consider only this source of penalty. Each node consists of a processor-cache pair and an attraction or local memory connected to the network. The systems are modeled with first-level caches of various sizes. We distinguish between coherence misses and capacity misses [1] to those caches. The fractions of misses from each of the two categories are multiplied by the average latency for that category, and their products are added in Equation 4.1 to form the *miss penalty* for the architecture. The miss penalty is the time the processor waits for memory and network accesses. By misses we mean the fraction of misses in the total number of accesses. The latency is measured as the number of processor clocks for servicing the misses, the miss penalty is the average number of idle cycles per data access.

Miss penalty:

$$miss\,penalty \quad = \quad misses_{coh} * L_{coh} + misses_{cap} * L_{cap} \tag{4.1}$$

All capacity misses are assumed to hit in the attraction memory of the COMA, and the AMs are assumed to be infinitely large. This is a valid simplification only if the data set per processor is smaller than the size of each AM for real-sized applications, assuming fully associative AMs. As a comparison it can be noted that the AMs of the KSR1 and DDM implementations are two orders of magnitude larger than ordinary sized NUMA caches. In cases where infinitely large attraction memories are not realistic due a large data set, a judgement can be made without any analytical studies, since the relatively small caches of a NUMA would suffer greatly.

For sequential consistency, the time a processor spends waiting to perform writes is also significant. That delay is not modeled here, and the comparison is therefore only valid for weaker forms of consistency where the write delay can be hidden. This is a potential disadvantage for the hierarchical topologies and their rapid responses to write requests [LHH91]. Other factors, like synchronization delay and barrier wait caused by uneven load balancing, also add delay not modeled here, but are less important for this kind of comparative study, because the different architectures are expected to have the same behavior in this respect. Our model does not take into

[1] We also included conflict misses into the capacity miss category. Compulsory misses will be discussed later.

account the positive *combining effect*, which can be found in hierarchical COMAs,[2] nor does it model contention for resources, such as network or memory. Not modeling contention may well be an advantage for the NUMA architecture, since its poor hit rate results in more network and directory activities. Contention might also be a problem in hierarchical systems, unless some bandwidth-increasing method is used at the top. Finally, the effect of locality in communication possible in hierarchical systems is not modeled.

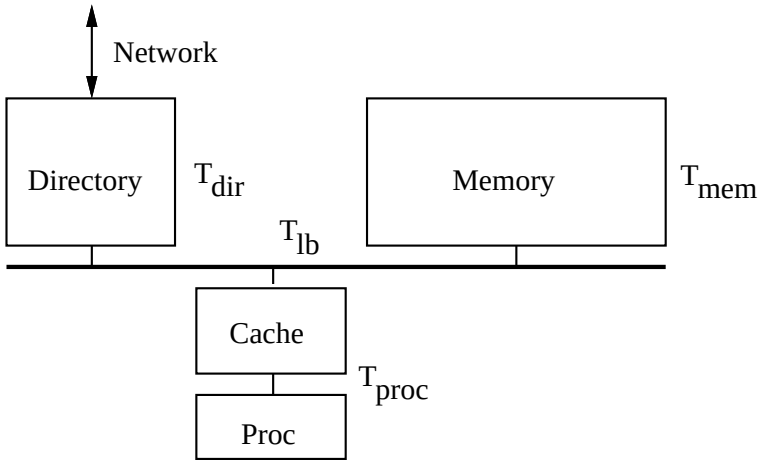


Figure 4.1: The structure of an ideal processor node of NUMA or COMA.

In this study we assume an ideal implementation of a processor node according to Figure 4.1. Details from several alternative COMA node implementations can be found in Appendix C. A hit in the processor cache is assumed to take one processor cycle, the time unit in which all other latencies are expressed. The time it takes to access the first word of a local NUMA memory or a local AM is denoted T_{mem}, and the time for a directory state lookup and protocol validation is T_{dir}. The time on the local bus T_{lb} includes arbitrating, sending the read request, and transferring the first word of data. The time spent in the requesting processor to initiate and terminate a transaction is denoted T_{proc}, including the time for reading the remaining words and to restart the processor. The equations also contain n, representing the number of processors. We assume an optimization where a NUMA can detect the need for a remote access by looking at the address on the local bus, adding no extra latency. A COMA has to perform a directory lookup before the need for the remote access is detected. For each topology, the average time for communication between two random processors is called T_{hop}.

[2]Discussed in Chapter 5.

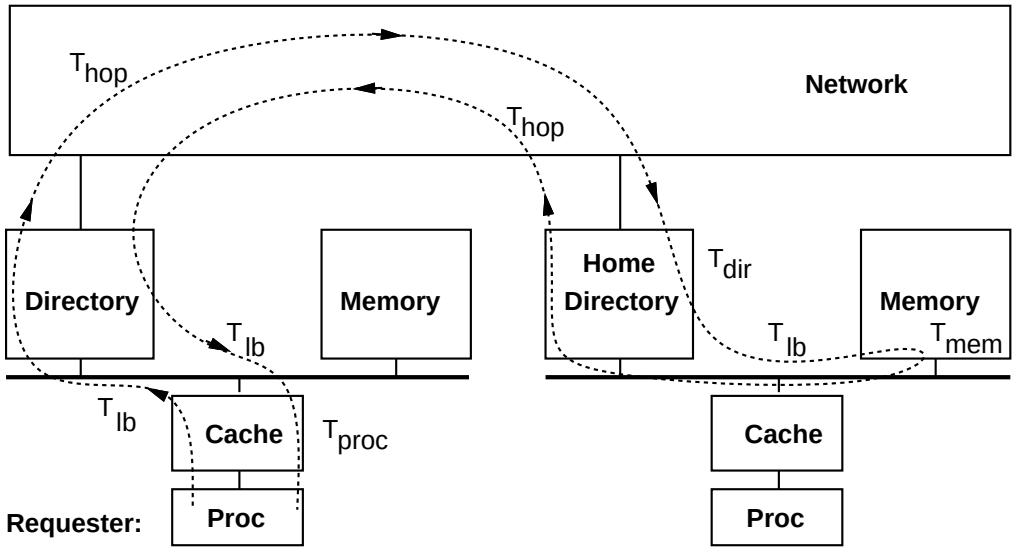


Figure 4.2: An illustration of a NUMA remote read access, when the requested datum resides clean in the home node.

4.1.2 Limitations of the Study

For sequential consistency, the time a processor spends waiting to perform writes is also significant. That delay is not modeled here, and the comparison is therefore only valid for weaker forms of consistency where the write delay can be hidden. This is a potential disadvantage for the hierarchical topologies and their rapid responses to write requests [LHH91]. Other factors, like synchronization delay and barrier wait caused by uneven load balancing, also add delay not modeled here, but are less important for this kind of comparative study, because the different architectures are expected to have the same behavior in this respect. Our model does not take into account the positive *combining effect*, which can be found in hierarchical COMAs [Hag92] nor does it model contention for resources, such as network or memory. Not modeling contention may well be an advantage for the NUMA architecture, since its poor hit rate results in more network and directory activities. Contention might also be a problem in hierarchical fat tree systems, unless the root is wide enough. Finally, the effect of locality in communication possible in hierarchical systems is not modeled.

4.1.3 Latency in General Networks

Using these parameters, we can now express the coherence and capacity miss latencies for a NUMA or COMA built of a general network.

Capacity misses are caused by a processor referencing a datum which there has not been space for in the processor's cache. In a NUMA, any miss to a datum that belongs to a remote node will force a remote read. Figure 4.2 illustrates such a read, where the remote copy is "clean", with the following latency components:
The processor cache arbitrates for the local bus and sends the read command and the address (T_{lb}). The request is transferred to the home node, i.e., the node in which the directory information for this datum resides (T_{hop}). The home node performs a directory lookup (T_{dir}) which indicates that the datum resides clean in the memory of the home. The directory arbitrates for its local bus, sends the read command, accesses its memory, and receives the first data word ($T_{lb} + T_{mem}$). The data is sent back to the requesting node (T_{hop}), which arbitrates for its local bus, sends the write command and transfers the first word of data (T_{lb}). The processor cache receives the remaining words, and the processor is restarted (T_{proc}).

Some optimizations are possible in the above example. Directory lookup and memory access in the home node can be overlapped; i.e., the access on the local bus can start before it is determined if the datum resides clean in its local memory. This optimization might cost unnecessary accesses on the local bus, but is probably worth paying for. In the formulas below, we include this optimization, and have not added T_{dir} in the equation for a NUMA capacity miss.[3]

In Equation 4.2 for $L_{cap-NUMA}$, the cache line is found in the memory of the requesting node with a probability of $1/n$ and found "clean" in a remote home node with a probability of $(n-1)/n$. When the datum is found in the memory of the requesting node, the latency is: $T_{mem} + T_{lb} + T_{proc}$.

Coherence misses are caused by some other processor writing to a datum. We assume that the datum of a coherence miss resides dirty in some processor's cache. In a general network, a NUMA needs three hops for most of the coherence misses, that is, when the datum resides dirty in a node other than the home, and the requesting node is not the home node. For $1/n$ of the coherence misses, the home and the requesting node are identical and only two hops are needed. For another $1/n$ misses, the dirty datum resides in the home node, resulting in another two-hop component, as can be seen in Equation 4.3 for $L_{coh-NUMA}$. For all cases, one directory lookup in the home node, T_{dir}, and one access to the cache with the dirty copy, T_{cache}, are also needed,[4] as well as two accesses on the local and one on the remote bus, $3 * T_{lb}$.

An example of a remote access of a COMA based on a general network can be found in Figure 4.3. $L_{coh-COMA}$ in a general network is quite similar to $L_{coh-NUMA}$, but one extra T_{dir} is required at the requesting node before the need for a remote request is detected. The $L_{coh-NUMA}$ differences are shown as dashed lines in the figure.

[3]Given the technology considered in Subsection 4.1.6, $T_{dir} \leq T_{lb} + T_{mem}$.

[4]The directory lookup and the cache access can be overlapped if the cache with the dirty copy is in the home, i.e., for 1/n of the cases. This has a negligible impact on performance, and has been left out of this equation.

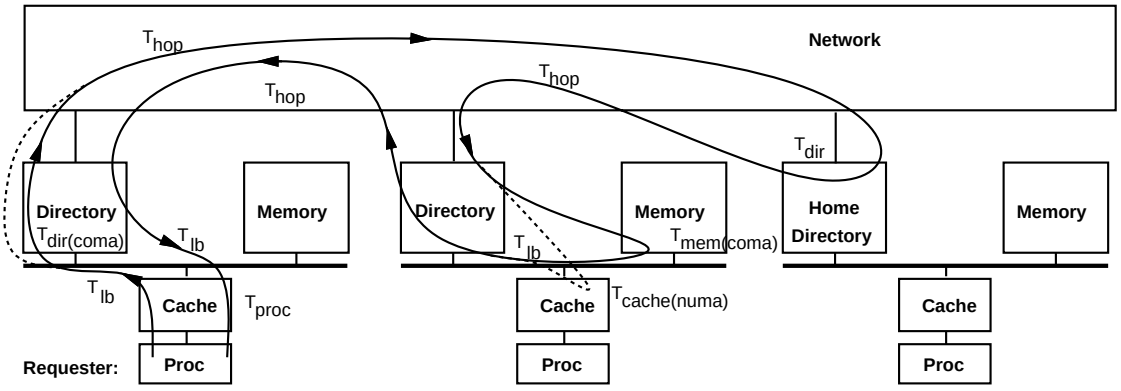


Figure 4.3: An illustration of a COMA remote read access, when the requested datum resides in a node other than the home node. The differences to a NUMA "dirty" coherence access are shown as dashed lines.

Equation 4.4 sums up the latency for $L_{coh-COMA}$ on a general network. All capacity misses are assumed to hit in the AM. A hit in the AM simply takes the memory access time, T_{mem}, the local bus time, T_{lb}, and the processor startup time, T_{proc}. The directory lookup is performed in parallel with the data access, the local bus action, and part of the processor start time, and its latency is hidden.[5]

NUMA and COMA based on general-networks

$$L_{cap-NUMA} = \frac{n-1}{n}(2T_{hop} + 2T_{lb}) + T_{mem} + T_{lb} + T_{proc} \quad (4.2)$$

$$L_{coh-NUMA} = \frac{n-2}{n}3T_{hop} + \frac{2}{n}2T_{hop} + T_{dir} + T_{cache} + 3T_{lb} + T_{proc} \quad (4.3)$$

$$L_{coh-COMA} = \frac{n-2}{n}3T_{hop} + \frac{2}{n}2T_{hop} + 2T_{dir} + T_{cache} + 3T_{lb} + T_{proc} \quad (4.4)$$

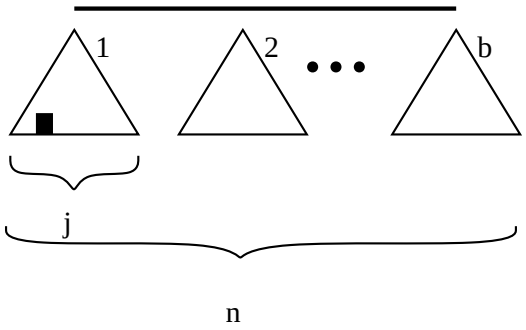
$$L_{cap-COMA} = T_{mem} + T_{lb} + T_{proc} \quad (4.5)$$

4.1.4 Calculating Communication Latency T_{hop}

T_{link} corresponds to the communication delay between two adjacent network nodes. It includes the latency of the link and the fall-through latency of the node. The average communication latency between two random processor nodes in a two-dimensional mesh, called one hop, is $2/3\sqrt{n} * T_{link}$, assuming random distribution of traffic. In a three-dimensional mesh, the average time per hop is about $\sqrt[3]{n}\,T_{link}$.

[5] This is discussed in detail in Appendix C.

In a serial link, the information is divided into small sequential packages. On a mesh, the first of these packages contains the destination, e.g., X and Y coordinates. Next comes the transaction code and address of the datum. Before the address and transaction code can be used to perform a directory lookup, an extra delay, T_{rec}, has to be accounted for. T_{rec} includes the time to receive the necessary information and to synchronize with the node. The total communication latency for 2- and 3-dimensional meshes is given in Equations 4.6 and 4.7.



$$n = j * b$$

$$P(\text{same top cluster}) = \frac{j-1}{j*b} \approx \frac{1}{b}$$

Figure 4.4: A request from a processor (the black box) in a subsystem of j processors has to pass the top with a probability of $(b-1)/b$. This is an approximation valid for large values of j.

A general network can also be a hierarchy of links or buses, similar to Figure 3.3.b and 3.3.c, but without the directories between the levels. The number of levels in a hierarchical architecture, l, is $\log_{bmax}(n)$, where $bmax$ is the maximum branching factor. We assume that the lower-level buses utilize the maximum branching factor. The top bus has the remainder branching, b, calculated as $n/(bmax^{(l-1)})$. Assuming a random distribution of communication in a hierarchy, approximately $1/b$ of the hops are within the same top cluster as shown in Figure 4.4. We only consider locality explored at the highest level in the hierarchy.

The mean value for inter-node latency, T_{hop}, is given in Equation 4.8. The equation for a hierarchy of buses (Equation 4.9) is similar to that of a hierarchy of links. T_{bus} is the time it takes to arbitrate for a bus and to transfer transaction code and address. Unlike the link hierarchy, there is no top to pass through, thus removing one T_{bus}. Within the same bus, all traffic is assumed to be word-wide and synchronous, removing the need for T_{rec}. A two-level bus hierarchy can be mapped nicely on to three-dimensional space.[6] Communication between levels two, three, and higher, however, is carried on links—similar to the link-based case. The equation for more than two levels (Equation 4.10) is therefore reminiscent of the equation for the links, but with a constant delay for the lower bus levels of 4 T_{bus}.

[6] As discussed further in Chapter 6, Figure 6.5.

Hops:

$$2\text{-}dim.mesh : T_{hop} = \frac{2}{3}\sqrt{n}\,T_{link} + T_{rec} \quad (4.6)$$

$$3\text{-}dim.mesh : T_{hop} = \sqrt[3]{n}\,T_{link} + T_{rec} \quad (4.7)$$

$$link\text{-}hier. : T_{hop} = \frac{b-1}{b} 2\,l\,T_{link} + \frac{1}{b} 2(l-1) T_{link} + T_{rec} \quad (4.8)$$

$$bus\text{-}hier. : T_{hop} = \frac{b-1}{b} 2\,l\,T_{bus} + \frac{1}{b} 2(l-1) T_{bus} - T_{bus} \quad (4.9)$$

$$bus\text{-}hier_{(l>2)} : T_{hop} = \frac{b-1}{b}(l-2) 2\,T_{link} + \frac{1}{b}(l-3) 2\,T_{link} + + 4\,T_{bus} + T_{rec} \quad (4.10)$$

The NUMA miss penalty for all topologies can be derived by inserting the T_{hop} of the corresponding topology into Equations 4.2 and 4.3. The COMA latencies for a protocol based on a general network can also be derived by inserting the corresponding T_{hop} into equation 4.4.

4.1.5 COMA with Hierarchical Protocol

Miss penalties for hierarchical COMAs with a hierarchical protocol are different, because of the directory structure. The hierarchical-protocol COMA assumes a directory memory between each level, to guide a request to a data value. The link-based implementation also has a directory at the top. So, when traveling between the levels in the hierarchy and through the top, directory lookups are needed. As mentioned before, a COMA also has to pay for one T_{dir} when leaving the requesting node, and an additional T_{dir} for each directory passed on its path to the datum. Before a lookup can be performed in the directory between the levels in the hierarchy, the address and transaction code must be received from the link, and the asynchronous message synchronized. All of this is included in the T_{dlevel} component together with the link delay between levels. The data reply transaction "knows" who requested the data, and does not need to pay for any directory lookups in its critical latency path when returning to the requesting node[7]—the *return to sender* feature. Assuming a random distribution of communication, about $1/b$ of the hops are in the same subsystem underneath the top, reflected in Equation 4.11. The effects of locality further down in the hierarchy are regarded as negligible for this discussion.

[7] Directory lookups and state changes can still be performed when forwarding data.

COMA with a hierarchical protocol:

$$link: L_{coh-COMA} = \frac{b-1}{b} 2\,l(T_{dir} + T_{dlevel} + T_{link}) + \frac{1}{b}2(l-1)(T_{dir} + T_{dlevel} + T_{link}) + \\ + T_{cache} + 3T_{lb} + T_{proc} \quad (4.11)$$

$$bus: L_{coh-COMA} = \frac{b-1}{b}(2\,l * T_{dir} + 4l\,T_{bus}) + \frac{1}{b}(2(l-1)T_{dir} + 4(l-1)T_{bus}) - 2T_{bus} + \\ + T_{cache} + 3T_{lb} + T_{proc} \quad (4.12)$$

$$bus: L_{coh-COMA\,(l>2)} = \frac{b-1}{b} 2\,(l-2)(T_{dir} + T_{dlevel} + T_{link}) + \frac{1}{b}2(l-3)(T_{dir} + T_{dlevel} + T_{link}) + \\ 4T_{dir} + 8T_{bus} + T_{cache} + 3T_{lb} + T_{proc} \quad (4.13)$$

A hierarchy of buses has an equation similar to that of a hierarchy of links (Equation 4.12), although there is no top to pass through, thus removing one T_{dir}. However, the receiving side needs to snoop in its directory before it is notified to service the request, adding a new T_{dir} again. Two T_{link} are then removed from Equation 4.12. The terms for T_{rec} are also removed, since both transaction code and address are assumed to be latched in the directory at the end of T_{bus}, why no extra synchronization latency is needed in this equation. For the higher levels, the bus-based and link-based COMAs account for the same delays, similar to the equations for the hops.

The extra delay for transferring the data part of a reply has only been accounted for in the receiving nodes of all architectures, included in the time T_{proc}. All along the path from the remote node to the receiving node, pipelining of the data part following the command part is assumed. Also, no extra time for error correction has been included in any of the equations. We assume that error detection is performed in parallel by hardware, and that recovery is handled by software, possibly using some check pointing scheme.

4.1.6 Choosing Technology Numbers

There is a large variety of parameters to choose from when describing the latency of memories and communication. In order to get a consistent set of parameters, we have adopted the technology numbers used in prototype projects where available, e.g., the link-based DASH [Len91] and our own experience with the bus-based DDM prototype. All numbers are translated to a 33 MHz processor clock, which is the clock rate of the DASH prototype. For the study of tomorrow, we use the technology assumed by Lenoski in his design of the future TeraDASH [Len91], where applicable.

TeraDASH assumes a DRAM speed of 50 ns and a clock frequency of 100 MHz. This technology i just around the corner. We also try to estimate technology available in the future comparable to 300 MHz. The numbers are presented in Table 4.1.

Parameters	Latency	Source	Brief Explanation
Parameters of 33 MHz Technology			
$T_{cache-hit}$	1	DASH	30 ns, the unit of latency
T_{lb}	6	DASH	arb for local bus(2), send cmd(2), send 1st word(2)
T_{mem}	6	DASH	get 1st word(6)
T_{proc}	4	DASH	get remaining words(2), restart processor(2)
T_{link}	2.5	DASH	fall through(45ns), network(35ns) 5'
T_{dir}	10	DDM	arb(1), get old state(6), decode, to FIFO(3)
T_{rec}	4	-	receive additional cmd info from a link(2), sync(2)
T_{bus}	4	DDM	arbitrate(2), transfer cmd and address(2)
T_{dlevel}	5	-	network(35ns), receive additional info(2), sync(2)
Parameters of 100 MHz Technology			
$T_{cache-hit}$	1	TDASH	10 ns, the unit of latency
T_{lb}	6	TDASH	arb for local bus(2), send cmd(2), send 1st word(2)
T_{mem}	9	TDASH	get 1st word(9)
T_{proc}	6	TDASH	get remaining words(4), restart processor(2)
T_{link}	2.5	TDASH	fall through(20ns), network(4ns)
$T_{link-hier}$	3.5	-	fall through(20ns), network(15ns)
T_{dir}	15	DDM+	arb(2), get old state(9), decode, to FIFO(4)
T_{rec}	4	-	receive additional cmd info from a link(2), sync(2)
$T_{rec-hier}$	5	-	receive additional cmd info from a link(3), sync(2)
T_{bus}	6	DDM+	arbitrate(2), transfer cmd and address(4)
T_{dlevel}	6.5	-	network(15ns), receive additional info(3), sync(2)
Parameters of 300 MHz Technology			
$T_{cache-hit}$	1	-	3.3 ns, the unit of latency
T_{lb}	15	MR	arb for local bus(5), send cmd(5), send 1st word(5)
T_{mem}	12	MR	get 1st word(12)
T_{proc}	7	-	get remaining words(5), restart processor(2)
T_{link}	5	-	fall through(10ns), network(4ns)
$T_{link-hier}$	8	-	fall through(10ns), network(15ns)
T_{dir}	18	-	arb(2), get old state(12), decode, to FIFO(4)
T_{rec}	4	-	receive additional cmd info from a link(2), sync(2)
$T_{rec-hier}$	5	-	receive additional cmd info from a link(3), sync(2)
T_{bus}	13	MR	arbitrate(5), transfer cmd and address(8)
T_{dlevel}	10	-	network(15ns), receive additional info(3), sync(2)

Table 4.1: The characteristics of two technologies. Sources are: DASH, the DASH implementation, TDASH, the TeraDASH proposal, and DDM, the DDM prototype project. MR numbers are from the Microprocessor Report magazine. The remaining numbers are estimated for this study.

4.2 Which Architecture is Better and When?

Using the above equations, we can calculate the average latency for coherence misses and capacity misses for both NUMA and COMA architectures. There is still not any obvious answer to the question of whether COMA is better, since we are lacking information about the applications. However, for a given network we can calculate the combinations of capacity and coherence miss rate for which both perform the same by making both architectures' miss-penalty equations equal (Equation 4.1). For all applications with a higher ratio of capacity misses, COMA can be expected to perform better.

COMA and NUMA perform the same:

$$\begin{aligned} miss\,penalty_{COMA} &= miss\,penalty_{NUMA} \\ (misses_{coh} * L_{coh} + misses_{cap} * L_{cap})_{COMA} &= (misses_{coh} * L_{coh} + misses_{cap} * L_{cap})_{NUMA} \\ misses_{cap} &= \frac{L_{cohNUMA} - L_{cohCOMA}}{L_{capCOMA} - L_{capNUMA}} * misses_{coh} \end{aligned} \quad (4.14)$$

This function of COMA and NUMA performing the same has been plotted in Figure 4.5 for 256-processors systems built with four different networks. The slower the network is, the lower is the curve, i.e., less capacity misses per coherence misses are required in order to make COMA better than NUMA. This effect can also be seen in Figure 4.5, where the number of processors is varied for a network built of a 3D mesh.

From the graphs we can see that the importance of COMA increases with the number of processors, i.e, when the network latency gets longer. The explanation for this can be seen in the formulas for capacity and coherence misses. For COMA the penalty for capacity misses is a constant, while the penalty paid by NUMA keeps increasing with n. The penalty for coherence misses keeps increasing with the number of processors both for NUMA and COMA, but their absolute difference is constant (only one T_{dir} differ). So, the overhead paid by COMA for locating the requested item is decreased relative to the coherence miss penalty.

Similarly, the importance of COMA is larger for network topologies with longer latencies. It also seems that COMA's importance increase with newer technologies, where the ability to remove remote accesses is of increased importance, since the speed gap between local communication and global communication is widened.

4.2.1 SPLASH Data

Classification of applications by cache-miss categories can be done through simulation. To our knowledge, the only such classification of multiprocessor applications

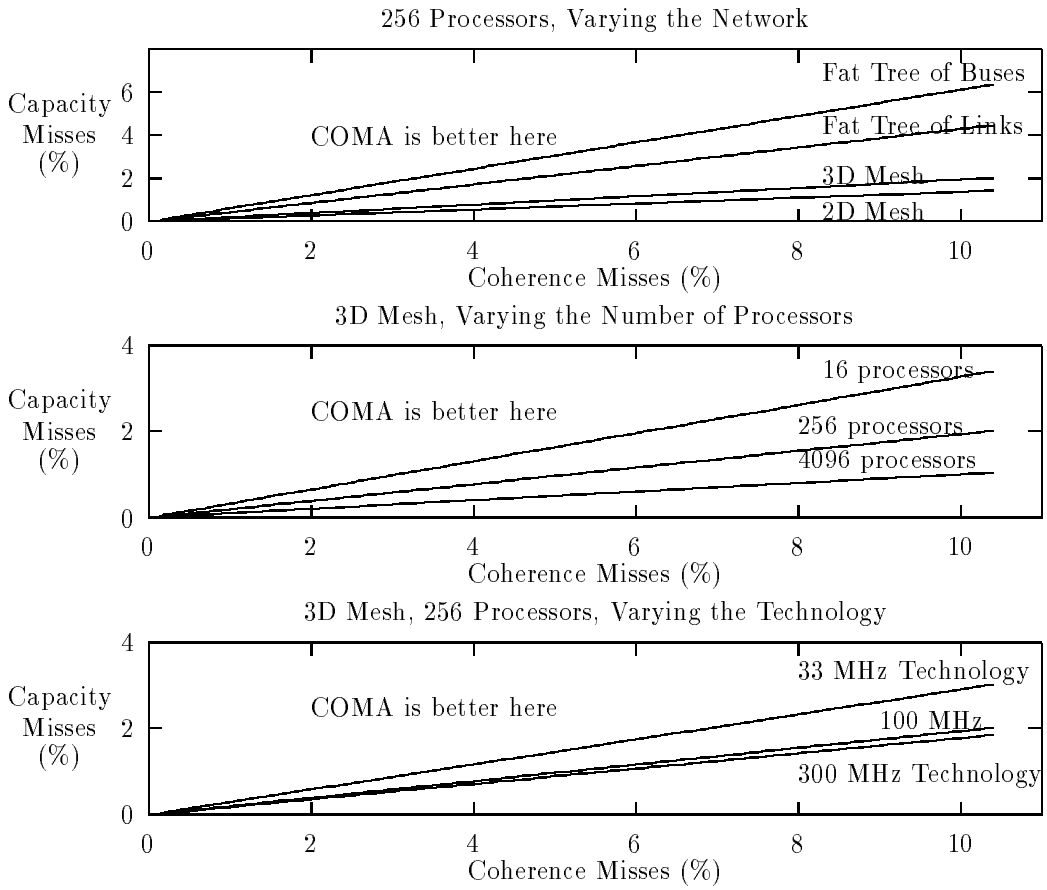


Figure 4.5: Showing where COMA and NUMA perform equally.

published is a study of SPLASH by Gupta et al. [GJS92], presented in Table 4.2. SPLASH contains the following programs:
MP3D, particle-based wind tunnel simulator;
PTHOR, logic simulator;
Locus Route, standard cell router;
Water, N-body molecular dynamics simulator;
Cholesky, factorizing sparse matrices;
LU, LU decomposition program;
Barnes-Hut, N-body problem solver; and,
Ocean, ocean basin simulator.

Using the SPLASH suite for quantitative comparison studies is strongly discouraged by its authors [SWG91]. Lacking other numbers, we hereby violate their advice, confess our sins, and ask the reader to keep this in mind. The processor caches in the Gupta et al. study are sized 4 kbytes, 16 kbytes, and 64 kbytes. The small cache sizes were chosen to compensate for the small data sets used in the simulation—often orders of magnitude smaller than realistically sized data sets (with the exception of PTHOR). Note that for five of the applications the data set per

processor is actually smaller than 64 kbytes. The read misses in the processor caches are divided into two categories: capacity misses and coherence misses. For the reported numbers, conflict misses and a small amount of compulsory misses are included in the capacity miss category. Compulsory misses were measured to be 0.9 percent for Cholesky, 0.26 percent for LocusRoute, and less than 0.1 percent for the remaining applications. We have moved one percentage point from capacity misses to coherence miss for Choleksy, since compulsory misses do not hit in the attraction memory. Only accesses to global shared data are included in the collection of cache statistics.The SPLASH numbers are compared to equality graphs in Figure 4.6 where it can be seen that the performance of NUMA is only comparable with COMA for a couple of applications even when caches sized 64 kbytes are used.

Applications: / **Data Set Size/Proc:**	**Applications and Data Set Size Per Node** MP3D 34k	PTHOR 199k	Locus 77k	Water 13k	Chol 62k	LU 40k	B-H 25k	Ocean 151k
Miss Type	**Cache Misses Divided into Categories (%)**							
Coherence	10	9	3	3	5	2	1	2
Capacity (4k)	7	12	11	10	22	21	41	52
	Reducing Local Cache+Memory Misses of a NUMA by Page Placement and Migration Strategies							
Capacity(Random 4k)	6	10	10	9	21	20	38	48
Capacity(Init. 4k)					21	15		9
Capacity(Migr. 4k)	5	8	6	6	5	16	37	9
	Reducing Capacity Misses by Increasing the Cache Size							
Capacity (16k)	6	7	6	5	6	8	20	28
Capacity (64k)	3	4	3	2	3	3	5	15

Table 4.2: The read misses of nonprivate data on a 16-processor machine divided into different categories by Gupta et al. [GJS92]. Figures are in percent. The coherence miss rate for Cholesky is here increased by one percent, and the capacity miss rate decreased by one percent to compensate for compulsory misses.

The reported numbers above rely on a random replacement of shared pages in a NUMA, while private pages are placed in the node of its processor. This is a common placement strategy for a NUMA, but the nature of the non-uniform memory allows more optimal distributions of data over nodes. The behavior of different page placing and migration strategies reported by Gupta et al. [GJS92] are also reported in Table 4.2. The migration strategy used assumes extra hardware support by associating N counters with each page, given N processing nodes [BGW89]. A remote access to a page increments the corresponding counter and decrements a randomly chosen counter. When a counter reaches a value corresponding to some threshold, the page is migrated. The migration capability results in COMA-like behavior for large items, but requires substantial hardware support. Read/write pages cannot, however, be duplicated, and can only exist in one copy. The effect of initial place-

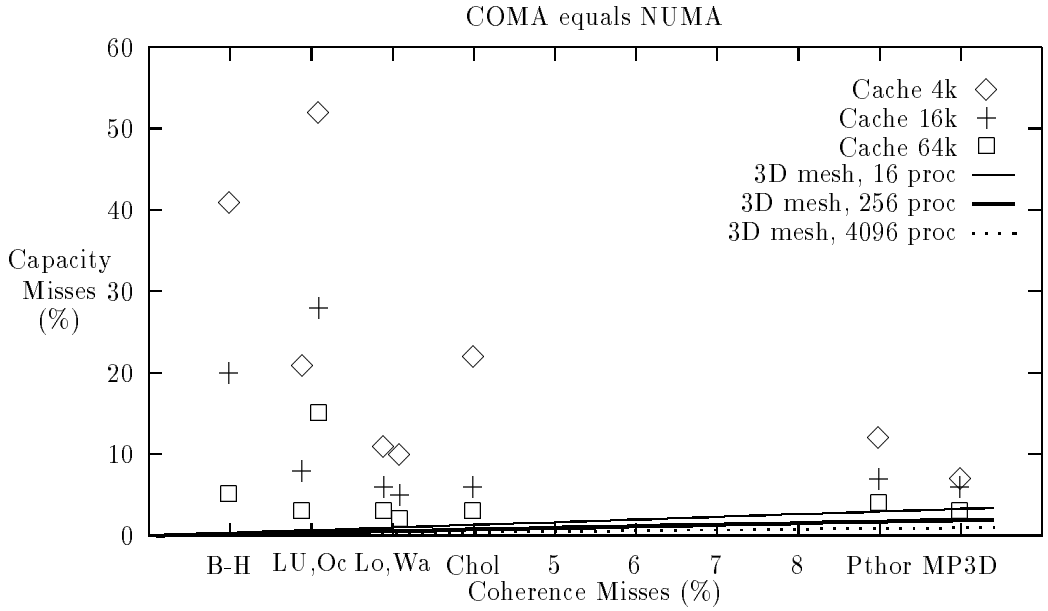


Figure 4.6: Comparing the SPLASH suite to the equality graph: COMA=NUMA. Each application is reported for three different cache sizes, with varying capacity misses as a result. The coherence misses are not affected by the cache size, which is why the applications names (initials) are marked along the coherence axis.

ment has also been studied. Here, the same algorithm as for page migration is used, but each page is only moved once. In Figure 4.7 it can be seen that Cholesky, which has a dynamic allocation of work, suits the migration strategy. Ocean gets most of its improvement from the initial placement. Still, most applications are far from the equality lines.

The statistics were collected from a simulation of 16 processors. A fully mapped directory scheme was used for the NUMA case. In this study we sometimes use the same miss statistics to calculate performance even for larger numbers of processors. This implies that the problem size increases with the number of processors, so that the data set per processor stays the same. We expect this to be a fair estimate for some applications, but not for others. For statically scheduled programs, like MP3D, increasing the problem size with the number of processors should have no effect on the calculation part of the program, while the synchronization will be affected somewhat by the increased number of processors. For dynamically scheduled programs with limited parallelism, like PTHOR, the miss characteristics will be different when the problem size is increased together with the number of processors.

Using a fully mapped directory scheme when the number of nodes increases to the hundreds is also clearly impractical. The effects of a limited pointer scheme should also have been taken into account for directory-based implementations [CFKA90].

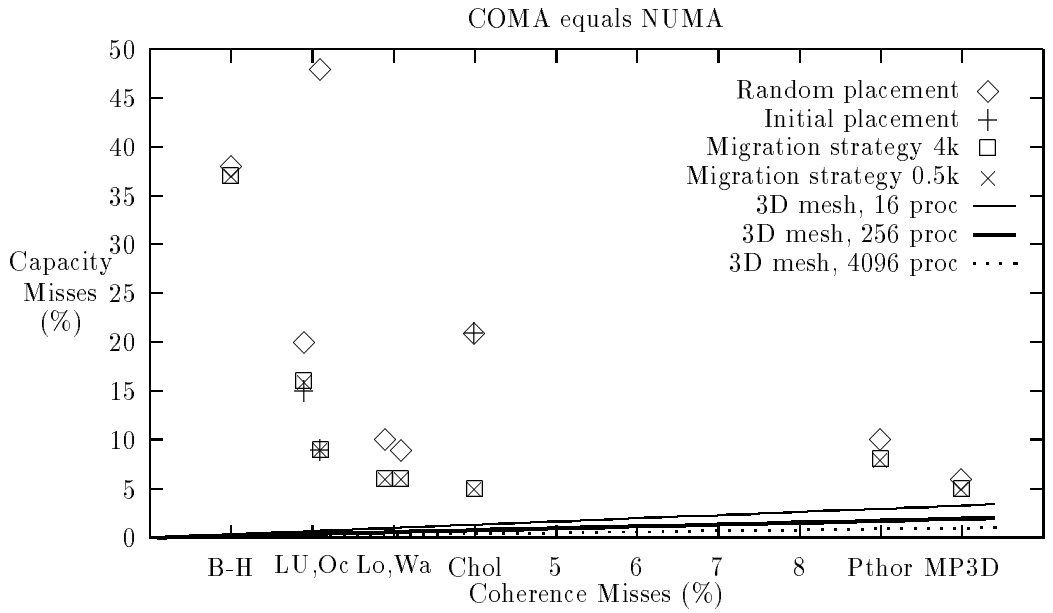


Figure 4.7: Comparing the SPLASH suite to the equality graph when migration strategies are applied.

This is especially important for the COMA architectures, where replication is more common, and thus the probability of many processors sharing a datum might be higher than for a NUMA.

4.3 Comparative Performance For Large Multiprocessors

Earlier we calculated the miss penalty for accesses to global shared data as the average number of processor cycles a processor spends on misses. We can calculate the average number of processor cycles required for accessing shared global data, including both hits and misses, by also adding the average time the processor spends on hits in its cache for accesses to global data.

Based on the miss-penalty equations, the mean time for a global data access can be calculated as:

$$mean\,global\,access\,time = miss\,penalty + fraction\,hits * cache\,access\,time.$$

The mean access time for the different applications can be found in Table 4.3. The column to the far left defines the ten different architectures compared in this

Architecture		L_{cap}/ L_{coh}	Applications and Data Set Size Per Node: MP3D DS:34k	Pthor 199k	Locus 77k	Water 13k	Chol 62k	LU 40k	B-H 25k	Ocean 151k	Avg all	Avg 4
33 MHz technology, cache size 4 kbytes, bmax = 16												
n = 256 processors												
N	mesh2	89/125	20	23	13	14	25	21	38	50	**25**	**18**
U	mesh3	68/ 92	15	17	10	10	19	16	29	38	**19**	**14**
M	h-link	55/ 74	12	14	8	9	16	13	24	31	**16**	**11**
A	h-bus	51/ 67	11	13	8	8	15	12	22	29	**15**	**11**
C	mesh2	16/135	15	15	5	7	10	5	8	13	**10**	**7**
O	mesh3	16/102	12	12	5	6	8	5	8	12	**8**	**6**
M	h-link	16/ 89	11	11	4	5	8	5	8	11	**8**	**6**
A	h-bus	16/ 82	10	10	4	5	8	5	8	11	**8**	**5**
Dir:	h-link	16/ 91	11	11	4	5	8	5	8	11	**8**	**6**
Dir:	h-bus	16/ 89	11	11	4	5	8	5	8	11	**8**	**6**
n = 4096 processors												
N	mesh2	249/365	55	64	36	37	70	57	106	141	**71**	**50**
U	mesh3	116/165	25	30	17	17	33	27	50	66	**33**	**23**
M	h-link	65/ 89	14	17	10	10	19	15	28	37	**19**	**13**
A	h-bus	69/ 95	15	18	10	11	20	16	30	39	**20**	**14**
C	mesh2	16/375	39	36	10	14	19	8	11	20	**20**	**13**
O	mesh3	16/175	19	18	6	8	11	6	9	14	**11**	**8**
M	h-link	16/104	12	12	5	6	8	5	8	12	**9**	**6**
A	h-bus	16/110	13	13	5	6	9	5	8	12	**9**	**6**
Dir:	h-link	16/126	15	14	5	6	9	5	8	13	**9**	**7**
Dir:	h-bus	16/128	15	14	5	6	9	5	8	13	**10**	**7**
100 MHz technology, cache size 4 kbytes, bmax = 16												
n = 256 processors												
N	mesh2	94/132	21	24	14	14	27	22	40	53	**27**	**19**
U	mesh3	73/ 99	16	18	11	11	21	17	31	41	**21**	**15**
M	h-link	70/ 96	15	18	10	11	20	16	30	40	**20**	**14**
A	h-bus	67/ 92	15	17	10	10	19	16	29	38	**19**	**14**
C	mesh2	21/147	17	17	6	7	11	7	11	16	**11**	**8**
O	mesh3	21/114	14	14	5	6	10	6	10	15	**10**	**7**
M	h-link	21/119	14	14	6	7	10	6	10	15	**10**	**7**
A	h-bus	21/115	14	14	5	6	10	6	10	15	**10**	**7**
Dir:	h-link	21/122	14	14	6	7	10	6	10	15	**10**	**7**
Dir:	h-bus	21/124	15	14	6	7	10	6	10	15	**10**	**7**
n = 4096 processors												
N	mesh2	254/372	56	65	36	37	72	58	109	144	**72**	**51**
U	mesh3	121/172	26	31	18	18	34	28	52	69	**34**	**24**
M	h-link	84/117	18	21	12	13	24	20	36	48	**24**	**17**
A	h-bus	94/132	21	24	14	14	27	22	40	53	**27**	**19**
C	mesh2	21/387	41	38	11	15	21	9	13	23	**21**	**14**
O	mesh3	21/187	21	20	7	9	13	7	11	17	**13**	**9**
M	h-link	21/140	16	16	6	7	11	7	11	16	**11**	**8**
A	h-bus	21/155	18	17	6	8	12	7	11	16	**12**	**8**
Dir:	h-link	21/172	19	19	7	8	12	7	11	17	**12**	**8**
Dir:	h-bus	21/180	20	19	7	8	13	7	11	17	**13**	**9**

Table 4.3: Studying the variation in the average number of processor cycles per global data access.

study. "Dir" marks COMAs implemented with a hierarchical protocol and the directories embedded in the hierarchy. The next column states their average delay for capacity and coherence misses. The rest of the numbers describe the average access time to global data. Applications to the left are the ones best suited for the NUMA architecture, since their ratios of coherence misses are the highest. As can be seen in the table, NUMA has a slightly shorter latency for coherence misses than

COMA, while COMA has a much shorter average delay for capacity misses. For the application furthest to the left, MP3D, coherence misses are more common than capacity misses. For MP3D, the average delay for accesses to global data for NUMA and COMA seems to be quite even. The further we move to the right, the more it favors COMA. The two rightmost columns report the mean value for all applications (Avg). The mean value for the middle four applications (Avg4) is also presented, cutting away the rightmost two applications with an unusually high capacity miss rate, and the leftmost two application with an unusually high coherence miss rate. All numbers are presented without decimals, to remind the reader that they are estimates.

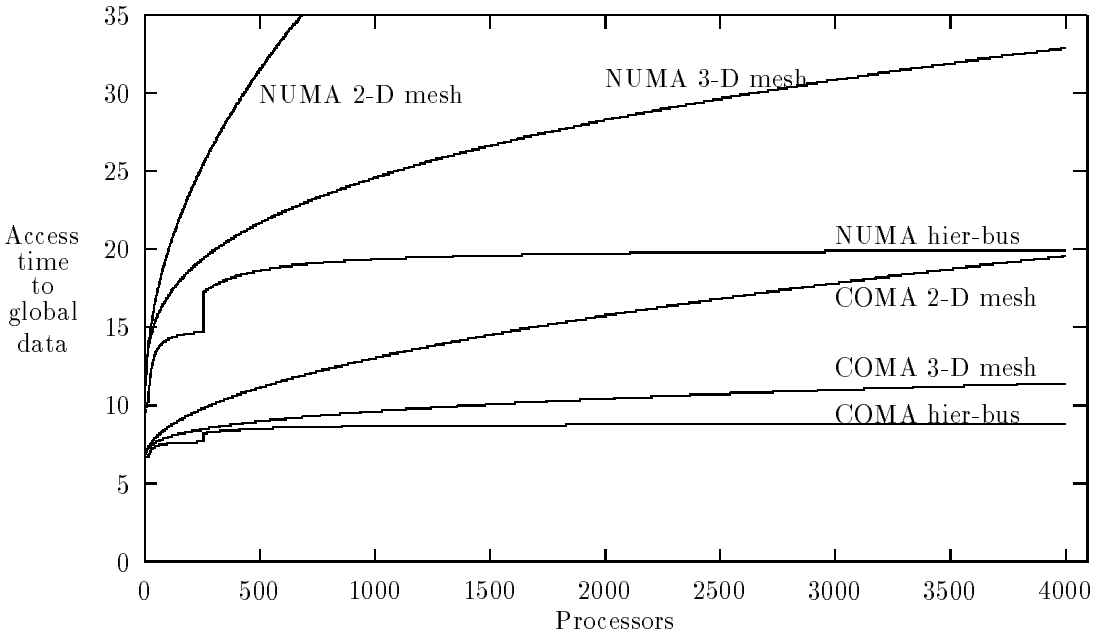


Figure 4.8: The average access time to global data for six of the studied architectures, varying the number of processors. The hierarchies of buses and links are similar in performance. The link-based architectures are left out for clarity. The curves are for 33 MHz technology and 4 kbytes caches.

The effect of varying the number of processors for a selection of the architectures can be seen in Figure 4.8. It is clear that the architectures are divided into two performance groups—COMAs and NUMAs. Within the groups, the differences between the architectures are small. In fact, the performances of the COMAs are so close that some of them have been left out of the figure for clarity. In Table 4.3 it is easier to identify the differences. From the results it appears that a COMA with a bus is the fastest for a small number of processors, while for large machines in the 100 MHz technology the 3-D mesh COMA has a comparable performance. However, this method is not sophisticated enough to distinguish such small differences. Before drawing general conclusions in this matter, a more detailed analysis

must be performed with a larger and more general set of programs. Many aspects important for the choice between the mesh and the hierarchy are not modeled here: contention, communication locality, write delay, and limited pointers. Also, if the performance is so close, the choice of COMA architecture may involve other aspects, such as which is the easiest/cheapest to implement.

A maximum branching factor of 16 is used for the hierarchical architectures in Figure 4.8. A knee can be observed for those architectures at 16 processors. The knee is largest for the hierarchical NUMA, since the increase in latency has a larger impact on NUMAs than COMAs. A larger discontinuity than the one shown here can be expected for real architectures. In all equations real numbers are used for the variables, resulting in branching factors of 1.25 at the top for 20 processors, instead of 2. Also, the latency for T_{hop} in the meshes are calculated using real numbers. Thus, a 2-D mesh of 20 processors is assumed to have the topology 4.47×4.47, instead of 4×5. This allows for simpler equations and smother curves. In this study we are interested in tendencies when the number of processors grow, rather than the absolute performance at a specific number of processors.

The relative difference between NUMA and COMA increases with the number of processors. The explanation for this can be seen in the column for capacity and coherence misses. For COMA the penalty for capacity misses is a constant, 16 cycles, while the penalty payed by NUMA keeps increasing. The penalty for coherence misses keeps increasing with the number of processors both for NUMA and COMA, but their absolute difference is constant (only one T_{dir} differ). So, the overhead payed by COMA for locating the requested item is decreased relative to the coherence miss penalty.

The performances of the hierarchical architectures depend on the number of levels in the hierarchy. Figure 4.9 shows how the average delay to global data is related to the branching factor. Since the number of levels is $\log_{bmax}(n)$, it is not surprising to see that the overhead levels off with the number of processors. As a comparison, the delays for the 2-D COMA and 3-D COMA are also included.

Figure 4.10 shows how performance varies for one NUMA and one COMA architecture with variably sized primary caches. It is not surprising to note that the differences between NUMA and COMA get smaller as larger caches are used. A cache size of 64 kbytes is larger than the data set per processor used for five of the eight applications. When the caches of the NUMA get larger than the data set, it adopts a COMA-like behavior.

In the performance graphs it looks like hierarchical COMAs build as fat trees have the biggest performance advantage for large systems. There are four alternative implementations of such available: with links or buses, and as a general network, or with directories between the layers of the hierarchy. We have avoided plotting all the hierarchical COMAs in the graphs, since their performance is very close.

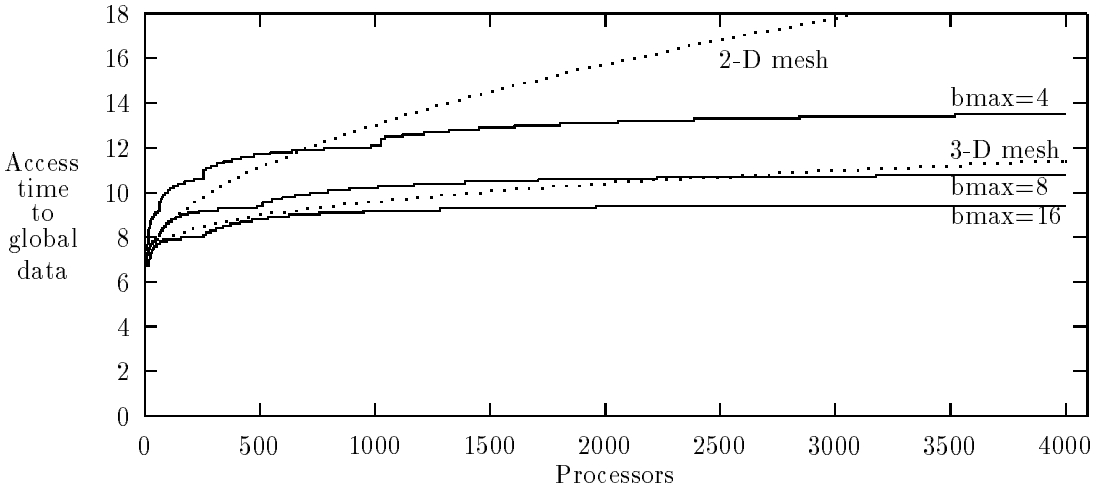


Figure 4.9: The access time to global data when varying the branching factor of a COMA architecture implemented with links. By comparison, the data for a 2-D mesh COMA (top) and a 3-D mesh COMA (bottom) appear with dotted lines.

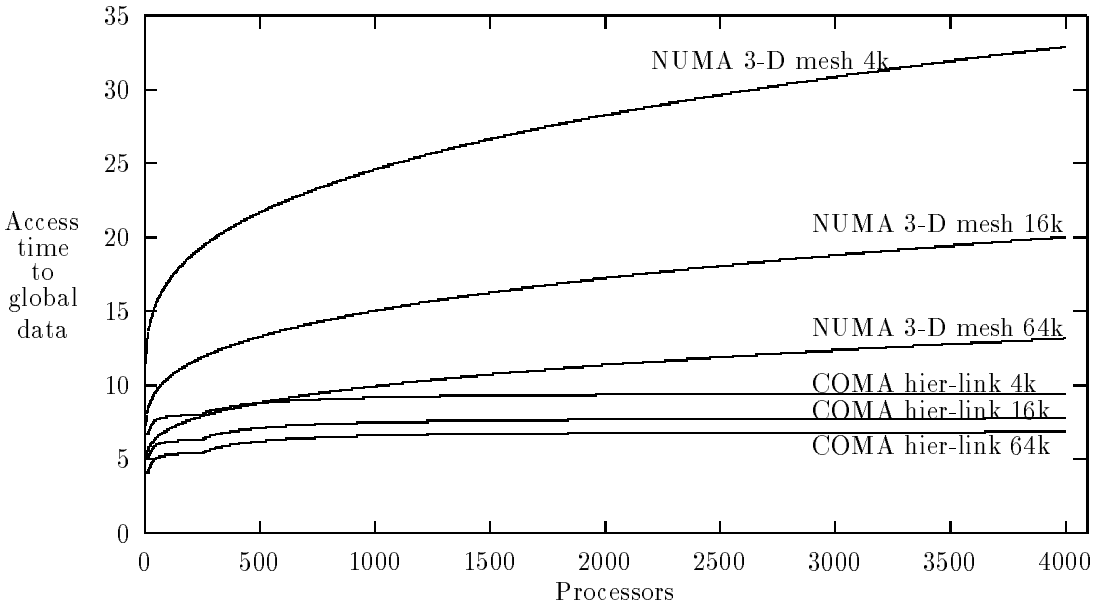


Figure 4.10: The access time to global data when varying the cache size.

Figure 4.11 shows a magnified performance comparison between the four. As can be seen, the performance is similar for all four implementations for small systems, while the general network solutions have an advantage for larger systems.

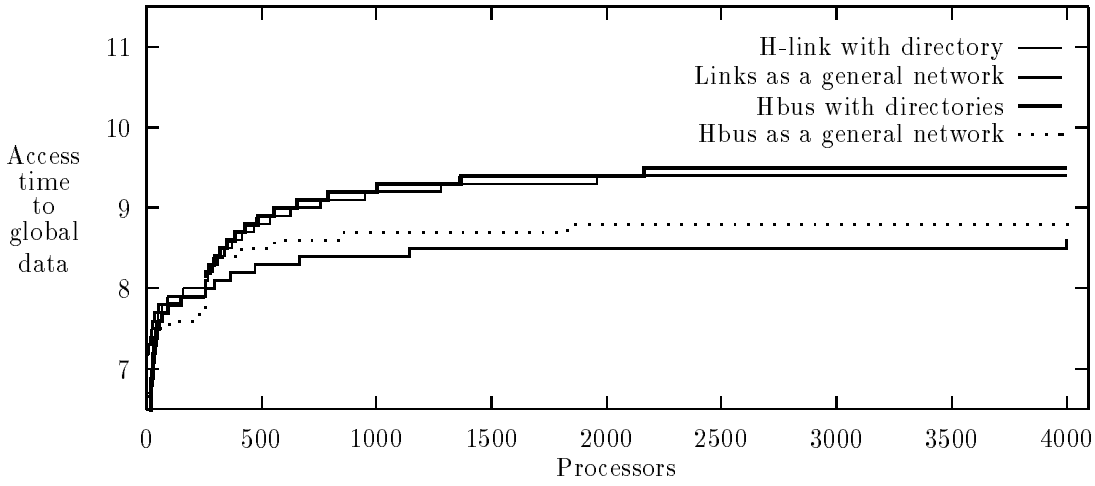


Figure 4.11: Comparing the four different strategies of hierarchical COMA implementation.

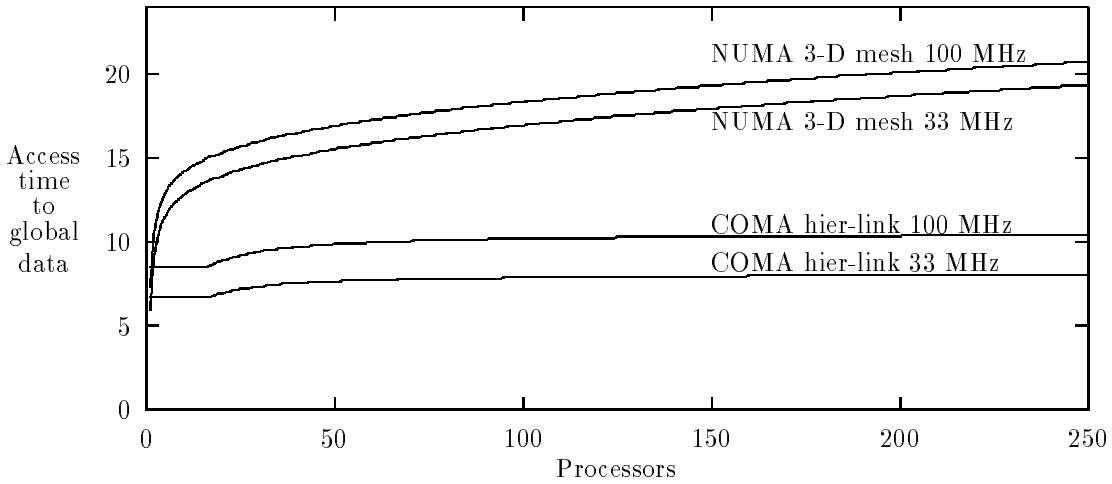


Figure 4.12: The access time to global data when varying the technology.

Figure 4.12 shows how the performance varies for one NUMA and one COMA when moving from 33 MHz technology to 100 MHz technology. Both suffer from a latency of more clock cycles for the faster technology.

The results from different parameter settings can be found in Appendix A.

The effect of the mean global access time on performance depends on how often the applications make global accesses. If we assume that all other instructions are performed in one cycle, assuming 100 percent hit rate in the cache for private data,

the average number of cycles per instruction can be calculated as:

$$CPI = (1 - global\,ratio) * 1 + global\,ratio * mean\,global\,access\,time.$$

Global ratio is the number of instructions performing a global access divided by the total number of instructions executed. The global ratio for MP3D is 0.16, for Water it is 0.07, and for Cholesky it is 0.20 [BS92]. As can be seen from Table 4.3, for Cholesky and MP3D the global access time is the dominant part of the CPI equation, while for Water it is a minor part. For all applications, the architecture with the fastest global access time performs best, however, since we assume that the busy times for the processors are identical.

4.4 Comparing Real Implementations

The analytical models have described ideal instances of a large variety of possible COMA and NUMA implementations. In this section we try to model existing machines, with all their limitations and drawbacks. We conclude by comparing these models with the ideal models.

4.4.1 KSR1

Recently, a commercial COMA architecture has been announced by Kendall Square Research [BFKR92]. The KSR1 architecture is different from the DDM in many ways. It uses large items of 128 bytes and suffers from a much larger remote-access delay. Its processors run at 20 MHz. The network consists of a ring-based hierarchical structure with a branching factor of 32 at each level. Its processor data caches, sized 256 kbytes, are accessed in two cycles. An access to its AM takes 18 cycles. A remote access satisfied by the lowest ring yields a delay of 126 cycles at 20 MHz, while an access climbing yet another level in the hierarchy takes 600 cycles [Dun92]. We have been unable to obtain any information from KSR, so the correct technology parameters for our model are lacking. However, translating available information to 33 MHz[8] allows for a rough estimate of KSR1's performance (Table 4.4).

We model smaller caches to compensate for the small data set, 16 kbytes or 64 kbytes. We assume the same hit rate as in Gupta's study, even though item sizes differ.

[8]By multiplying all latencies by 1.65.

Parameter	Value in this comparison	Explanation
KSR1, $n = 64$ processors		
1st level cache	16k/64k	The actual size is 256 kbytes.
$L_{COMA\,1\,level}$	208	$1.65 * 126$
$L_{COMA\,2\,levels}$	990	$1.65 * 600$
b	2	$b = n/32$
$L_{cap} = L_{cap-COMA}$	30	$1.65 * 18$
L_{coh}	599	$\frac{b-1}{b} * L_{COMA\,2\,levels} + \frac{1}{b} * L_{COMA\,1\,level}$
DDM, $n = 64$ processors		
1st level cache	4k/16k	The actual size is 64 kbytes.
$L_{COMA\,1\,level}$	105	$1.65 * (60 + 4)$
$L_{COMA\,2\,levels}$	196	$1.65 * (115 + 4)$
b	3	$b = n/24$
$L_{cap} = L_{cap-COMA}$	20	$1.65 * (8 + 4)$
L_{coh}	169	$\frac{3}{n-1} L_{cap} + \frac{n-4}{n-1}(\frac{b-1}{b} * L_{COMA\,2\,levels} + \frac{1}{b} * L_{COMA\,1\,level})$
DASH, $n = 64$ processors		
1st level cache	4k/16k	The actual size is 64 kbytes.
2nd level cache	16k/64k	The actual size is 256 kbytes.
$L_{2nd\,level\,cache}$	15	Access time to 2nd level cache.
$L_{local\,fill}$	29	Fill from local cluster memory.
c	16	Number of clusters is $n/4$.
$L_{2\,hop}$	101	$77 + 6\sqrt{c}$ (estimated)
$L_{3\,hop}$	132	$96 + 9\sqrt{c}$ (estimated)
L_{coh}	130	$\frac{3}{n-1} L_{local\,fill} + \frac{n-4}{n-1}(\frac{c-1}{c} L_{3\,hop} + \frac{1}{c} L_{2\,hop})$
L_{cap}	96	$\frac{3}{n-1} L_{local\,fill} + \frac{n-4}{n-1}(\frac{c-1}{c} L_{2\,hop} + \frac{1}{c} L_{local\,fill})$

Table 4.4: Estimates of real machine parameters, translated to a clock rate of 33 MHz.

4.4.2 DDM

A prototype design of the DDM is near completion at SICS. The hardware implementation of the processor/attraction memory is based on the TP881V system by Tadpole Technology, U.K. Each such system has up to four Motorola 88100 20 MHz processors, each with two 64 kbytes cache/MMUs with access times of two cycles. The attraction memory in each node is 32 Mbytes in size.

Read accesses from a cache to the attraction memory take eight cycles per cache line. A remote read to a node on the same DDM bus takes 60 cycles at best, most of which are spent making local bus transactions (a total of four accesses). Read accesses climbing one step up and down the hierarchy take approximately 115 cycles. For all accesses an extra four cycles is added for the latency of going through the first-level cache.

In Table 4.4, the timing is specified for the DDM. The cache sizes modeled are four times smaller than those modeled for the KSR1 machine. Six nodes of four processors are hosted on the lowest level bus, i.e., 24 processors. The next level bus

is expected to have a branching factor of eight. Since the DDM has a cluster size of four, $(n-1)/3$ of the capacity misses will be handled locally. This effects only small values on n.

4.4.3 DASH

DASH is a NUMA architecture built at Stanford University. It has 16 processor clusters connected by two two-dimensional meshes; one mesh for requests and one for replies [Len91]. Each cluster is built from a commercial product from Silicon Graphics which has four 33 MHz MIPS R3000 processors and 16 Mbytes of memory. Each processor has write-through 64 kbytes instruction and data caches, and a unified second-level cache of 256 kbytes. DASH implements a fully mapped directory cache-coherence protocol. An interface designed at Stanford connects the processor node to the network. The interface contains the directory information and a remote-access cache, mastering the remote actions. A read hit in the second-level cache takes 15 cycles. The second-level cache cuts the access time for some data missing in the smaller first-level cache, but it also adds extra delay to the local memory. A hit in the local memory takes 29 cycles, while a two-hop remote read access takes 101 cycles and a three-hop read takes 132 cycles.

Like the DDM, DASH has a cluster size of four and $(n-1)/3$ of the capacity misses and coherence misses are handled locally. Unlike the DDM, this research project already has a machine running with 48 processors.

It is modeled with the same size first-level cache as the caches in the DDM, and with second-level caches sized equally to the KSR1 machine.

4.4.4 Estimated Performance

The equations for L_{coh} and L_{cap} for each of the machines in Table 4.4 can be inserted in Equation 4.1, and their mean global access time can be calculated. For the DASH we calculate how many accesses hit in the second-level cache, and add its second-level delay for those accesses.

The average mean global access time for each of the architectures is shown in Figure 4.13. More detailed figures are given in Table 4.5. Note that the numbers reported are the access times to global data normalized to processor cycles at 33 MHz. The processors of the DDM and the KSR1 run at 20 MHz. As can be seen, the DDM has the lowest latency, while the DASH and KSR1 architectures alternate as second. Up to 32 processors, the KSR1 only needs one level, and its penalty for coherence misses is constant. Above 32 processors, the huge latency

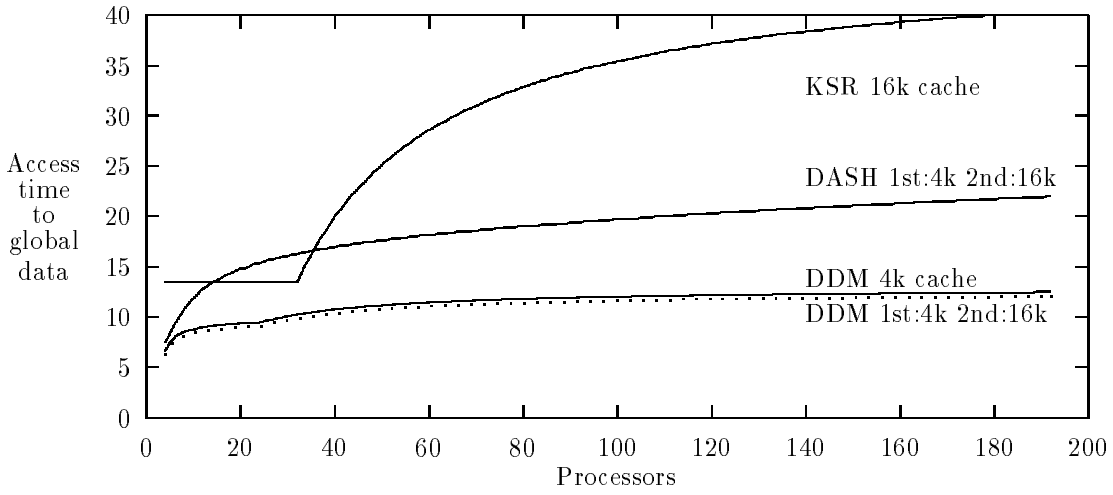


Figure 4.13: The average of the mean global access time over all applications calculated using the performance-estimation method described for different numbers of processors.

for the second-level ring has effect. The exploration of locality is a necessity for this architecture. For the DDM and DASH architectures, the effect of clustering is visible for the performance of few processors. This gives DASH an advantage over KSR1 for up to 16 processors. The next crossover point, after which DASH performs better, lies around 38 processors.

A large second-level cache could delay the access time of a node's local memory. In applications exploring improved hit rates in local memories—possibly with a successful page-migration strategy in a NUMA or by an attraction memory—introducing a second-level cache might actually slow down execution. We wanted to try this hypothesis for the DDM and modeled a DDM equipped with a second-level cache of four times the size of its first-level cache. It has an access time of ten cycles and adds five extra cycles of latency for all accesses missing in the second-level cache. As can be seen, the second-level cache adds only marginally to performance.

What if KSR1 had been a NUMA? A NUMA with a latency similar to KSR's would have resulted in terrible performance. In Figure 4.14 we compare the three architectures with a NUMA with latencies similar to KSR1's. The T_{hop} for two levels is 400 cycles, the T_{hop} for one level is 70 cycles and the cache size is 16 kbytes.

Architecture	L_{cap}/ L_{coh}	Applications and Data Set Size Per Node								Avg all	Avg 4
		MP3D DS:34k	Pthor 199k	Locus 77k	Water 13k	Chol 62k	LU 40k	B-H 25k	Ocean 151k		
n = 16 Processors											
DDM 4k	20/ 82	11	11	6	6	9	7	10	14	**9**	**7**
DDM 4k,16k	20/ 82	11	10	5	5	6	4	6	9	**7**	**5**
Dash 4k,16k	65/ 92	15	15	8	9	11	10	18	25	**14**	**9**
KSR 16k	30/208	24	22	8	10	12	6	10	16	**14**	**9**
DDM 16k	20/ 82	11	10	5	5	6	4	6	9	**7**	**5**
DDM 16k,64k	20/ 82	11	10	5	5	6	4	6	9	**7**	**5**
Dash 16k,64k	65/ 92	13	13	6	7	7	5	8	16	**9**	**6**
KSR 64k	30/208	23	22	7	9	11	5	5	12	**12**	**8**
n = 32 Processors											
DDM 4k	20/106	14	14	6	7	10	7	10	14	**10**	**7**
DDM 4k,16k	20/106	14	13	6	7	9	6	9	13	**10**	**7**
Dash 4k,16k	80/109	18	18	9	10	12	11	21	30	**16**	**11**
KSR 16k	30/208	24	22	8	10	12	6	10	16	**14**	**9**
DDM 16k	20/106	14	13	5	6	7	4	7	10	**8**	**6**
DDM 16k,64k	20/106	13	12	5	6	7	4	6	10	**8**	**6**
Dash 16k,64k	80/109	15	15	7	8	8	6	9	19	**11**	**7**
KSR 64k	30/208	23	22	7	9	11	5	5	12	**12**	**8**
n = 64 Processors											
DDM 4k	20/137	17	16	7	8	11	7	11	15	**12**	**8**
DDM 4k,16k	20/137	17	16	6	8	11	6	10	14	**11**	**8**
Dash 4k,16k	93/125	20	20	11	12	14	12	24	34	**18**	**12**
KSR 16k	30/599	63	58	16	22	27	10	14	28	**30**	**19**
DDM 16k	20/137	17	15	6	7	8	5	7	11	**10**	**7**
DDM 16k,64k	20/137	16	15	6	7	8	5	6	10	**9**	**6**
Dash 16k,64k	93/125	17	17	8	9	9	7	10	21	**12**	**8**
KSR 64k	30/599	63	57	15	21	26	9	9	24	**28**	**18**
n = 192 Processors											
DDM 4k	20/160	19	18	7	9	12	7	11	16	**12**	**9**
DDM 4k,16k	20/160	19	18	7	8	11	7	10	15	**12**	**8**
Dash 4k,16k	115/156	24	24	13	14	16	14	29	41	**22**	**14**
KSR 16k	30/860	89	81	21	29	38	13	16	36	**40**	**25**
DDM 16k	20/160	19	17	6	8	9	5	7	12	**10**	**7**
DDM 16k,64k	20/160	19	17	6	8	9	5	6	11	**10**	**7**
Dash 16k,64k	115/156	21	21	9	10	11	8	11	25	**14**	**9**
KSR 64k	30/860	89	80	20	29	37	11	12	32	**39**	**24**

Table 4.5: The mean global access time estimated for the different architecutres. The unit of delay is 30 ns, i.e., comparable to 33 MHz.

4.5 Comparing the Models of the Ideal and Real Systems

As can be seen in Table 4.4, the latency numbers achieved modeling the ideal systems earlier do not correspond to the latency numbers of the real prototypes. This is mostly because the equations model ideal systems, while the architectures are built on top of commercially available ones, leading to a number of compromises.

Both the DDM and DASH suffer from one extra T_{lb} when a requested datum reaches the requesting node. Also, when a requested datum arrives at a node, it is first written to the local memory before the processor can access it. The interface logic of DASH (remote access cache) adds delays that are not included in the ideal model.

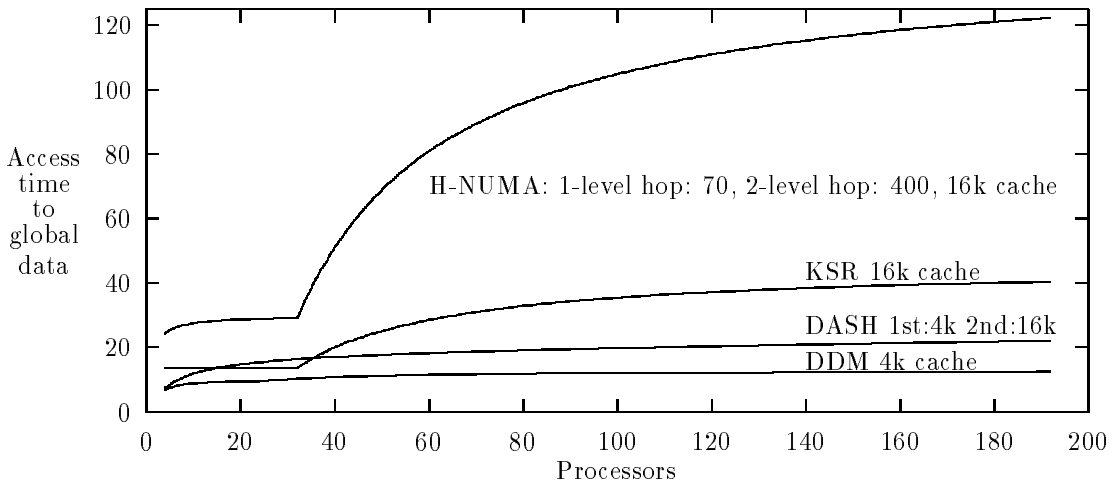


Figure 4.14: What if KSR1 had been a NUMA? A NUMA with a remote delay similar to the KSR1 caches of 16 kbytes is compared to the other three architectures. Note the scale on the y axis.

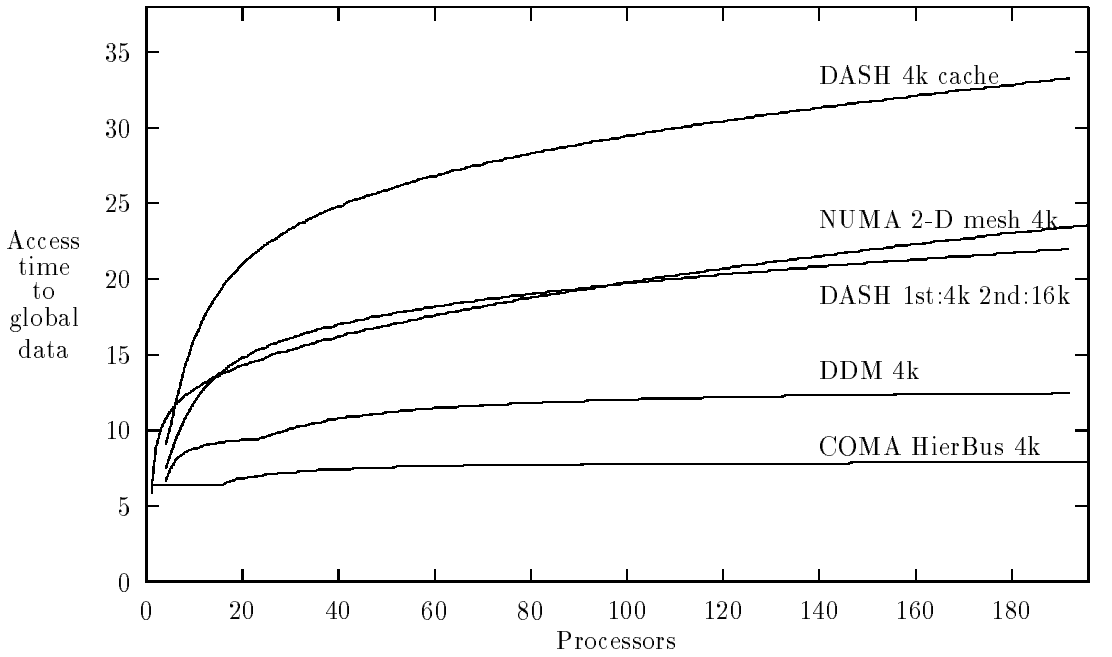


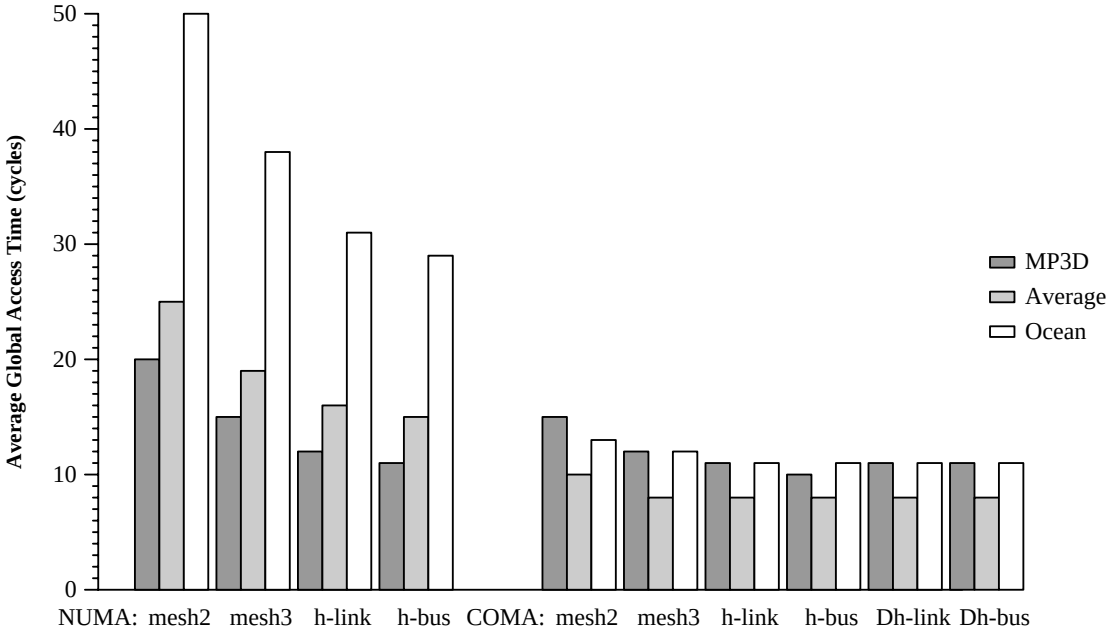
Figure 4.15: Validating the two analytical methods by comparison.

DASH also implements error checking and recovery each time a message enters a node. The DDM prototype does not implement the return-to-sender feature, adding extra terms of T_{dir}.

We compare the access times to the global data of the parameterized analytical model, and the model based on real machines, in Figure 4.15. In the figure, the curve for a DASH with only its first-level cache is included since two-level caches are not modeled by the ideal models. Note that a second level cache has a large effect on the performance in a NUMA.

We can see that the ideal models for both implementations are about 50 percent better than the real machines, except for when the clustering effect is seen for the lowest number of processors.

⋆ ⋆ ★ ⋆ ⋆



Our analytical study defined a set of equality lines, defining the capacity miss rate where NUMA and COMA have comparable performance as a function of the coherence miss rate. The study shows that COMA is clearly better than NUMA for most of the SPLASH benchmarks assuming reasonably sized data sets. The study did not take network contention into account. If network contention is included, an even larger advantage of COMA can be expected since its large caches reduce communication. The study shows a larger difference between NUMA and COMA than between different network variation studied. The optimum network seems to be a fat tree in combination with the COMA protocol for a general network.

Many aspects important for the choice of network are not modeled here: contention, communication locality, write delay, and limited pointers. Also, if the performance is so close, the choice of network and protocol may involve other aspects, such as which is the easiest/cheapest to implement.

We also studied the performances of three real architectures, DDM, DASH, and KSR1. Here, the high latency of its network made the performance of the KSR1 COMA worse than the DASH NUMA if the number of processors is greater than 32. The DDM, which is a COMA with a latency comparable with DASH, performs the best for the whole range .

Part II

IMPLEMENTING COMA–THE DDM

5

A DDM Primer

THE Data Diffusion Machine (DDM) [WH88] is a hierarchical COMA with its directory information distributed in the network. The bus-based DDM architecture is introduced in this chapter by first looking at its smallest part, the DDM with a single bus.

The single-bus DDM consists of several large attraction memories connected by a bus. Each attraction memory has a processor connected to it. The properties of the attraction memory architecture, bus behavior, and protocol are discussed. The larger, hierarchical DDM is then introduced as an extension to the single-bus architecture. The hierarchical DDM connects several single-bus DDMs in a hierarchical network.

The chapter ends with a detailed description of the architecture and a discussion of some important COMA issues like handling the shared memory space.

5.1 Single Bus DDM–a Small COMA

We introduce the DDM architecture by first looking at a small machine built around a single bus. The cache-coherence protocol for a COMA can adopt existing techniques used in other cache-coherence protocols and be extended with the functionality of finding a datum on a cache-read miss and handling replacement. The system described here therefore has many similarities with existing cache-coherent single-bus architectures, such as the Sequent Symmetry.

5.1.1 The Architecture of a Node

A DDM node in this description consists of a processor and an attraction memory. The attraction memories of the minimal DDM are connected by a single bus. Data distribution and coherence among 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*.

A cache line of an attraction memory, here called an *item*, is viewed by the protocol as one unit. The attraction memory stores a small state field per item and an address tag for implementing its associativity. The architecture of the nodes in the single-bus DDM is shown in Figure 5.1.

The DDM uses a split-transaction bus, where the bus is released between a requesting transaction and its reply, e.g., between a read request and its data reply. Delays between request and reply may 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, assuring 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 to avoid deadlock. Figure 5.1 shows how several nodes are connected together by the DDM bus. Selection and arbitration are handled by two state machines. Subsystems on a DDM bus can try to get selected with different priorities. Arbitration logic will then determine which subsystem to select. On top of the bus is the *top protocol*. It is a simple protocol containing no state memory.

5.1.2 The Protocol of the Single Bus DDM

The protocol of the DDM is similar in many ways to the write-once protocol (Figure 3.2). The write-coherence part of the protocol described here is of the

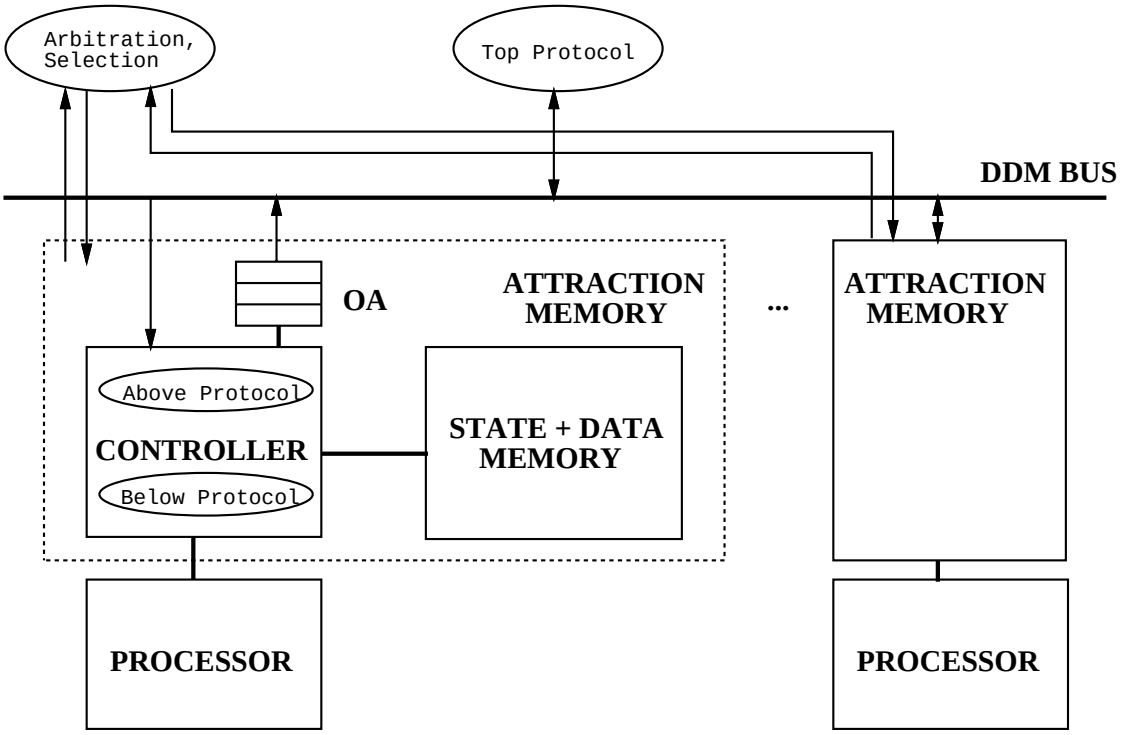


Figure 5.1: Architecture of a single-bus DDM. Processors are below the attraction memories. Located on top of the bus are arbitration and selection.

write-invalidate type so that, in order to keep data coherent, all copies of the item but the one to be updated are erased on a write.

The protocol also handles the data attraction (read) and replacement when a set in an attraction memory becomes 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 current state of the item in the attraction memory:

$$PROTOCOL : current\,state \times transaction \rightarrow new\,state \times new\,transaction.$$

The different states are explained in Table 5.1. The first three states, Invalid (I), Exclusive (E), and Shared (S), correspond to the Invalid, Reserved, and Valid states in the write-once protocol. The Dirty state in that protocol, meaning "this is the only cached copy and its value differs from that in the memory," has no correspondence in a COMA. New states in the protocol are the *transient states* Reading (R), Waiting (W), and Reading and Waiting (RW). The transient states were created to serve the split-transaction bus and remember outstanding requests.

The transactions carried on a DDM bus are listed in Table 5.1.

A processor writing an item in state E or reading an item in state E or S will proceed without interruption. A read attempt of an item in state I will result in a

	State	Description
I	Invalid	This subsystem does not contain the item.
E	Exclusive	This subsystem and no other subsystem contains the item.
S	Shared	This subsystem and possibly other subsystems contain the item.
R	Reading	This subsystem is waiting for a data value after having issued a read.
A	Answering	This subsystem has promised to answer a read request.
W	Waiting	This subsystem is waiting to be exclusive after having sent an erase.
RW	Reading and Waiting	This subsystem is waiting for a data value,later to become exclusive.

	Transaction	Description
e	*erase*	Orders recipient to erase all copies of the item.
x	*exclusive*	Acknowledges an *erase* request.
r	*read*	Requests to read a copy of the item.
d	*data*	Carries the data in reply to an earlier read request.
i	*inject*	Carries the only copy of an item and is looking for a subsystem to move into, caused by a replacement.
o	*out*	Carries the data on its way out of the subsystem, caused by a replacement. It will terminate when another copy of the item is found.

Table 5.1: Transactions and states of the DDM. For the lowest level in the hierarchy, "subsystem" refers to the attraction memory.

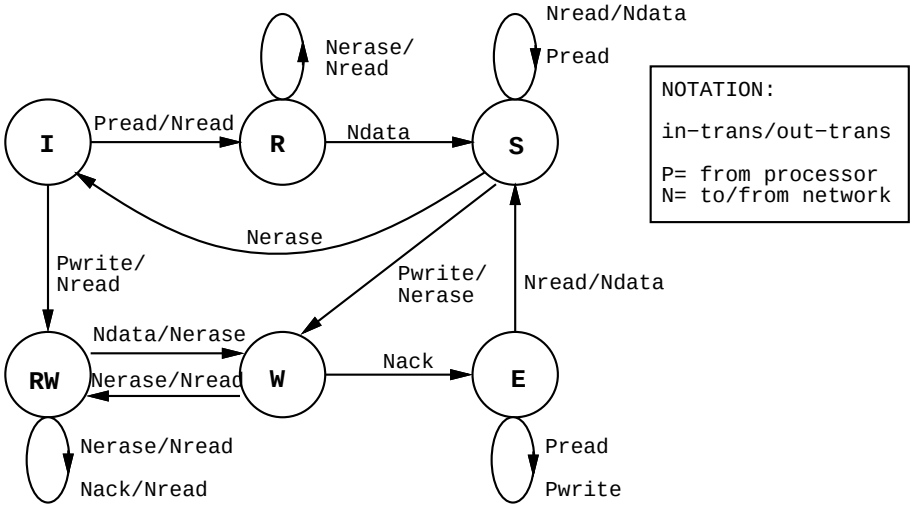


Figure 5.2: A simplified representation of the attraction memory protocol not including replacement.

read request and a new state R as shown in Figure 5.2. The selection mechanism of the bus will select one attraction memory which contains the item to service the request, eventually putting *data* on the bus. The requesting attraction memory, now in state R, will grab the *data* transaction, change state to S, and continue.

Processors are only allowed to write to items in state E. If the item is in S, all other copies have to be erased and an acknowledge received before the writing is allowed.

The attraction memory sends an *erase* transaction and waits for the acknowledge transaction *exclusive* in the new state, W. Many simultaneous attempts to write the same item will result in many attraction memories in state W, all with an outstanding *erase* transaction in their output buffers. The first erase to reach the bus is the winner of the write race. All other transactions bound for the same item are removed from the small *output above* buffers, OA. Therefore, the buffers also have to snoop transactions. The OA can be limited to a depth of three, and deadlock can still be avoided with a special arbitration algorithm. The losing attraction memories in state W change state to RW while one of them puts a *read* request in its output buffer. Eventually, the *top protocol* at the top of the bus replies with an *exclusive* acknowledge, telling the only attraction memory left in state W that it may now proceed. Writing to an item in state I results in a *read* request and a new state RW. Upon the *data* reply, the state changes to W and an *erase* request is sent.

5.1.3 Replacement

Though large, the attraction memories run out of space, forcing some items to leave room for more recently accessed ones. If the set where an item is supposed to reside is full, one item in the set is selected to be replaced.

Preferably, the oldest item in state S, of which there might be other copies, is selected for replacement. Replacing an item in state S generates an *out* transaction. The space used by the item can now be reclaimed. If an *out* transaction sees an attraction memory with a copy of the item, it terminates; if not, it is converted to an *inject* transaction by the top protocol. An *inject* transaction can also be produced by replacing an item in state E. The *inject* transaction is the last copy of an item trying to find a new home in a new attraction memory. In the single-bus implementation it will do so firstly by choosing an empty space (state I), and secondly by replacing an item in state S; i.e., it will decrease the amount of sharing. If the item-identifier space that corresponds to the physical address space of conventional architectures is not made larger than the sum of the attraction memory sizes, it is possible to devise a simple scheme that guarantees a physical location for each item.

Often, only a portion of the physical address space is used in a computer. This is especially true for operating systems eagerly reclaiming unused work space. In the DDM, the unused item space may be used to increase the degree of sharing if the unused items are purged. The operating system might even change the degree of sharing dynamically, as discussed in Section 3.3.3.

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

5.2 Hierarchical DDM–a Large COMA

The single-bus DDM, as described above, can become a subsystem in a large hierarchical DDM by replacing the top with a directory, which interfaces between the bus described and a higher level bus of the same type in a hierarchy; see Figure 5.3. The directory can answer the questions: "Is this item below me?" and "Does this item exist outside my subsystem?"

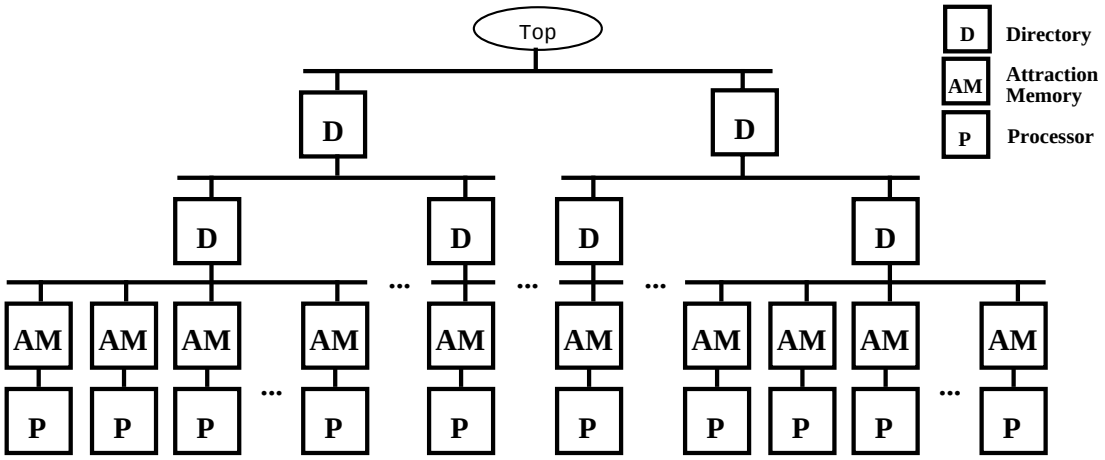


Figure 5.3: Hierarchical DDM, here with three levels.

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.

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

The directory as shown in Figure 5.4 has a small output buffer, *output above* (OA), to store transactions waiting to be sent onto the higher bus. Transactions for the lower bus are stored in the buffer *output below* (OB), and transactions from the lower bus are stored in the buffer *input below* (IB). A directory reads from input below 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.

5.2.1 Multilevel Read

If a *read* request cannot be satisfied by the subsystems connected to the bus, the next higher directory retransmits the *read* request onto the next higher bus. The directory

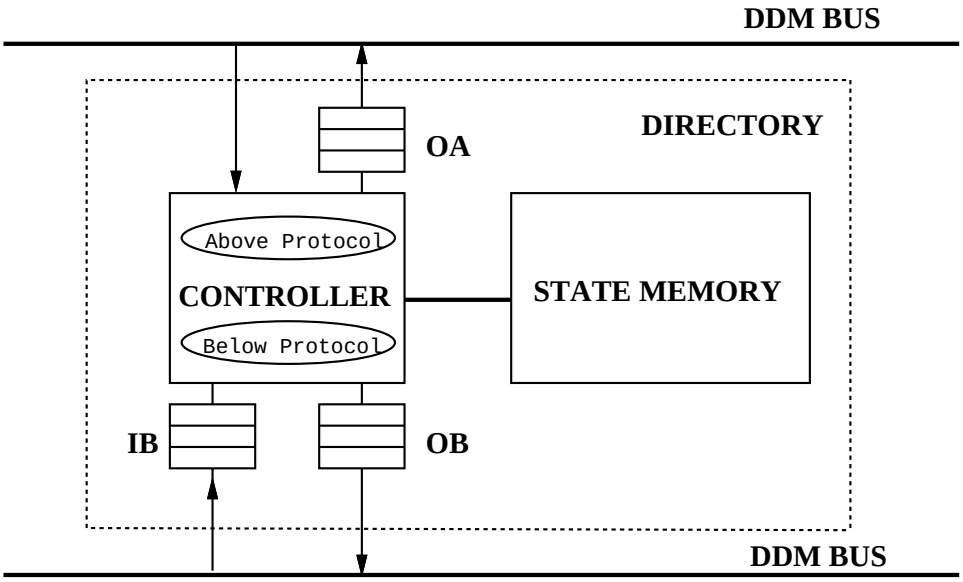


Figure 5.4: The architecture of a directory.

also changes the item's state to Reading (R), marking the outstanding request. Eventually, the request reaches a level in the hierarchy where a directory, containing a copy of the item, is selected to answer the request. The selected directory changes the state of the item to Answering (A), marking an outstanding request from above, and retransmits the *read* request on its lower bus. Transient states R and A in the directories mark the request's path through the hierarchy, shown in Figure 5.5, like unrolling a red thread while walking in a maze [HomBC].

A flow-control mechanism in the protocol prevents deadlock if too many processors try to roll out a red thread to the same set in a directory. When the request finally reaches an attraction memory with a copy of the item, its *data* reply simply follows the red thread back to the requesting node, changing all the states along the path to Shared (S). Many processors often try to read the same item, creating the "hot-spot" phenomenon [P+85]. Combined reads and broadcasts are simple to implement in the DDM. If a *read* request finds the red read thread rolled out for the requested item (state R or A), it simply terminates and waits for the *data* reply which will eventually follow that path on its way back. Stenström reports that combining situations are rare for most studied programs, but found one application, LU, where 47 percent of the read requests got combined in the DDM hierarchy [SJG92b].

Extra latency is introduced with every directory lookup. If the directories are implemented with slow memories and/or if the number of levels in the hierarchy is large, this latency will become the dominant part of the read latency. However, half of the lookups can be removed from the critical path. If a *read* request carries the identity of the requesting node, the returning *data* knows which node is the requester. The data can find its way back without any lookups. As the data returns, the directory

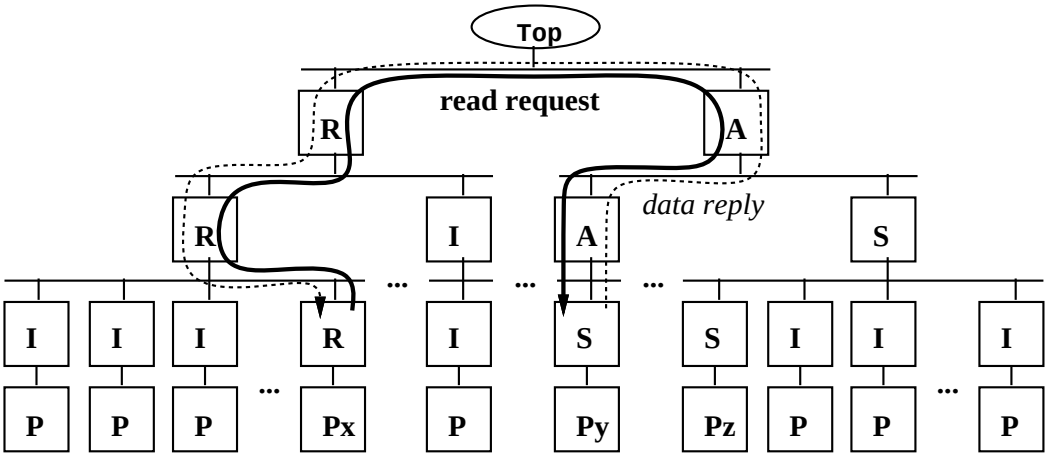


Figure 5.5: A read request from processor Px has found its way to a copy of the item in the attraction memory of processor Py. Its path is marked with states Reading and Answering (R and A), which will guide the data reply back to processor Px.

lookup can be performed in parallel with the forwarding of the transaction, and not on the critical latency path. Only requests utilizing the combining effect will suffer from the directory lookup latency. We call this technique *return to sender*.

5.2.2 Multilevel Write

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

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 5.6. The losing attraction memory (now in state RW) will restart a new write action automatically upon receiving the erase.

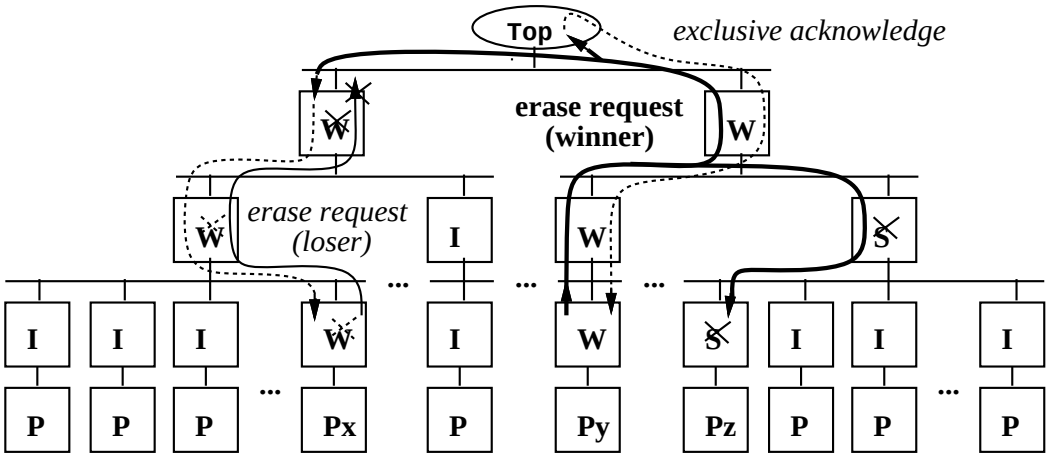


Figure 5.6: Write race between two processors, Px and Py, resolved when the request originating from Py reaches the top bus (the lowest bus common to both processors). The top can now send the acknowledge, *exclusive*, which follows the path marked with Ws back to the winning processor Py. The W states will be changed to E by the *exclusive* acknowledge. The *erase* will erase the data in Px and Pz, forcing Px to redo its write attempt.

5.2.3 Replacement in the Hierarchical DDM

Replacement of a shared item in the hierarchical DDM will result in an *out* transaction propagating up the hierarchy and terminating when a subsystem with a copy of the item is found. If the last copy of an item marked with state S is replaced, an *out* that fails to terminate will reach a directory in state E and be turned into an *inject*.[1] Replacing an item in E generates an *inject* transaction, which then tries to find an empty space in a neighboring attraction memory. *Inject* transactions will first look for an empty space in the attraction memories of the local DDM bus, as in the single-bus DDM. Unlike the single-bus DDM, an *inject* failing to find an empty space on the local DDM bus will turn to a special bus, its home bus, determined by the item identifier. On the home bus, the *inject* will force itself into an attraction memory, possibly by throwing a foreigner and/or shared item out. The item home space is equally divided among the bottommost buses, and space is therefore guaranteed on the home bus.

The home bus, as described, is different from memory locations in NUMAs in that the notion of a home is only used after failing to find space elsewhere. When the item is not there, its place can be used by other items. Another difference is that the home is a bus and not a node, i.e., any attraction memory on that bus will do.

The hierarchy as described here has a single top bus which easily could become the bottleneck of the system. This bottleneck can be widened using several techniques discussed in Chapter 10.

[1] Note that the top of the DDM hierarchy considers all items to be in state E.

5.2.4 Arbitration

A centralized arbiter chooses the next subsystem to send a transaction on the bus. The number of outstanding requests must be limited to avoid deadlock and guarantee progress in the execution. Filling all buffers with requests would make it impossible to service any of the requests. It is also important that a request transaction not result in more than one data reply. We have selected a bus arbitration for the DDM bus with three priority levels to avoid the deadlock problem:

1. The subsystem selected to service the previous transaction (if any) is given the highest priority. The last transaction on the bus might have resulted in a new transaction for its output-above buffer. Being given the highest arbitration priority entitles it to send a new transaction during the next cycle; i.e., there can never be an increased number of transactions in the output-above buffer caused by snooping on the DDM bus above.

2. Transactions from the above directory are given the second highest priority. This prevents new transactions from the subsystems from entering the bus, unless a subsystem is selected, while there are requests or replies from higher subsystems arbitrating for the bus.

3. Lowest priority is given to the nonselected subsystems which are serviced in a round-robin fashion.

5.2.5 Flow control

In order to guarantee the completion of a DDM bus transaction, there should always be enough space left in the output-above buffer to accept a lookup from above. Because of arbitration priority, a selected request from above would only temporarily increase the number of transactions in that buffer. A small amount of extra space in the output-above buffer will thus guarantee the completion of a DDM transaction. A transaction from below to either a directory or an AM is only handled if there is enough space in the output-above buffer. This might prevent a transaction from the processor from completing; i.e., the processor might be stalled.

Any device connected to a DDM bus can temporarily halt the bus by asserting the brake signal of the bus. When there are only a few spaces left in the input-below buffer underneath a directory, the DDM bus below will be stalled by a *brake below* signal. If the output-below buffer of a directory is almost full, the bus above is suspended by a *brake above* signal. A processor node may also pull the *brake above* when it gets congested, e.g., if reads and writes to the AM take longer than a single DDM transaction.

5.3 A Closer Look at the Protocol

The protocol is defined as four state machines: *attraction memory below*, *attraction memory above*, *directory below*, and *directory above*. The state machines define a selection priority, a new state, and a new transaction as a function of the current transaction type doing the lookup and the current state in the corresponding state memory. The directory-below protocol also looks at whether or not a subsystem tried to be selected. The regularity of the protocol simplifies its representation and implementation. In writing, the protocol can be defined in a table format indexed by the current state and the current transaction type. Each state machine can be implemented by a PROM, addressed by the current state and the current transaction, plus some additional logic. The number of different states and transactions thus only affects the size of the PROM, not how difficult an implementation would be. A fifth table, *top*, defines a simple stateless function at the top of the hierarchy.

The output-above buffers snoop the transactions on the DDM bus above them. Some transactions on this bus purge other transactions involving the same item in the output-above buffers. The arbitration scheme used allows for these buffers to be short. The output-above buffer should have a minimum depth of three in order to avoid deadlock. None of the other buffers perform any snooping and can be made deep at a low cost. They are not part of the atomic snooping action, which also allows for an asynchronous coupling between the levels in the DDM.

The state machines of a single-level DDM are defined in Table 5.2. Each state has its own column and each transaction its own row. Actions have the format: *guard*→NEWSTATE:transaction-to-send$_{Index}$, where Index A means to the bus above and Index B means to the bus below. The subsystems can try to be selected with different priorities. The action taken for a granted selection and the action taken for a refused selection can differ. The guard for a successful selection of priority n is ($seln\rightarrow$), and the action for a refused selection is ($\neg sel\rightarrow$). No guard means that the action defined in the square will always be carried out. An empty square means no action and $\emptyset$ represents impossible combinations.

Table 5.3 defines the additional functionality required by a multilevel DDM. The state machine directory below is not part of the atomic selection action, but it monitors attempts at selection. "None tried to be selected" ($a = 0\rightarrow$) and its inverse ($a \geq 1\rightarrow$) are part of the directory-below protocol.

The directory operates like a filter, keeping transactions that can be serviced locally from spreading to the rest of the system. A simple scheme can prevent the directory from looking at all transactions. The scheme, called the input-below filter (IBF), can save some of the bandwidth of the directory state memory. Its functionality is defined in Table 5.3, and its location is shown in Figure 5.7.

In table	Transaction
r	read request
d	data reply
e	erase request
x	exclusive acknowledge
o	out (replacement of shared)
i	inject (replacement of exclusive)

In table	State
I	Invalid
E	Exclusive
S	Shared
R	Reading
W	Waiting
RW	Reading and Waiting
A	Answering

ATTRACTION MEMORY BELOW

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

TOP

Trans-action	$a = 0 \rightarrow$	$a \geq 1 \rightarrow$
r	new item	
e	x_B	x_B
d		
x		
o	i_B	
i	go home	

ATTRACTION MEMORY ABOVE

Trans-action	**State** I	E	S	R	W	RW
r		$sel1 \rightarrow$S:d_A $\neg sel \rightarrow \emptyset$	$sel1 \rightarrow$S:d_A $\neg sel \rightarrow$S:d_A		$sel2 \rightarrow$ $\neg sel \rightarrow$	
e [3]		$\emptyset$	I:-	$sel1 \rightarrow$R:r_A $\neg sel \rightarrow$	$sel1 \rightarrow$RW:r_A $\neg sel \rightarrow$RW:-	$sel1 \rightarrow$RW:r_A $\neg sel \rightarrow$
d		$\emptyset$		S:$\perp$		$sel1 \rightarrow$W:e_A $\neg sel \rightarrow$
x		$\emptyset$	$\emptyset$	$sel1 \rightarrow$R:r_A $\neg sel \rightarrow$	E:$\perp$	$sel1 \rightarrow$RW:r_A $\neg sel \rightarrow$
o [4]		$\emptyset$	$sel0 \rightarrow$ $\neg sel \rightarrow$	$sel1 \rightarrow$S:$\perp$ $\neg sel \rightarrow$S:-	$sel0 \rightarrow$	$sel2 \rightarrow$W:e_A $\neg sel \rightarrow$
i	$seln \rightarrow$E:- $\neg sel \rightarrow$	$\emptyset$	$\emptyset$	$sel1 \rightarrow$S:$\perp$ $\neg sel \rightarrow$	$\emptyset$	$sel2 \rightarrow$W:e_A $\neg sel \rightarrow$

$\emptyset$ This situation is impossible.
$\perp$ The processor may continue with its operation.
$seln \rightarrow$ Selection of the nth priority succeeded. The priority increases with the number.
$\neg sel \rightarrow$ The attempt to become selected failed.
[1] Preceded by a replace of the old item.
[2] The processor is suspended. The processor will be revived by a $\perp$ for this very item.
[3] An e on the DDM bus kills r, e, d and o in OA.
[4] An o on the DDM bus kills r and d in OA.

Table 5.2: The protocol of the single-bus DDM.

DIRECTORY BELOW

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

INPUT BELOW FILTER

Transaction	$a = 0 \rightarrow$	$a \geq 1 \rightarrow$
r	r	
e	e	e
d	d	d
x		
o	o	o
i	i	i

DIRECTORY ABOVE

Transaction	States I	E	S	R	W	A
r		$sel1 \rightarrow$ A:r_B $\neg sel \rightarrow \emptyset$	$sel1 \rightarrow$ A:r_B $\neg sel \rightarrow$		$sel2 \rightarrow$ $\neg sel \rightarrow$	$sel2 \rightarrow$ $\neg sel \rightarrow$
e[2]		$\emptyset$	I:e_B	$sel1 \rightarrow$ R:r_A $\neg sel \rightarrow$	I:e_B	I:e_B
d		$\emptyset$		S:d_B		S:-
x		$\emptyset$	$\emptyset$	$sel1 \rightarrow$ R:r_A $\neg sel \rightarrow$	E:x_B	$\emptyset$
o[3]		$\emptyset$	$sel0 \rightarrow$	$sel1 \rightarrow$ S:d_B $\neg sel \rightarrow$ S:d_B	$sel0 \rightarrow$	$sel1 \rightarrow$ S:- $\neg sel \rightarrow$ S:-
i	$seln \rightarrow$ E:i_B	$\emptyset$	$\emptyset$	$sel1 \rightarrow$ S:d_B $\neg sel \rightarrow$ S:d_B	$\emptyset$	$\emptyset$

$a = 0 \rightarrow$ No subsystem tried to be selected or had the item.
$a \geq 1 \rightarrow$ At least one subsystem tried to be selected or had the item.
[1] The transaction might be sent by the directory itself.
[2] e on the DDM bus kills r, e, d and o in OA.
[3] o on the DDM bus kills r and d in OA.

Table 5.3: The protocol of directories of a hierarchical DDM.

5.4 Protocol Examples

A few examples will help explain what happens at the different levels of the DDM. The figures in the examples show the state changes and transitions for just one address. State transitions and bus transactions are indexed to indicate in which order they take place.

Table 5.4 shows a picture of an initial, two-level system with the item residing in the attraction memories of two of the twelve processors, P2 and P4. Thus, the copies of the item in both P2 and P4 are in the S state, and, in the parent directories above them, the item is in the E state. Everywhere else the item is in state I (nonexistent). Note that the top actually does not store any state information. Here it is still marked with an E, indicating that our picture of the system can very

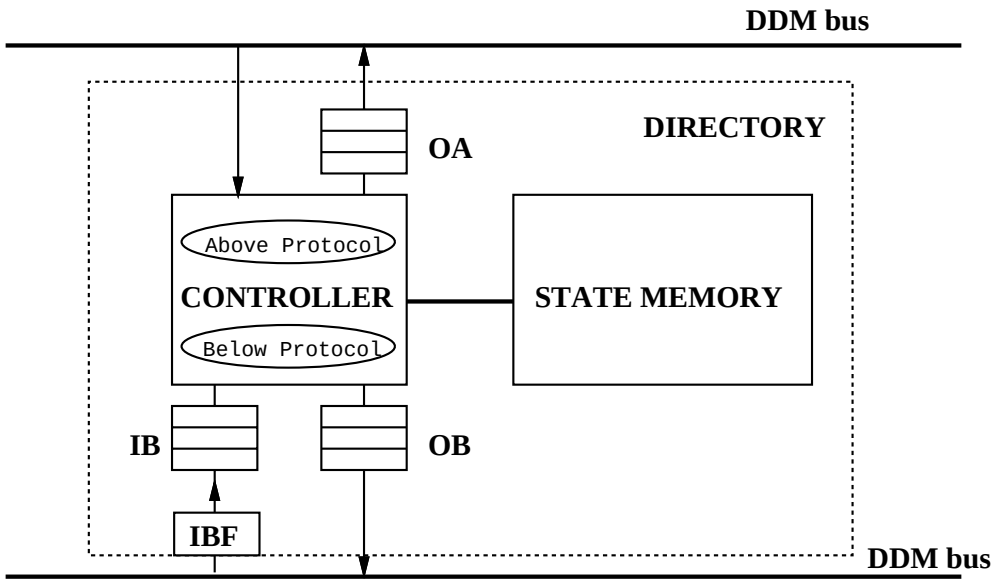


Figure 5.7: Connecting the input below filter (IBF).

				E							
E				I				I			
I	S	I	S	I	I	I	I	I	I	I	I
P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12

Table 5.4: The initial system.

well be a subsystem of a bigger machine.

When any of the processors P1–P4 read or write the item, actions similar to those of the single-bus DDM occur. The directory above the bus has the item in state E and anticipates a response to a *read* request from somewhere in its subsystem. Upon an *erase* request from below, the directory will issue the acknowledge transaction, *exclusive*, since the item is exclusive to its subsystem. No unnecessary bus traffic will be generated outside the subsystem.

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

r^1 Indicates a *read* transaction on the bus in phase 1.
I$\xrightarrow{1}$R Indicates a state transition from the *invalid* state to Reading in phase 1.
The actions in the different phases are explained below.
[1] A read by P6 to an item I generates a *read* and changes the state to R.
[2] The directory detects a nonlocal action and repeats the *read* upward, changing its state to R.
[3] A directory in state E answers the request by changing its state to A, sending *read* below.
[4] One of the memories, P2, is selected to service the *read*. It stays in S and sends *data*.
[5] The directory in state A has promised to answer. It sends *data* above and changes its state to S.
[6] The directory in state R is waiting for the *data*. It changes state to S and sends the *data* below.
[7] The attraction memory in state R is waiting for the *data*. It receives the *data* and changes state to S.
NOTE 1: Many subsystems on a bus may have an item in state S. Letting all of them reply with the *data* would produce unnecessary bus transactions; instead, one is selected in phase 4.
NOTE 2: After phase 3, the return path for *data* is marked with As and Rs.

Table 5.5: Multilevel read.

5.4.1 Reads

Multilevel reads involve several buses. The choice of a split-transaction bus means that there is no need to lock all the buses for the entire read operation. In Table 5.5 we start with the same state as Table 5.4. Now P6 tries to read the item. The path of the request is marked: state R marks the path of the *read* request on its way up, and state A marks the path on its way down. The *data* transaction may use these states to find its way back to the requesting attraction memory.

If the return-to-sender technique is used, i.e., if the *read* request carries the identity of the requesting processor, the reply can be forwarded more quickly, since its path is known before the directory lookup. The state transitions shown here will still take place, even on the return path. However, the forwarding of *data* is done in parallel with the lookup, instead of after the lookup.

Table 5.6 shows the combining read implementation. A *read* request finds the red read thread rolled out for the same item (state R or A), and terminates. Table 5.6 differs from Table 5.5 only in that P10 also reads the item between phases 1 and 4.

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

$^{<4}$ P10 also reads before phase 4.
$^{<5}$ A second *read* request will appear on the top bus generating no extra action.
^{6,7} The *data* originally intended for P6 will also be received by P10.

Table 5.6: Combining read and broadcasting.

E$\overset{3}{\rightarrow}$**E** e^2x^3															
S$\overset{3}{\rightarrow}$**I** e^3				**S**$\overset{2}{\rightarrow}$**W**$\overset{4}{\rightarrow}$**E** e^1x^4				**S**$\overset{3}{\rightarrow}$**I** e^3			
I	**S**$\overset{4}{\rightarrow}$**I**	**I**	**S**$\overset{4}{\rightarrow}$**I**	**I**	**S**$\overset{1}{\rightarrow}$**W**$\overset{5}{\rightarrow}$**E**	**I**	**I**	**I**	**S**$\overset{4}{\rightarrow}$**I**	**I**	**I**
P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12

[1] P6 tries to write to the shared item, generates an *erase* (e), and changes state to W.
[2] The directory detects a nonlocal *erase*, changes its state to W, and retransmits *erase* above.
[3] Directories in state S receiving an erase from above change state to I and repeat the *erase* below. The top directory detects a local *erase* in its subsystem and replies with an *exclusive* (x) below.
^{4,5} The *exclusive* transaction uses the trail of Ws to find the "winning" processor. It is guaranteed that there is at most one path of Ws to the leaves of the net.
NOTE: The acknowledge of the *erase* (*exclusive*) is sent when the *erase* reaches the top, not when it reaches the memories.

Table 5.7: Multilevel write.

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

[1–2] Like Table 5.7, both *erases* work their way up toward the top bus.
[3] The *erase* originating in P6 is the winner and is carried on the top bus. All other directories change their states to I and retransmit the *erase* below.
[4] P10 receives the bad news (*erase*). Instead of just invalidating it starts a *read* transaction.
[5] P6 becomes the exclusive owner of the item and carries out the write.
[7] The *read* from P10 reaches P6, which changes state to S and sends *data* containing the new value.
[10] The *data* reaches P10, which changes state to W and once more sends an *erase*. We wish it better luck this time.

Table 5.8: Write race.

5.4.2 Writes

While the main goal of a *read* is to find and deliver an item, writing involves worrying about consistency. Processors are allowed to write to an item only when it is in state E. If the item is in state S, all other copies are erased before writing is allowed. A subsystem waiting for all other copies to be erased uses the transient state W to mark its intention. Table 5.7 starts with the final state of Table 5.6: P6 writes to a shared item.

Table 5.8 is identical to Table 5.7, except that both P6 and P10 try to write at the same time. It shows how a write race is handled by the protocol.

5.5 Directory Replacement and Mapping

As discussed earlier, replacement of a shared item will result in an *out* transaction propagating up the hierarchy and terminating when a subsystem with a copy of the item is found.

Replacing an item in state E generates an *inject* transaction. If an *inject* transaction fails to find a new home on the lowest level, as described earlier, the directory converts it to a *go home* transaction, just like in the single-bus case. If the last copy of an item marked S is replaced, an *out* will be unable to find another copy in the system. However, when the *out* eventually reaches a directory in state E, it will be converted into an *inject*. The *out* carries the data value, which is rarely used, to

make this conversion possible. Moving the last two items out at the same time is a special case of the above.

5.5.1 Preferred Home Location

The refugee transaction *inject* that failed to find a new home is converted to the *go home* transaction as described earlier. Although we have tried to avoid any notion of a home location for items, this emergency situation requires it. The item space is equally divided among all directories at the same level, so that each item has one preferred bus. The directories can check if a transaction is in their portion of the item space and can guide the *go home* to its home bus. The home item space of a bus is smaller than the sum of the sizes of the attraction memories connected to that bus, in that each item is guaranteed a space, possibly by throwing shared and/or foreign items out. An attraction memory seeing a *go home* belonging to its home bus will try to be selected with the following priority:

sel2→it has an empty space in the corresponding set,
sel1→it has a shared item in the corresponding set, and
sel0→it has an item not belonging to the subsystem in the corresponding set.

A replacement might take place to make space for the home-coming item. A subsystem with the item in a transient state does not try to be selected. This will create a rare situation in which none of the subsystems try to be selected in support of the *go home* transaction, which will turn upwards again. It will be returned to the home bus soon again. The *go home* transaction is thus buffered in the IB and OB buffers for these rare situations. The preferred location, as described, is different from the memory location of NUMAs in that the item only goes there after failing to find space elsewhere. When the item is not there, its place can be used by other items.

5.5.2 Replacement in a Directory

Baer and Wang have studied the multilevel inclusion property [BW88] with the following implications for our system: a directory at level $i+1$ must 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.

$$\begin{aligned} size(Dir_{i+1}) &= B_i * size(Dir_i) \\ associativity(Dir_{i+1}) &= B_i * associativity(Dir_i) \end{aligned}$$

Even if implementable, higher level memories become expensive and slow if those properties are fulfilled for large hierarchical systems. The effects of the multilevel inclusion property are limited in the DDM, however, since it only stores state information in its directories and does not replicate data at higher levels. Yet another way to limit the effect is to use directories with smaller sets (less number of ways) than what is required for multilevel inclusion. These so-called imperfect directories can be endowed with the ability to perform replacement. The probability of replacement can be kept at a reasonable level by increasing associativity moderately higher up in the hierarchy. A higher degree of sharing also helps keep probability low. A shared item occupies space in many attraction memories, but only one space in the directories above them. Directory replacement can be implemented in the DDM by an extension to the existing protocol, which requires one extra state and two extra transactions [HHW90].

A drawback of directory replacement is the problem of finding a good replacement strategy. The least recently used algorithm (LRU) would not work. The directories simply do not monitor which items are used by the processors. Random replacement is probably the best one can do. The effects of directory replacement have yet to be studied.

5.5.3 Implicit State in a Directory

The directories, as described earlier, store state information about all the items residing in their subsystems, with the most common states being Exclusive and Shared. The absence of state information for an item is interpreted as the Invalid state. Having Invalid as the implicit state saves space, since most items are expected to be in that state, and subsequently will not occupy any state memory. Introducing a preferred location opens up the possibility of saving more memory.

If an item most commonly resides in its home and nowhere else, the implicit state for all items in the home item space of a directory should be Exclusive. The implicit state for items outside the home item space is still Invalid.

This does not change the ordinary protocol, only the representation of the states. Subsequently, only items residing outside their subsystems will need entries in the directory, which will drastically reduce its size. The size of the directory decides how many foreign items should be allowed in the subsystem and how many items should be allowed to move out. Freeing up space in the directory is equivalent to bringing an item home or throwing a foreigner out. The technique is only practical to a limited extent, however, since having too small directories prevents sharing and migration, resulting in drawbacks similar to those of NUMAs.

5.6 Handling the Memory System

An architecture with only caches and no physical memory may have a "physical" address space larger than the sum of the memories in the machine. This is appealing to applications where a large "name space" can be beneficial. With a large name space, the need for garbage collection can be suppressed, since garbage will eventually be sent to secondary storage, and not occupy any primary memory resources. We call such a memory system a *large address space* system. The implementation of such a system must include some *item reservoir* with some backing storage where all the overflow items turn.

The intention of the DDM is to produce a general-purpose architecture that looks like a shared-memory machine to the user in order to suit many of the existing programs and operating systems. This also requires some traditional aspects of a memory system to be implemented, e.g., virtual memory and protection. A *limited address space* is more appropriate here where the address space is smaller or equal to the amount of memory in the system. In order to allow for effective *direct memory access* (DMA), to/from input/output devices, some extra overhead is needed in a COMA.

5.6.1 Virtual and Physical Memory

Virtual memory is used for many reasons. It allows for a larger data set than the physical memory available. It also allows for multiple address spaces and memory protection. Two processes, possibly located on the same processor, may use identical virtual addresses for accessing their own private address spaces. These virtual address spaces may be mapped to disjoint physical addresses, or they may partially overlap. This behavior of virtual addresses causes difficulties for coherence protocols. Instead, the translation from virtual to physical addresses is often performed by a memory management unit (MMU), located between the processor and the coherent cache. The operating system maps pages of virtual memory to pages of physical memory. Address translation is kept in tables in global memory. The MMU translates a virtual address to a physical address through a lookup in the global translation table, but may also cache translation information locally, thus avoiding accesses to global memory. During execution, the page might get unmapped (paged out) and remapped (paged in) to a new physical page. The operating system keeps a free list of physical pages not in use.

A COMA does not need all the functionality of the mapping of virtual addresses to physical addresses, but does require that the addresses be unique, i.e., a translation from multiple address spaces to one unique address space. This is part of the functionality implemented in an MMU, so MMUs could be used to translate to unique addresses.

5.6.2 Limited Address Space by DMA Nodes

Trying to allocate more virtual memory than physical memory in a machine introduces the need for secondary storage. Dedicated nodes interface the diffusion world to the sequential world. They allow for DMA transfers of contiguous sequential parts of the item space and are subsequently called *DMA nodes*.

From the DDM bus the DMA node looks like any attraction memory, containing a protocol similar to that of the AMs. From the other side, pages can be DMAed to and from its memory. The memory of the DMA node is organized like fully associative page frames.

When the operating system decides to send a page to secondary storage, it first invalidates its entries in the MMUs and then gives an order to the DMA node of the home bus to send it to secondary storage. The DMA node first allocates a page frame for that item page (physical page), then gets all the items into state E, and finally writes the page frame to disk and erases the page frame.

Trying to access a virtual page that has been paged out reverses the process. Initially, a page frame is allocated for the item page (physical page), and the page is read to the frame from the disk. Secondly, the virtual page is mapped to the item page in page tables. Thirdly, the process might start accessing that page by using the DDM protocol. Items are attracted to the attraction memories of the processors using it. To reuse the page frame later for paging a new page in or out, any remaining items in the frame—not yet attracted by any attraction memory—must be forced into the system by *inject* transactions.

5.6.3 Distributed Limited Address Space

A more appealing strategy is *distributed limited address space*. Instead of some dedicated DMA node per home bus, any or many of the nodes may host secondary storage.

When the operating system decides to send a page to secondary storage, it first invalidates all translation entries of the page in all the MMUs, and then gives an order to a selected node to send the page to its secondary storage. The selected node first gets all the items into the new state Exclusive-immune to make them immune to replacement, possibly by replacing other items. Then, it starts the DMA transfer to disk. Each item read by the DMA will automatically be invalidated, freeing up space again.

Trying to access a paged-out page reverses the process. The operating system orders the node holding the page on its disk to write the page to a selected item page. The node first allocates space in its attraction memory by putting the items of the selected item page in the state Exclusive-immune, possibly by replacing other items.

Then, it starts the DMA transfer from disk, changing states to Exclusive. Finally, the virtual page is mapped to the item page by the operating system.

5.6.4 Birth of New Items

Upon receiving a request to create a new virtual page, the operating system takes a page from the free list and maps the virtual page to it. If the virtual page corresponds to a page on disk, the schemes described above will bring the contents of that page into the machine. If no page corresponds to the virtual page, none of the items are yet in the machine. The first access to each item on that page will return a transaction *new item* as a response, interpreted as a zero-filled item. Thus, a new item is born the moment it is first accessed, such as when a stack grows. The data value of a new item is by definition zero, making a compact representation for sparse arrays and eliminating the need to zero-fill pages.

⋆ ⋆ ★ ⋆ ⋆

The Data Diffusion Machine is a hierarchical COMA based on buses with directories between each level in the hierarchy. The buses are of split-transaction type and released between a request and its reply. Its cache-coherence protocol of write-invalidate type. The DDM protocol also handles attraction of data and replacement. The directories between the levels in the hierarchy serves as filters, and only requests not satisfied otherwise are transferred through the directories.

We described the DDM protocol, the DDM bus and how to implement a limited address space. We also described some possible optimizations in the directory structure.

6

DDM Prototype Implementation

BUILDING a prototype is time-consuming, but gives invaluable insights to where the real difficulties of parallel architectures lie. The real behavior of a large parallel architecture can never be simulated. Nor can real questions about the operating system be understood. Our development effort has been reduced by taking a commercial product as a starting point.

The hardware implementation of the processor/attraction memory is based on the TP881V system by Tadpole Technology, U.K. Each system has up to 32 Mbytes of data memory, and up to four Motorola MC88100 20 MHz processors, each with 64 kbytes instruction and 64 kbytes data cache.

A single VME-sized board is being designed at SICS, the DDM Node Controller (DNC). It interfaces the Tadpole node to the DDM bus. The DNC contains protocol state machines and memory for address tags and state code. It turns the data memory into the data part of the attraction memory. The DDM bus is implemented with the electrical specification of the Futurebus in the rack backplane.

6.1 The Motorola 88000 Family

The processor of the DDM prototype was chosen for the functionality of its caches. The processor cache of the Motorola 88000 family simplifies the design of a large second-level cache, like the AM. It also has an MMU well suited to multiprocessors. We decided we would be able to produce a DDM prototype faster and with less design effort using this family than with any other commercial processor available at the time of the decision.

The Motorola MC88100 CPU is a RISC processor with Harvard architecture; i.e., the CPU has two separate buses: one for data, and one for instructions [Mot89a]. Both buses, called P buses, have identical characteristics. The processor is referred to as a 15–17 VAX-MIPS machine; this, of course, heavily depends on how fast the memory system is and on what programs are run. There is a combined cache and memory management unit (CMMU) chip MC88204 between the P buses and a memory bus called M bus.

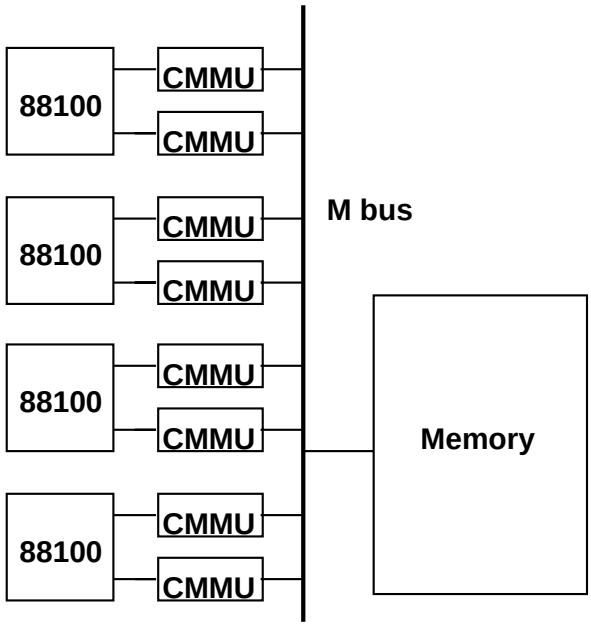


Figure 6.1: Multiprocessor configuration of the 88000 family.

The CMMU is supplied with a copy-back cache-coherence protocol similar to the write-once protocol discussed earlier (Figure 3.2). It allows many CMMUs to be connected to the same memory, as shown in Figure 6.1. A centralized arbiter selects one CMMU at a time to be the master of the M bus. CMMUs that are not masters are called slaves. Slaves can snoop the transactions on the bus. If any of the slaves sees a transaction on the bus that interferes with its private updated copy of a datum, it gets a snoop hit. The CMMU that got the snoop hit stops the master by asserting a retry signal on the bus (marked "intercepted" in Figure 3.2). 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 got the snoop hit is granted the bus and can update the memory during the next cycle.

An input pin on each CMMU can turn off the snooping. This is useful to the instruction cache. A page can also be declared as being local instead of global. A control bit, set by the master, will tell the slaves whether a transaction on the M bus is global (has to be snooped) or local (no snooping necessary). The bus carries the following transactions: *read, write*, and *read with intent to modify*. The transactions can be for a single byte, or the whole cache line. The master can "lock" the bus and transfer several transactions during one tenure. An atomic operation, *xmem*, exchanges the contents of one register and a memory word.

The MMU of the MC88204 contains a *page address translation cache* (PATC) with 56 page-translation entries for 4 kbytes-sized pages and a *block address translation cache* (BATC) of six entries, holding a translation for 512 kbytes-sized blocks [Mot89]. The page address translation cache uses the FIFO replacement strategy. Entries in the address translation caches can be invalidated on a page or segment basis—or the whole address-translation cache can be invalidated by the processor writing to a control word. The control word is also mapped into the address space of the M bus and may be accessed by the other processors.

6.2 The 88000 Family and the DDM

In our implementation, the rest of the DDM looks like yet another CMMU to the local M bus of a cluster. A miss in the local attraction memory looks like a coherence action from that CMMU. The key to the design is the retry signal of the M bus, used to integrate the coherence protocols of the CMMU and the protocol of the attraction memory.

6.2.1 Interfacing the DDM and the M bus

The DDM Node Controller (DNC) contains the *memory below protocol* (MBP) and an associative state memory (ASM). The combined functionality of an 88000 multiprocessor configuration and the DNC can be found in Figure 6.2. The MBP checks each transaction on the bus for validity. If it is a *read* of an Invalid item, for example, the DNC asserts the retry signal. The retry signal makes the current bus master stop and release the bus, while the MBP initiates necessary actions. The DDM system will then start to retrieve the requested item; meanwhile, the DNC that asked for the value will not be granted the bus. Only after the item is retrieved, and the DNC has written it to memory, can the CMMU be granted the bus and redo its transaction.

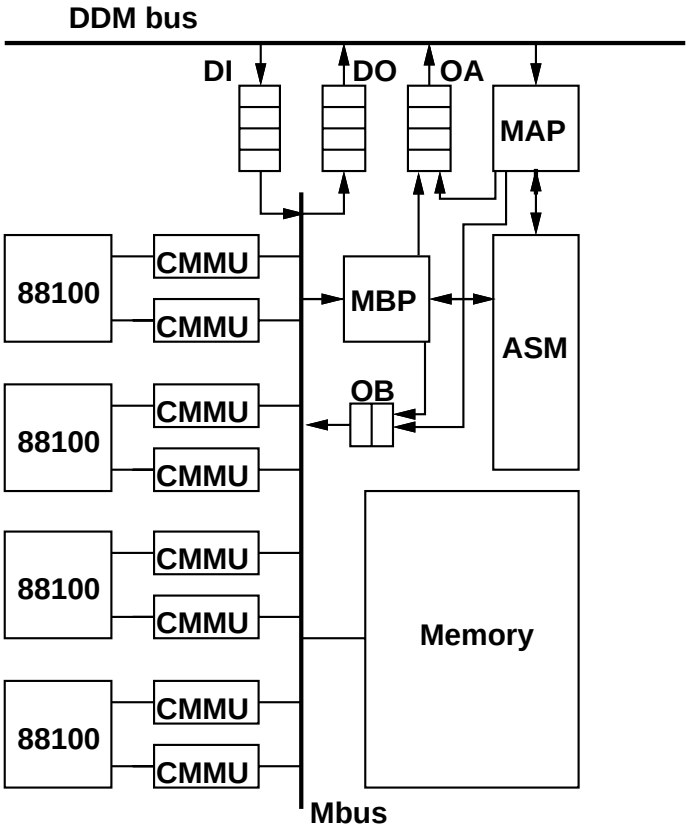


Figure 6.2: DDM implementation based on the 88000 family.

The DNC also hosts the *memory above protocol* (MAP) and 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) and *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* in the OB. Data is written from DI to the node's memory by putting a *write line* in the OB.

6.2.2 Implementing the DDM Protocol

There are several ways an AM and its protocol can be implemented based on the functionality of the retry signal. We will start by looking at 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 sequen-

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

A *read* request on the M bus is snooped by the memory below protocol, which keeps asserting wait states on the bus until the transaction has been approved by the protocol. 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. If the transaction is approved, the MBP simply drops its wait signal, allowing for the transaction to be completed. 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 order 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:

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. During the time from its first request for the item until the data are returned on the M bus, the requesting CMMU is blocked from arbitrating for the M bus. When the CMMU can repeat its request again, it will not be interrupted by the memory below protocol.

A write transaction on the M bus to an item in an inappropriate state is intercepted in a similar way by the memory below protocol, and necessary actions are taken before the CMMU is released to arbitrate for the bus again. So, from the viewpoint of the CMMUs, the DNC and the rest of the DDM looks like yet another CMMU, only slower and noisier.

If the node sees a *read* request on the DDM bus, for which it has the item in state Exclusive, the node should change state to Shared and send a *data* transaction on the DDM bus according to the DDM protocol. The memory above protocol, thus :

1. puts a *read line* transaction in the OB,

2. puts a *data* transaction in the OA–its item identifier only—and,

3. changes the item's state to Shared.

The OA will not arbitrate for the DDM bus before the data out (DO) contains data. The data is put in the DO by the OB performing its *read line* for the item on the M bus. Note that performing a *read line* on the M bus gives all the CMMUs a chance of snooping the transactions. If a CMMU has the item in cache state Dirty, it can assert the retry signal and force the OB to redo its *read line*. Before it has a chance of doing so, the dirty CMMU updates the memory and changes cache state to Valid. Now the OB can redo its read line without any complaints from any CMMU.

The snooping facility of the CMMUs is also be used when a MAP snoops an *erase* transaction on the DDM bus, for which it has the item in state Shared. It not only changes the ASM state to Invalid, but also transmits a dummy *write word* on the M bus to make sure that all copies of the item are erased from the CMMUs.

A complete specification of the protocol as implemented in the prototype and in the performance simulator can be found in Appendix B. Real-world problems have forced us to make some modifications to the protocol explained in Chapter 5. The biggest differences are the four new states:

EW	Exclusive Writing	The awaited write acknowledge has arrived, but the item has still not been modified.
EA	Exclusive Answering	A read request was received while in state EW.
WP	Waiting Prefetcher	Corresponds to state W, but this write request was initiated by the prefetcher.
RWP	Reading and Waiting Prefetcher	Corresponds to state RW, but this write request was initiated by the prefetcher.

The reason for the two first states is to avoid write bouncing. We found cases in our simulator where two nodes trying to write to the same item resulted in a large amount of transactions between the two nodes, but none succeeded with a single write. When a node received the acknowledge for its write request, a *read* request from the other node arrived before the writing was performed. When finally the CMMU tried to write to the item, the item's state had already changed to Shared, and the writing was prohibited. This protocol deficiency was found through simulations. The first and simplified simulator of the DDM was not detailed enough to model this behavior. It took a second, and very detailed, simulator [Löf90], to monitor the write bouncing. This implementation-specific problem exemplifies the difficulties of formal verification of a protocol, unless the formal proof includes implementation details.

To avoid the write bouncing, we introduced the new state Exclusive Writing, to which an item is changed on a write acknowledge. When the CMMU writes the first time, the item changes state to Exclusive. If a *read* request is received before the first write is performed, the request for the item is remembered by the new state Exclusive Answering, and the reply sent after the first successful write has been performed by the node. With this mechanism, we guarantee progress in that at least one write will be performed each time the item is moved.

The other two new states are related to the prefetcher discussed in Chapter 8. If a write is stimulated by the prefetcher, it would be fatal to require a write before the item was released from the node. The prefetching might have been speculative, and the item never written to by any CMMU, resulting in deadlock. The protocol is also extended with a new bus signal, *has data* (hd), asserted on the DDM bus by any subsystem which has a data copy of the item in its AM and/or OA.

6.2.3 A Direct-Mapped Attraction Memory

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 the transaction will succeed and start the *read line* before approval from the MBP is received. A processor cache 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.

It seems that the latency for accessing the ASM cannot be hidden on a write, since overwriting parts of another item would be fatal. There is, however, full inclusion between a processor's (data) cache and its AM; in other words, there can be no copy of an item in the processor's cache unless there is also a copy of the item in the AM. This, together with the fact that a *write* to memory (AM) is never performed by the CMMU unless it already contains a copy of the item [Mot89], hides the ASM access—even from write accesses. Accesses from the DNC to the AM can also be performed without additional delay on the M bus, since the state of the item is checked before the access starts, e.g., the erase of a Shared item by the *write word* on the M bus.

A direct-mapped cache is advantageous over a multiway associative implementation for shortening access time to the AM, but it also increases conflict misses. More associativity is expected to increase the hit rate in the AM. One can imagine situations for which a directly mapped attraction memory could be fatal for performance, for example mapping code segment of a tight loop and its data to the same location

in the attraction memory. However, if code pages are handled the right way by the operating system, instruction caches need not be coherent, i.e., the instructions of that tight loop can still reside in the instruction cache while they are being pushed out of the attraction memory by the data. Still, mapping two data areas "on top of" each other could be fatal.

Another drawback of a directly-mapped attraction memory is its limitation for replication of popular items. In order for one item to get replicated in all attraction memories, no other item for the same set of the attraction memory can be present in the machine, i.e., the item space cannot be larger than the size of one attraction memory. For two-way attraction memories, the item space can be $0.5 * \sum AM\ size$, and for four-way attraction memories, the item space can be $0.75 * \sum AM\ size$.

6.2.4 Set-Associative Attraction Memory

It is possible to implement a set-associative memory with almost no extra access latency for read accesses. If AM organization is multiway set-associative, part of the address of the physical memory location in the data part of the AM must be generated based on the result of a multiway comparison in the ASM. This means putting the ASM lookup on the critical path.

We propose a set-associative AM implementation that is optimized for the read access. The algorithm used is the MRU algorithm—guessing that the entry last used in the set will be the one asked for next time as well. There is no fixed location in the set for the MRU entry; instead, a fast last-accessed memory (LAM) is added. It contains one pointer of $\log_2(ways)$ bits per set, pointing to the entry last accessed in the set. The contents of the LAM are used as part of the address to start the read access immediately.

If the AM's data part is built of DRAMs, access to the LAM can be hidden. Half the DRAM address[1] is not needed during the first access cycle. By putting the LAM pointer in that part of the address, no extra delay is introduced. The comparisons of the address tags in the ASM are started in parallel with the data access. The comparison tells which set—if any—contains the item and if the item is in the correct state. If the LAM guess turns out to be the right one, no further action is taken. If another entry contains the right address tag, a retry signal is asserted, and the LAM is updated. The same transaction will then be restarted, but with the correct LAM pointer.

The LAM can also be used to implement a better replacement strategy. It contains enough information to implement the least-recently-used replacement algorithm (LRU) for a two-way cache. For an AM with greater associativity, it can be used to implement the not-most-recently-used (NMRU) algorithm. When the M bus

[1]The column-access strobe (CAS) part.

above erases an entry in the AM, the LAM should be made to point to a different way (instead of pointing to an item in Invalid).

During our simulation studies, the hit rate of the LAM guesses is is high. However, we have not yet studied data sets of a realistic size, which disables any conclusions from being drawn. The positive effects of returning the MRU data first in large multiway caches have been studied by Chang et al. [CCS87].

6.3 Prototype Implementation

Attempting 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 lacked the possibility of connecting yet another master board to the M bus. Tadpole Technology, U.K., had a design that suited us.

6.3.1 Building on TP881V

All cards in the Tadpole design TP881V have an interboard bus connector that carries a proprietary M bus. Many cards can be stacked together to build a complete system. One card, the base module, is connected to the VME backplane. The TP881V is divided into the following cards:

- The base module contains interfaces among the M bus and Ethernet, VME bus, two SCSI buses, and four TTY interfaces. The board also contains EPROM and timers.
- The processor module contains up to 32 Mbytes DRAM and a Hypermodule containing MC88100 and CMMUs. Hypermodules can host from one to four MC88100 processors.
- The memory module can host up to 64 Mbytes DRAM. This card is not directly needed in the DDM design, but could be useful in future upgrades.

Tadpole made some modifications to its product before shipping it to us. They mounted most of the programmable PAL components on sockets. This allows us to replace the PALs and change address mapping, DRAM timing, and arbitration. They also disconnected the VME signals from the backplane, freeing up the backplane for the DDM bus. In order to avoid large redesigns, we limited our changes to reprogramming the PALs and avoided adding extra connections and circuits. Thus, we have been unable to implement the functionality of disallowing a requesting CMMU from arbitrating for the bus until its reply comes back from the DDM bus.

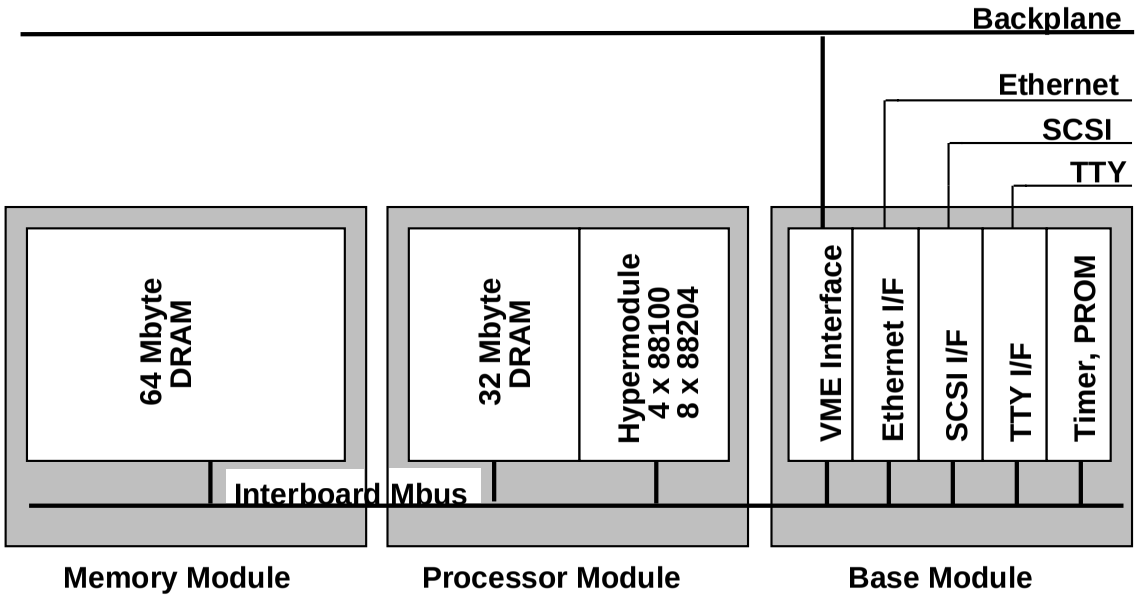


Figure 6.3: Basic TP881V system from Tadpole

Instead, we have modified the arbitration algorithm from a prioritized scheme to a round-robin scheme. This enables other CMMUs to get the bus between successive accesses from a CMMU waiting for a DDM reply to come back. We estimate this approximation of the arbitration procedure to account for a performance drop in the five-percent range.

6.3.2 A Pipelined DDM Bus Implementation

The split transaction bus makes pipelined implementation straightforward. The pipelining of the prototype's DDM bus was proposed by Anders Landin [Lan92]. The pipeline can be divided into several distinct phases. If the different phases use different resources they might very well overlap. A transaction is divided into five phases: Arbitration, Transaction, Lookup, Selection, and Data, as shown in Table 6.1.

Phase:	1	2	3	4	5	6	7	8	9
Transaction 1	Arb	Trans	Lookup	Sel	Data	Data			
Transaction 2			Arb	Trans	Lookup	Sel	Data	Data	
Transaction 3					Arb	Trans	Lookup	Sel	Data
Transaction 4							Arb	Trans	Lookup

Table 6.1: An example of a pipelined DDM bus

During the first phase, the sender of Transaction 1 asserts its arbitration line and gets the grant back. During the next phase, it puts the transaction code and the

item identifier on the bus. All nodes read the current state of this item and do a protocol lookup during the Lookup phase. During the Selection phase the nodes try to get selected according to the result of the protocol lookup. At the end of this phase, receivers are determined. During the (optional) Data phases, data is transferred from the FIFO of the sending node to FIFOs of the receiving nodes. The Data phases are only needed if the transaction carries data. In order to allow the Data phases of Transaction 1 to occur at the same time as the Trans phase of Transaction 3, the data and address of a transaction must be carried on separate lines; i.e., a nonmultiplexed bus. If a subsystem knows that it will be selected during the Selection phase, it can forward a new transaction at the end of the Lookup phase; otherwise, it must wait until the end of the Selection phase before forwarding the transaction—if it was selected.

We decided on a conservative bus design initially, since high 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. Each phase takes 100 ns. We use the drivers developed for the Future bus for all parallel signals, like 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, which is enough for connecting up to eight nodes, i.e., up to 32 processors.

6.3.3 The DDM Node Controller

The organization of the DDM node based on the TP881V system can be found in Figure 6.4. The DDM Node Controller (DNC) [Lan92] is a heavily integrated and pipelined implementation. We use Xilinx chips for implementing the data paths and Mach chips from AMD for implementing the control logic. The design integrates the functionality of MAP and MBP in one pipeline design. The ASM is a 60 ns single-ported SRAM. The choice of SRAM, rater that DRAM, simplifies the implementation. Its relatively fast access time is motivated by the modest implementation technology used elsewhere (Xilinx), consuming much of the available access time. The ASM is interleaved between lookups from MAP and MBP in such a way that each one can do a lookup every fourth cycle. This lookup frequency matches the transaction frequency of the DDM bus. Transactions on the M bus take between eight and twelve cycles. The lookup bandwidth required by the M bus is much lower than its available bandwidth. All the write-back operations to the ASM are mapped into M bus slots, so M bus transactions might sometimes be delayed four cycles. For simplicity, we decided to make the very first implementation of the attraction memory direct mapped, as described in Section 6.2.3. The implementation details of the DNC lie outside the scope of this dissertation [LH91].

The TP881V has a small part of its address space allocated to control registers. The control registers of the device circuits, the interrupt circuit, and the control registers

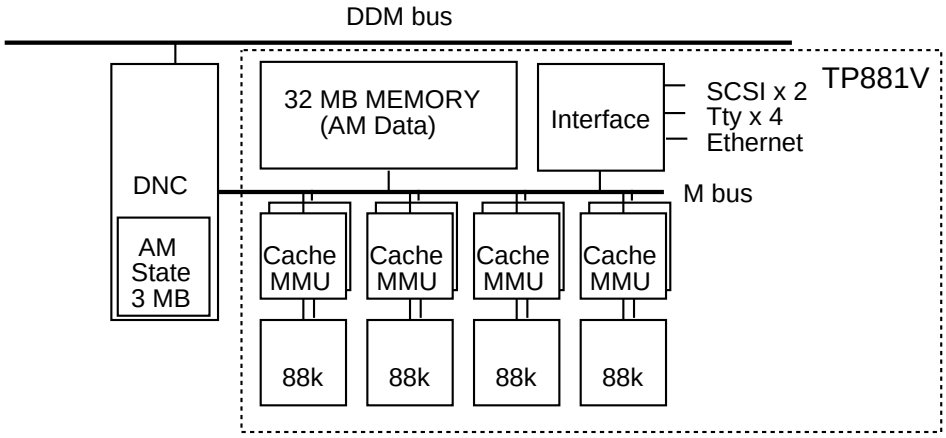


Figure 6.4: The implementation of a DDM node consisting of four processors sharing one attraction memory.

of the CMMU are mapped into this space. The original system uses this address space for sending software interrupts between the processors, and for managing the address-translation caches of the CMMUs. The DNC maps the control spaces of all the other nodes into a special "non-attraction" part of each node's address space, allowing for remote write accesses. This entitles any processor to send a software interrupt to any other processor. It also enables any processor to invalidate the contents of the address translation cache of any other processor, helpful for "TLB coherence," i.e., the coherence actions involving the MMUs necessary when a page table entry is modified.

The protocol in the DNC has one more optimization than the protocol described earlier. The device sending an erase to the topmost bus can send a write acknowledge to its subsystem immediately, removing the need to involve the top. This speeds up the write response for sequential consistency.

A DDM node consists of three VME-sized cards: a processor module including 32 Mbytes DRAM, a base module card with SCSI and Ethernet interfaces, and the DNC card. We can fit six DDM nodes, power, and directory into a VME rack with 21 slots. The directories are yet not designed, but could be implemented using DNCs with small modifications. Figure 6.5 shows a possible packing technology for a large DDM. 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.

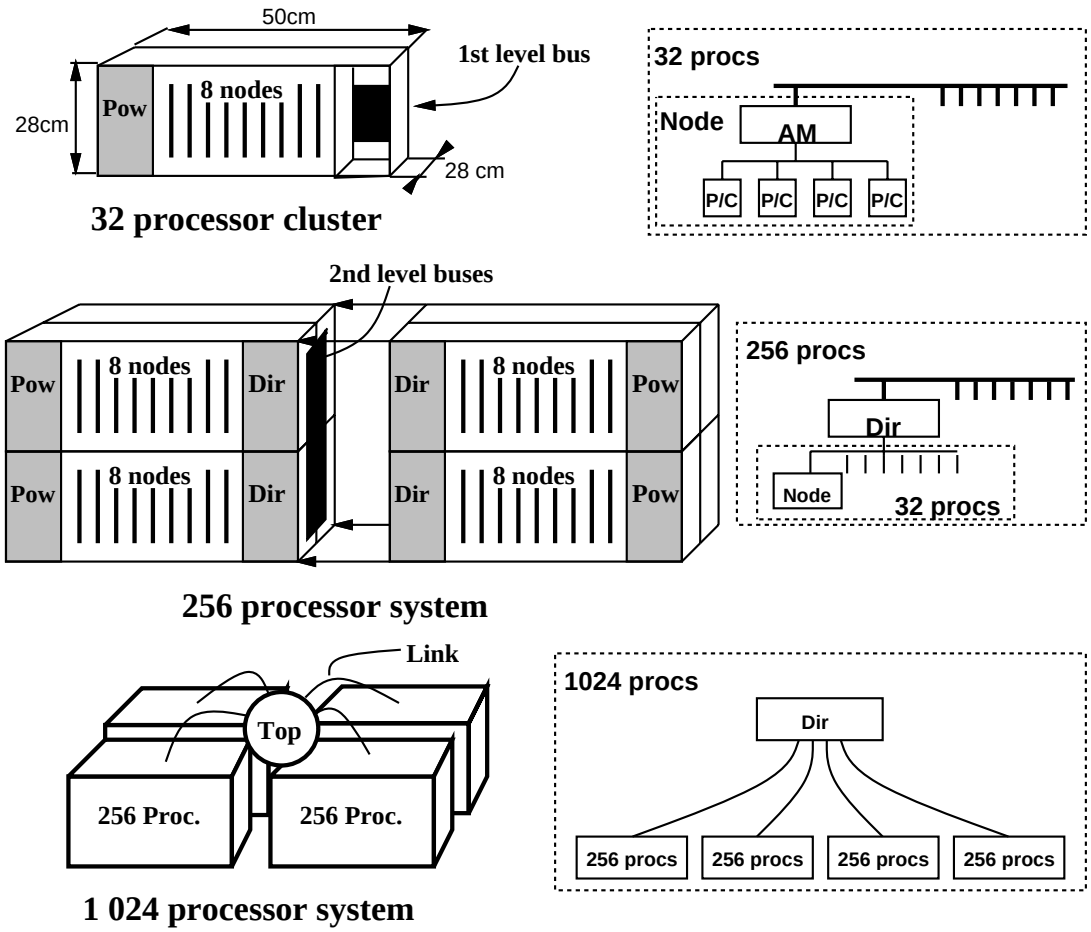


Figure 6.5: The packing technology used for the DDM prototype, and its possible extension.

6.3.4 The Performance of the DDM Node

Read accesses to the attraction memory take eight cycles per cache line—one more than the original TP881V system. Write accesses to the attraction memory take twelve cycles compared to ten cycles in the original system. A read/write mix of 3/1 to the attraction memory results in access time to the attraction memory being on the average 16 percent slower than the original TP881V memory.

Since the directory has yet not been designed, its latency is estimated based on the numbers from the DNC design. A remote read to a node on the same DDM bus takes 60 cycles at best, most of which are spent making M bus transactions (a total of four accesses). Read accesses climbing one step up and down the hierarchy add about 55 extra cycles. Write accesses to state S take at best 35 cycles for one level

CPU access	State in AM	Delay, one level (cycles)	Delay, two levels (cycles)
read	I	60	115
write	S	35	60
write	I	70	145

Table 6.2: The remote latency in the prototype DDM.

and 60 cycles for two levels, as shown in Table 6.2. To these numbers must be added an extra latency of four cycles for going through the processor caches.

6.4 Memory Overhead

Extra memory is required to store state bits and address keys for the set-associative attraction memories, as well as for the directories. We have calculated the extra bits needed if all items reside only in one copy (worst case). An item size of 128 bits is assumed; it is the cache-line size of the Motorola MC88204.

A 32-processor DDM, i.e., a one-level DDM with a maximum of eight direct-mapped attraction memories, needs three bits of address tag per item, regardless of attraction memory size. As stated before, item space is no larger than the sum of the sizes of the attraction memories. Each attraction memory is one eighth of the item space. There are eight items that could reside in one set. Three bits are needed to tell them apart. Each item also needs four bits of state. An item size of 128 bits gives an overhead of (3+4)/128 = 5 percent.

By adding another layer with eight 8-way set-associative directories, the maximum number of processors comes to 256. The size of the directories is the sum of the sizes of the attraction memories in their subsystems. A directory entry consists of six bits for the address tag and four bits of state per item, using a 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: six bits of address tag and four bits of state. The total overhead per item is (6+4+6+4)/128 = 16 percent. A larger item size would, of course, decrease these overheads.

⋆ ⋆ ★ ⋆ ⋆

The DDM prototype is near its completion at the Swedish Institute of Computer Science. The prototype DDM is built with commercially available MC88100 processors. The design ties the copy-back caches and instruction caches of four processors to an attraction memory sized 32 Mbytes. We base our design on a computer system by Tadpole Technology to cut the design effort. A small overhead in access time and memory size is observed in the design.

7

Simulated Performance of the DDM Prototype

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. An attempt in that direction was presented in the quantitative study in Chapter 4. Rather than cover a large design space, which was the driving force of the quantitative study, this chapter presents a simulation study aimed at accuracy and detail for one specific architecture—the prototype DDM.

We developed an execution-driven simulation environment, which can run parallel programs written in C. A detailed DDM simulator describes the prototype DDM. It allows us to study the behavior of the prototype and to collect statistics one cannot gather in a real implementation. However, it has a slowdown of approximately 1000 times, which limits the problem size of the studied programs. Nor does it take the effect of the operating system into account, which is another major drawback.

The simulation of the DDM shows encouraging behavior for the studied programs. We present our simulation method, the performance of the DDM, and some internal dynamic statistics. We also study the effects when the data set is changed and the communication locality explored in the SPLASH programs. Finally, the behavior of the TLB accesses on a COMA is studied.

7.1 Simulation Technique

In order to produce interesting simulation results, the study of a parallel architecture should involve the following components:

1. Efficiency, allowing for long runs and large working sets.
2. Large real applications.
3. A realistic model of the architecture.
4. Efficient and flexible analyzing tools.

The DDM simulator has gone through three distinct phases. A first trace-driven simulator, implemented in Prolog, described and debugged a protocol similar to the one described in Chapter 5. This simulator was based on hierarchical time control. Based on the same time control, a second, more efficient and more detailed, trace-driven simulator was developed in C++. It modeled the prototype DDM as described in Chapter 6. As a third step, this simulator was turned into an execution-driven simulator.

7.1.1 Modeling the DDM

Controlling time can be time-consuming in a simulation of a parallel architecture. One method is to advance each processor one cycle at a time, another method is to order events in some event queue. Both methods can lead to high overhead in time control. Instead, our simulator is built using a hierarchical time control, and explored the hierarchical structure of the DDM architecture in the time control. The simulator is driven from the top of the hierarchy. The OA buffers underneath the top bus first arbitrate for the bus. If any of them wants to send a transaction, the time on the top bus is advanced with the time the bus is occupied for sending that transaction. Otherwise, the time is advanced just one time step. During that time, none of the subsystems can talk to each other, and therefore their time can individually be advanced the same amount of time. This hierarchical time control method is used throughout the hierarchy all the way down to the roots. It allows for time to advance in larger steps at the roots.

The C++ simulator [Löf90] is parameterized with data from our ongoing prototype project, and accurately describes its behavior. It has partly been used for performance evaluation, and partly for protocol debugging. The M bus and the protocols of the CMMU are modeled according to the user manual [Mot89]. The page-address translation cache, corresponding to a translation-lookaside buffer (TLB), is modeled

with 56 entries. On a TLB miss, all necessary transactions are sent on the M bus. A page-translation descriptor is retrieved from a faked page handler, implemented as a C++ object. The page handler has a scrambled free list containing item-identifier pages not in use. The first time a virtual page is touched, a new page is allocated from the free list. 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 item is sent to a special AM, which makes all necessary transactions for returning the *data*.

7.1.2 Modeling the Processors

Traditionally, the simulation of multiprocessors has been trace-driven [EK89]. Files containing address traces from execution on a shared-memory architecture are produced by some tracing tool. The files are later used as inputs to an architectural simulator. Trace-driven simulation has three major drawbacks.

1. A multiprocessor with an appropriate amount of processors is needed to produce the traces, i.e., it takes a supercomputer to build one.

2. The files with address traces are huge. Modern processors can execute around 100 Minstructions and make 25 Mdata references per second. Simulating one multiprocessor consisting of 100 processors running for one second would require trace files of around 50 Gbytes (125 Maddresses * 4 bytes * 100 processors).

3. Trace-driven simulation of a multiprocessor system faces serious validity issues since it cannot represent the interaction of processes in the target architecture correctly [Bit90]. The relative speed between processors on the architecture generating the traces might differ by as much as 20 percent from that of the simulated target architecture.

Originally, we used Abstract Execution (AE) [Lar90], a modified GNU-CC compiler, to produce memory traces and adapted the trace-driven method. Programs compiled with AE write a minimum of information to a file when executing. We ran our parallel application as several parallel time-sliced processes on a SPARC station, and produced one file per process. The AE tool also produces an unpacking program, a *road map*, that turns the saved information into address traces, both data and instruction addresses. We used the road map to later unpack and feed the traces into the simulator. AE traces virtual addresses. We modified the unpacking program to translate from virtual addresses to unique addresses, a necessity for simulating a COMA. UNIX has a function called *mmap* to set up its shared memory. The translation tables were generated by trapping the mmap function.

Using AE to produce memory traces from the execution of parallel applications has several advantages. AE can be used on an ordinary SPARC station, i.e., it does not take a supercomputer to build one. AE only saves a small amount of information on the file, which reduces the problem of the large files by one order of magnitude. The validity problem of trace-driven simulation remains, however. Using the AE trace-method on a single workstation may magnify the error, since each process is run for a time slice at a time.

7.1.3 Execution-Driven Simulation

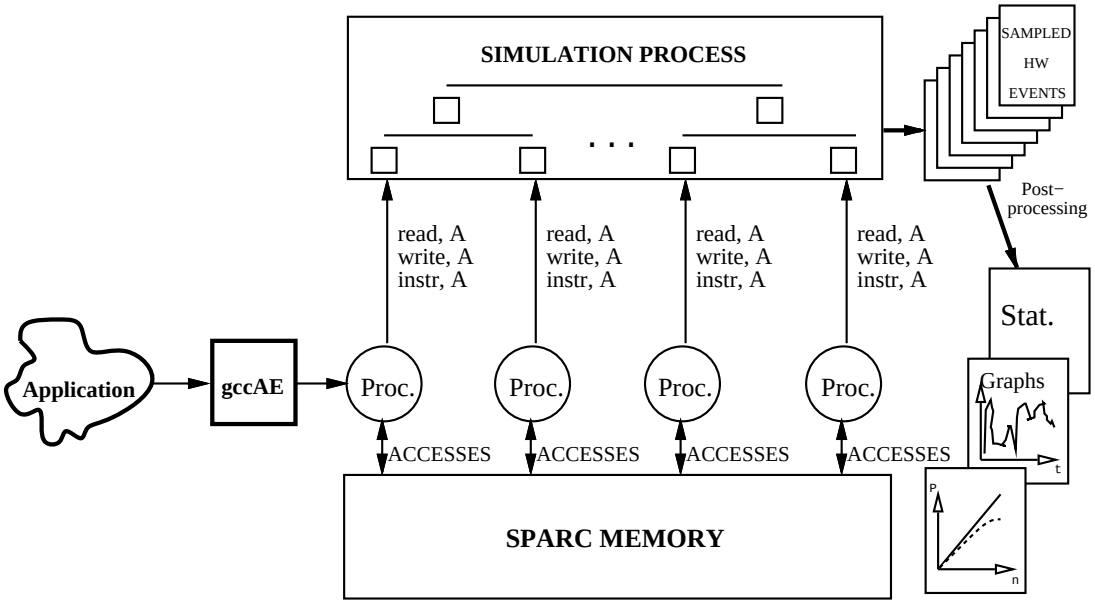


Figure 7.1: The structure of execution-driven simulation.

Inspired by the Tango simulator at Stanford [DGH90], we modified our trace-driven system to become execution driven. It models the parallel applications as if they were running on a real physical implementation of the architecture.

Instead of having each application process writing to a file, their trace information is immediately translated to unique addresses, and sent to a process simulating the DDM on the fly, as shown in Figure 7.1 The execution speed of each process is determined by how fast the information in its stream is consumed by the simulated DDM architecture, stalling the application process if necessary and making the relative execution speeds of the processes that of their execution on the DDM architecture. All simulation results reported here are produced using this method.

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 196 processors running programs of up to 2 CPU minutes simulated single-processor

time. The simulation currently runs the programs 1000 times slower than execution on a single SPARC station. The number of simulated processors has a small effect on the slowdown if the application simulated has an ideal speedup, which allows for large machines running large applications to be studied.

The sampled statistics are processed after the completion of the simulation runs. From each simulation run, several hundred plot files may be extracted, allowing for the dynamic internal behavior of the architecture to be studied. Only a fraction of the information obtained is presented here.

7.2 Simulating the Prototype

We studied the behavior of the DDM by running three of the programs used in the quantitative study (Chapter 4). We identified one of them, MP3D, as the toughest one for a COMA. The other two, Cholesky and Water, appeared midway through the applications in that study. 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.

7.2.1 Pitfalls

In Chapter 2 we discussed pitfalls to watch for when scalability and efficiency were calculated. Identical pitfalls exist when simulating an architecture. The simulation speed limits the simulation study to a small and often unrealistic problem size. Often, the data-set size per processor is comparable to the size of the first-level caches. If the number of processors is increased while the problem size is held constant, the data set per processor will decrease. This leads to the positive effect of increased hit rates in the small caches when the data set suddenly starts fitting into the caches. One can imagine applications for which the hit rate can change from zero percent to almost one hundred percent as the number of processors increases. By choosing parameters correctly, any architecture with small first-level caches can display an attractive speedup. Actually, superlinear speedup, i.e., efficiency greater than one, can be achieved.

If the problem size is small enough to fit in the processor cache of a single processor, the positive effect of an increased hit rate in the first-level caches will not be so large while increasing the number of processors. However, if the whole problem set fits in the small caches, COMA-like behavior can be seen even for non-COMA architectures.

A third problem arises if the problem size is too large. If the data set for the whole application cannot fit into the local memory of a node, the cost of remote accesses is paid when the application is run on a single processor. When the number of processors is increased, the data set per processor might suddenly fit in local memory. The result is similar to the positive caching effect. This may also result in superlinear speedup.

Ideally, problem size should be scaled to the number of processors, unless the application is run normally with a small data set. This can be done on real machines, but is harder to achieve on a simulator, since the simulation time for a large system would be too long. When simulating with constant problem size, care must be taken to identify—or avoid—these effects.

7.2.2 Simulation Setup

Here, three programs from Stanford Parallel Applications for Shared Memory (SPLASH) [SWG91], MP3D, Water, and Cholesky, and one matrix multiplication program are used. The SPLASH 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. In Chapter 4, we identified MP3D as the toughest one for a COMA which makes it interesting to study, 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.

The original versions of the programs are used. One of them, MP3D, was also studied in two versions that were rewritten to make better use of the data diffusion ability of a COMA. An additional application, Cholesky, was simulated using a modified hierarchical scheduler.

We 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 identify 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," should have made the DDM less sensitive to the small-cache effect. Actually, a better hit rate in the data caches resulted in a poor hit rate in the AMs, and vice versa.

The numbers 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]. The simplest protocol, used here, provides sequential consistency, as described in Appendix B. The tables in the appendix are, in fact, LaTeX output produced by the simulator.

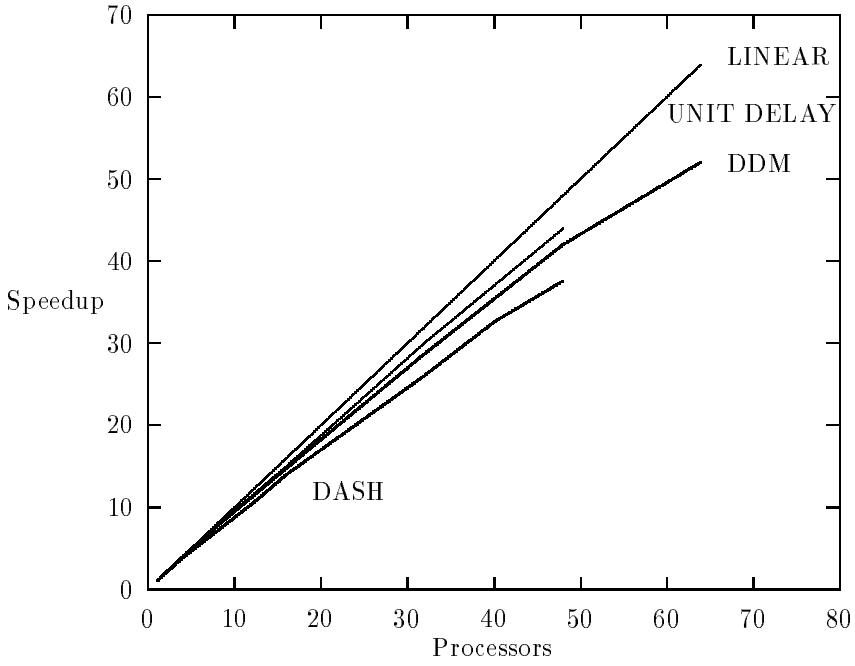
For configurations 1×1 and 4×1, the DDM network has not been simulated. Instead, a 100 percent hit rate in the AM is assumed. The speedups presented in the graphs are self-relative, i.e., compared to the execution time for 1×1. A hit is defined as a read or write that can be completed without stalling 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].

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, while the prototype implements a directly mapped attraction memory. DNC functionality is greatly simplified in the simulator. 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] of 16 processors 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. DASH was presented in more detail in Chapter 4.

7.2.3 Application Performance



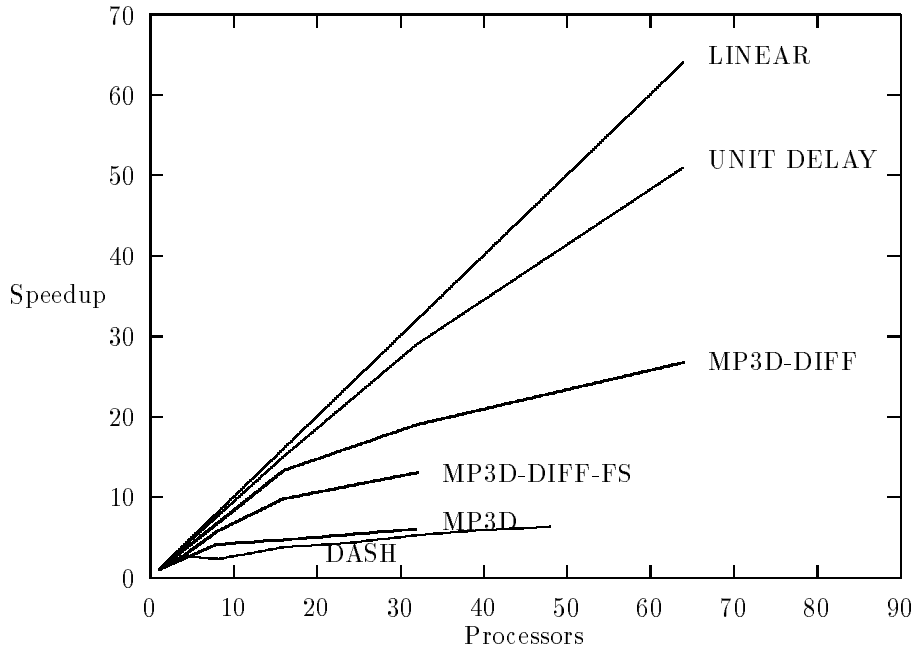
Data Set	192 molecules				384 mols.	
Topology	1×1	8×4	2×8×4	64×1	2×8×4	4 × 8 × 4
Hit rate Dcaches (%)	99	99	99	99	98.9	98.9
Hit rate in AM (%)	100	50	44	12	65	58
Node miss rate (%)	-	0.5	0.6	0.9	0.4	0.5
Busy rate:M bus (%)	2	21	31	32	26	37
Busy rate:DDM bus (%)	-	24	39	80	30	40
Busy rate:Top bus(%)	-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

Figure 7.2: 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.

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, which the AM can do nothing about. The speedup shown for WATER in Figure 7.2 is almost ideal. Some statistics are presented in Figure 7.2. 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.



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

Figure 7.3: 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 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 the move phase, resulting in 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 tables, we present the steady-state behavior of the last four simulation steps.

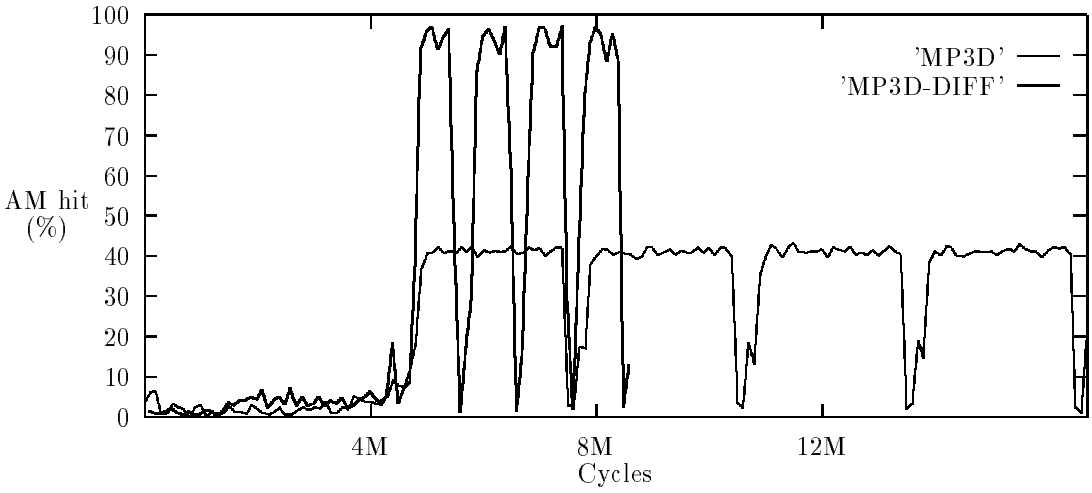


Figure 7.4: Dynamic behavior of the hit rate in one attraction memory over time for the original MP3D and the modified version MP3D-DIFF on the topology 2x8x2. The move phases of the last four simulation steps can easily be identified by their higher hit rates. The first step includes the cold-start 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).

MP3D-DIFF is a rewritten 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 the particle state and the space-cell state is now local to the processor. When a particle is moved across a processor border, its data are handled by a new processor; i.e., the particle

data diffuse to the attraction memory of the new processor. The rewriting adds some 30 extra lines.

The move phase is now optimized, since most operations are local to the processors. Only the diffusion of particles 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. 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 that is the dominant phase of the application 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 7.3, where all the different runs are compared. Figure 7.4 compares the hit rates in the AM of the MP3D and MP3D-DIFF. Figure 7.5 shows the bus-busy rate for MP3D and MP3D-DIFF. The MP3D clearly suffers from a serious bus-contention problem on the second-level bus of our prototype. By changing the algorithm, the communication rate drops, and the contention is gone.

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.

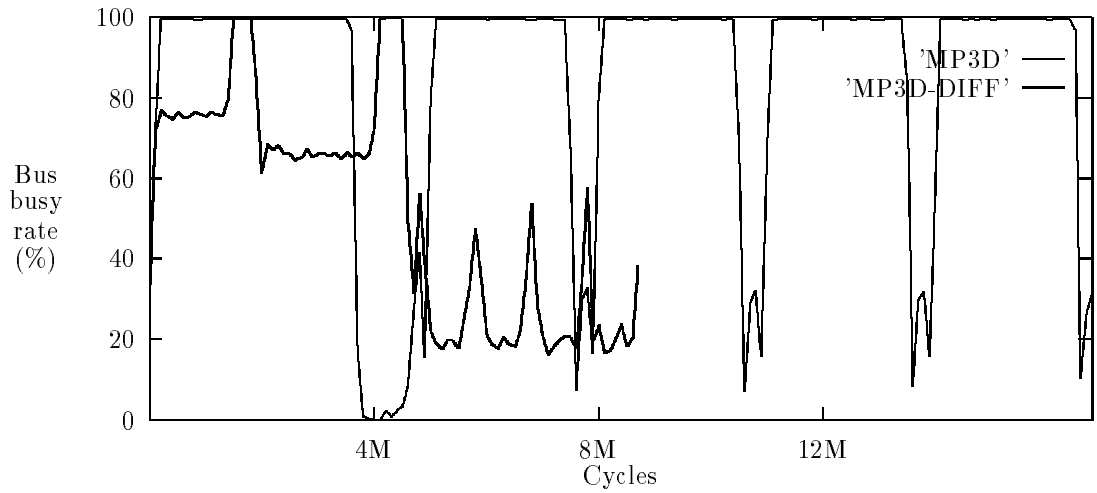


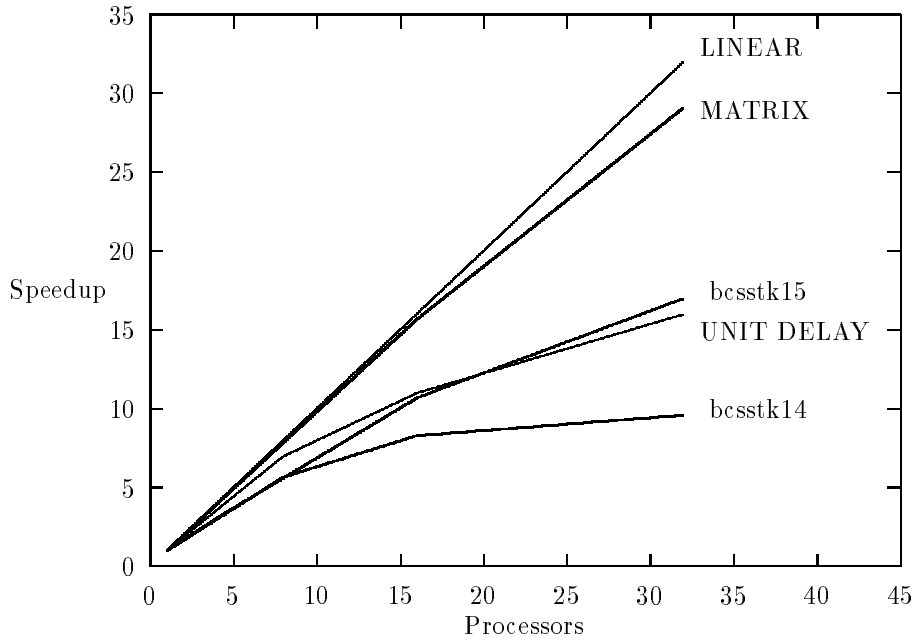
Figure 7.5: The load on the second-level bus over time for MP3D and MP3D-DIFF

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 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 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 7.6). 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 Table 7.6 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 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



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

Figure 7.6: Statistics for matrix programs. The unit delay is for bcsstk14.

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 Table 7.6 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 makes 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.

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 7.6), 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.

7.3 Communication Locality

In the quantitative study we assumed that communication between processors was randomized and that the retrieval of data on a read miss did not explore the locality in the hierarchy. How true that is for these applications can be seen by counting the number of read transactions at each level and calculating their distribution. The studied topology is $2 \times 8 \times 2$. The reads on the buses are counted and divided into three categories: reads on the top bus (top), reads on the two intermediate DDM buses (ddm), and reads to disk (disk). The disk node is located on the top bus. A read request going from one subsystem to the other appears twice on the DDM buses and once on the top bus. A read request to the disk node appears once on the DDM bus and once on the top bus. A read request that is local to a DDM bus only appears on one DDM bus. Read attempts for the disk were deducted and the fraction of the read attempts local to the lowest bus and not passing the top was calculated as locality. Locality is the probability that a read request to another attraction memory will be local to the lowest bus.

$$
\begin{aligned}
reads\,through\,top &= top - disk \\
local\,reads &= ddm - disk - 2\,reads\,through\,top = ddm + disk - 2\,top \\
locality &= \frac{local\,reads}{reads\,through\,top + local\,reads} = \frac{ddm + disk - 2\,top}{ddm - top}
\end{aligned}
$$

If the distribution of reads is random, 7/15 (= 0.47) of reads are local to a bus and 8/15 pass the top bus. As can be seen in Table 7.1, there exists some locality in the communication, but it is not excessive. This is not surprising, since the applications were written for an architecture where communication locality did not pay off. Extensive communication locality was only found in the modified versions of Cholesky-H and MP3D-DIFF. In Cholesky-H, a read request is local to the lowest bus with a probability of 0.78, while the probability for MP3D-DIFF is 0.80.

Application	Top Bus	Disk Read	DDM Bus	Actual Locality	Random Locality
MP3D, 40000 particles	66385	598	203277	0.52	0.47
MP3D-DIFF, 40000 particles	4360	415	24118	0.80	0.47
Cholesky, bcsstk14	358911	91990	952729	0.55	0.47
Cholesky-H, bcsstk14	197277	92411	668020	0.78	0.47
Water, 196 molecules	80482	13785	295246	0.69	0.47

Table 7.1: Calculating locality: What is the probability that a read access will be local to the lowest-level DDM bus? The topology for all runs has been 2x8x2. For MP3D and MP3D-DIFF steady-state numbers are reported, while the rest of the reported numbers are for the full run.

7.4 Speeding up TLB Fills

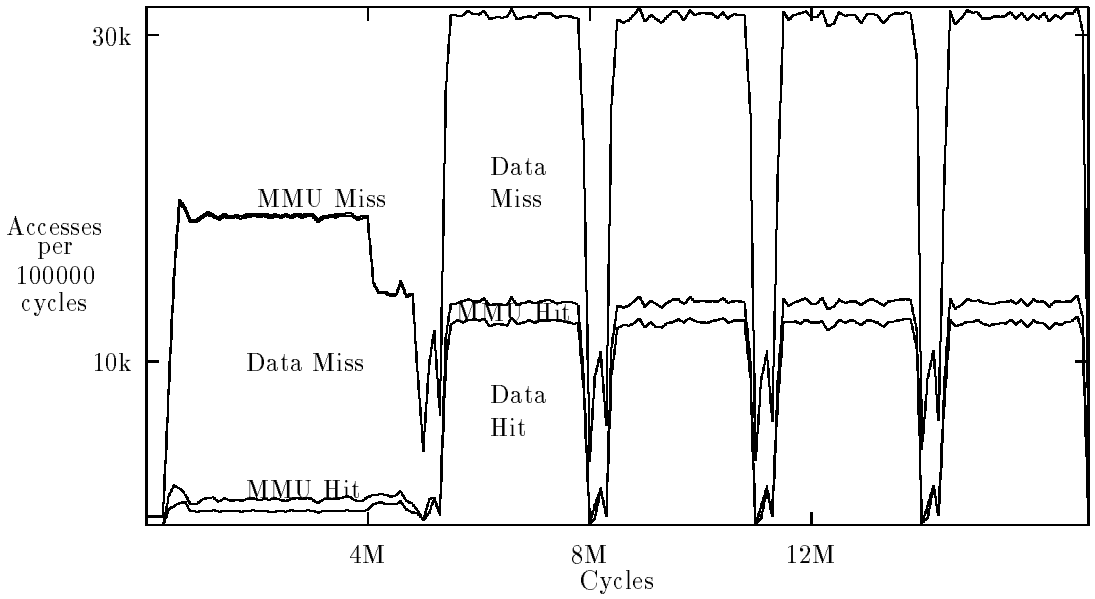


Figure 7.7: The sum of accesses to all attraction memories when running MP3D divided into different categories. The area below the bottom line is the number of hits made by data accesses. The area between the two lowest lines is the number of access hits by the MMU. The next area is the number of data accesses missing in the AM, while the small topmost area is MMU accesses missing in the AM.

We have stressed the importance of simulating large working sets for appropriate behavior in cache accesses. This is also true when simulating correct behavior for TLB accesses. Our simulator models the MMU in the Motorola 88204 chip. The

TLB has 56 entries and a FIFO replacement algorithm. Page size is 4 kbytes, so a TLB might contain translations for 224 kbytes at the same time. In some of the simulations, each processor accesses a much larger address space than that. This is true of the MP3D with 75000 molecules. Figure 7.7 shows the accesses to the AM when running MP3D, and Figure 7.8 shows the results from running MP3D-DIFF. Data Miss and Data Hit represent the ordinary data accesses missing and hitting in the AM. MMU Hit indicates the accesses of the MMU's table walk that hit in the AM, while MMU Miss indicates the missing ones. As can be seen in the figure, a small fraction of the MMU accesses miss in the AM during the first time step, after which the AM contains all the page descriptors of interest to the processor and will serve as second-level cache for the MMU as well. During the following time step, the hits in the AM by the MMU account for 10 percent of all its hits. If these accesses instead had resulted in remote accesses, performance would have been decreased by five to ten percent.

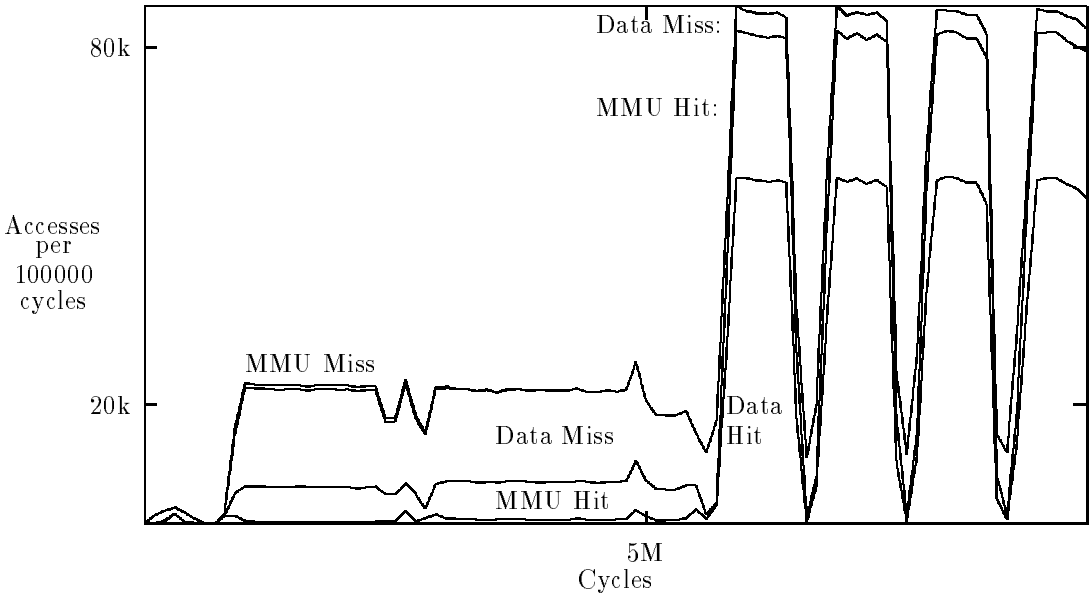


Figure 7.8: Accesses to attraction memory when running MP3D-DIFF divided into different categories.

The task of caching MMU accesses is even more important in MP3D-DIFF, where each processor makes almost random accesses in the data space of 4 Mbytes. If every TLB fault made a TLB fill using slow remote accesses, much of the gain of the modified algorithm would be lost. However, the data for the page table walk is cached in the AMs, cutting delay and communication needs. As can be seen in Figure 7.8, accesses made by the MMU account for about one third of all accesses to the second-level cache. They are five times as frequent as data accesses that miss in the AM. Fortunately, only a small portion of the MMUs—mostly when execution

begins—miss in the AM. After studying such behavior by the MMU, we should ask ourselves if TLB and/or page sizes really are suited for multiprocessor applications with large data sets.

⋆ ⋆ ★ ⋆ ⋆

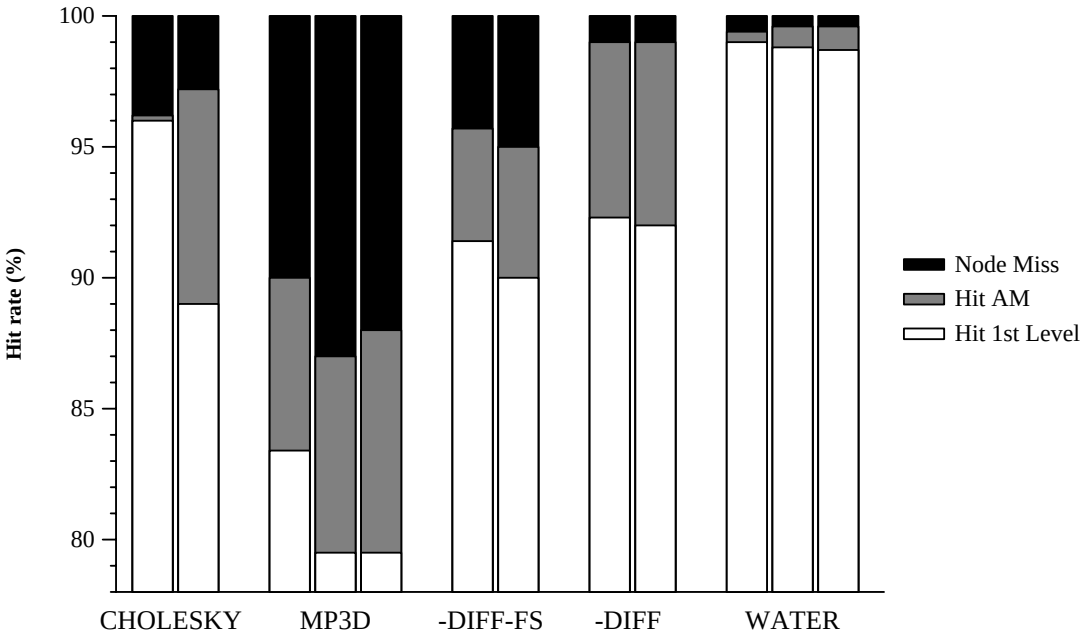


Figure 7.9: The effects of the large attraction memories increase with the size of the data set. Here, the behavior of a small data set (to the left) is compared to larger data sets (to the right) for some applications. For the MP3D applications, the number of particles is increased while the number of space cells is held constant.

A detailed execution-driven simulator of the DDM has been developed. It models a detailed implementation of the DDM. Programs from the SPLASH suite with the largest bearable problem size were used to evaluate the DDM. The importance of the attraction memory increased with a larger data set (Figure 7.9). Good speedup was reported for two of the three studied SPLASH applications. The poor locality of the third application, MP3D, was tamed by restructuring its distribution of work from being static to being dynamic. That restructuring also increased its communication locality. Communication locality was also enhanced for Cholesky by adding hierarchical knowledge to its dynamical scheduler. Finally, the importance of a second-level cache for the accesses made by the MMU was shown to be significant to performance.

Part III

OPTIMIZING COMA

8

Prefetching—ROT

MISSES REMAIN even when caches of the largest possible size are used. The goal of a one-percent miss rate, stated in Chapter 2, is met in the prototype DDM only for a few parallel applications. For others, additional latency-hiding techniques are needed. Prefetching a datum to the register bank or the cache prior to a request by the processor is one method of reducing the misses.

The hardware-based prefetching technique presented here detects access patterns made by the processor and brings expected continuation data closer to the processor. The technique also adjusts the prefetch distance of the pattern so that the prefetched data arrive right on time to be used. We call this right-on-time *prefetching (ROT).*

This chapter explains the technique and its usage. Results from uniprocessor simulation and its effect on the DDM simulator are presented. Finally, some suggestions for implementation end the chapter.

8.1 Introduction

The hit rate in the second-level caches of uniprocessors increases with their size, but it has been shown generally to be much lower than that of a first-level cache. This is true even when second-level caches are one order of magnitude larger. Most of the locality is exploited by the first-level cache, and therefore never reaches the second-level cache [SL88].

As mentioned earlier, uniprocessor cache misses are classified into three categories [HS89]. *Capacity misses* are caused by a fixed cache size (the cache is too small), *conflict misses* by too many active blocks mapping to a fraction of the sets (not enough associativity), and *compulsory misses* by the cache line being touched for the first time. The first two categories of misses can be avoided by increasing the size of the caches, and the second category can also be decreased by more associativity (more ways). Compulsory misses cannot be decreased without prefetching and are the dominant cause of misses for uniprocessors with large caches [HS89].

All misses removed by the first-level caches belong to the first two categories, but none of the compulsory misses are removed. Subsequently, accesses to the second-level cache will have a higher ratio of compulsory misses. Table 8.1 shows the numbers reported by Hill and Smith [HS89] for a trace from VAX 11 interactive users under ULTRIX.

Miss Rate Components for a First Level Cache				
Cache size (bytes)	Degree of associativity	Total miss ratio (%)	Compulsory misses (%)	Ratio of misses being compulsory (%)
16k	4	5.0	0.9	18
32k	4	3.8	0.9	23
64k	4	2.8	0.9	32
128k	4	1.6	0.9	55

Table 8.1: Numbers reported by Hill and Smith, page 1625.

From the table we can see that:

- **First-level cache: 32 kbytes, second-level cache: 128 kbytes**. A two-level cache system with a first-level cache of 32 kbytes has a miss rate in the first level of 3.8 percent. A cache of 128 kbytes has a 1.6 percent miss rate; in other words, the miss rate in the second-level cache, sized 128 kbytes, is $1.6/3.8 = 42$ percent. Of the misses in the second-level cache, 56 percent are compulsory misses (0.9/1.6).

- **First-level: 32 kbytes, second-level:** ∞. An infinite second-level cache connected to the 32 kbytes first-level cache would have a 24-percent miss rate(0.9/3.8), completely compulsory.
- **First-level: 128 kbytes, second-level:** ∞. A first-level cache of 128 kbytes and an infinite second-level cache would have resulted in a 55-percent miss rate in the second-level cache, completely compulsory.

From these numbers we can conclude that the benefits of techniques removing compulsory misses in uniprocessors can be equal in importance to the benefits of large second-level caches.

This introduces a need for second-level prefetch strategies, just as there is a need for second-level caches. Unnecessary traffic is to be avoided; yet, the strategy might require prefetching data far ahead of its use. The head start by the prefetcher is determined by three factors: frequency of accesses to data to be prefetched, size of the prefetch quantum, and the remote delay. The miss frequency is difficult to determine and might be as frequent as every cycle. The remote delay might vary for different parts of the address space (e.g., NUMAs and architectures with a cache hierarchy) and might also be affected by current workload.

8.2 Existing Prefetching Techniques

Prefetching is discussed below divided into two groups: hardware-controlled and software-controlled prefetching. These have different properties and might very well coexist in a complementary fashion.

8.2.1 Hardware-Controlled Cache Prefetching

Fetch Always prefetches the next consecutive address for each access to a cache regardless of whether it is a hit or miss in the cache. This technique removes many of the misses, but is clearly impractical since too many prefetches have to be performed.

Prefetch On Miss fetches the next cache line as well as the current one on a cache miss. It halves the number of misses for a purely sequential address reference stream [SL88].

Tagged Prefetch associates a tag with each cache line. When a block is fetched (or prefetched), a tag associated with the block is set to zero. Accessing a block with a tag equal to zero advances the tag to one and causes a prefetch of the successor block. This method differs from the previous one in that it prefetches on a hit rather than on a miss. This can reduce the number of misses in a purely sequential reference stream to zero, given that the latency of the prefetch is short enough.

Stream Buffers [Jou90] can provide a larger prefetch distance. The prefetching distance is the number of pipelined prefetches needed to avoid misses. Each miss triggers a prefetch stream to start. A stream buffer organized as a FIFO prefetches a fixed number of cache lines starting at a cache-miss address. The buffer is placed by the side of the cache. The address of the head entry of the FIFO buffer is compared to all accesses from the processor. If there is a match, the data of the head entry is removed from the buffer and sent to the processor, and another cache line is prefetched from the memory to the buffer.

8.2.2 Virtual or Physical Addresses?

Should the prefetching be performed on virtual or physical addresses? In general, that question cannot be answered by the prefetch designer, who must adapt to the existing architecture. Most computers have physical caches, i.e., the translation from virtual to physical addresses takes place between the processor and the first-level cache. In such systems, the prefetching scheme monitors and prefetches physical addresses. A prefetched stream of addresses passing a physical page border might continue on to another physical page, and the prefetching should be discontinued. Passing a page border can be detected by the least significant bit of the page-address toggle when the next address to prefetch is calculated. Some operating systems can allocate adjacent virtual pages to adjacent physical pages; crossing a page border would then be no problem.

Some systems have their MMUs located in the second-level cache [WBL89], allowing the prefetcher to monitor virtual addresses. This has the advantage of monitoring contiguous addresses when page boundaries are crossed. Indirectly, prefetching virtual addresses automatically prefetches TLB entries as well. Fatal MMU errors caused by eager prefetching must be avoided, however, possibly by adding a different type of prefetch operation.

8.2.3 Software-Controlled Schemes

Software-controlled schemes prefetch virtual addresses. Typically, the compiler tries to move the loads as early as possible, thus prefetching the values to registers. The CPU is not stalled on loads. Only a register without a value that is used as an operand stall the CPU.

The values can also be brought into the processor cache prior to use. Special prefetching instructions—nonblocking reads and writes provided by modern CPUs—are used for this purpose [GHG+91]. The prefetch instructions can be inserted into the code by the programmer or the compiler. Other software-controlled schemes

involve extra hardware support, where a special prefetching cache, separate from the processor cache, is implemented [KL91].

Most software prefetching inserts the prefetch instruction manually. The compiler technology of tomorrow may be able to insert prefetch instructions automatically. Most of the prefetching is done in loops, where data for the next iteration is prefetched. The next iteration may access the next consecutive word, or a word on a fixed offset from the previous access—the *stride* between accesses. The difference between accessed addresses can be any distribution function, but can still often be determined at the cost of some extra address calculations. Very few unnecessary prefetches are generated outside the processor node. However, in order to cover all misses, many more prefetch instructions than what is necessary are sent from the processor to the processor cache. This puts an extra burden on the scarce bandwidth between the processor and the data cache.

Another interesting approach has been introduced to the WM architecture [Wul88]. Hardware streams initiated by software represent a sequence of addresses. The stream is accessed like any register. Reading a stream returns the data value of the current address and automatically generates the prefetching of the next value in the sequence. The stream therefore both works as an automatic address incrementer and as a prefetcher.

8.3 Right-On-Time Algorithm

A simple algorithm can meet the needs of second-level prefetching. Called *right on time* (ROT), it monitors accesses to a cache. Prefetching activity is started or terminated. The prefetching distance is adjusted to suit the behavior of the application and the load of the machine at the time. ROT can be designed to make prefetches to any cache. This study focuses on prefetching to the second-level cache, based on design experience from the DDM project.

ROT snoops the traffic on the bus between first- and second-level caches. Its input is:

- Operation (read or write),
- Address accessed, and,
- Information on whether the access caused a hit or a miss in the second-level cache.

ROT outputs an operation and an address to prefetch, sent to the second-level cache as shown in Figure 8.1. If a separate input to the second-level cache cannot

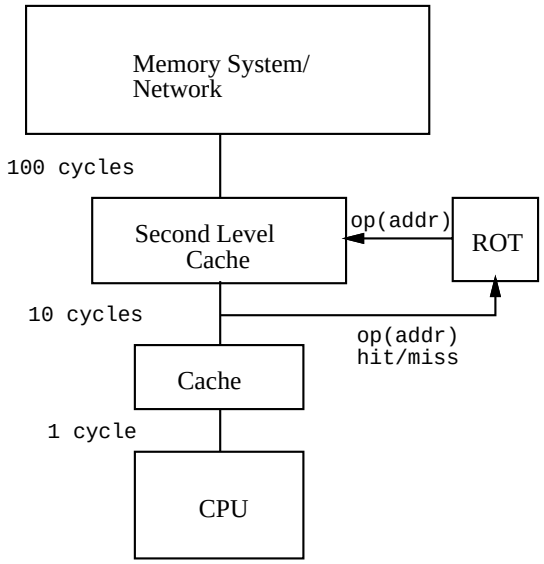


Figure 8.1: Connecting a right-on-time prefetcher (ROT) to a generic three-level memory system.

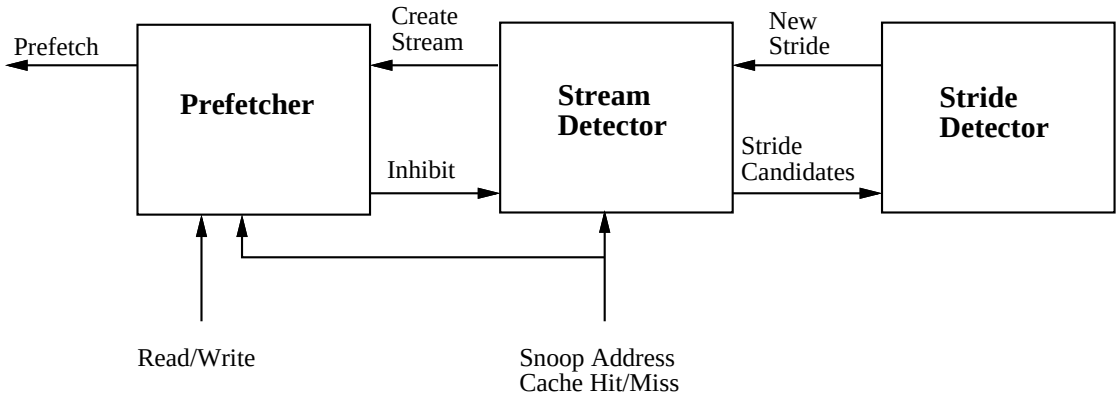


Figure 8.2: The right-on-time prefetcher consists of three parts.

be provided, ROT can perform its prefetches by issuing its prefetch operations on the bus between the first- and second-level caches.

ROT is designed with the hardware in mind. For now, we will concentrate on the algorithm and leave the implementation issues for later. The algorithm consists of three parts. One part detects new access patterns and initiates prefetching. Another part keeps the prefetching alive as long as the accesses to the pattern continue. It also dynamically adjusts the prefetching distances to meet the requirements of the program and the architecture. A third part detects new strides between accesses. See Figure 8.2.

8.3.1 Finding a Prefetch Stream

At this point we assume a *stride list* of popular strides most commonly found in access patterns. Initially, the stride list contains only one value: +1.

The addresses of accesses missing in the second-level cache are recorded on a *miss list*. The address of each new miss in the second-level cache is compared with the members of the miss list, starting with the most recent one. If a distance equal to any stride in the stride list is found, a stream is suspected, and a prefetch stream is created. The corresponding address is removed from the miss list. If no stream is detected, the address is added to the miss list. Old elements of the miss list are purged on a FIFO basis. Only addresses missing in the second-level cache check the miss list.

8.3.2 Keeping a Prefetch Stream Alive

When a prefetch stream is created, the next address of the stream is calculated by adding the matching stride to the most recently accessed address, and its value is prefetched to the cache. The prefetched address is stored as the *stream head* of the stream, and the stride is recorded as the *stream stride*. Each new access is compared with the stream head. A match in the stream head, called a *stream hit*, indicates that it is indeed a stream. The next item to prefetch is calculated by adding the stride to the stream head. The prefetch is performed and the stream head updated. The prefetch is performed by issuing an operation of the same kind as the one that stimulated the prefetch; in other words, a snooped read access results in a read prefetch and a write results in a write prefetch. A stream hit in combination with a miss in the second-level cache indicates a late prefetch, causing multiple prefetches to be performed. Thus, the stream will adjust its prefetch distance until it has the right amount of outstanding prefetches. The *stream tail* will store the most recent prefetched address. When a prefetch distance of one is used, its value will be identical to that of the stream head. A stream hit will inhibit any stream detection.

A large number of streams may be active concurrently. Old streams are reused using the LRU algorithm.

8.3.3 Finding New Strides

The stride +1 is by far the most popular stride, but many applications may make heavy use of other strides as well. The task is to find new, interesting strides quickly and add them to the stride list. We assume that accesses with a new stride will miss in the second-level cache and fail to detect a stream. Each such access will produce a

set of *stride candidates* by subtracting the current address from each of the addresses on the miss list.

The stride candidates are compared with earlier stride candidates, kept on a *stride-candidate list*. A frequency counter for each entry on the list tells how many times that entry has been a candidate. When the counter passes a threshold, the candidate is promoted to the stride list.

The number of stride candidates can be reduced greatly by examining only those candidates whose absolute value is lower than a threshold. If ROT operates on physical addresses, the threshold for interesting stride candidates should be much less than half the address size of a physical page. Strides greater than that cannot benefit by the ROT algorithm anyway, since they pass the page border before the stream is detected.

In order to find interesting strides which are not multiples of the item size, the word or byte address must be monitored by the prefetcher. Consider a stride of six words between accesses and an item size of four words. The access pattern looks as follows:

```
   item 0        item 1         item 2       item 3          item 4
|0  1  2  3 | 4  5  6  7 | 8  9 10 11 | 12 13 14 15 | 16 17 18 19 |
 ^                  ^                      ^                   ^
```

Monitoring the item addresses only produces the following sequence:

item 0, item 1, item 3, item 4, item 6, item 7 ...

which is not easy to detect. If the word address had been monitored instead, a stride of six would have been found.

8.3.4 Making Use of Prefetching

Prefetching can speed up several different categories of accesses, some of which are listed below. Expected behaviors are described for software-based prefetching (SW) and hardware-based prefetching (ROT).

- Strict sequential accesses, like some accesses to vectors or files, are handled well by both techniques.
- Nonsequential accesses in a loop may require extra address calculation for SW, which otherwise performs well. ROT performs well only if a stride can be detected in the access pattern.

- If a dynamic scheduler is used in a multiprocessor application, the need for a prefetcher is accentuated, since the data set will move dynamically throughout the computation. Given the constraints above, both might perform well.

- Process migration—the operating system decides to move the execution of a process to a new processor. ROT can perform well, if the constraints above are fulfilled. SW may fail, since it is impossible to determine at compile time exactly when a process migration might take place.

- Cold-start effects, appearing when a program is started, are similar to process migration. The ratio of compulsory misses is higher when the program is initiated. ROT has a high potential to perform well.

- Copying blocks between processors can be used to implement message-passing algorithms on shared-memory machines or used by the operating system to handle pages. Both techniques can perform well.

- Prefetching contiguous blocks when reused after being replaced from a (direct-mapped) cache. ROT can perform well, while SW might be completely unprepared for the misses.

8.4 A Simple Example

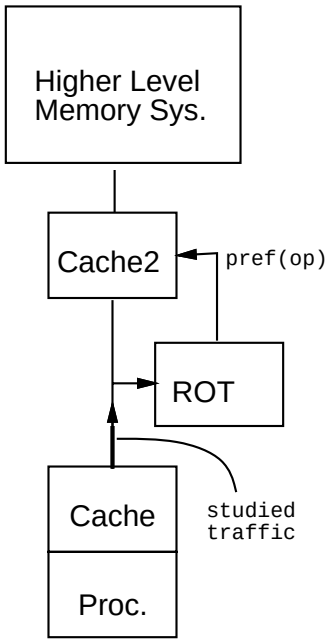
Figure 8.3 shows an access pattern to the second-level cache applied to an architecture with and without ROT. In addition to "Hit Cache2," ROT activity is presented. The stride list contains two elements: +1 and -1.

The following ROT activities take place during execution:

read A The access results in a miss in Cache2. Cache2 fetches A. Address A is compared to all *stream heads* (at this point none) and to the *miss list*, resulting in no match. A is added to the miss list.

write B A hit in Cache2. (The item was already in the cache.) Address B is compared to all *stream heads* (at this point none).

read A+1 The miss in Cache2 makes Cache2 fetch A+1. Address A+1 is compared to addresses in the miss list (A) in combination with the stride list (+1,-1), resulting in a match. A stream is created with *stream tail* = *stream head* = A+2 and *stream stride* = 1. A+2 is prefetched by performing a read(A+2). Address A is removed from the miss list.



Studied Traffic	Hit Cache2 w/o ROT	Hit Cache2 with ROT	ROT activity
read A			add miss list: A
read B	x	x	
read A + 1			pref: (read, A+2)
read A + 2			pref: (read, A+3 & A+4)
write C			add miss list: C
read A + 3		x	pref: (read, A+5)
read A + 1	x	x	
write C -1			pref: (write, C-2)
read A + 4		x	pref: (read, A+6)
write C - 2		x	pref: (write, C-3)

Figure 8.3: Study of access behavior with and without ROT.

read A+2 The miss in Cache2 in combination with the match for *stream head* = A+2 makes the prefetcher fetch two items: A+3 and A+4. The *stream head* of that stream is changed to A+3 and its *stream tail* is changed to A+4.

write C The access results in a miss in Cache2. Cache2 fetches C. Address C is compared to all *stream heads* (A+3), resulting in no match. C is compared to the miss list (currently none), resulting in no match. C is added to the *miss list*.

read A+3 A hit in Cache2. Address A+3 is compared to all *stream heads* (A+3), resulting in a hit, changing *stream head* to A+4, *stream tail* to A+5, and prefetching A+5 by read(A+5).

read A+1 A hit in Cache2. Address A+1 is compared to all *stream heads* (A+4), resulting in a miss.

write C-1 A miss in Cache2. Address C-1 is compared to all *stream heads* (A+4), resulting in no hit. C-1 is compared to the miss list (C), resulting in a hit for step -1. A stream is initiated with *stream tail* = *stream head* = C-2, and *stream stride* = -1. Item C-2 is prefetched by performing a write(C-2). Address C is removed from the miss list.

read A+4 A hit in Cache2. Address A+4 is compared to all *stream heads* (A+4 and C-2), resulting in a hit for A+4. Its *stream head* is changed to A+5, *stream tail* to A+6, and A+6 prefetched by a read(A+6).

Applications	2nd L. MR w/o ROT (%)	2nd L. MR with ROT (%)	Increased traffic (%)	Speedup with ROT (%)
grep	100	6	0.4	15
diff	52	2	0.5	12
awk	8	5	2	2
cc1	27	18	1.3	8
gas	55	12	0.15	18
bcopy	100	0	0.1	550

Table 8.2: Behavior of a two-level cache system using ROT with 8 streams and 32 addresses on the miss list. The reported miss rate is for the second level cache.

write C-2 A hit in Cache2. Address C-2 is compared to all *stream heads* (A+5 and C-2), resulting in a hit. The *stream head* of the corresponding stream is changed to C-3 and *stream tail* to C-3. Item C-3 is prefetched by a write(C-3).

8.5 Uniprocessor Simulation

We will first study a model similar to Figure 8.1 to determine the behavior of ROT on a uniprocessor running UNIX utilities. Access delays of 1 cycle to the first-level cache, 10 cycles to the second-level cache, and 100 cycles to the higher-level memories are modeled. Accesses to higher-level systems can be issued only every tenth cycle. The first-level cache is 16 kbytes, four ways, while the second-level cache is 8 Mbytes, two ways. Cache-line size is 16 bytes for both caches. Grep, awk and diff are well-known text-processing utilities. CC1 is the first part of the GNU-CC compiler, and gas is the GNU-assembler. Bcopy is a highly optimized version of the copying from one memory block to another.

The study, as shown in Table 8.2, only detected strides of +1 (i.e., cache-line-sized strides of 16 byte). The system is configured with 32 addresses on the miss list and 8 stream objects. The achieved hit-rate improvements and speedups vary greatly. All applications but awk make good use of ROT, while all of them produce small amounts of extra traffic.

The bcopy application shows a much greater speedup than the others. By only looking at the improvement in hit rate, an equally large speedup could have been expected from grep. The answer to the difference lies in how often the programs make their accesses to the second-level cache. The version of bcopy used is highly optimized and makes its accesses to the second-level cache often, which also lies in the nature of the application. The other applications have a low access frequency to the second-level cache, resulting in less speed improvements.

Appl.	MR for n streams (%)			
	n=0	n=1	n=2	n=8
grep	100	7	6	6
diff	52	17	2	2
awk	8	6	5	5
cc1	27	17	17	18
gas	55	18	12	12
bcopy	100	40	0	0

Appl.	MR for n miss addr.(%)			
	n=0	n=2	n=4	n=8
grep	100	6	5	5
diff	52	2	2	2
awk	8	5	5	5
cc1	27	19	19	19
gas	55	13	13	12
bcopy	100	0	0	0

Table 8.3: ROT miss rates with a miss list of 32 addresses while varying stream objects and when holding the number of stream objects constant at eight while varying the length of the miss list. The reported miss rate is for the second-level cache.

Appl.	2nd L. MR w/o ROT (%)	2nd L. MR with ROT, stride =1 (%)	2nd L. MR with stride detection (%)
grep	100	6	6
diff	52	2	2
awk	8	5	4
cc1	27	18	15
gas	55	12	4
bcopy	100	0	0

Table 8.4: Behavior of a system with a two-level cache using ROT with 8 streams and 32 addresses on the miss list, with and without stride detection. The reported miss rate is for the second-level cache.

Table 8.3 shows the applications' sensitivity to the number of available streams. For diff and bcopy, where two separate streams of data are processed, having two streams instead of one is clearly advantageous. Other applications gain little from additional streams. None benefits by more than two streams. The table also shows that a miss list of two was enough for these applications.

Until now, only stride +1 has been simulated. Table 8.4 shows the benefits of a stride detector in the different applications. The only application showing any major reduction in miss rate is gas. Its miss rate is decreased from 12 percent down to 4 percent.

8.6 Prefetching in Multiprocessors

Prefetching in a multiprocessor differs in some respects from prefetching in a uniprocessor:

1. **Bandwidth Overhead**. Unnecessary speculative prefetches in a uniprocessor, performed when the memory bus is idle, will not worsen performance. In a multiprocessor, however, a processor wasting the bandwidth of the common network might worsen the performance of some other processor. For many multiprocessors, the bandwidth is the bottleneck of the system.

2. **Cache Coherence**. A prefetch scheme using data buffers in a coherent system must make sure that stale copies are not kept in the buffers and that coherence is not violated. Prefetches that bring data outside the scope of cache coherence are said to be *binding*, while prefetching to a cache is called *nonbinding*. Prefetching to registers is one example of the binding type.

3. **Prefetch Distance**. The prefetch distance, defined as the number of iterations between issuing a prefetch and using it, must be large for many applications because of frequent accesses to the stream and a long latency for remote accesses.

The introduction of caches into computer systems often results in behaviors that are difficult to predict. By just looking at a code sequence, it is hard to estimate the corresponding execution time without knowing the contents of the caches. The contents of the caches depend not only upon past execution but upon the exact organization of the caches.

In multiprocessors, such prediction is even more difficult, since the prior executions of other processors and their invalidation patterns must also be taken into account. The introduction of hierarchical caches [CGB89, Wil85] further complicates the picture, considering the positive effect of active sharing, i.e., that the cache line may already be prefetched by a cache miss issued by another processor in the same cluster.

For nonuniform memory architectures (NUMA) like the DASH [LLG+90], access time to shared memory varies for different parts of the address space—yet another complication.

Existing hardware-based prefetching techniques are not tuned to prefetching for multiprocessors. Prefetching by stream buffers complicates the cache-coherence task for multiprocessors, involving the contents of the buffers in the protocols. Stream buffers also produce a large amount of unnecessary prefetches. Tagged prefetching does not allow for prefetching early enough to fill the speed gap. Both techniques only allow for a stride of +1 between addresses of consecutive prefetches.

Software-based prefetching can perform better, but requires complicated analyses at compile time to determine if a cache line is already in the cache and, if not, how far away it is, i.e., how early the prefetching must be performed. That task is even more difficult if processes are allowed to migrate at run time. Often, more prefetches than necessary are generated to cover all cases, sometimes even resulting in a slowdown.

Application	2nd L. MR w/o ROT (%)	2nd L. MR with ROT (%)	Increased top traffic (%)	Speedup with ROT (%)
Cholesky	93	52	7	9
Cholesky-H	87	51	14	11
MP3D	61	60	6	-5
MP3D-DIFF-FS	54	53	1	-2
MP3D-DIFF	12	12	3	3
Water	72	63	0	1

Table 8.5: Behavior of a $2 \times 8 \times 1$ DDM system using ROT with 10 streams and 10 addresses on the miss list.

8.6.1 DDM Simulations

ROT has been incorporated into the DDM prototype simulator [Gri92]. Details of the DDM architecture, simulator setup, and applications can be found in Chapter 7. ROT default parameters include a miss list of 10 addresses and 10 stream objects. No stride detection is enabled. Only strides of +1 are checked. ROT snoops the transactions on the M bus. It monitors the retry signal of the M bus to detect hits or misses in the AM. ROT can issue a new prefetch every eighth cycle. It disables prefetching when a page border is crossed. A DDM configuration of 2×8 is simulated here, chosen to more easily isolate contention problems related to the DDM prototype, not the ROT algorithm.

The applications are the same three as in the performance study of Chapter 7.

8.6.2 Simulation Results

The simulation results are shown in Table 8.5. Only the two versions of Cholesky show any major gain from ROT. Their miss rates are halved, while communication increases seven percent. They show a speedup in the 10-percent range. Compared to results from other prefetch studies reporting speedups in the 30- to 40-percent range [GHG+91], 10 percent sounds like a minor improvement. Ten percent is about halfway to the theoretical maximum of the algorithmic speedup for 16 processors, however, which is about 20 percent above the speedup achieved in the simulation study reported in Figure 7.6. Other prefetch studies simulate much smaller caches, which leaves room for greater improvements.

Water also shows a minor improvement in execution speed, while the extra traffic generated is negligible compared to total communication. MP3D has a more irregular use of data. Of the three versions of MP3D, only MP3D-DIFF shows a gain in execution time when ROT is used. The original MP3D actually shows a slowdown of 5 percent caused by the extra 6 percent of traffic generated on the already heavily loaded system. MP3D is a major disappointment for the ROT algorithm. This is where a software-based technique can perform much better than ROT. Gupta

et al. reported that execution speed improved 37 percent when they applied their software-based technique to MP3D [GHG+91].

MP3D-DIFF shows a strange behavior. Its hit rate is the same with and without ROT, and its traffic increases with ROT. Still, it shows a speedup of three percent. We have no explanation of this other than possible side effects from dynamic scheduling and that some hot spots in the time domain possibly have been avoided.

It is difficult to justify ROT for a multiprocessor based on the above results. A clear gain in one application does not motivate extra hardware if another application shows a clear loss, although a slowdown for some applications is not uncommon for prefetching schemes. To remove the negative results of ROT, we propose a very simple enabling mechanism turning on and off the prefetching of data, called *ROT enabling*. The mechanism adds two counters, one counting issued prefetches, and one counting prefetch hits. After a certain number of issued prefetches, the number of prefetch hits should be above a certain threshold, or the prefetching activity is turned off. The ROT prediction algorithm is still in operation, however, creating and removing prefetching streams—only the prefetching action is disabled. This enables the prefetch and hit counters to continually evaluate what the effect would have been if ROT had been turned on. Counter results indicate when the prefetching can be turned on again. A second simulation study was run with the ROT enabling added. After twenty issued prefetches, the prefetch hit counter is checked. Eight or less prefetch hits result in the prefetching being turned off.

The results from the run with modified ROT can be found in Table 8.6. Communication overhead for MP3D is cut to only one percent. Slowdown has also dropped to one percent. Communication overhead for Cholesky has been cut to two percent, while its speedup is maintained. The reason is that ROT prefetching is turned off dynamically when access patterns poorly suited to ROT are observed and turned on again when the pattern looks more promising. We believe that ROT enabling effectively prevents large communication overheads and avoid negative effects on performance.

Once more, MP3D-DIFF shows strange behavior with the ROT by decreasing its traffic, while its hit rate is constant. The net result is a speedup of one percent.

8.7 Implementation

The ROT implementation described here is intended for prefetching physical addresses, a page size of 4 kbytes, and a cache line of 16 bytes. ROT is implemented in three distinctive parts as described in the algorithm: prefetcher, stream detector, and stride detector, as shown in Figure 8.2.

Appl.	2nd L. MR w/o ROT (%)	2nd L. MR with ROT (%)	Increased top traffic (%)	Speedup with ROT (%)
Cholesky	93	52	2	9
Cholesky-H	87	50	8	12
MP3D	61	61	0	-1
MP3D-DIFF-FS	46	47	1	-1
MP3D-DIFF	12	12	-1	1
Water	72	72	0	0

Table 8.6: Behavior of a $2 \times 8 \times 1$ DDM system using ROT with 10 streams and 10 addresses in the miss list and the ROT enabling algorithm.

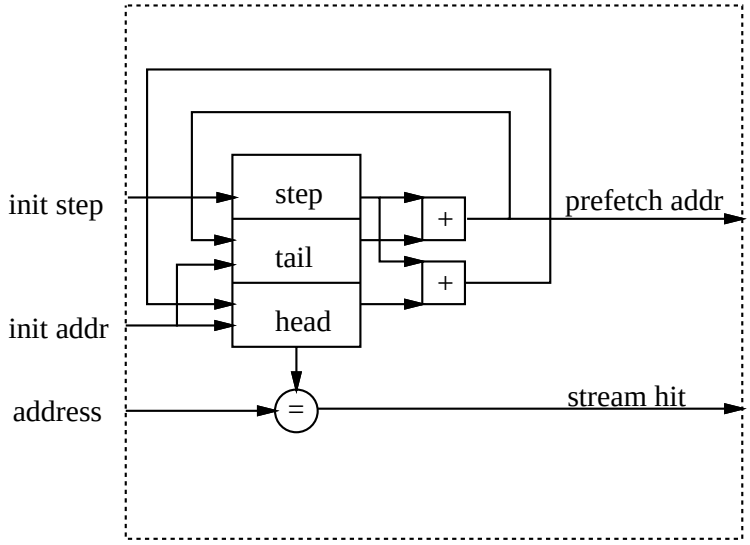


Figure 8.4: Implementation of one stream object.

The stream detector is active only for accesses missing from the second-level cache. In the DDM prototype, a new transaction is started on the M bus at a maximum rate of approximately every eighth cycle. This rate can only be kept up if all accesses hit in the AM, in which case the stream detector is completely inactive. Assuming a hundred-cycle delay for remote accesses, the maximum frequency of misses is every twenty-fifth cycle, assuming four processors which may miss every hundredth cycle. The miss list need not be searched more frequently than every twenty-fifth cycle. A miss list of 10 addresses can therefore be searched sequentially. It can be implemented with a small memory and an ALU.

The prefetcher contains several stream objects. Each object contains a stream-head register of 30 bits (a word address), a stream-tail register of 10 bits (the lower bits of a word address), and a stream-stride register of 10 bits. The stream objects are searched for a match between the current address and the contents of the stream-

head register. A match (stream hit) will increment the contents of the stream head with the contents of the stream stride.

The stream objects must be checked every hit, which might be as frequent as every eighth cycle. Many stream objects may therefore be implemented in parallel. Their simple structure, shown in Figure 8.4, nonetheless allows for cheap implementation.

Implementing a stride detector is quite similar to implementing the stream detector. A stride list contains the most recently reported stride candidates, each with its own counter. The entries in the stride list are replaced in a FIFO manner. If the counter of an entry passes a threshold before it is replaced, a new stride is detected. This list can also be implemented sequentially, given the time constraints.

The enable logic simply consists of two counters, one counting the prefetches and the other counting the stream hits. When a certain number of prefetches have been performed, the value of the stream-hit counter is checked to determine if the prefetching should be turned on or off.

⋆ ⋆ ★ ⋆ ⋆

A hardware-based prefetching scheme was introduced. Its algorithm detected access patterns and prefetched data ahead of use. The algorithm dynamically adjusted its prefetch distance and turned itself off for sequences of poor prefetch statistics.

It performed well for most UNIX utilities and compilers studied. Bcopy showed an improved performance of 550 percent. The only SPLASH applications studied that benefit by ROT were the different versions of Cholesky, ROT removing half the deficiencies toward the goal of a parallel efficiency of 1. Water gained marginally, while the negative effects for MP3D were avoided by the ROT enabling strategy.

9

An Adaptive Write Update Protocol

WRITE UPDATE and write invalidate are the two most common techniques for maintaining cache coherence in a multiprocessor. With the write-update strategy, the other shared copies are updated for each write. The write-invalidate strategy instead invalidates the other copies on a write, making the updated datum the only cached copy in the system that entitles local writes in the future. Both strategies suffer from drawbacks. By invalidating the other copies, write invalidate introduces a new category of misses, invalidation misses*, i.e., a read miss to a datum that would have been in the cache if it had not been invalidated by another processor's writing. Write-update strategy does not have this miss category. The write-update strategy can, on the other hand, generate unnecessary traffic—the updated values might never be used.*

Strategies have been proposed to limit the traffic overhead of write update, by detecting where the update strategy is not a good fit, and instead switch to a write-invalidate strategy, e.g., competitive snooping [KMRS86]. Competitive snooping relies on the functionality of a dedicated signal on the common bus.

Here we describe an alternative adaptive protocol. The protocol is write invalidate by default, but detects where write update is a good match and changes strategy. It can return to the write-invalidate strategy in a manner similar to competitive snooping. The proposed strategy does not rely on a single bus and has added support for migratory objects.

Yet another source of cache misses in a multiprocessor is the share miss*, i.e., in order to maintain sequential consistency, a processor writing to a Shared cache line is stalled until acknowledgement of its invalidations are received from all other caches with a copy. We implement our write-update strategy using a weaker form of consistency, and can therefore also remove this miss category.*

9.1 Introduction

With the write-invalidate cache-coherence policy, a processor writing to a Shared copy invalidates all other copies in the system and therefore can guarantee coherence. With the write-update policy, all copies are instead updated for each write. Both strategies for maintaining coherence have their advantages. Write invalidate is believed to produce less traffic for most applications, since communication between different caches only takes place when needed, i.e., on a read or write miss. Traffic is only generated the first time a processor writes to a Shared datum, after which the writes will be local to the processor's cache. Write invalidate suffers, however, from *invalidation misses* caused by write invalidations, i.e., a read miss to a datum which would have been in the cache if it had not been invalidated by some other processor's writing.

With the write-update strategy, update transactions keep remote cache contents up-to-date on a write to shared data. Write update does not result in any invalidation misses. The price is that traffic is generated for each write to a shared cache line, not just for the first write. The updated values in the remote caches may never be used, either because the values are not read by the remote processors before the next update, or because the data is replaced before they are read—or simply because the cache lines are no longer in active use by the remote processors.

In a COMA with large caches, many of the other causes of cache misses are reduced. Capacity and conflict misses are small for reasonably-sized data sets. Chapter 8 described a method of removing some of the remaining misses. Weaker consistency models have been shown to effectively to reduce share misses [GGH91]. In this chapter we will explore the possibility of also reducing the invalidation misses by means of write update. The multicast ability of the bus hierarchy is a good help in that task.

This chapter first looks at the remaining misses in a COMA and a review of previous work in the area of write-update protocols. Secondly, a first attempt at a write-update protocol for the DDM is sketched, followed by a discussion about weaker consistency models. A second attempt at a write-update protocol combines weaker consistency models with some improved write-update methods. The chapter ends with a performance study of the proposed protocols.

9.1.1 Misses in COMA Architectures

As discussed previously, the cache misses of uniprocessors are grouped 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]. We refer to the sum of these three categories as "uni misses."

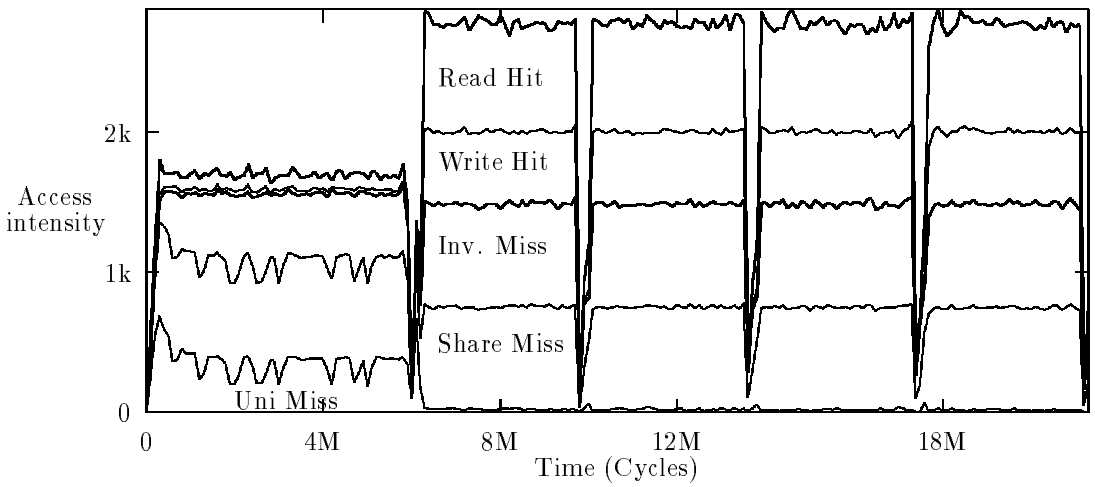


Figure 9.1: Studying the number of accesses to an AM per 100000 cycles for MP3D running on an 8×2 DDM. The accesses are divided into different categories.

When running statically scheduled parallel programs in steps, like WATER and MP3D, compulsory misses disappear after the first step, since the whole problem set has been touched. If the caches are large, the other two categories of uni misses are also rare. Instead, the major sources of misses are *invalidation misses* and *share misses*. The sum of these two categories is sometimes referred to as *coherence misses*. In Figure 9.1 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. Only the coherence misses remain.

Coherence misses can be caused by the following forms of sharing:

1. Producer-consumer sharing—where one processor writes to data while one or many other processors read the datum.

2. Migratory sharing [GW92]—where many (or all) processors write (and read) the datum. Using a write-invalidate protocol for migratory sharing has the effect that the exclusiveness of an item will move among the processors.

3. False sharing—with two or more processors accessing different data from the same cache line. At least one processor performs writes.

The different causes of misses and their traffic behavior are discussed in Section 9.4.4.

Write-update protocols have been proposed to eliminate coherence misses. However, write broadcast may also create unnecessary traffic for many applications, throwing a shadow over the positive effects of the improved hit rate. This is especially true

for large caches. The performances of different cache-coherence strategies have been studied by Eggers and Katz [EK89].

9.1.2 Write Update

The best-known write-update protocol is probably the protocol used in the DEC SRC Firefly [TSS88], a UMA architecture based on a single bus. The protocol uses only three cache states, *Valid Exclusive*, *Dirty*, and *Shared*. A write is performed locally to the first two states. If the cache line is in state Shared, the local value and all remote values are updated by sending a *write broadcast* on the bus, updating the remote copies. For each write broadcast, the other caches with a copy of the cache line activate a wired-OR *share line* on the bus. If the share line is not activated during a write broadcast, sharing ceases, and the cache line in the writing cache goes into state Valid Exclusive, which enables a local write next time. The Firefly protocol removes many cache misses but was shown by Eggers and Katz to produce a large overhead in bus traffic.

9.1.3 Competitive Snooping

In competitive snooping [KMRS86], unnecessary updating is detected and stopped. The updates to a cache line without any intermediate reads by the local processor are counted. When the number of unused updates to the same cache line passes a threshold, the copy is invalidated. The broadcasting cache detects when there are no longer any receivers of its updates and puts the cache line in an exclusive state, performing writes locally in the future. Eggers and Katz identified a large reduction in communication overhead for one application when competitive snooping, instead of the plain write update, was used. The remaining three applications in that study showed a minor gain or loss.

9.1.4 Read Broadcast

Eggers and Katz also studied a write-invalidate protocol with a read-broadcast extension [SR84]. Read broadcast distinguishes between the implicit Invalid state (Figure 3.2), with no address-tag match in a cache, and the explicit Invalid state, where the address-tag matches and the state bits in the cache correspond to the Invalid state. We refer to the explicit Invalid state as *Deleted*. A cache snooping a read transaction on the bus for a cache line it has in the Deleted state will grab a copy of the data and change to state Shared. Eggers and Katz identified this method as being advantageous primarily in cases with a single producer and many consumers, e.g., the pessimistic spin locks used in their study.

9.1.5 Summary of this Study

Step by step, we show how we developed an adaptive write-update protocol for the DDM simulator, including pitfalls and their solutions. Working toward possible implementation proved to be very beneficial in understanding the problem and in identifying the bottlenecks of each individual step.

We propose an aggressive strategy aimed at removing all coherence misses with a small communication overhead. We evaluate the strategy and its effect through a detailed implementation in the DDM prototype simulator.

Our study differs from that of Eggers and Katz [EK89] on several points:

1. Our architecture is multibus rather than single-bus; thus, the penalty for a miss is much greater.
2. The protocol we study is a write-invalidate protocol by default and uses a write-update strategy only for cache lines in which such behavior is detected as being beneficial.
3. Our protocol uses a slightly modified turn-off strategy, thus supporting a wider range of applications.
4. We introduce a special update cache cutting down the local traffic to a reasonable level for especially difficult applications.
5. Our system has much larger caches: the attraction memories.
6. Our caches are second-level caches.
7. Our protocol does not rely on a single global wired-OR signal of a single bus.

9.2 A First Attempt

The goal is an adaptive algorithm for the DDM that turns on the write-update policy for items for which it is beneficial, yet has no negative effect on other items. For this purpose, the protocol of the DDM prototype has to be extended with a write-update part. An item should be able to move back and forth between the two strategies. The default strategy for an item should be write invalidate.

9.2.1 Adaptive Write-Update Strategies

Only cache lines involved in coherence misses benefit from a write-update strategy. The first problem is to detect which items suffer from coherence misses, and to

change their strategy to write update. An invalidation miss occurs if a datum, which would have stayed in the cache, is invalidated between two consecutive reads by the same processor. A share miss[1] occurs if a datum, not otherwise replaced, is read by another processor between two consecutive writes by the same processor. Both types of coherence misses can be detected in a cache-coherence protocol.

- **Invalidation miss detection**
  Here we make use of the new state *Deleted*, used in a read broadcast. An item that is invalidated by a coherence action is put in the Deleted state (D). The item no longer contains a valid data value, and its space may be reclaimed if needed. If the item remains in Deleted when a new read is performed by the processor, a coherence read miss is detected. This is quite similar to the read-broadcast strategy, where Deleted was interpreted as: "this processor read this datum (not long ago), but it was (recently) invalidated—it might need it again."

- **Share miss detection**
  This detection also requires a new state to be introduced. A cache line in the Exclusive state (writing is allowed) that receives a read request from the network is put in the *Shared Was Exclusive* state (SE) in the local cache, having the same privileges as the Shared state. If the item is still in this state on the next write by the processor, a write miss caused by the coherence protocol is detected.

Either method can be used for detecting the need for a write-update strategy. A share miss is easy to detect and simple to integrate into a protocol. It would effectively detect producer-consumer relationships. However, it would not detect share misses for migratory objects, as will be seen later. Since we are aiming at removing misses for migratory items, we chose to detect invalidation misses. When an invalidation miss is detected, a *subscribe* request is sent to the system. All occurrences of the item are then changed to write-update mode, if they were not in that mode before, and the data value is returned to the requesting node, as described in more detail in Section 9.4.3.

9.2.2 Write-Update Turn-Off Strategies

Another problem is to determine when an item in write-update mode should be moved back again to write-invalidate mode. We use a back-off strategy similar to the competitive snooping proposal to move back to write-invalidate mode. When an update transaction is received by a node, the node increments a counter associated

[1] A miss here means an action that makes the processor stall in order to maintain sequential consistency.

with the item. Each time a node reads a data value locally in update mode, it resets the counter to the corresponding item. When the counter value passes a threshold, an *unsubscribe* transaction is sent to the sender, and the item is put in the Invalid state. When only one copy of the item is left, the strategy switches back to write invalidate, described more in detail in Section 9.4.3.

9.2.3 A First Attempt at a Write-Update Protocol

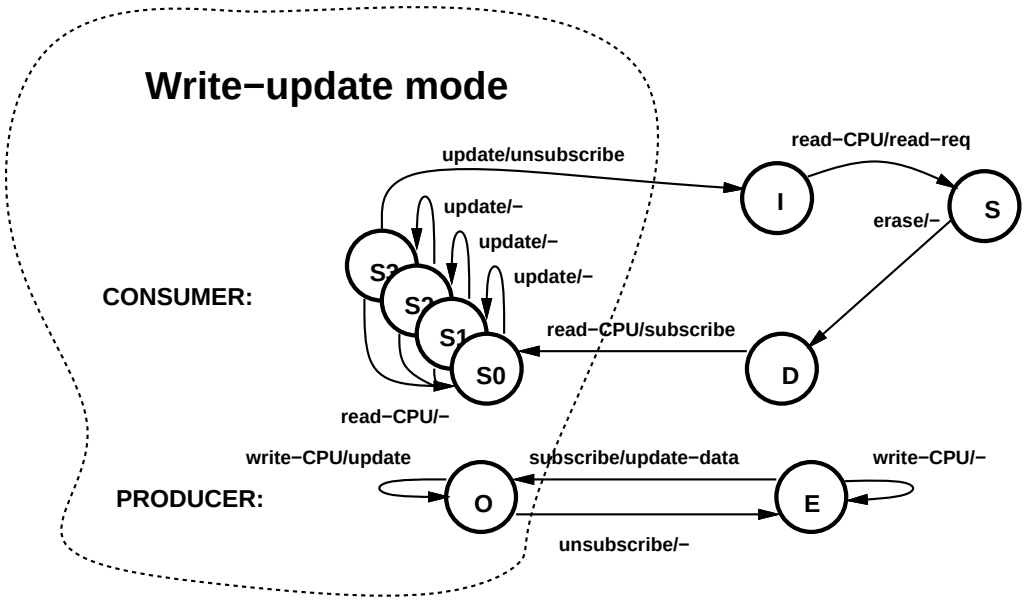


Figure 9.2: A fraction of a first attempt toward a subscribing write-update protocol as an extension to the DDM protocol. None of the necessary transient states are shown in this picture.

A first attempt at designing a protocol is shown in Figure 9.2. An invalidated item is put in the Deleted state (D). A read miss to an item in Deleted results in a subscribe request being sent, instead of a normal read request. A node receiving a subscribe request in the Exclusive state (E) replies with *update-data*, and changes to the Owner state (O). The requesting node receiving the *update-data* will end up in state Subscribe0 (S0), as can be seen in the figure. Additional writes by the "owner" node generate *update* transactions to all nodes in the Subscribe state. Receiving an *update* changes the subscribing node's state from Sx to Sx+1. A local read by the node will reset the state to S0. Receiving an update in state S*threshold* (in the figure the threshold is three) results in an *unsubscribe* to the owner and a new Invalid state (I). When the last subscriber sends an unsubscribe request, the owner returns to the Exclusive state. Note that *update-data* contains the whole item, while *update* contains only the word written.

In order to achieve sequential consistency, a processor writing to a shared item is stalled until the acknowledge is received. So, the share misses remain.

A protocol similar to the one briefly described here would remove invalidation misses for producer-consumer sharing and false sharing, but would have problems with some of the other kinds of sharing.

For false sharing where more than one processor writes to the shared item, "ownership" must be moved between the writing processors in an efficient way; this is not solved by this protocol. Migratory items would also be a problem. After writing to a migratory item, the item might be written to by several other processors before it is needed again. According to the back-off strategy, the item is invalidated before being requested again. However, there will always be some node requesting updates. The net effect is that a large amount of traffic is generated, yet few misses removed.

9.3 Introduction to Weaker Consistency

The protocol of the DDM prototype described in Chapter 6 provides a *sequentially consistent* [Lam79] system to the programmer. While fulfilling the strongest memory-access model, performance is degraded, for instance, by waiting for the acknowledge before the write can be performed. Weaker forms of consistency can provide higher performance.

9.3.1 Weaker Consistency Models

The weaker consistency models that have been proposed rely on special synchronization operations recognized by hardware. The synchronization operations implement stronger consistency for a selection of variables. Synchronization can be a fence operation, like the one used in RP3 [P+85]. In *weak ordering* [DSB86], the programmer declares explicitly which variables are *synchronization variables*. These variables are used to implement critical regions and to synchronize the processors. *Release consistency* [GLL+90] has two synchronization primitives, one for entering a critical region and one for leaving the region.

The weaker consistency models allow certain memory operations to bypass each other and thus stall the processor less often, improving performance. Optimized solutions require extra hardware support such as write buffers and dynamic scheduling of instructions, and introduce a new model to the programmer.

9.3.2 Processor Consistency

Goodman introduced an intermediate consistency level called *processor consistency* [Goo89]. He also noted that existing processors (e.g., VAX 8800) rely on this consistency model.

> A multiprocessor is said to be *processor consistent* if the result of any execution is the same as if the operations of each individual processor appear in the sequential order specified by its program.
>
> Thus the order in which writes from two processors occur, as observed by themselves or a third processor, need not be identical, but writes issuing from any processor may not be observed in any order other than that in which they are issued.

Processor consistency provides a programming model closer to that of sequential consistency, while providing nearly the same performance as the weaker orders of consistency [GGH91]. We assume a programming model including some synchronization primitives. Reads can bypass writes in systems with general networks, and in systems with a race-free network, e.g., a hierarchical network, writes may also be pipelined [LHH91]. The definition of a race-free network can be found in Chapter 10.

We will show here how transient cache states can be used to implement processor consistency in a multiprocessor based on a race-free network without any extra cost in hardware. We call this the *fast write* scheme [HLH91].

9.3.3 Processor Consistency and Hierarchical Networks

Processor consistency requires all processors to observe two consecutive writes: first A, then B from a processor P in program order. A processor Q can violate processor consistency by reading first the new value of B and then the old value of A. The hierarchical network, however, guarantees that in order for Q to observe the new value of B, the erase for A must also have reached Q. This ensures that Q cannot contain any old value of A. The properties of the race-free network also ensure that no other old copy of A can reach Q.

In hierarchical race-free networks, processors need not stall on a write to a Shared item, yet processor consistency is maintained. The newly changed item is marked as not yet public, while an erase is sent to the network. The item turns public upon receipt of an acknowledge.

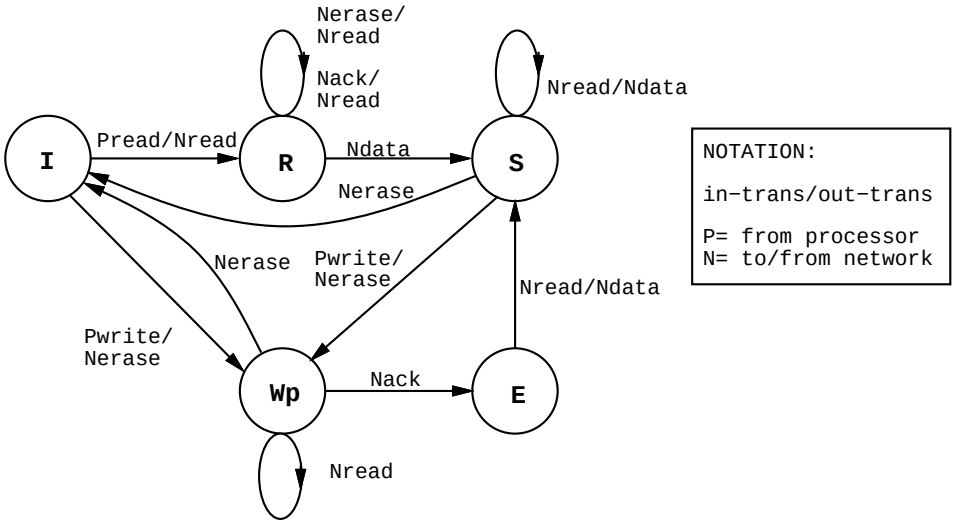


Figure 9.3: Fast-write protocol for item-sized writes. Note that writes use the state Write Pending without stalling the processor.

9.3.4 A Protocol for Items of one Word

The protocol for the DDM shown in Figure 5.2 can be modified slightly to support processor consistency efficiently. A fast-write protocol for items equal in size to the smallest writable entity, here assumed to be one word, is shown in Figure 9.3. The protocol assumes a hierarchical network, like the non-split hierarchical buses of the DDM, to handle write races. The new transient state *Write Pending* (Wp) is used to mark an item not yet public. Upon receipt of the acknowledge, the item is made public by its transition to Exclusive. A *read* request for an item in Write Pending will not be answered until an acknowledge has been received from the network. The probability of thus delaying remote *read*s is of course much lower than that of stalling the processor for each write miss as in the sequentially consistent protocol.

Note that the processor does not stall when writing to items in state I. This is of great importance when an item is first used (e.g. cold start) but of less importance at steady state.

The protocol described above has been shown to provide processor consistency and *causal correctness*[2] by Landin et al. [LHH91], where an alternative and more aggressive implementation of processor consistency is also proposed. That proposal removes the need for the Write Pending state. Instead, a processor with a Shared copy of the item it is writing simply goes to Exclusive and sends out an *erase* message in the network. This scheme also assumes item-sized writes for its functionality. The proof presented also holds for write-update strategies.

[2]Guarantees no side effect if more than two nodes are involved, i.e., node X writes A' then B', node Y reads B' then writes C'; then node Z will "receive" write A' before write C'.

Processor-consistent protocols for items larger than the smallest writable entity [HLH91] have been designed, but become very complicated, and have not yet been fully implemented. However, we need not solve this problem in order to make use of the fast write in the update protocol. When an item is in update mode, the write update unit is word-sized, with a few, and well-defined exceptions.

9.4 A Second Shot at a Write-Update Protocol

The weaker consistency models can be used to remove share misses in a multiprocessor. One of them—processor consistency—in combination with the update protocol also removes invalidation misses. Any weaker model could have been used. We selected processor consistency simply because it suited our architecture and did not require modifications to the applications.

9.4.1 Allowing for Multiple Writers

A write-update protocol where each datum written updates the other copies in the system meets the prerequisite of the smallest writable entity of the fast-write protocol. A write race between two processors writing to the same word of an item is solved in a similar fashion to a write race in the ordinary DDM protocol (Chapter 5), i.e., the winning update will have full effect and the losing update is not seen by any processor. Updates by two processors to two different parts of the item will both have effect. Thus, caches with shared copies in update mode can write to their data—not just the Owner—and yet maintain processor consistency. Thus, the ownership state is removed. A simplified state diagram is shown in Figure 9.4. A protocol along these lines efficiently removes share misses for producer-consumer sharing and false sharing. Atomic operations, like atomic swap, still need to be handled in a sequentially consistent fashion. The transaction *update-mode*, which changes all copies of an item to update mode, is described in Section 9.4.3.

9.4.2 Supporting Migratory Objects

Removing misses for migratory items is more difficult but important for some applications. The protocols described so far do not do this. Let's assume an application where a migratory item "jumps" between n sharing processors. Each of the processors writes to the item every nth time. A node detects the coherence misses and switches to write-update strategy. After receiving a couple of *update* writes from the other processors, the node destroys its value according to the back-off strategy described. However, the value is eventually needed again the next time the node writes.

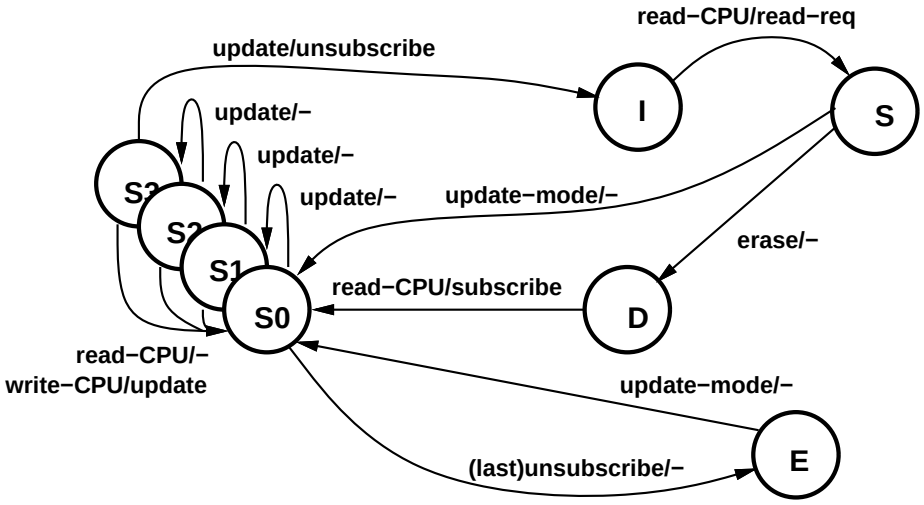


Figure 9.4: A fraction of the implemented subscribing write-update protocol allowing for multiple writers as an extension to the DDM protocol. No transient states are included in this picture.

When a migratory item is detected, the choice is between two strategies: either support migratory items to avoid their misses, or turn off write update completely for these items and avoid unnecessary traffic.

We chose to support migratory items with write update. If data is broadcast to other nodes anyway—possibly on the same bus—why not keep updated values alive until nodes use them again? We use the new state *Subscribe Writing* (SW) to mark items which the processor has written. A write to an item in state Sx results in the new state SW and the transaction *update-reset*, while a write to an item already in state SW results in an *update* transaction.

A subscriber in state Sx receiving an *update* moves to Sx+1, similar to the earlier proposal. However, receiving an *update-reset* in state Sx puts the item in state S0, i.e., resets the back-off strategy counter. An *update-reset* is generated each time a migratory item is moved, and thus keeps the remote copies "alive" by reseting the state to S0. A fraction of the protocol implemented can be found in Figure 9.5.

Supporting migratory items is not cheap, as will be seen in the performance evaluation, since a lot of communication is created. The biggest communication bottleneck is caused by the buses of the local nodes. The top bus, suspected of being the bottleneck in a hierarchical system, has moderate overhead. If the communication overhead for supporting migratory items is unbearable for an architecture, *update-reset* can be used merely to detect migratory items and to switch back to write invalidate for those items, possibly marking them as not suitable for the update strategy and

thus preventing others from applying write update to them. This technique will not be further evaluated here.

Supporting migratory objects in the manner described here is not scalable, in that writes to migratory items from every processor are sent to every node. However, for a limited number of processors it will result in an increased performance.

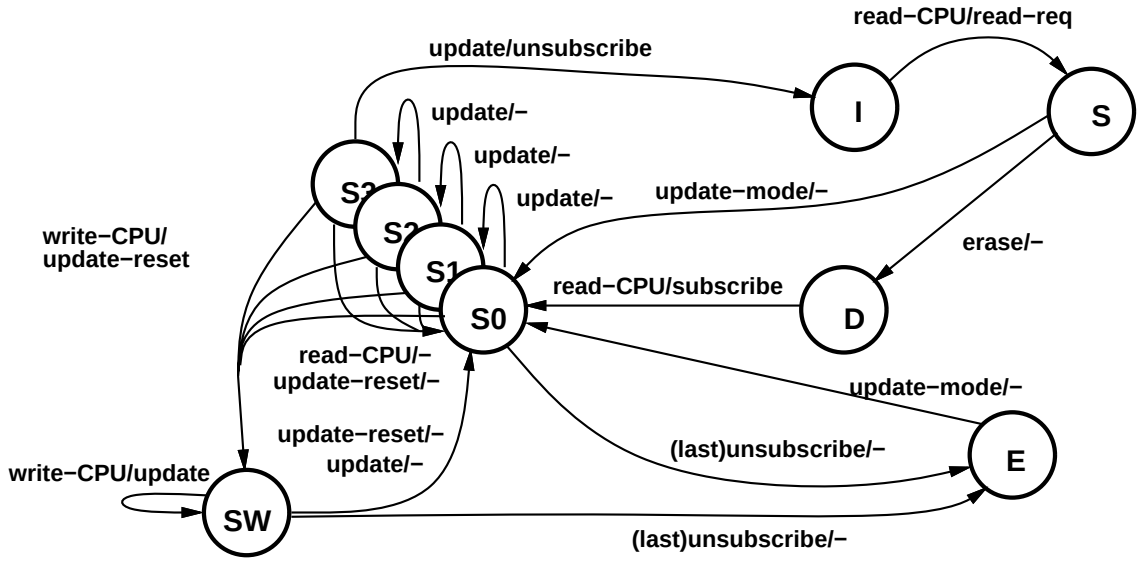


Figure 9.5: A fraction of the implemented subscribing write-update protocol as an extension to the DDM protocol. No transient states are included in this picture.

9.4.3 Protocol Internals

The protocol is built as an extension to the DDM protocol described earlier. The main additional transactions of this protocol are:

subscribe – A request from a node to subscribe on updates to this item.

update-mode – A global transaction putting all copies of the item in update mode.

update-data – Returning the whole item as a response to a *subscribe*.

update – Updating a word of the item.

update-reset – Updating a word of the item, and resetting to state S0.

unsubscribe – A node has stopped subscribing to this item in update mode. This transaction carries the data value of the whole item since this might be the last copy of the item.

The protocol developed in this study soon became large and complicated. It was designed with the flexibility to fully evaluate many different variations rather than implement one kind of update protocol. The details of the implementation will not be described here.

Switching to and from update mode was the most difficult. A slow and safe transition between the two modes was chosen to protect the protocol from processor disagreement about an item's mode. The first *subscribe* request for an item goes either to a directory in the Exclusive state or to the top. From there, the transaction *update-mode* is transferred down to all processors not in state I, changing their states to the corresponding state in update mode. The requesting processor also gets the *update-mode* transaction. This will force it to resend its *subscribe* transaction.

A *subscribe* transaction finds its path to a copy of the item in the same manner as a *read* request found its path to an item, as described in Chapter 5. A *subscribe* transaction reaching a node in state Sx results in a *subscribe-data*, containing the whole item, to be returned to the requester.

An *update* is first sent to the top (or a directory in state E) and from there sent down to, and thus updating, all subscribing nodes.

Changing from write-update mode back to write-invalidate mode uses a technique similar to item replacement in the prototype DDM protocol, for detecting when the last listener sends its *unsubscribe*. Like the *out* transaction, the *subscribe* keeps climbing up the hierarchy, changing the directory state to Invalid, as long as no subsystem on the bus has a copy of the item. If it reaches the top (or a directory in state E) and still no subsystem has the data, it is converted to an *inject* transaction. An *update* on its way up through the hierarchy can detect if there are no "listeners" to its update, and can change state to Exclusive (like the Firefly), forcing the exclusiveness further down in the system. The only copy left in update mode will eventually detect its situation and change its state to E.

9.4.4 Traffic Generated in the DDM

We compare traffic generated by write-invalidate and write-update policies in the DDM architecture for three cases of sharing: one producer and one consumer sharing, false sharing, and migratory sharing. The topmost example in Figure 9.6 compares producer-consumer sharing with the two strategies. The full cycle of consumer reading followed by producer writing is described. The write-invalidate policy requires four DDM transactions, while the write-update policy can perform the same task with a single transaction. However, only one writable entity is transferred for each *update* transaction, in this case a word. If all four words of an item in the DDM are written by the producer, four *update* transactions are needed.

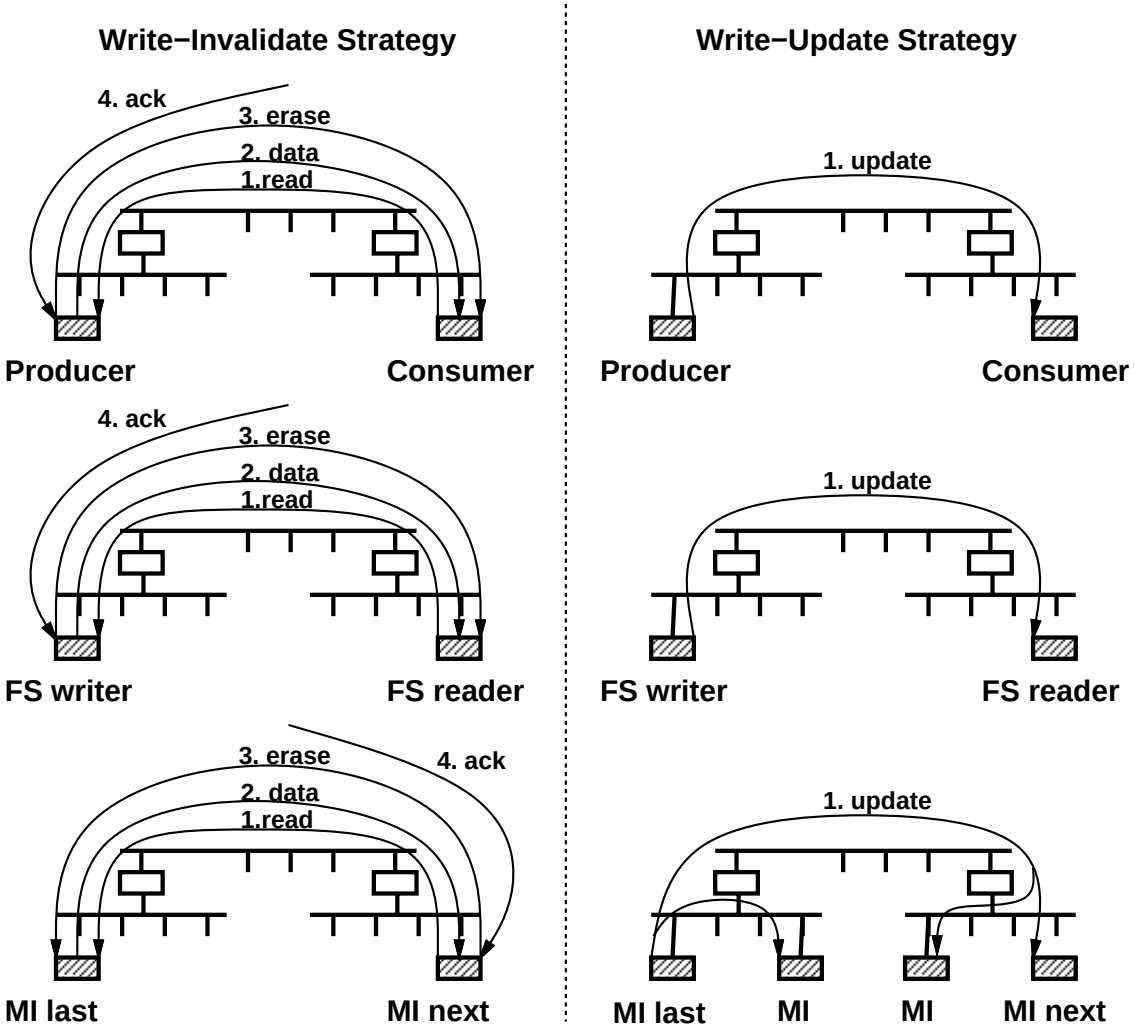


Figure 9.6: Traffic generated by write-invalidate and write-update policies for producer-consumer sharing, false sharing, and migratory sharing in the DDM architecture.

The second example in Figure 9.6 shows a false-sharing situation in which one of the sharing processors is writing (and perhaps reading) while the other processor is reading. The write-invalidate policy needs four transactions for the full cycle of a write by one processor and a read by the other processor. Here again, the write-update policy needs only one transaction to solve this problem. More transactions may be needed if more than one word of the shared item is written. If both processors sharing the item write, twice the amount of transactions are needed in the write-invalidate case. Write update does it in two *update* transactions.

The last example in Figure 9.6 exemplifies migratory sharing among four processors. Write-invalidate policy gives only the last writer (MI last) an exclusive copy of the item. The next owner (MI next) must first retrieve a copy, later made exclusive, in four transactions. With the write-update policy, all four processors involved in

Action #1, read to state I		
Local M bus	read line	a read from a cache to the AM–retry
DDM bus	read request	the protocol generates a read request
Remote M bus	read line	the remote services the request
DDM bus	data	data reply to the requesting node
Local M bus	write line	data to the attraction memory
Local M bus	read line	the read now succeeds

Action #2, write to state S, the Dcache already had a shared copy		
Local M bus	write word	retry
DDM bus	erase	erase request generated by the protocol
Remote M bus	write word	erasing all copies in the remote nodes
DDM bus	acknowledge	changing state to E
Local M bus	write word	updating the memory

Action #3, write to state Sx, the Dcache already had a shared copy		
Local M bus	write word	OK
Local M bus	read word	the protocol puts the Dcache back in state Valid and reads the value to broadcast
DDM bus	update	update containing a word to all remote nodes
Remote M bus	write word	all nodes in state Sx update their copies

Table 9.1: Three frequent actions in the DDM prototype with only one DDM bus.

migratory sharing have copies, marked MI. All are updated with a single *update* transaction. As above, up to four *updates* may be needed if all words of the item are updated.

In these examples, it looks like the write-update policy does not produce more traffic than write invalidate. In the $2 \times 8 \times 1$ topology, this is not necessarily so. After discussing locality in Chapter 7, we concluded that a request in such a topology had a 0.47 probability of being local to the lowest-level DDM bus, assuming random distribution. The actual ratio of locality was measured to be higher than that. For migratory items, all four transactions of the write-invalidate policy have a 0.47 probability of being local to the lowest-level DDM bus, while the *update* transaction is always broadcast over the top bus if at least one of the MI items exists in another subsystem: it always passes the top and is carried on at least two of the lowest-level DDM buses.

9.4.5 Actions in the DDM Prototype

The adaptive write-update protocol has three frequent actions described for a single-level DDM in Table 9.1.

Action #1 describes the actions caused by a read to an item in the Invalid state in the attraction memory, e.g., an invalidation miss in a write-invalidate protocol. The read attempt on the M bus is retried by the DDM protocol, which puts a read

request on the DDM bus. A remote node services the request by a read line on its local M bus. The data transaction is sent on the DDM bus to the requesting node, which writes the data to its attraction memory by a write line on its M bus. Finally, the requesting processor may read the item by a read line on its M bus. Action #1 corresponds to "1.read" and "2.data" in the producer-consumer study for the write-invalidate strategy in Figure 9.6. Action #2, a write to a shared item, e.g., a share miss, corresponds to the "'3.erase" and "'4.ack" in Figure 9.6, and action #3 corresponds to the necessary actions caused by the update of a single word. The read word of action #3 is needed to put the processor cache back in the Valid state again after its write has been approved, otherwise the next write to the cache line will not show up on the M bus. Alternative implementations of the processor cache could avoid this extra transaction.

In a producer-consumer situation, the consumer performs action #1, while the producer performs action #2 for each item transferred between the two of them, involving a total of seven M bus transactions and four DDM bus transactions. Both the producer and the consumer experience misses in the attraction memories.

If instead the item is in write-update mode, action #3 will be performed for each word written. If only one word of the item is updated, three M bus transactions and one DDM bus transaction will be needed. Updating all four words of an item results in a total of twelve M bus transactions and four DDM bus transactions. Neither the producer nor the consumer experience misses in the attraction memories, but the producer will detect a miss in its Dcache.

A modified Dcache, which automatically puts itself in a Valid state in an update situation, would take away the read word transaction in action #3, thus reducing the number of M bus transactions to two per updated word.

Such calculations can be made for other kinds of sharing as well. If the same node writes to all words of the same item in a batch, write invalidate has a minor advantage in terms of communication cost; however, if less words are written before the item is read by another node, write update can be superior in terms of communication.

Note that writes to items in update mode create more traffic on the local bus of the writer, since writes are not performed locally in their Dcaches. The caches used in this study implement the write-once protocol, as described earlier, and experience an update write as a write miss in the Dcache and a write hit in the attraction memory. Together, these observations normally result in a lower hit rate in the Dcache (less local writes), a higher hit rate in the AM, and a slight increase in communication on the M bus. The net effect is a lower node miss rate.

9.4.6 Reducing Traffic with an Update Cache

For migratory items a combination of actions #1 and #2 is replaced by action #3, like for the producer-consumer case. Supporting migratory items results in not just one, but many—even all—other nodes being updated for each write performed. This does not increase the communication cost of the DDM bus in a single-bus system. In a multibus DDM, a slight increase in communication on the DDM buses may be observed if all four words of the migratory items are updated. Some of the #2/#1 combinations previously local to the lowest-level DDM bus will now broadcast to many (all) buses, including the top bus.

Migratory items put a heavy burden on all the M buses, since each write to every migratory item performed by any processor is multicast to all other M buses.

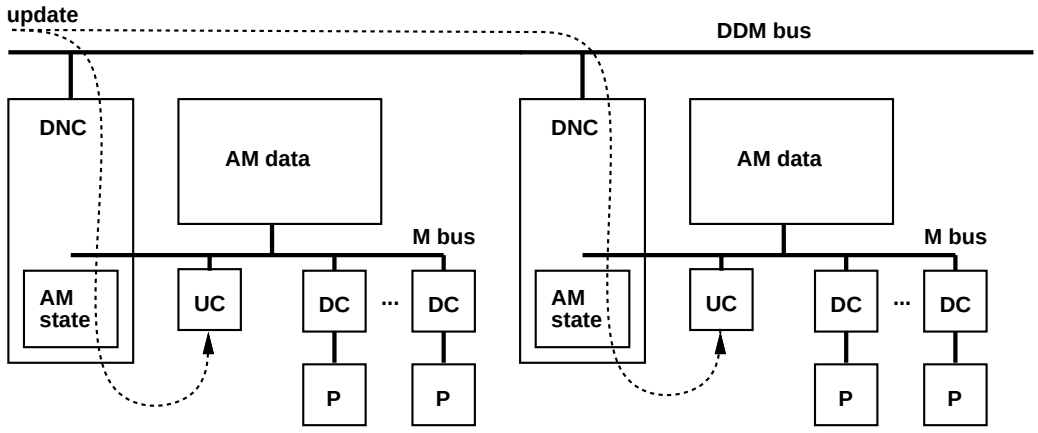


Figure 9.7: Connecting an update cache (UC) to a DDM node (DC= Data Cache, P=Processor, DNC=DDM Node Controller).

Increased communication on the DDM buses can be accepted as the price of avoiding many misses in the attraction memories. However, frequent write words on the M buses will congest all meaningful activity in the nodes. If all *updates* instead are sent to a specially assigned Dcache on the M bus, called the update cache (Ucache), much of the traffic on that bus can be avoided. The Dcache chip, MC88200 of 16 kbytes, is simulated here to function as a Ucache. The Ucache is part of the regular coherence protocol of the M bus. The first update to a cache line puts the line in the "Dirty" state and erases all other copies in other Dcaches with a write word on the M bus. All following updates to the same cache line will be performed locally in the Ucache. The AM gets the updated values when the line is replaced. If a Dcache tries to read the cache line while in the Dirty state in the Ucache, M bus

coherence actions will 1) retry the Dcache, 2) have the Ucache update the AM, and 3) let the Dcache arbitrate for the bus again. This method could be improved by building a specialized Ucache. A better scheme might be the Berkeley protocol for SPUR [Kat85]. Such a protocol would allow the Ucache to update the Dcache directly, instead of first retrying the Dcache, followed by a write-back to the AM so the Dcache can successfully complete the read line in yet another M bus transaction, i.e., the access time to a cache line that is Dirty in the Ucache hit would be reduced by roughly two thirds.

This use of Ucache resembles the possible combination effects of delayed consistency [Dub91]. In delayed consistency, sending the update from the writing node can be delayed. If a new update for the same item is detected before the update is sent off, the two (or more) updates can be combined in a single update message. This saves bandwidth not only on the remote M bus but also in the global network.

9.5 Performance Study

The SPLASH programs studied in Chapter 7 match this study well. They have been characterized by Gupta and Weber [GW92]. MP3D has a fair amount of migratory sharing caused by all processors updating the states of all space cells, i.e., space-cell data becomes migratory shared data. The few misses remaining in WATER are of migratory nature as well. MP3D-DIFF-FS has removed the migratory nature of the space-cell data, but instead suffers from massive false sharing between particle data. MP3D-DIFF removes false sharing of particle data. Can the update improve its performance?

The simulated topology is $2 \times 8 \times 1$, and none of the suggested improvements for the Dcache and Ucache were included in the default setup. The statistics cover the whole run, including cold-start effects, except for the three MP3Ds, where the statistics are for the last time-step only.

Simulation results for each application are reported in tables. The tables present hit rates (HR) for the data cache (Dcache), attraction memory (AM), and update cache (Ucache). A hit in the Ucache is defined as an update to a node that does not generate any transactions on the M bus.

The tables also report the node miss rate, i.e., the ratio of data transactions missing in both the Dcache and the AM.

The total numbers of read misses (action #1), write misses (action #2), and updates (action #3) are also reported for each run. The weighted sum for the network is calculated by adding the number of DDM transactions that each action requires $(2 * \#1 + 2 * \#2 + \#3)$. All four categories of transactions on the M bus are also counted. The weighted sum for the M bus is calculated by multiplying the write

line and read line numbers by two and adding them to the read word and write word. The ratio between how many cycles a word and line transaction occupy on the M bus is approximately 1:2.

The busy rates for all three categories of buses are also reported. Note that the busy rate of the M bus includes the repeated retry polling of processors waiting for a remote access to be completed. The busy rate for the M bus might therefore look unproportionally high. Bus busy rates are highlighted if they are believed to be saturated. Note that a bus may be saturated for long periods, despite its busy rate for the whole run still being much lower than 100 percent (compare with Figure 7.5.)

Each application is run with a number of different configurations, specified at the top of each column.

Abbreviation	Description
Old	The old protocol described in Chapter 6
w/o	Without *update reset*
Mod	Using the modified caches, removing the need for some extra read lines
NW weighted sum	The weighted sum of transactions sent to the network
M bus weighted sum	The weighted sum of used transactions on the M bus (excluding retry polling)
2xBW	Using a network with twice the transaction frequency

Table 9.2: Abbreviations for performance tables.

MP3D resulted in the statistics shown in Table 9.3. The first column shows the results from the old protocol. The node miss rate of 11 percent limits the achievable speedup of this application. The poor locality produces a lot of communication, which almost saturates the DDM bus (84 percent).

A write-update protocol without update reset cuts the number of node misses to 6.3 percent, i.e., a cut of 4.7 percentage points. This cut mainly comes from false-sharing gains. Interestingly, the node miss rate for MP3D-DIFF-FS is exactly 4.7 percent with the old protocol (see Table 9.4). The remaining misses in MP3D are caused by accesses to migratory objects. Although miss rates decrease, execution times increase. The limiting factor is the DDM bus, which is saturated at 93 percent. Just removing false sharing does not increase the load on the network; in fact, it produces less traffic. The extra traffic comes from trying to remove migratory sharing. This version of the protocol is without the update reset. Migratory data are updated in the network, but the updated value is usually destroyed before it is used by the processor again.

The third column is with update reset. When update reset is used, the node miss rate is cut to a mere 0.6 percent, but execution time is still very long. The reason for

MP3D, 40000 particles, 2x8x1								
	Old	w/o	With Update Reset				Old	New
		No Ucache		With Ucache		Mod	2xBW	2xBW
		4S states			8S			8S
HR Dcache (%)	82	73	73	71	73	73	83	73
HR AM (%)	40	77	98	98	99	98	39	98
HR Ucache (%)	-	-	-	88	88	88	-	89
Node miss (%)	11	6.3	0.6	0.7	0.3	0.5	11	0.5
Node read miss	125k	112k	13k	12k	5k	10k	127k	9k
Node write miss	121k	25k	2k	2k	1k	2k	122k	2k
Updates	0	273k	377k	370k	376k	376k	0	376k
NW weighted sum	492k	546k	406	398k	388k	400k	495k	398k
Read line	354k	329k	237k	232k	229k	232k	348k	232k
Read word	16k	400k	393k	463k	399k	23k	17k	23k
Write line	291k	187k	131k	717k	678k	543k	295k	547k
Write word	317k	788k	5658k	1049k	1081k	658k	319k	661k
M bus weighted sum	1.6M	2.2M	6.8M	3.4M	3.3M	2.2M	1.6M	2.2M
Busy rate M bus	74	80	**92**	82	78	71	64	71
Busy rate DDM	84	**93**	65	**93**	**90**	**87**	46	62
Busy rate Top	64	65	63	88	88	84	35	57
Exec. time (rel)	1.00	1.30	1.30	1.05	1.01	1.01	1.00	0.91

Table 9.3: Statistics from the subscribe protocol running MP3D.

this is simply the high number of updates on the M buses, i.e., the number of write words (5.7 M). The fourth column shows how the introduction of Ucache effectively cuts this to about 18 percent of the original number (1.0 M). Execution time now drops compared to the earlier column, but is still longer than when the old protocol is used. The bottleneck has once more moved to the DDM bus.

The fifth column shows behavior when using eight Sx states instead of four. This decreases the node miss rate by 0.4 percentage points down to 0.3 percent. The migratory items move randomly among the 16 processors. An item is used twice in a row by the same processor with a probability of 1/16. For such a case, update reset is not generated as frequently as usual, and remote items are destroyed. With eight Sx states, the probability of destroying a migratory item is much lower. It is interesting to note that dividing the node miss rate of the second column (6.3 percent), which is migratory misses, by 16 equals 0.4 percentage points—i.e., identical to the node miss rate gained by introducing eight S states.

In the sixth column, the effect of modifying the Dcaches is tested. The communication on the M bus drops. The number of read lines is cut from 399k down to 23k, and the weighted sum is brought down to 2.2M. That did not help the execution speed, however, since the bottleneck was the DDM bus. The execution time is now almost identical to the original one. Comparing this column with the results from the old protocol, the weighted sum for the network shows a decrease of almost 20

percent in the number of generated transactions. Still, the busy rate on the top bus is increased from 64 percent to 84 percent. The actual increase on the top bus caused by the lack of locality is thus 33 percent ($= 84 - (64 * 0.8)$). The locality loss can be calculated as 33/84=39 percent, i.e., close to the expected 47 percent.

So, for MP3D we managed to cut the miss rate from 11 percent down to 0.3 percent, but started off with an almost congested system which turned the small communication increase into the limiting factor and no improvement in speed was observed.

For this application we also ran additional simulations where the bandwidth of the DDM buses was doubled by increasing the transaction frequency. The results can be seen in the last two columns. For the new protocol, the setup is identical to that with the modified caches. Now a speedup of around 10 percent can indeed be observed—still discouraging. A much larger speedup was expected from cutting the node miss rate from 11 percent down to 0.5 percent, and this time we cannot blame the contention of the buses. Part of the answer can be seen in the Dcache hit rate, which is decreased from 83 percent to 73 percent. Neither the weaker access order nor the update protocol have been implemented in the Dcache. The processor is stalled on every write to broadcast data for 10–12 cycles until the attraction memory is updated. A much larger speed improvement would be expected if the update protocol were incorporated in the Dcache and write buffers added to the cache. This would result in an increased hit rate[3] in the Dcache, since the processor would only rarely be stalled for writes.

MP3D-DIFF-FS in Table 9.4 shows great improvement already in the second column. The original node miss rate of 4.7 percent is cut to only 0.6 percent. Since this application suffers mostly from false sharing, migratory item support is not needed and actually slows things down slightly compared to the second column. Note that traffic decreases on all buses in this application. When we apply the modified caches to this application, the burden on the M bus is greatly reduced. This time a gain of 17 percent in execution time can be observed.

The adaptive protocol thus proves to handle false sharing well and actually reduce both communication needs and execution time.

MP3D-DIFF differs from MP3D-DIFF-FS in that most false sharing in the program is removed. The improvements measured for MP3D-DIFF are reported in Table 9.5. We believed MP3D-DIFF was already heavily optimized for the DDM and did not expect much gain with this protocol. Using write update results in a slight gain on 16 processors. We have measured greater improvements for MP3D-DIFF on larger topologies. MP3D-DIFF on the $4 \times 8 \times 2$ topology runs about 20 percent faster using the described protocol with update reset and update cache than with the old protocol.

[3]Remember that we define a miss as an access that makes the processor stall.

MP3D-DIFF-FS, 40000 particles, 2x8x1						
	Old	Without	With Update Reset			
		No UCache		With Ucache	Mod. Caches	
		4S states			8S	
HR Dcache (%)	90	85	83	84	84	86
HR AM (%)	54	96	98	97	98	97
HR UCache (%)	-	-	-	60	65	65
Node miss (%)	4.7	0.6	0.4	0.4	0.4	0.4
Node read miss	78k	10k	8k	9k	7k	8k
Node write miss	62k	5k	3k	4k	3k	3k
Updates	-	183k	256k	245k	222k	182k
NW weighted sum	260k	213k	278k	270k	241k	204k
Read line	232k	178k	178k	180k	173k	176k
Read word	86k	280k	344k	344k	317k	91k
Write line	149k	52k	55k	111k	370k	84k
Write word	193k	378k	542k	393k	109k	142k
M bus weighted sum	1.0M	0.96M	1.4M	1.3M	1.5M	0.75M
Busy rate M bus	47	38	46	41	42	35
Busy rate DDM	61	60	67	65	64	58
Busy rate Top	44	30	29	30	30	31
Exec. time (rel)	1.00	0.84	0.88	0.87	0.86	0.83

Table 9.4: Statistics from the subscribe protocol running MP3D-DIFF-FS.

MP3D-DIFF, 40000 particles, 2x8x1						4x8x2	
	Old	Without	With Update Reset			Old	With
		No UCache		With Ucache			With
		4S states			8S		4S
HR Dcache (%)	92	91	89	89	89	93	92
HR AM (%)	88	93	96	96	96	74	96
HR UCache (%)	-	-	-	77	81	-	67
Node miss (%)	1.0	0.6	0.4	0.4	0.4	1.7	0.3
Node read miss	18k	11k	8k	7k	7k	50k	10k
Node write miss	13k	7k	4k	4k	4k	12k	5k
Updates	-	35k	76k	67k	64k	-	104k
NW weighted sum	62k	60k	99k	89k	86k	123k	134k
Read line	169k	164k	155k	153k	154k	284k	210k
Read word	88k	134k	163k	165k	159k	87k	207k
Write line	111k	101k	100k	111k	111k	168k	116k
Write word	118k	158k	273k	178k	177k	187k	245k
M bus weighted sum	0.77M	0.82M	0.95M	0.87	0.87 M	1.2M	1.1M
Busy rate M bus	31	29	36	33	32	43	41
Busy rate DDM	13	18	26	25	24	21	40
Busy rate Top	8	8	8	8	9	21	30
Exec. time (rel)	1.00	0.97	1.00	0.97	0.97	1.00	0.82

Table 9.5: Statistics from the subscribe protocol running MP3D-DIFF.

Cholesky, bcsstk14, 2x8x1							
	Old	Without	With Update Reset			Without	With
		No Ucache	With Ucache				
		4S states			8S	ROT	
HR Dcache (%)	96	94	92	93	92	95	95
HR AM (%)	7	42	64	62	64	73	77
HR UCache (%)	-	-	-	89	91	66	89
Node miss (%)	4.1	3.4	2.8	2.8	2.8	1.3	1.2
Node read miss	510k	510k	440k	440k	430k	125k	120k
Node write miss	550k	340k	250k	250k	250k	185k	160k
Updates	-	516k	1070k	1010k	1060k	170k	330k
NW weighted sum	1924k	2206k	2510k	2400k	2420k	(790k)	(890k)
Read line	1150k	1160k	2160k	1060k	1070k	1240k	1180k
Read word	18k	955k	36k	1450k	1420k	690k	860k
Write line	1010k	905k	885k	1320k	1260k	1120k	1180k
Write word	911k	1450k	6470k	1760k	1820k	690k	890k
M bus weighted sum	5.2M	6.5M	12.6M	7.9M	7.9M	6.1M	6.5M
Busy rate M bus	40	42	52	44	45	38	38
Busy rate DDM	60	76	77	77	79	80	82
Busy rate Top	53	55	57	57	63	54	59
Exec. time (rel)	1.00	1.08	1.11	1.09	1.09	0.94	0.92

Table 9.6: Statistics from the subscribe protocol and ROT prefetching running Cholesky, bcsstk14.

Cholesky with its dynamic scheduling of work does not suit write update at all. Dynamically moving the data is detected as migratory, keeping items in update mode throughout the computation. Update values have very small chances of ever being used again. However, by combining the update protocol with the ROT prefetcher, these negative effects can be avoided (and the positive effects of ROT explored), as can be seen in Table 9.6. ROT is given a higher priority in that data prefetched by ROT are never put into update mode; i.e., data that successfully can be retrieved by ROT do not need the update strategy.

Water already shows good speedup with the old protocol—an improvement of only 1 percent in execution time for the most successful combination of update strategies. Its node hit rate was cut from 0.5 percent to only 0.08 percent as can be seen in Table 9.7. In this application we see a larger increase in the communication of the top bus—11 percent compared to the original 6 percent. The increase of 5 percent relates almost exactly to the 47 percent expected by losing the locality.

The scheduling of work differs for the three versions of the same program, MP3D, MP3D-DIFF, and MP3D-DIFF-FS, all result in roughly the same node miss rate using the new protocol (0.3 percent-0.4 percent) as can be seen in Figure 9.8. The three versions all implement the same algorithm and differ only in the way they distribute their work. The adaptive write update compensates for poor distribution,

Water 192 Molecules 2x8x1					
	Old	Without	With Update Reset		
		No Ucache		With Ucache	
		4S states			8S
HR Dcache (%)	99.3	99.0	98.5	98.6	98.6
HR AM (%)	27	64	94	93	94
HR UCache (%)	-	-	-	93	94
Node miss (%)	0.5	0.3	0.09	0.09	0.08
Node read miss	240k	220k	75k	75k	70k
Node write miss	180k	70k	13k	13k	13k
Updates	-	316k	770k	700k	720k
NW weighted sum	840k	896	946k	876k	846k
Read line	630k	610k	460k	462k	460k
Read word	2k	500k	460k	790k	810k
Write line	380k	300k	170k	410k	390k
Write word	410k	840k	4556k	990k	1000k
M bus weighted sum	2.4M	3.2M	6.3M	3.5M	3.5M
Busy rate M bus	7	7	12	7	7
Busy rate DDM	10	15	15	14	14
Busy rate Top	6	8	11	11	11
Exec. time (rel)	1.00	0.995	0.995	0.990	0.990

Table 9.7: Statistics from the subscribe protocol running Water 192 Molecules.

unavoidably paid for in terms of communication. The Dcache hit rates differ between the versions, though, producing different performance results.

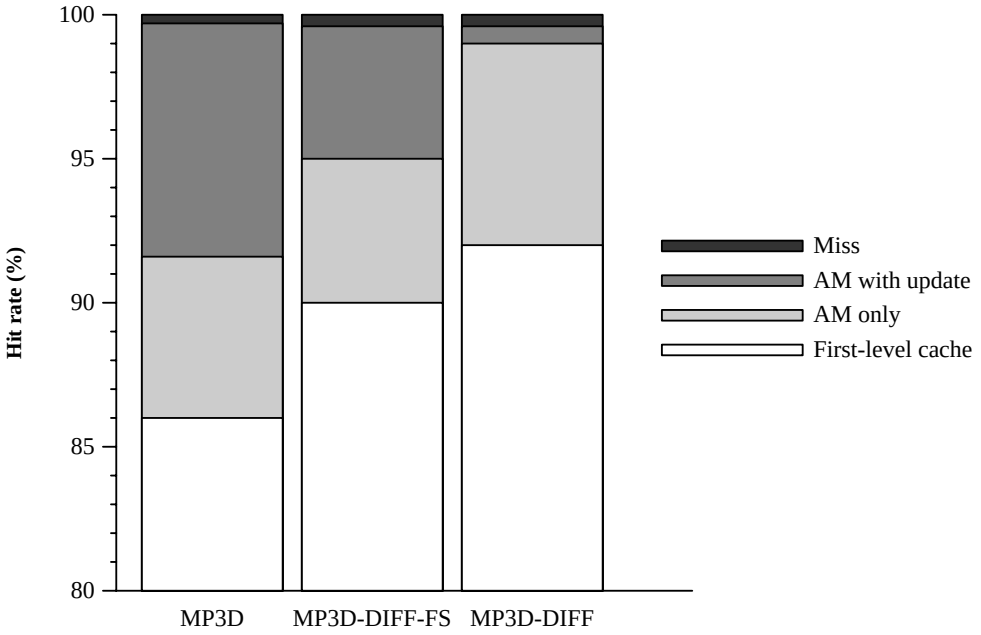


Figure 9.8: Comparing the effects of the first-level cache, the second-level attraction memories, and the update protocol for the three versions of MP3D. Despite their different distribution algorithms, the sum of these effects is constant.

⋆ ⋆ ★ ⋆ ⋆

An adaptive write-update protocol was introduced to remove all coherence misses. The described protocol removed misses from very difficult applications, while producing an unexpectedly low communication overhead. Most effects of false sharing and migratory objects were tamed by the protocol. Lacking examples, we could not demonstrate its suitability for producer-consumer applications. MP3D-DIFF gained an improved execution speed of 18 percent. Sometimes, improvements in the execution speed could not be observed, even though the misses were gone. One reason was the communication bottleneck of our relatively slow buses. Another source is efficiency reduction in the processor caches, which are unmodified and blocking and thus not able to utilize the pipelining of writes possible in this protocol.

The described protocol would benefit by a higher transaction frequency in the network, but higher bandwidth by an increased item size would not help much. It would also benefit by processor caches with a write buffer for pipelining writes, and with the ability for reads to bypass writes.

10

High-Performance Hierarchical Networks

THREE important properties of a multiprocessor network are remote-access latency, available communication bandwidth, and transaction frequency. Another—seldom discussed—property is the ability to preserve the order among transactions, helpful in protocol design. The properties of an order-preserving network can reduce latency, but also communication, since cache coherence can be implemented in a more straightforward manner.

The top bus, or top node, of a hierarchy would become a bottleneck unless significant locality in communication were explored by the application. Locality in communication has been explored in some applications, of course, as a result of hierarchical abstraction; however, demanding such behavior of an application would degrade the generality of the architecture. Thus, a hierarchical topology should be supplied with a higher bandwidth closer to its root.

COMA's ability to migrate data to where they are used increases the importance of communication locality. The extensive replication and migration in a COMA also lowers the demand for bandwidth somewhat.

10.1 Properties of a Hierarchical Network

A positive property of a hierarchical network is its ability to explore communication locality in an application. Two processors in a subsystem of the hierarchy may interact with each other without generating any traffic outside their common subsystem. In Chapter 7 we showed that some of the applications studied had some communication locality even if they were originally written for a uniform-memory architecture with one common bus. We also evaluated two modified versions of the applications, where as much as 80 percent of all traffic on the lowest bus was local to that bus (Table 7.1).

Another positive property is the ability of a hierarchical network to preserve the ordering of transactions. This simplifies cache-coherence protocol design, but could also have performance benefits for write delay in a sequentially-consistent protocol.

For a COMA cache-coherence protocol, a hierarchical search algorithm can be implemented in the network, by which the item can be located on a read miss.

As discussed previously, one of the major disadvantages of a hierarchical network is the bottleneck at its root. In a mesh network, the bisectional bandwidth, i.e., the bandwidth available for communication between the two halves of the system, increases by the square root of the number of processors [Len91]. This is not true for a hierarchical network. Its topmost component in the hierarchy could easily become the system's bottleneck since its available bandwidth does not increase with the number of processors in a natural way. Here, we will look at several methods for improving the bandwidth at the root of a hierarchy, while preserving some of its positive properties.

10.1.1 Sequential Consistency in a Multiprocessor

Caching allows multiple copies of data in a multiprocessor, and therefore several accesses can be performed in parallel to the shared memory. But what ordering of parallel accesses can be assumed by the programmer? Several *memory-access order models* of different consistency levels exist. A consistency level can be viewed as a specification of a memory's behavior. Hardware designs fulfilling this specification can correctly execute programs assuming that behavior. As we shall see, it can cost performance to maintain a high consistency level in multiprocessors.

Sequential consistency is the strongest consistency level and the one most often used by programmers. The term was first defined by Lamport [Lam79]:

Definition 1 (Sequential consistency) *[A system is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program.*

Uniprocessor systems with a single memory fulfill this requirement in a natural way. Multiprocessors, especially those with arbitrary networks, often have to forfeit performance in order to maintain a stronger consistency level. Normally, sequential consistency is maintained in a system if all processors issue their accesses in program order, and no access is performed by a processor until its previous access has been globally performed [SD87].[1] This is an often used but not necessary condition.

Sequential consistency is often maintained in an architecture built with a general network upon a write to a shared datum, a write request is sent to the home node which in turn sends out invalidation messages to all caches with shared copies. The caches respond with acknowledge messages. All acknowledges must be received before the write can be performed in order to maintain sequential consistency. The latency of a write is thus:

$$L_{write} = L_{to.home.node} + max(L_{inv.to.share.node} + L_{ack.to.req.node}).$$

10.1.2 Sequential Consistency in Hierarchical Networks

A hierarchical network is a *race-free network* as defined by Landin et al. [LHH91]:

Definition 1 (Race-free Network) *A race-free network is a network with the topology of any acyclic, undirected, network graph. Transactions propagate on the arcs in the network without the possibility of overtaking each other. Transactions may be buffered in the network nodes, but buffers must maintain a strict FIFO ordering of transactions.*

In a race-free network, sequential consistency can be maintained more easily than in a general network. First of all, the ordering of transactions from a node is observed by all other nodes. Secondly, *causal correctness* [Sch89] is maintained, i.e., any transaction that has reached a node X in a race-free network before X issues a new transaction B will also have reached any other node Y before it receives B. Lastly, sequential consistency is maintained if 1) a reading processor is stalled until the data reply is received, and 2) a writing processor is stalled until the write is performed on the *root node*; i.e., the node in a hierarchy that currently is the top of the subsystem in which all copies of the datum reside.

[1] This also assumes a level of coherence called *general coherence* [Sch89].

This property of a race-free network implies that a write acknowledge can be sent directly by the directory on top of the subsystem that contains all the copies of the datum—the root node. The latency of a write is thus:

$$L_{write} = L_{to.root.node} + L_{from.root.node}$$

Worst-case latency for the two terms above is proportional to the number of levels in the hierarchy, i.e., $\log_b n$ where b is the branching factor and n is the number of processors in the system.

10.2 Increasing the Bandwidth

A hierarchical network has two apparent bottlenecks:

- Directory size grows the higher one climbs in the hierarchy. The largest directories are close to the top. Practical size limit on these directories therefore limits the size of the system.
- Although some memory accesses can be localized in such a network, the higher level may nevertheless demand a higher bandwidth than what can be provided by the single root, creating a bottleneck. Snooping in the large directories makes the top bus slower rather than faster.

A way of taking the load off the higher buses is to have a smaller branch factor at the top of the hierarchy [VJS88]. This solution, however, makes the higher directories larger rather than smaller. It also increases the number of levels in the system, which has a negative impact on latency.

The bandwidth of a network is the product of the possible transaction frequency and the size of the data block communicated. Often, transaction frequency is the limiting factor on communication means in a multiprocessor. This is the case for the DDM bus of the prototype. Bandwidth can be increased simply by increasing the size of the communicated data block and widening the bus. A higher bandwidth per se is not beneficial for all applications and protocols. An equally important property is transaction frequency. Applications with fine-grained sharing of read/write data do not benefit at all by the high bandwidth of a large communication block. Transaction frequency is also important in a write-update protocol. Large items may create positive prefetching effects and negative false-sharing effects. If a ROT prefetcher is used, prefetching effects can be explored and false-sharing effects avoided with small items.

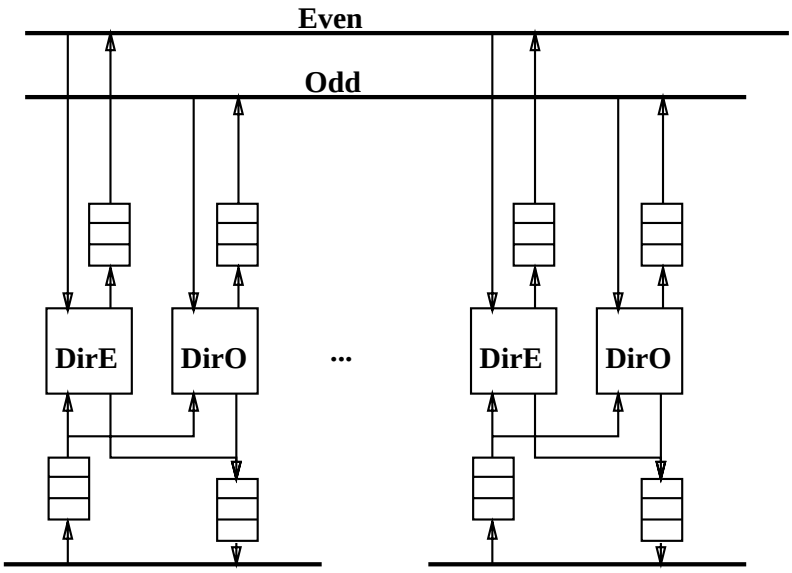


Figure 10.1: Hierarchy built of a fat tree with two top buses.

10.2.1 Fat Tree

Splitting the higher buses into a fat tree [Lei85] solves both the directory growth and root-bandwidth problems. Each directory is split into two directories of half the size. The directories deal with different halves of the address space (even and odd). The number of buses above is doubled, and each bus deals with its own address space, as shown in Figure 10.1. Repeated splits will make a bus as wide as needed, and directories as small as needed. Splitting is possible at any level. Regardless of the number of splits, the architecture is still hierarchical to each specific address. A fat tree increases not only the bandwidth, but also transaction frequency. It also supports communication locality.

The drawback of a fat tree compared to a plain hierarchy is that transactions from the same subsystem, but carried on different top buses, may overtake each other. The race-free property is violated. The protocol described so far would not fulfill sequential consistency on a fat tree. A protocol similar to that of a general network with acknowledges from the leaves must be used. There is no practical limit to how many times the split can be performed for a fat tree. Thus, a fat tree can be regarded as a scalable network.

One example with a fat tree used for the network is the new architecture by Thinking Machines Inc., the CM5 [Mac91]; another is the Wisconsin Multicube [GW88]. The Multicube architecture has $\sqrt{n}$ buses running horizontally and $\sqrt{n}$ buses running vertically. The n nodes are located at the interconnection points. The node has the combined functionality of a processor node and a directory. The network can therefore be labeled a fat tree with $\sqrt{n}$ top buses.

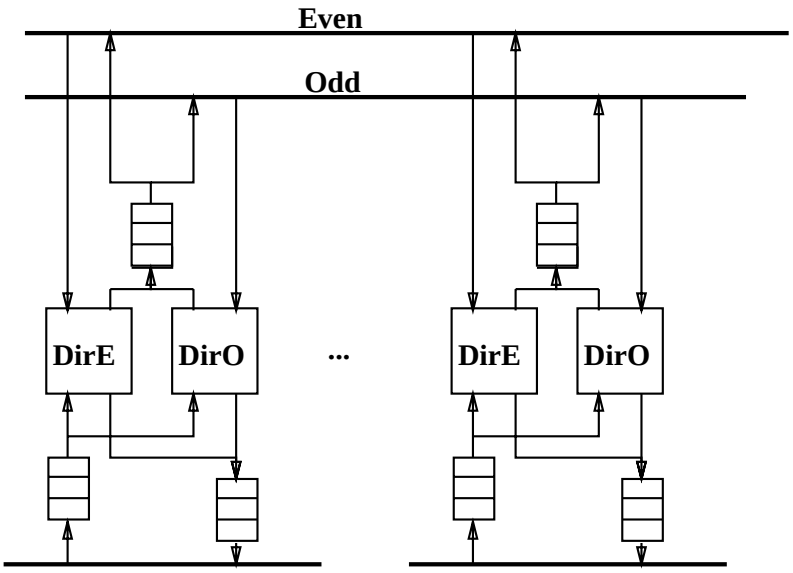


Figure 10.2: Increasing the bandwidth of a bus by splitting buses while maintaining transaction order.

10.2.2 Race-free Split

Transactions on different split buses can be prevented from overtaking each other by restrictions in the buffers, as shown in Figure 10.2. Each subsystem has a joint buffer for the two top buses. The subsystem arbitrates for the top bus in accordance with the first transaction in its buffer, which is sent before the second transaction is handled. To fully utilize available bandwidth, the different subsystems must supply transactions bound for different split buses.

Restricted splitting relies on split buses running synchronously and transactions initiated first being finished first. A need for synchronous implementation limits the physical distances between split buses, which might cause problems in a multiway split. Splitting at several layers also becomes complicated, introducing the need for extra sequencing logic [LHH91].

10.2.3 Logical Splitting

Splits can also be performed logically on a single bus. The limiting factor on a snooping bus is not the available bandwidth of the bus, but the transaction rate. The transaction rate is limited by snoop-lookup time rather than bus-transfer time. Split-state memory could help here as well. If each node has a split directory and the transactions on the bus are interleaved (even/odd), transaction frequency on the bus will be almost doubled. A balanced bus design using today's memory and bus technology would probably employ a four-way split on a single bus, almost

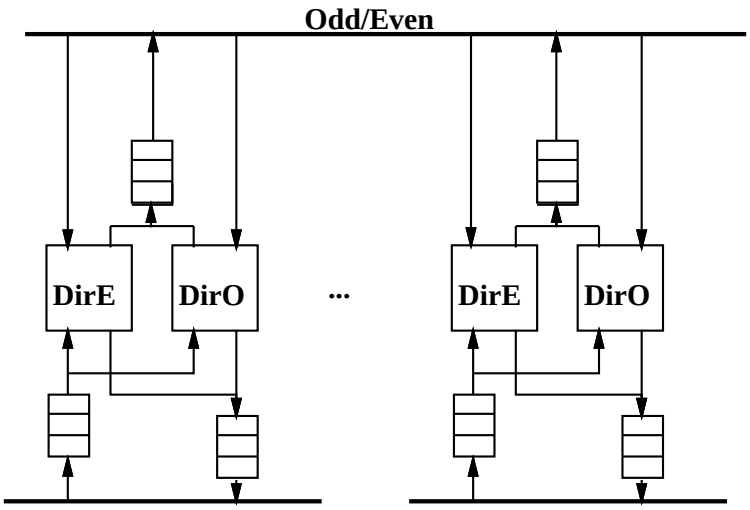


Figure 10.3: Increasing the transaction frequency of a single bus with a logical split.

quadrupling transaction frequency. This leaves us with a demand to transfer four times the amount of data.

Transferring a data burst on a bus, where you know the receiver is ready to receive, could be very fast—like the FIFO-to-FIFO transfers of the DDM bus. Letting the sender generate the data clock at the rate at which the receiver is guaranteed to receive removes most of the clock skew and allows for high clock frequencies for transferring the data.

10.2.4 Heterogeneous Hierarchical Networks

If unable to preserve ordering of transactions, a protocol cannot rely on the technique described for the race-free networks. Instead, a protocol similar to the general-network protocol, with acknowledges from the leaves, can be used [LHH91].

A network may consist of several nonsplit hierarchies connected by a general network—a heterogeneous network. The leaf-acknowledge protocol need only incorporate the tops of the hierarchies. When the top of the hierarchy performing the write has received acknowledges from all the other tops, the acknowledge can be sent to its lower levels, as in the nonsplit hierarchical case.

Write latency in the general part of the network is identical to those of cache-coherence on a general network:

$$L_{write} = L_{to.dir.node} + max(L_{inv.to.share.subs} + L_{ack.to.req.sub}).$$

However, this latency is accounted for solely in the general part of the network.

⋆ ⋆ ★ ⋆ ⋆

A hierarchy can explore the communication locality found in an application. A plain hierarchy has advantages for protocol design in that the ordering of transactions is preserved, but suffers from contention at its root. The fat tree is a scalable solution to the bandwidth problem, but does not preserve the ordering of transactions.

We proposed several schemes for improving the bandwidth of a hierarchy by splitting its root, while preserving the ordering of transactions. The methods can improve the transaction frequency by a large factor, but none of them can be said to be scalable and are only practical up to a limited number of processors. The heterogeneous network combined part of the advantage from the hierarchy with the bandwidth available from a general network.

SUMMING UP

11

Related Work

The potential and promises of parallel processing have attracted researchers since the days of ENIAC. The proposed solutions have changed as technology has shifted. Proposed architectures also differ according to what programming paradigm is supported. Single-instruction multiple-data *(SIMD) machines offer a data-parallel view to the programmer wherein all processors perform identical instructions on different parts of the data simultaneously, e.g., the Connection Machine. In* multiple-instructions multiple-data architectures *(MIMD), the processor may perform different operations on different sets of data simultaneously. MIMD architectures can be divided into two subgroups, those with a private address space for each processor,* message-passing architectures, *and those with one common address space shared by all processors,* shared-memory architectures. *In message-passing architectures, communication takes place through explicit messages between the processors, while communication in shared-memory architectures is implicit and often handled by the cache-coherence protocol integrated into the processor caches. Vector processing and heavily pipelined computers can also be regarded as parallel processing.*

This chapter will try to cover two areas related to the work presented here: hierarchical architectures and COMA-related architectures.

11.1 Hierarchical Architectures

It was natural to pursue the hierarchical route when the bandwidth of a single bus became the limiting factor.

11.1.1 Encore Architectures

Wilson proposed a hierarchical architecture in his thesis in 1985 [Wil85]. That work later became the Gigamax proposal at Encore Computer Corporation. The Gigamax is an architecture with a hierarchy of buses with snooping caches between the two levels of buses in the hierarchy. There is full inclusion between the layers in the hierarchy. At the lowest level, 16 processors are tied together by a bus to form a cluster. The large cluster caches are second-level caches common to all processors in the cluster and the interface to the top bus. The state memory in the cluster cache is duplicated and snoop transactions on the top bus. Thus, invalidations and interceptions needed by the hierarchical cache-coherence protocol can be implemented.

The Gigamax was never brought to market. Instead, elements were used in the S93 architecture [Cor91]. S93 is also a hierarchical architecture but with a somewhat different structure. At the lowest level are clusters, similar to a DDM cluster,[1] hosting four MC88100 processors and their caches. The processor caches are tied together by a proprietary bus (Ibus), rather than the Motorola M bus, and subsequently have interfaces between each processor-memory pair and the bus. The cluster hosts a secondary cache of 1 Mbytes, which is interfaced to the top bus, called the Nanobus, with a throughput of 100 Mbyte per second. The cluster caches snoop the traffic on the Nanobus. The Nanobus also hosts a common shared memory, sized from 32 Mbytes up to 640 Mbytes. Up to eight cluster can be tied together hosting a total of 32 MC88100 processors.

DDM relates to this work in its use of snooping caches and the manner in which the cache-coherence protocol of the MC88204 is integrated with the higher-level cache-coherence protocol.

11.1.2 TREEB

The TREEB architecture [VJS88] is a hierarchical cache architecture, extendible to any depth, with the shared memory at its top. The caches at the leaves are said to be large given "the density of modern-day dynamic RAMs." All caches (at the leaves and between layers) have two separate two-bit state memories per block

[1] Described in Chapter 6.

(item), one for above and one for below. Both status memories are updated in an atomic action. A bus request is either serviced by another processor cache or by the directory above. The directory might either service the request itself or send the request above. Other directories on the same level service the request, or it continues on up the hierarchy.

Reads are allowed if the cache above is in the Clean or Dirty state. If not, the request is issued on the next higher bus. Read requests on a bus are serviced by the owner (there is only one owner per bus); otherwise, they are issued to the next higher bus.

Writes are only allowed for the Dirty state. Otherwise, an invalidate signal is issued on the bus above. The invalidate will propagate upwards until it reaches a directory in Exclusive, or the shared memory. The directory will reply with an invalidate acknowledge. The invalidate will also propagate downwards to all caches underneath the directory.

The DDM is related to TREEB by its being a hierarchy. However, TREEB has a shared memory at the top and its interlevel directories store both state information and data. There are no transient states for reads and writes in the system. The caches, directories, and buses are suspended and wait for replies. One interesting aspect of the architecture is its one owner per bus, removing the need for selection on a read request. This cuts down the bus cycle and makes the consistency somewhat simpler.

Instead of simulating a TREEB with thousands of processors, an analytical model is used, combined with a synthetic work load.

11.1.3 Paradigm

Paradigm [CGB91] is a continuation of an earlier hierarchical architecture named VMP-MP [CGB89] under development by the Distributed Systems Group at Stanford. Paradigm also consists of a hierarchy of buses with full-inclusion caches between each level. It differs from the DDM and the Wilson architectures by utilizing a directory-based cache-coherence protocol, rather than a snooping protocol. A directory entry in the hierarchy has one presence bit for each subsystem of the lower level (toward the leaves of the hierarchy). The directory entry also has two additional code bits and a lock bit. The code bits can be in private, shared, request_notification, or undefined states.

Paradigm also has an additional switching network tying the leaves of the hierarchy together. This entitles each leaf to have access to the I/O system. A leaf consists of four processors, each with its own on-chip virtual cache, hosted on a board together with a shared on-board virtual cache of 512 kbytes and some local memory. Up to eight board caches are tied together by the *group bus* to the *interbus cache module*.

The interbus cache module, hosting a large cache, interfaces to the top bus, the *memory bus*, connected to the shared memory. The Paradigm protocol integrates synchronization primitives into all levels of caches.

The DDM is related to the Paradigm in its being a hierarchy. The delay of the Paradigm network is similar to the one reported for our DDM prototype in Chapter 6.

11.1.4 Memory Hierarchy Network

Data moving closer to the processor accessing it can be found in the *memory hierarchy network* architecture of Mizrahi [MBLZ89]. The network of this architecture is a binary tree with storage capabilities in each network node. The main idea is to move the datum one step closer to the processor accessing it. Different thresholds for when, and how far, to move the datum have been evaluated. The main idea is to move read-write shared data to the top of the subsystem including all processors sharing the datum.

The memory overhead of that architecture is much bigger than in the DDM, because of a low branch factor and full inclusion. The architecture is equipped with a shared memory at its top.

11.2 COMA-related Architectures

11.2.1 Distributed Virtual Shared-Memory Systems

Distributed virtual shared-memory (DVSM) systems implement one form of COMA in software on standard workstations connected by a local-area network (LAN). Item size in DVSM is equal to page size in the MMU. Traps from the MMU are the hooks into the cache-coherence protocol. A shared item (page) resident in the local memory is tagged with read permission, and an exclusive item is tagged with read and write permission. A software-based implementation of this type has been proposed by Li and Hudak [LH89]. A page in Shared state in a node has a page descriptor pointing to the physical location in the local memory of the node and its page privileges set to *read only*. Reads to the page are performed by a normal read operation, including an indirection by the MMU to the physical location in the local memory where the datum is kept. A write to the page will cause an MMU fault, waking up the coherence mechanism. Other processors storing copies of a page are sent invalidation messages. Upon the reception of acknowledges from all other nodes with a shared copy, the processor may perform its write.

In the case of a read miss to a page, i.e., the page is not in the local memory of the node, the current location must be searched for, similar to a COMA read miss. One proposed scheme is to allocate a specific home node for each page, storing information about where the page currently resides. A read request first goes to the home node of the page, to find out in which node a valid copy exists. One extra hop to that node is often needed before the data value can be returned to the requesting node by a third hop. In DVSM terms this is called the *fixed distributed manager algorithm* [LH89].

Yet another solution has been proposed, where the home has the ability to move and where sometimes the third hop can be avoided. Instead of having a fixed home location for the page, the global page information can be stored in any node. In order to avoid searching all the nodes sequentially on a read miss, a dynamic per-page pointer scheme is maintained in the nodes, pointing to the node that is believed to currently have a copy of a page. This is called the *dynamic distributed manager* [LH89].

The DDM is related to the DVSM work in its relaxation of mapping to physical memory.

11.2.2 J-machine

The J-machine is a message-passing architecture developed at M.I.T. [DW89]. It supports a low overhead context switch and has communication integrated into its instruction set, enabling it to efficiently emulate caches and cache coherence in software. A hierarchical cache-coherence protocol has been developed [Wal90]. The network of the J-machine is a 3-D mesh; the hierarchy only exists logically. Different address domains have different logical hierarchies, removing some of the hot-spot phenomena of a single top in a hierarchy.

The protocol has a hierarchical directory representation similar to Paradigm's. There is, however, no shared memory at the top introducing the COMA problems of finding a datum upon a read miss or not losing the last copy of a datum upon replacement. It finds a copy of an item by a hierarchical scheme similar to the DDM's, but the data reply is returned on the available shortcuts in the physical topology rather than back along the logical search path. The last datum is kept track of by always having one responsible node, *the owner*, for shared data. The owner replaces its copy of the datum with care, while others can simply destroy their copies on replacement.

The DDM is related to this work by being a hierarchical COMA.

11.2.3 Link-Based DDM

An alternative link-based DDM scheme has been developed at the University of Bristol [RW91]. Compared to the bus-based DDM, its protocol has a different representation, which is suited to a link-based architecture structured like a tree, rather than a bus-based one. It has certain similarities with the Paradigm and the J-machine protocols.

An emulator of the DDM has been developed at the University of Bristol [RW91]. The emulator runs on the Meiko Transputer platform. The modeled architecture has a tree-shaped link-based structure with Transputers as directories. Its four links allow for a branch factor of three at each level in the model. 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 can evaluate large data sets running on many Transputers. Even though the overhead of emulating the protocol on a Transputer is high, some actual speedups have been reported. A newly started ESPRIT project, EMI/Horn, will define a hardware-based architecture based on this idea.

The bus-based and link-based DDMs are related through their common origin [WH88] and common protocol [HHW90].

11.2.4 KSR1

As mentioned earlier, a commercial COMA architecture has recently been announced by Kendall Square Research [BFKR92]. The KSR1 architecture is similar to the DDM in many ways. It differs as well. It uses large items of 128 bytes and suffers from a much larger remote-access delay. Its processors run at 20 MHz. The network consists of a ring-based hierarchical structure with a branching factor of 32 at each level. Its processor data caches, sized 256k bytes, are accessed in two cycles. An access to its AM takes 18 cycles. A remote access satisfied by the lowest ring yields a delay of 126 cycles at 20 MHz, while an access climbing yet another level in the hierarchy takes 600 cycles [Dun92]. We have been unable to obtain any detailed information from KSR.

The KSR1 offers an attractive bandwidth of 1 Gbytes per second on its rings, but suffers greatly from its long remote-access delay. It is built of proprietary processors rather than commercially available ones. We believe in adapting to existing microprocessors to stay on the processor technology curve, which has recently taken a few giant steps. We do not fully understand the causes of the long delays on the rings. The proposed ring bus by Barossa and Dubois [BD91] would drastically cut that delay.

The DDM is related to this work by being a physically hierarchical COMA. However, it has much shorter remote latency, smaller item size, and uses commercially available processors. Further details of the KSR1 implementation is to be found in Appendix C.

11.3 Cache-Coherent NUMAs

The work presented here also has a lot in common with cache-coherent NUMAs in general in that the same cache-coherence problems are solved. A survey of cache-coherence protocols can be found in Chapter 3, and a brief description of one of them, Dash, is found in Chapter 4.

Summary

We have identified and described a new class of architectures—cache-only memory architectures (COMA)—with its memory system comprised solely of caches. We isolated four COMA properties enabling it to suit different types of applications. COMA showed a performance superior to non-uniform memory architectures (NUMA) for a selection of programs in a quantitative analytical study. Our study also showed that COMA architectures are less sensitive to network delays and also less dependent on large second-level caches.

One instance of COMA, the Data Diffusion Machine, was described down to the level of an implementation proposal. The proposal included a cache-coherence protocol as well as solutions to the COMA-specific problems of finding data and replacing with care. The DDM's performance was evaluated in a detailed simulation study. The DDM performs well for programs written with a completely different architecture in mind. The COMA benefits increase with the problem size. It was also shown how even better performance can be achieved for some applications by only slight modification of the programs.

The misses remaining in an architecture with large caches were attacked by supplying a dynamic hardware-prefetching scheme and an adaptive write-update cache-coherence protocol. The prefetching removed half of the remote misses for one application, while the adaptive write-update strategy reduced the misses for another application by more than ten times. However, both methods need a network with higher transaction frequency than what can be offered by the relatively slow busses of the current DDM design in order to have full effect. We have also presented several methods for achieving higher transaction frequency.

A prototype implementation of the DDM is near its completion at SICS. Although we presented detailed simulation studies in this work, the real performance can only be studied when running real-sized problems on a real machine.

Conclusion

The three negative attributes of multiprocessors—they are hard to program, they do not achieve the expected performance, and they take a long time to develop—can be reduced by a COMA.

COMA can be regarded as more general than NUMA in that it supports a larger variety of program behaviors. It provides the shared memory paradigm to the programmer and does not require advanced optimizations to run well.

COMA performance is superior to NUMA for the studied applications. It is less sensitive to network latency—important for large machines and future technology.

If some of the suggested optimization techniques are applied, its sensitivity for network latency is further reduced, but demands higher transaction frequency. None of the optimizations put any extra demand on the programmer.

A COMA implementation can be made simple in that it integrates the three functionalities found in a NUMA—local memory, directory, and remote access cache—into one unit: the attraction memory. The development complexity of a COMA is therefore low.

We believe that a COMA of the future focuses on simplicity, to allow for a short development time, and transaction frequency, to improve performance, while providing an efficient shared-memory abstraction to the programmer.

Epilogue

"Fascinating," responds the computer freak. We are the only ones left at the party. The host is asleep. "What do *you* work with?" I ask to be polite. "I work with computers," he replies while watching my face ...

References

[AI83] Arvind and R.A. Iannucci. Two Fundamental Issues in Multiprocessing: the Dataflow Solution. MIT/LCS/TM 241, MIT, 1983.

[Amd67] G. M. Amdahl. Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities. In *AIFPS Conference Proceedings*, pages 483–485, 1967.

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

[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.

[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.

[BGW89] D. L. Black, A. Gupta, and W-D. Weber. Compatitive management of distributed shared memory. In *Proceedings of Compcon*, 1989.

[Bit90] P. Bitar. A Critique of Trace-Driven Simulation for Shared-Memory Multiprocessors. In *Cache and Interconnect Architectures in Multiprocessors*, pages 37–52. Kluwer Academic Publisher, Norwell, Mass, 1990.

[BS92] M. Brorsson and P. Stenström. Visualising Sharing Behavior in relation to Shared Memory Management. In *ICPADS '92, International Conference On Parallel And Distributed Systems*, 1992.

[BSF+91] W.J. Bolosky, M.L. Scott, R.P. Fitzgerald, R.J. Fowler, and A.L. Cox. NUMA Policies and Their Relation to Memory Architecture. In *Proceedings of the 4th Annual Architectural Support for Programming Languages and Operating Systems*, pages 212–221, 1991.

[BW88] J-L. Baer and W-H. Wang. On the Inclusion Properties for Multi-Level Cache Hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 73–80, 1988.

[CCS87] J.H. Chang, H. Chao, and K. So. Cache Design of A Sub-Micron CMOS System/370. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 208–213, 1987.

[CF78] L.M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, 27(12):1112–1118, December 1978.

[CFKA90] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-Based Cache Coherence in Large-Scale Multiprocessors. *IEEE Computer*, 23(6):49–58, June 1990.

[CGB89] D.R. Cheriton, H.A. Goosen, and P.D. Boyle. Multi-Level Shared Caching Techniques for Scalability in VMP-MC. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 16–24, 1989.

[CGB91] D.R. Cheriton, H.A. Goosen, and P.D. Boyle. Paradigm: A Highly Scalable Shared-Memory Multicomputer Computer. *IEEE Computer*, 24(2):33–46, February 1991.

[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.

[Cor91] Encore Computer Corporation. Encore 93 Series Technical Summary. 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.

[DSB86] M. Dubois, C. Scheurich, and F.A. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, 1986.

[Dub91] M. Dubois. A New Solution to Coherence Problems in Multicache Systems. In *Supercomputing '91*, pages 197–206, 1991.

[Dun92] T. H. Dunigan. Kendall Square Multiprocessor: Early Experiences and Performance. Technical Report ORNL/TM-12065, Oak Ridge National Laboratory, 1992.

[DW89] W. J. Dally and D. S. Wills. Universal Mechanism for Concurrency. In *Proceedings of Parallel Architecture and Languages Europe*, pages 19–33. Springer-Verlag, 1989.

[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.

[FW78] S. Fortune and J. Wyllie. Relations Between Concurrent-Write Models of Parallel Computation. In *Proceedings of the Tenth ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[GGH91] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *Proceedings of the 4th Annual Architectural Support for Programming Languages and Operating Systems*, 1991.

[GGH92] K. Gharachorloo, A. Gupta, and J. Hennessy. Hiding Memory Latency using Dynamic Scheduling in Shared-Memory Multiprocessors. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 22–33, 1992.

[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.

[GJS92] A. Gupta, T. Joe, and P. Stenström. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. TR #CSL-TR-92-524 Stanford University, 1992.

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 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.

[Goo89] J. R. Goodman. Cache Consistency and Sequential Consistency. Technical Report 61, SCI Committee, 1989.

[Gri92] M. Grindal. Evaluation of Latency-Hiding Techniques for the Data Diffusion Machine. Final work for Undergraduate Studies, KTH, February 1992.

[GVW89] J. R. Goodman, M. K. Vernon, and P.J. Woest. Efficient Synchronization Primitives for Large-scale Cache-Coherent Multiprocessors. In *Proceedings of the 3rd Architecture Symposium for Programming Languages and Operating Systems*, pages 64–75, 1989.

[GW88] J.R. Goodman and P.J. Woest. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture, Honolulu, Hawaii*, pages 422–431, 1988.

[GW92] A. Gupta and W-D. Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 41, 1992.

[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.
Also in the Proceedings of Paralle Architectures and Languages Europe (PARLE) 1989.

[Hil90] M. D. Hill. What is Scalability? *Computer Architecture News*, 18(4):18–21, December 1990.

[HL91] E. Hagersten and A. Landin. An Initial Attempt to a General Network COMA. DDM-memo, Swedish Institute of Computer Science, August 1991.

[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.

[JLGS90] D. James, A.T. Laundrie, S. Gjessing, and G.S. Sohi. Scalable Coherence Interface. *IEEE Computer*, 23(6):74–77, June 1990.

[Jou90] N. Jouppi. Improving Direct-Mapped Cache Performance by Addition of a Small Fully-Associative Cache and Prefetch Buffer. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, 1990.

[Kat85] R.H. Katz. Implementing a Cache Consistency Protocol. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 276–283, 1985.

[KL91] A.C. Klaiber and H.M. Levy. An Architecture for Software-Controlled Data Prefetching. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 43–53, 1991.

[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.

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

[Lan92] A. Landin. Personal Communication. Swedish Institute of Computer Science, 1989-1992.

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

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

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

[LH89] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[LH91] A. Landin and E. Hagersten. The DDM Node Controller User Manual. DDM-memo, Swedish Institute of Computer Science, Feb 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.

[Löf90] M. Löfgren. A Simulator Written in C++ for a Parallel Architecture. Final work for Undergraduate Studies, KTH, November 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.

[Mac91] Thinking Machines. The Connection Machine CM-5 Technical Summary. October 1991.

[Mag93] P. Magnusson. A Design for Efficient Simulation of a Multi-Processor. In *Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 1993.

[MBLZ89] H. E. Mizrahi, J-L Baer, D.E. Lazowska, and J. Zahorjan. Introducing Memory into the Switch Elements of Multiprocessor Interconnection Networks. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 158–176, 1989.

[MCS91] J.M. Mellor-Crummey and M.L. Scott. Synchronization Without Contention. In *Proceedings of the 4th Annual Architectural Support for Programming Languages and Operating Systems*, pages 269–278, 1991.

[Mot89a] Motorola. *MC88100–RISC Microprocessor, User's Manual.* Prentice Hall, New Jersey, 1989.

[Mot89b] Motorola. *MC88200–Cache/Memory Management Unit, User's Manual.* Prentice Hall, New Jersey, 1989.

[NA91] D. Nussbaum and A. Agarwal. Scalability of Parallel Machines. *Communication of the ACM*, 34(3):57–61, March 1991.

[NPA92] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 156–167, 1992.

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

[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.

[Sch89] C. Scheurich. *Access Ordering and Coherence in Shared Memory Multiprocessors.* PhD thesis, University of Southern California, 1989.

[SD87] C. Scheurich and M. Dubois. Correct Memory Operation of Cache-based Multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 234–243, 1987.

[SJG92a] P. Stenström, T. Joe, and A. Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 80–91, 1992.

[SJG92b] P. Stenström, T. Joe, and A. Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Forth Swedish Workshop on Computer System Architecture*, January 1992.

[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.

[Smi78] B.J. Smith. A Pipelined, Shared Resource MIMD Computer. In *Proceedings of International Conference on Parallel Processing*, 1978.

[SR84] Z. Segall and L. Rudolph. Dynamic Decentralized Cache Schemes for an MIMD Parallel Processor. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 340–347, 1984.

[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.

[TD90] M. Thapar and B. Delagi. Stanford Distributed-Directory Protocol. *IEEE Computer*, 23(6):78–80, June 1990.

[TSS88] C.P. Thacker, L.C. Stewart, and E.H. Satterthwaite. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[VJS88] M.K. Vernon, R. Jog, and G.S. Sohi. Performance Analysis of Hierarchical Cache-Consistent Multiprocessors. In *Conference Proceedings of International Seminar on Performance of Distributed and Parallel Systems*, pages 111 – 126, 1988.

[Wal90] D. Wallach. A Scalable Hierarchical Cache Coherence Protocol. SB Thesis. MIT AI lab, May 1990.

[WBL89] W-H. Wang, J-L. Baer, and H. M. Levy. Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 140–148, 1989.

[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.

[Wil85] A.W. Wilson. *Organization and Statistical Simulation of Hierarchical Multiprocessors*. PhD thesis, CMU, 1985.

[Wul88] W.A. Wulf. The WM Computer Architecture Definition and Rationale Computer Science. Department of Computer Science, University of Virginia, TR-88-19, July 1988.

[ZB92] R.N. Zucker and J-L. Baer. A Pereformance Study of Memory Consistency Models. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 2–12, 1992.

A

Tables from the Analythical Model

Architecture		L_{cap}/ L_{coh}	MP3D DS:34k	Pthor 199k	Locus 77k	Water 13k	Chol 62k	LU 40k	B-H 25k	Ocean 151k	Avg all	Avg 4
33 MHz technology, cache size 4 kbytes, bmax = 16												
n = 16 processors												
N	mesh2	47/ 64	11	12	7	8	14	11	21	27	**14**	**10**
U	mesh3	47/ 63	10	12	7	7	13	11	20	27	**14**	**10**
M	h-link	44/ 59	10	11	7	7	13	11	19	25	**13**	**9**
A	h-bus	35/ 44	8	9	6	6	10	9	15	20	**10**	**7**
C	mesh2	16/ 74	9	9	4	5	7	5	8	11	**7**	**5**
O	mesh3	16/ 73	9	9	4	5	7	5	8	11	**7**	**5**
M	h-link	16/ 74	9	9	4	5	7	5	8	11	**7**	**5**
A	h-bus	16/ 60	8	8	4	4	7	5	8	11	**7**	**5**
Dir:	h-link	16/ 58	8	8	4	4	7	5	8	11	**7**	**5**
Dir:	h-bus	16/ 51	7	7	4	4	6	5	8	10	**6**	**5**
n = 64 processors												
N	mesh2	62/ 84	14	16	9	10	18	15	27	35	**18**	**13**
U	mesh3	55/ 75	12	14	8	9	16	13	24	31	**16**	**12**
M	h-link	53/ 71	12	14	8	8	15	13	23	30	**15**	**11**
A	h-bus	48/ 63	10	12	7	8	14	11	21	27	**14**	**10**
C	mesh2	16/ 94	11	11	5	5	8	5	8	12	**8**	**6**
O	mesh3	16/ 85	10	10	4	5	8	5	8	11	**8**	**5**
M	h-link	16/ 86	11	10	4	5	8	5	8	11	**8**	**6**
A	h-bus	16/ 78	10	10	4	5	7	5	8	11	**7**	**5**
Dir:	h-link	16/ 84	10	10	4	5	8	5	8	11	**8**	**5**
Dir:	h-bus	16/ 82	10	10	4	5	8	5	8	11	**8**	**5**
n = 256 processors												
N	mesh2	89/125	20	23	13	14	25	21	38	50	**25**	**18**
U	mesh3	68/ 92	15	17	10	10	19	16	29	38	**19**	**14**
M	h-link	55/ 74	12	14	8	9	16	13	24	31	**16**	**11**
A	h-bus	51/ 67	11	13	8	8	15	12	22	29	**15**	**11**
C	mesh2	16/135	15	15	5	7	10	5	8	13	**10**	**7**
O	mesh3	16/102	12	12	5	6	8	5	8	12	**8**	**6**
M	h-link	16/ 89	11	11	4	5	8	5	8	11	**8**	**6**
A	h-bus	16/ 82	10	10	4	5	8	5	8	11	**8**	**5**
Dir:	h-link	16/ 91	11	11	4	5	8	5	8	11	**8**	**6**
Dir:	h-bus	16/ 89	11	11	4	5	8	5	8	11	**8**	**6**
n = 1024 processors												
N	mesh2	143/205	31	36	21	21	40	33	61	81	**41**	**29**
U	mesh3	86/121	19	22	13	13	25	20	37	49	**25**	**18**
M	h-link	63/ 86	14	16	10	10	18	15	27	36	**18**	**13**
A	h-bus	67/ 92	15	17	10	10	19	16	29	38	**19**	**14**
C	mesh2	16/215	23	22	7	9	13	6	9	15	**13**	**9**
O	mesh3	16/131	15	14	5	6	9	5	8	13	**10**	**7**
M	h-link	16/101	12	12	5	6	8	5	8	12	**8**	**6**
A	h-bus	16/107	13	12	5	6	9	5	8	12	**9**	**6**
Dir:	h-link	16/119	14	13	5	6	9	5	8	12	**9**	**6**
Dir:	h-bus	16/121	14	14	5	6	9	5	8	12	**9**	**6**

Table A.1: Studying the variation in the average number of processor cycles per global data access.

Architecture		L_{cap}/ L_{coh}	Applications and Data Set Size Per Node: MP3D DS:34k	Pthor 199k	Locus 77k	Water 13k	Chol 62k	LU 40k	B-H 25k	Ocean 151k	Avg all	Avg 4
33 MHz technology, cache size 16 kbytes, bmax = 16												
n = 16 processors												
N	mesh2	47/ 64	10	10	5	6	6	5	11	16	**9**	**5**
U	mesh3	47/ 63	10	10	5	6	6	5	11	16	**8**	**5**
M	h-link	44/ 59	9	9	5	5	5	5	10	15	**8**	**5**
A	h-bus	35/ 44	7	7	4	4	4	4	8	12	**6**	**4**
C	mesh2	16/ 74	9	9	3	4	5	3	5	7	**6**	**4**
O	mesh3	16/ 73	9	8	3	4	5	3	5	7	**6**	**4**
M	h-link	16/ 74	9	9	3	4	5	3	5	7	**6**	**4**
A	h-bus	16/ 60	8	7	3	4	4	3	5	7	**5**	**3**
Dir:	h-link	16/ 58	8	7	3	4	4	3	5	7	**5**	**3**
Dir:	h-bus	16/ 51	7	7	3	3	4	3	4	7	**5**	**3**
n = 64 processors												
N	mesh2	62/ 84	13	13	6	7	7	7	14	21	**11**	**7**
U	mesh3	55/ 75	12	11	6	6	7	6	13	18	**10**	**6**
M	h-link	53/ 71	11	11	6	6	6	6	12	18	**9**	**6**
A	h-bus	48/ 63	10	10	5	6	6	5	11	16	**9**	**5**
C	mesh2	16/ 94	11	10	4	5	5	3	5	8	**6**	**4**
O	mesh3	16/ 85	10	10	4	4	5	3	5	8	**6**	**4**
M	h-link	16/ 86	10	10	4	4	5	3	5	8	**6**	**4**
A	h-bus	16/ 78	10	9	3	4	5	3	5	8	**6**	**4**
Dir:	h-link	16/ 84	10	10	4	4	5	3	5	8	**6**	**4**
Dir:	h-bus	16/ 82	10	9	4	4	5	3	5	8	**6**	**4**
n = 256 processors												
N	mesh2	89/125	19	18	9	10	10	9	20	29	**16**	**10**
U	mesh3	68/ 92	14	14	7	8	8	7	15	22	**12**	**7**
M	h-link	55/ 74	12	11	6	6	7	6	13	18	**10**	**6**
A	h-bus	51/ 67	11	10	5	6	6	6	12	17	**9**	**6**
C	mesh2	16/135	15	14	5	6	7	4	5	9	**8**	**5**
O	mesh3	16/102	12	11	4	5	6	3	5	8	**7**	**4**
M	h-link	16/ 89	11	10	4	5	5	3	5	8	**6**	**4**
A	h-bus	16/ 82	10	9	4	4	5	3	5	8	**6**	**4**
Dir:	h-link	16/ 91	11	10	4	5	5	3	5	8	**6**	**4**
Dir:	h-bus	16/ 89	11	10	4	5	5	3	5	8	**6**	**4**
n = 1024 processors												
N	mesh2	143/205	30	29	14	16	16	14	31	47	**25**	**15**
U	mesh3	86/121	18	18	9	10	10	9	19	28	**15**	**9**
M	h-link	63/ 86	13	13	6	7	8	7	14	21	**11**	**7**
A	h-bus	67/ 92	14	14	7	8	8	7	15	22	**12**	**7**
C	mesh2	16/215	23	21	6	8	10	4	6	12	**11**	**7**
O	mesh3	16/131	15	14	4	6	7	3	5	9	**8**	**5**
M	h-link	16/101	12	11	4	5	6	3	5	8	**7**	**4**
A	h-bus	16/107	13	12	4	5	6	3	5	8	**7**	**5**
Dir:	h-link	16/119	14	13	4	5	6	3	5	9	**7**	**5**
Dir:	h-bus	16/121	14	13	4	6	7	3	5	9	**8**	**5**

Table A.2: Studying the variation in the average number of processor cycles per global data access.

Architecture		L_{cap}/ L_{coh}	Applications and Data Set Size Per Node: MP3D DS:34k	Pthor 199k	Locus 77k	Water 13k	Chol 62k	LU 40k	B-H 25k	Ocean 151k	Avg all	Avg 4
33 MHz technology, cache size 64 kbytes, bmax = 16												
n = 16 processors												
N	mesh2	47/ 64	9	8	4	4	4	3	4	10	**6**	**4**
U	mesh3	47/ 63	9	8	4	4	4	3	4	10	**6**	**4**
M	h-link	44/ 59	8	8	3	4	4	3	4	9	**5**	**4**
A	h-bus	35/ 44	6	6	3	3	3	2	3	7	**4**	**3**
C	mesh2	16/ 74	9	8	3	4	4	2	2	5	**5**	**3**
O	mesh3	16/ 73	9	8	3	4	4	2	2	5	**5**	**3**
M	h-link	16/ 74	9	8	3	4	4	2	2	5	**5**	**3**
A	h-bus	16/ 60	7	7	3	3	4	2	2	5	**4**	**3**
Dir:	h-link	16/ 58	7	7	3	3	4	2	2	5	**4**	**3**
Dir:	h-bus	16/ 51	6	6	2	3	3	2	2	5	**4**	**3**
n = 64 processors												
N	mesh2	62/ 84	11	11	4	5	6	4	5	13	**7**	**5**
U	mesh3	55/ 75	10	10	4	5	5	3	4	11	**7**	**4**
M	h-link	53/ 71	10	9	4	5	5	3	4	11	**6**	**4**
A	h-bus	48/ 63	9	8	4	4	4	3	4	10	**6**	**4**
C	mesh2	16/ 94	11	10	3	4	5	2	3	6	**6**	**4**
O	mesh3	16/ 85	10	9	3	4	5	2	3	6	**5**	**4**
M	h-link	16/ 86	10	9	3	4	5	2	3	6	**5**	**4**
A	h-bus	16/ 78	9	9	3	4	4	2	3	6	**5**	**3**
Dir:	h-link	16/ 84	10	9	3	4	5	2	3	6	**5**	**3**
Dir:	h-bus	16/ 82	10	9	3	4	5	2	3	6	**5**	**3**
n = 256 processors												
N	mesh2	89/125	16	16	6	7	8	5	7	18	**10**	**7**
U	mesh3	68/ 92	12	12	5	6	6	4	5	14	**8**	**5**
M	h-link	55/ 74	10	10	4	5	5	3	4	11	**7**	**4**
A	h-bus	51/ 67	9	9	4	4	5	3	4	10	**6**	**4**
C	mesh2	16/135	15	14	4	5	7	3	3	7	**7**	**5**
O	mesh3	16/102	12	11	3	4	5	2	3	6	**6**	**4**
M	h-link	16/ 89	10	10	3	4	5	2	3	6	**5**	**4**
A	h-bus	16/ 82	10	9	3	4	5	2	3	6	**5**	**3**
Dir:	h-link	16/ 91	10	10	3	4	5	2	3	6	**5**	**4**
Dir:	h-bus	16/ 89	10	9	3	4	5	2	3	6	**5**	**4**
n = 1024 processors												
N	mesh2	143/205	26	25	9	11	12	7	10	28	**16**	**10**
U	mesh3	86/121	16	15	6	7	7	5	6	17	**10**	**6**
M	h-link	63/ 86	11	11	5	5	6	4	5	13	**7**	**5**
A	h-bus	67/ 92	12	12	5	6	6	4	5	14	**8**	**5**
C	mesh2	16/215	23	21	6	8	10	4	4	10	**11**	**7**
O	mesh3	16/131	14	13	4	5	6	3	3	7	**7**	**5**
M	h-link	16/101	11	11	3	4	5	2	3	6	**6**	**4**
A	h-bus	16/107	12	11	4	5	6	3	3	6	**6**	**4**
Dir:	h-link	16/119	13	12	4	5	6	3	3	7	**7**	**4**
Dir:	h-bus	16/121	13	12	4	5	6	3	3	7	**7**	**4**

Table A.3: Studying the variation in the average number of processor cycles per global data access.

Architecture		L_{cap}/ L_{coh}	Applications and Data Set Size Per Node: MP3D DS:34k	Pthor 199k	Locus 77k	Water 13k	Chol 62k	LU 40k	B-H 25k	Ocean 151k	Avg all	Avg 4
100 MHz technology, cache size 4 kbytes, bmax = 16												
n = 16 processors												
N	mesh2	52/ 71	12	13	8	8	15	12	23	30	**15**	**11**
U	mesh3	52/ 70	11	13	8	8	15	12	22	29	**15**	**11**
M	h-link	55/ 74	12	14	8	9	16	13	24	31	**16**	**11**
A	h-bus	44/ 57	10	11	7	7	13	10	19	25	**13**	**9**
C	mesh2	21/ 86	11	11	5	6	9	6	10	14	**9**	**6**
O	mesh3	21/ 85	11	11	5	6	9	6	10	14	**9**	**6**
M	h-link	21/ 98	12	12	5	6	9	6	10	14	**9**	**7**
A	h-bus	21/ 80	10	11	5	5	9	6	10	14	**9**	**6**
Dir:	h-link	21/ 75	10	10	5	5	8	6	10	14	**8**	**6**
Dir:	h-bus	21/ 67	9	9	5	5	8	6	10	13	**8**	**6**
n = 64 processors												
N	mesh2	67/ 91	15	17	10	10	19	16	29	38	**19**	**14**
U	mesh3	60/ 82	13	15	9	9	17	14	26	34	**17**	**13**
M	h-link	67/ 91	15	17	10	10	19	16	29	38	**19**	**14**
A	h-bus	62/ 85	14	16	9	10	18	15	27	35	**18**	**13**
C	mesh2	21/106	13	13	5	6	10	6	10	15	**10**	**7**
O	mesh3	21/ 97	12	12	5	6	9	6	10	14	**9**	**7**
M	h-link	21/114	14	14	5	6	10	6	10	15	**10**	**7**
A	h-bus	21/108	13	13	5	6	10	6	10	15	**10**	**7**
Dir:	h-link	21/112	14	13	5	6	10	6	10	15	**10**	**7**
Dir:	h-bus	21/114	14	14	5	6	10	6	10	15	**10**	**7**
n = 256 processors												
N	mesh2	94/132	21	24	14	14	27	22	40	53	**27**	**19**
U	mesh3	73/ 99	16	18	11	11	21	17	31	41	**21**	**15**
M	h-link	70/ 96	15	18	10	11	20	16	30	40	**20**	**14**
A	h-bus	67/ 92	15	17	10	10	19	16	29	38	**19**	**14**
C	mesh2	21/147	17	17	6	7	11	7	11	16	**11**	**8**
O	mesh3	21/114	14	14	5	6	10	6	10	15	**10**	**7**
M	h-link	21/119	14	14	6	7	10	6	10	15	**10**	**7**
A	h-bus	21/115	14	14	5	6	10	6	10	15	**10**	**7**
Dir:	h-link	21/122	14	14	6	7	10	6	10	15	**10**	**7**
Dir:	h-bus	21/124	15	14	6	7	10	6	10	15	**10**	**7**
n = 1024 processors												
N	mesh2	148/212	32	38	21	22	42	34	63	84	**42**	**30**
U	mesh3	91/128	20	23	13	14	26	21	39	52	**26**	**19**
M	h-link	81/113	18	21	12	12	23	19	35	46	**23**	**17**
A	h-bus	91/128	20	23	13	14	26	21	39	52	**26**	**19**
C	mesh2	21/227	25	24	8	10	14	7	11	18	**15**	**10**
O	mesh3	21/143	17	16	6	7	11	7	11	16	**11**	**8**
M	h-link	21/136	16	16	6	7	11	7	11	15	**11**	**8**
A	h-bus	21/151	17	17	6	7	11	7	11	16	**12**	**8**
Dir:	h-link	21/162	19	18	6	8	12	7	11	16	**12**	**8**
Dir:	h-bus	21/170	19	19	7	8	12	7	11	16	**12**	**8**

Table A.4: Studying the variation in the average number of processor cycles per global data access.

Architecture		L_{cap}/ L_{coh}	MP3D DS:34k	Pthor 199k	Locus 77k	Water 13k	Chol 62k	LU 40k	B-H 25k	Ocean 151k	Avg all	Avg 4
100 MHz technology, cache size 16 kbytes, bmax = 16												
n = 16 processors												
N	mesh2	52/ 71	11	11	5	6	6	6	12	17	**9**	**6**
U	mesh3	52/ 70	11	11	5	6	6	6	12	17	**9**	**6**
M	h-link	55/ 74	12	11	6	6	7	6	12	18	**10**	**6**
A	h-bus	44/ 57	9	9	5	5	5	5	10	15	**8**	**5**
C	mesh2	21/ 86	11	10	4	5	5	3	6	9	**7**	**4**
O	mesh3	21/ 85	11	10	4	5	5	3	6	9	**7**	**4**
M	h-link	21/ 98	12	11	4	5	6	4	6	9	**7**	**5**
A	h-bus	21/ 80	10	10	4	5	5	3	6	9	**6**	**4**
Dir:	h-link	21/ 75	10	9	4	4	5	3	6	9	**6**	**4**
Dir:	h-bus	21/ 67	9	8	4	4	5	3	6	9	**6**	**4**
n = 64 processors												
N	mesh2	67/ 91	14	14	7	8	8	7	15	22	**12**	**7**
U	mesh3	60/ 82	13	12	6	7	7	7	14	20	**11**	**7**
M	h-link	67/ 91	14	14	7	8	8	7	15	22	**12**	**7**
A	h-bus	62/ 85	13	13	6	7	7	7	14	21	**11**	**7**
C	mesh2	21/106	13	12	4	5	6	4	6	10	**8**	**5**
O	mesh3	21/ 97	12	11	4	5	6	4	6	9	**7**	**5**
M	h-link	21/114	14	13	4	6	7	4	6	10	**8**	**5**
A	h-bus	21/108	13	12	4	5	6	4	6	10	**8**	**5**
Dir:	h-link	21/112	13	12	4	6	6	4	6	10	**8**	**5**
Dir:	h-bus	21/114	13	13	4	6	6	4	6	10	**8**	**5**
n = 256 processors												
N	mesh2	94/132	20	19	9	11	11	10	21	31	**16**	**10**
U	mesh3	73/ 99	15	15	7	8	9	8	16	24	**13**	**8**
M	h-link	70/ 96	15	14	7	8	8	7	16	23	**12**	**8**
A	h-bus	67/ 92	14	14	7	8	8	7	15	22	**12**	**7**
C	mesh2	21/147	17	16	5	7	8	4	6	11	**9**	**6**
O	mesh3	21/114	14	13	4	6	7	4	6	10	**8**	**5**
M	h-link	21/119	14	13	5	6	7	4	6	10	**8**	**5**
A	h-bus	21/115	14	13	4	6	7	4	6	10	**8**	**5**
Dir:	h-link	21/122	14	13	5	6	7	4	6	10	**8**	**5**
Dir:	h-bus	21/124	14	13	5	6	7	4	6	10	**8**	**5**
n = 1024 processors												
N	mesh2	148/212	31	30	14	16	17	15	32	48	**25**	**15**
U	mesh3	91/128	19	19	9	10	11	9	20	30	**16**	**10**
M	h-link	81/113	17	17	8	9	9	9	18	27	**14**	**9**
A	h-bus	91/128	19	19	9	10	11	10	20	30	**16**	**10**
C	mesh2	21/227	25	23	7	9	11	5	7	13	**12**	**8**
O	mesh3	21/143	16	15	5	6	8	4	6	11	**9**	**6**
M	h-link	21/136	16	15	5	6	7	4	6	11	**9**	**6**
A	h-bus	21/151	17	16	5	7	8	4	6	11	**9**	**6**
Dir:	h-link	21/162	18	17	5	7	8	4	7	11	**10**	**6**
Dir:	h-bus	21/170	19	18	6	7	9	4	7	12	**10**	**6**

Table A.5: Studying the variation in the average number of processor cycles per global data access.

Architecture		L_{cap}/ L_{coh}	Applications and Data Set Size Per Node: MP3D DS:34k	Pthor 199k	Locus 77k	Water 13k	Chol 62k	LU 40k	B-H 25k	Ocean 151k	Avg all	Avg 4
100 MHz technology, cache size 64 kbytes, bmax = 16												
n = 16 processors												
N	mesh2	52/ 71	10	9	4	5	5	3	4	11	**6**	**4**
U	mesh3	52/ 70	9	9	4	5	5	3	4	11	**6**	**4**
M	h-link	55/ 74	10	10	4	5	5	3	4	11	**7**	**4**
A	h-bus	44/ 57	8	8	3	4	4	3	4	9	**5**	**4**
C	mesh2	21/ 86	10	9	3	4	5	2	3	7	**5**	**4**
O	mesh3	21/ 85	10	9	3	4	5	2	3	7	**5**	**4**
M	h-link	21/ 98	11	10	4	4	5	3	3	7	**6**	**4**
A	h-bus	21/ 80	10	9	3	4	5	2	3	6	**5**	**4**
Dir:	h-link	21/ 75	9	8	3	4	4	2	3	6	**5**	**3**
Dir:	h-bus	21/ 67	8	8	3	4	4	2	3	6	**5**	**3**
n = 64 processors												
N	mesh2	67/ 91	12	12	5	6	6	4	5	14	**8**	**5**
U	mesh3	60/ 82	11	11	4	5	5	4	5	12	**7**	**5**
M	h-link	67/ 91	12	12	5	6	6	4	5	14	**8**	**5**
A	h-bus	62/ 85	11	11	5	5	6	4	5	13	**7**	**5**
C	mesh2	21/106	12	11	4	5	6	3	3	7	**6**	**4**
O	mesh3	21/ 97	11	10	4	4	5	3	3	7	**6**	**4**
M	h-link	21/114	13	12	4	5	6	3	3	7	**7**	**4**
A	h-bus	21/108	12	11	4	5	6	3	3	7	**6**	**4**
Dir:	h-link	21/112	13	12	4	5	6	3	3	7	**7**	**4**
Dir:	h-bus	21/114	13	12	4	5	6	3	3	7	**7**	**4**
n = 256 processors												
N	mesh2	94/132	17	16	6	8	8	5	7	19	**11**	**7**
U	mesh3	73/ 99	13	13	5	6	6	4	6	15	**8**	**5**
M	h-link	70/ 96	13	12	5	6	6	4	5	14	**8**	**5**
A	h-bus	67/ 92	12	12	5	6	6	4	5	14	**8**	**5**
C	mesh2	21/147	16	15	5	6	7	3	3	8	**8**	**5**
O	mesh3	21/114	13	12	4	5	6	3	3	7	**7**	**4**
M	h-link	21/119	13	12	4	5	6	3	3	8	**7**	**4**
A	h-bus	21/115	13	12	4	5	6	3	3	7	**7**	**4**
Dir:	h-link	21/122	14	13	4	5	6	3	3	8	**7**	**5**
Dir:	h-bus	21/124	14	13	4	5	6	3	3	8	**7**	**5**
n = 1024 processors												
N	mesh2	148/212	26	26	10	12	12	8	10	29	**17**	**10**
U	mesh3	91/128	16	16	6	8	8	5	7	18	**11**	**7**
M	h-link	81/113	15	14	6	7	7	5	6	16	**9**	**6**
A	h-bus	91/128	16	16	6	8	8	5	7	18	**11**	**7**
C	mesh2	21/227	24	22	6	8	10	4	4	11	**11**	**7**
O	mesh3	21/143	16	15	4	6	7	3	3	8	**8**	**5**
M	h-link	21/136	15	14	4	6	7	3	3	8	**8**	**5**
A	h-bus	21/151	17	15	5	6	7	3	3	8	**8**	**5**
Dir:	h-link	21/162	18	16	5	6	8	3	4	9	**9**	**6**
Dir:	h-bus	21/170	19	17	5	7	8	3	4	9	**9**	**6**

Table A.6: Studying the variation in the average number of processor cycles per global data access.

The Protocol of the Prototype

THE PROTOCOL OF THE DDM PROTOTYPE

Z:y_A,x_B	Change state to Z, send y above and x below.
Retry	Retry the CMMU.
!	The M bus transaction has the Intend to Modify bit set.
ww!$_B$	Writeword is used as an erase on the M bus.
$sX\rightarrow$Action	If selected after trying to be selected (with priority = X) — Do Action.
$\neg sX\rightarrow$Action	If not selected after trying to be selected (with priority = X) - Do Action.
$\perp$	Release the CPU.
$\emptyset$	The combination is impossible.
¬IM	Transaction without the intent to modify, e.g. ordinary read.
IM	Transaction with intet to modify, e.g. a write.
IM-P	Transaction with intet to modify by the prefetcher.
¬ hd	No subsystem contained data with same address as the transaction.
hd	At least one subsystem contained data with same address as the transaction.

MEMORY BELOW 1

Trans.	States					
	I	E	S	R	W	RW
¬ IM	R:r$_A$Retry			R:Retry	W:Retry	RW:Retry
IM	RW:r$_A$Retry		W:e$_A$Retry	R:Retry	W:Retry	RW:Retry
replace		I:i$_A$,rl!$_B$	I:o$_A$,rl!$_B$	R:Retry	W:Retry	RW:Retry
IM-P	RWP:r$_A$Retry		WP:e$_A$Retry	R:Retry	W:Retry	RW:Retry

MEMORY BELOW 2

Trans	States			
	EA	EW	RWP	WP
¬ IM			R:Retry	WP:Retry
IM	S:d$_A$,rl$_B$	E:	RW:Retry	W:Retry
replace	EA:Retry	EW:Retry	RWP:Retry	WP:Retry
IM-P	EA:Retry	EW:Retry	RWP:Retry	WP:Retry

OA - KILLING CONTROL

Trans	Trans trying to kill					
	r	e	d	x	o	i
r	-	K	-	-	K	-
e	-	K	-	-	-	$\emptyset$
d	-	K	-	-	K	$\emptyset$
x	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
o	-	K	-	-	-	$\emptyset$
i	-	$\emptyset$	-	-	$\emptyset$	$\emptyset$

MEMORY ABOVE 1

Tr.	States					
	I	E	S	R	W	RW
r		$s1\rightarrow$S:d$_A$,rl$_B$ $\neg s1\rightarrow\emptyset$	$s1\rightarrow$S:d$_A$,rl$_B$ $\neg s1\rightarrow$		$s2\rightarrow$ $\neg s2\rightarrow$	
e		$\emptyset$	I:ww!$_B$	$s1\rightarrow$R:r$_A$ $\neg s1\rightarrow$	$s1\rightarrow$RW:r$_A$,ww!$_B$ $\neg s1\rightarrow$RW:ww!$_B$	$s1\rightarrow$RW:r$_A$ $\neg s1\rightarrow$
d		$\emptyset$		S:wl!$_B\perp$		$s1\rightarrow$W:wl!$_B$,e$_A$ $\neg s1\rightarrow$
x		$\emptyset$	$\emptyset$	$s1\rightarrow$R:r$_A$ $\neg s1\rightarrow$	EW:$\perp$	$s1\rightarrow$RW:r$_A$ $\neg s1\rightarrow$
o		$\emptyset$		$s1\rightarrow$S:wl!$_B\perp$ $\neg s1\rightarrow$S:wl!$_B\perp$		$s2\rightarrow$W:wl!$_B$,e$_A$ $\neg s2\rightarrow$
i	$sx\rightarrow$S:wl!$_B$[1] $\neg s1\rightarrow$	$\emptyset$	$\emptyset$	$s2\rightarrow$S:wl!$_B\perp$ $\neg s2\rightarrow$S:wl!$_B\perp$	$\emptyset$	$s3\rightarrow$W:wl!$_B$,e$_A$ $\neg s3\rightarrow$

[1] The home node priority scheme.

MEMORY ABOVE 2

Trans	States			
	EA	EW	RWP	WP
r	$s1\rightarrow$ $\neg s1\rightarrow\emptyset$	$s1\rightarrow$EA: $\neg s1\rightarrow\emptyset$		$s2\rightarrow$ $\neg s2\rightarrow$
e	$\emptyset$	$\emptyset$	$s1\rightarrow$RWP:r$_A$ $\neg s1\rightarrow$	$s1\rightarrow$RWP:r$_A$,ww!$_B$ $\neg s1\rightarrow$RWP:ww!$_B$
d	$\emptyset$	$\emptyset$	$s1\rightarrow$WP:wl!$_B$,e$_A$ $\neg s1\rightarrow$	
x	$\emptyset$	$\emptyset$	$s1\rightarrow$RWP:r$_A$ $\neg s1\rightarrow$	E:$\perp$
o	$\emptyset$	$\emptyset$	$s2\rightarrow$WP:wl!$_B$,e$_A$ $\neg s2\rightarrow$	
i	$\emptyset$	$\emptyset$	$s3\rightarrow$WP:wl!$_B$,e$_A$ $\neg s3\rightarrow$	$\emptyset$

DIRECTORY BELOW - sub NOT selected								DIRECTORY BELOW - sub selected						
Trans		States						States						
		I	E	S	R	W	A	I	E	S	R	W	A	
r	$\neg\ hd$	R:r$_A$							$\emptyset$	$\emptyset$		$\emptyset$	$\emptyset$	
	hd				$\emptyset$						$\emptyset$			
e	$\neg\ hd$		E:x$_B$	W:e$_A$	I:	$\emptyset$	W:e$_A$		E:x$_B$	W:e$_A$	I:	$\emptyset$	W:e$_A$	
	hd		E:x$_B$	W:e$_A$	I:	$\emptyset$	W:e$_A$		E:x$_B$	W:e$_A$	I:	$\emptyset$	W:e$_A$	
d	$\neg\ hd$					$\emptyset$	S:d$_A$						S:d$_A$	
	hd					$\emptyset$	S:d$_A$						S:d$_A$	
x	$\neg\ hd$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	
	hd	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	
o	$\neg\ hd$		E:i$_B$	I:o$_A$	$\emptyset$	$\emptyset$	I:o$_A$				$\emptyset$	$\emptyset$	S:d$_A$	
	hd				$\emptyset$		S:d$_A$				$\emptyset$	$\emptyset$	S:d$_A$	
i	$\neg\ hd$	$\emptyset$	I:i$_A$	$\emptyset$	$\emptyset$	$\emptyset$	I:i$_A$	$\emptyset$		$\emptyset$	$\emptyset$	$\emptyset$	S:d$_A$	
	hd	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	

DIRECTORY ABOVE

Trans	States					
	I	E	S	R	W	A
r		$s1 \rightarrow$ A:r$_B$ $\neg s1 \rightarrow \emptyset$	$s1 \rightarrow$ A:r$_B$ $\neg s1 \rightarrow$		$s2 \rightarrow$ $\neg s2 \rightarrow$	$s2 \rightarrow$ $\neg s2 \rightarrow$
e		$\emptyset$	I:e$_B$	$s1 \rightarrow$ R:r$_A$ $\neg s1 \rightarrow$	I:e$_B$	I:e$_B$
d		$\emptyset$		S:d$_B$		S:
x		$\emptyset$	$\emptyset$	$s1 \rightarrow$ R:r$_A$ $\neg s1 \rightarrow$	E:x$_B$	$\emptyset$
o		$\emptyset$		$s1 \rightarrow$ S:d$_B$ $\neg s1 \rightarrow$ S:d$_B$		S:
i		$\emptyset$	$\emptyset$	$s1 \rightarrow$ S:d$_B$ $\neg s1 \rightarrow$ S:d$_B$	$\emptyset$	$\emptyset$

IBF - dir is master

Transaction		Sub sel ? NO	Sub sel ? YES
r	$\neg\ h\,d$	$\emptyset$	$\emptyset$
	$h\,d$	-	-
e	$\neg\ h\,d$	-	-
	$h\,d$	-	-
d	$\neg\ h\,d$	-	-
	$h\,d$	$\emptyset$	$\emptyset$
x	$\neg\ h\,d$	-	-
	$h\,d$	-	-
o	$\neg\ h\,d$	$\emptyset$	$\emptyset$
	$h\,d$	$\emptyset$	$\emptyset$
i	$\neg\ h\,d$	i	-
	$h\,d$		$\emptyset$

IBF - sub is master

Transaction		Sub sel ? NO	Sub sel ? YES
r	$\neg\ h\,d$	r	$\emptyset$
	$h\,d$	-	-
e	$\neg\ h\,d$	e	e
	$h\,d$	e	e
d	$\neg\ h\,d$	d	d
	$h\,d$	d	d
x	$\neg\ h\,d$	$\emptyset$	$\emptyset$
	$h\,d$	$\emptyset$	$\emptyset$
o	$\neg\ h\,d$	o	o
	$h\,d$	o	o
i	$\neg\ h\,d$	i	i
	$h\,d$		

TOP BELOW

Transaction		Sub sel ? NO	Sub sel ? YES
r	$\neg\ h\,d$	$\emptyset$	
	$h\,d$		
e	$\neg\ h\,d$	x$_B$	x$_B$
	$h\,d$	x$_B$	x$_B$
d	$\neg\ h\,d$		
	$h\,d$		
x	$\neg\ h\,d$		
	$h\,d$		
o	$\neg\ h\,d$	i$_B$	
	$h\,d$		
i	$\neg\ h\,d$	$\emptyset$	
	$h\,d$	$\emptyset$	$\emptyset$

Simple COMA

By Erik Hagersten, Ashley Saulsbury and Anders Landin
Swedish Institute of Computer Science
Box 1263
164 28 KISTA
SWEDEN

July 1993

Abstract

Shared memory architectures often have caches local to the processors to remove some of the potential penalty for slow remote accesses and also to reduce the traffic in the network. The positive effect of caches increases with their size. The largest possible caches exist in architectures called Cache-Only Memory Architectures (COMAs), where all the memory resources are spent implementing large caches, also forming the "shared memory" illusion. However, these large caches also have their price. Due to its lack of physically shared memory, COMA might suffer from a longer remote access latency than alternatives. The large COMA caches might also introduce an extra latency for local accesses, unless the node architecture is designed with care.

The goal of this work is to find the right trade-off between software/hardware solutions in order to find the minimal COMA node architecture, i.e., a simple architecture with hardware support only for the functionality frequently used. Such a system will achieve outstanding performance (and price/performance) by its simplicity which enables a low development cost, a short time to market and the possibility of using more exclusive technology. These properties are comparable to those of early RISC microprocessors of the eighties.

The solution presented here not only achieves the simplicity property, but also enables existing technologies to be used, i.e., commercial microprocessors, programming models, operating systems, and compiler technology and proposed latency-hiding techniques.

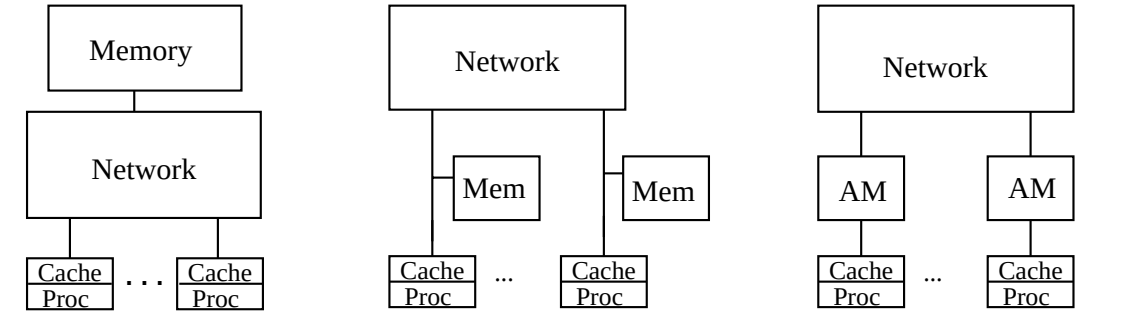


Figure C.1: Comparing COMA to more conventional architectures.

C.1 Introduction

Multiprocessors with cache-coherent shared memory can be built in many ways. Systems based on a single bus suffer from bus saturation and therefore typically have only some tens of processors, each 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]. This 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 (NUMAs), 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. Early 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, the memory 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 unit of coherence, called an *item*, is comparable to a cache line, and is moved around by the protocol. 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. There are cases where multiple attraction memories could have matching items, i.e., the item is replicated. Examples of such architectures are the DDM [HLH92] and the KSR1 [Res92].

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. However, a non-optimized version of the same program would give a similar behavior on a COMA, since the data are attracted to the processor and used 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.

The new requirements of building efficient large caches have led to the design of a proprietary processor cache in the KSR1, as will be described in more detail later. Other proprietary parts of the KSR1 are the processor and the network. This proprietary choice results in a processor three to four times slower than today's commercial offerings, and remote latencies three to four times longer than necessary. In spite of this, the KSR1 has proven a performance comparable to the Dash architecture for the SPLASH [SWG91] applications [JSGH93]. A comparative analytical study of general implementations of NUMA and COMA, covering a large design space, also reports a performance advantage for COMA architectures for the same set of programs [Hag92].

The objectives of this work are to find a *simple* COMA implementation which is compatible with existing computer technology and knowledge, such as cache-coherence protocols, microprocessor implementations, programming paradigms, and operating systems.

In the remainder of this paper, we first discuss some general issues for COMA architectures; the next section reviews some existing COMA node implementation proposals, followed by a description of our simple COMA proposal. The paper is concluded by a complexity and performance study followed by a summary of related work and our conclusions.

C.2 COMA Properties

This paper does not take a position on network topology and/or choice of coherence protocol. However, a short discussion about these two important topics might be appropriate.

Recent years have seen extensive study of the problem of maintaining coherence among read-write data shared by different caches—for example directory-based and snooping-based techniques [Ste90].

Even though both COMAs being built today, KSR1 and DDM, rely on a hierarchical network topology, COMAs can be built upon general networks [HL91, GJS92]. The cache-coherence protocol for a COMA can adopt the techniques of other cache-coherence protocols [LLG+90, CFKA90, TD90, JLGS90] and add functionality for finding an item on a cache-read miss and for handling replacement [HLH92]. The search for the item compensates for the lack of a home for data in a COMA. The problem of finding an item on a read miss has already been addressed in the NUMA protocols for situations where a dirty copy of the requested item resides in a node other than the home node. Most accesses missing in COMA's attraction memory at steady state execution are likely to be coherence misses [HS89], caused by true or false data sharing between one or more processors. For those misses, chances are high that dirty data will reside in a remote node in a NUMA architecture, i.e., about the same amount of overhead can be expected for both architectures. Therefore, the overhead for locating the data on a read miss in a COMA architecture is not expected to be significant.

A COMA protocol also must have a replacement strategy which makes sure that the last copy of an item is not lost when replacement occurs. One solution to this has been suggested by Hagersten et al. [HHW90] where all shared copies are replaced with care. In order to guarantee some space for all items in a COMA, the address space in use cannot be larger than the sum of the attraction memories and the distribution of addresses evenly distributed over the sets in the attraction memories. Another solution has been proposed by Wallach and Dally [Wal90], where each shared item has one tagged owner who replaces with care. Gupta et al. propose a strategy where each item has a defined home which is the synchronization point for the replacement action [GJS92].

One important, and unique, property of COMA is its ability to dynamically adjust its ratio between replication and memory size according to the needs of the current applications [Hag92]. Some applications have no need for massive sharing (replication), but need a large shared (physical) memory to avoid frequent disk accesses. Other applications, e.g., some database applications, benefit by large portions of their data being replicated among all the processors. In a NUMA architecture, the replication is limited by the size of its (second-level) caches, and its shared memory size is also fixed. The two application behaviors described above would need

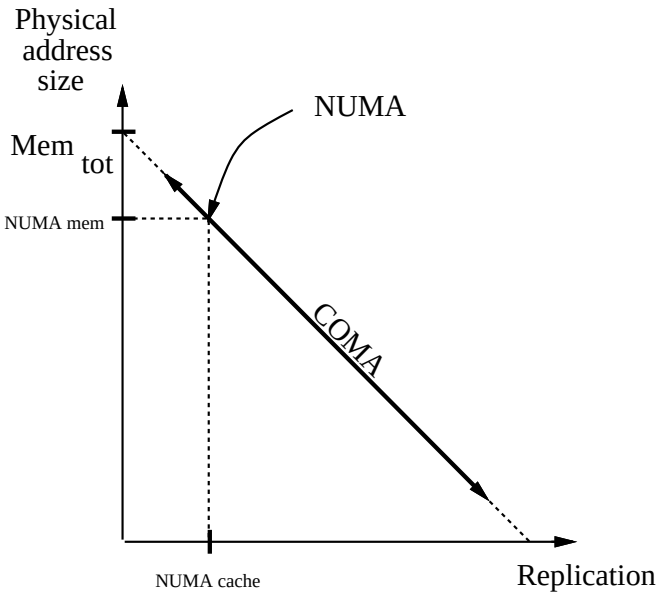


Figure C.2: A NUMA has a statically fixed relationship between the size of the caches (replication) and the physical memory, while a COMA dynamically can change its working point to suit the application.

two different parametrizations of NUMA in order to run well. In a COMA, both behaviors could run well on the same architecture, thanks to its dynamic property.

A COMA will increase its replication until space runs out in the attraction memories. At this point, the amount of replication is determined by the size of the item space presently mapped by the operating system. A large, mapped item space results in a lower amount of replication, and vice versa, as shown in Figure C.2. The operating system of the COMA can decrease the item space by reclaiming more pages, and increase the space again by mapping more pages.

A mapping from virtual addresses to physical addresses is needed to map the accesses to the limited physical shared memory of conventional architectures. Most cache-coherent architectures keep the coherence among physical addresses to overcome a problem known as the "aliasing problem", where the same physical data might be referred to by different virtual addresses. In a COMA the need to map virtual addresses to a limited set of physical addresses does not exist, since the caches might host any address, and only the aliasing problem needs to be overcome. In the KSR1, software conventions are used to get around aliasing problems and virtual addresses are used as the cache-coherent shared addresses. In the DDM, the translation of virtual addresses to physical addresses still exists. This provides a more common software view of the system, and simplifies adaption of existing operating systems, compilers and applications. It also enables control over the usage of the cache-coherent memory size at run time, which is used for the adjustment of replication

and also plays an important part in the replacement mechanism, described earlier. Further, the cost for the translation is limited, since that functionality often comes with the commercial processors.

C.3 Proposed COMA Node Implementations

So far, we have explained why dynamic memories in the shape of attraction memories are to be preferred over statically bound memories of NUMA architectures. However, if the dynamical property has to be payed for by a significantly longer access time for local accesses, the advantage of a COMA architecture over a NUMA is not at all obvious. In this section we will review a few existing proposals for how the associativity of the attraction memories can be achieved. We will describe the implementations based on a baseline architecture similar to the DDM prototype implementation [HLH92] based on the processor MC88100 and its cache circuit MC88200 by Motorola.

C.3.1 Baseline Architecture

The baseline system consists of several processors, each one with its own snooping copy-back physical cache/MMU (CMMU), connected by a common bus, (M bus) to a common attraction memory. The MMUs translate from virtual addresses to "shared physical addresses"[1] The attraction memory implementation is divided into two parts: the *data memory* (DM) and the *protocol handler* (PH), which implements the protocol and interfaces to the rest of the system.

One important functionality used in the designs presented here is the retry signal of the M bus, used in the coherence protocol among the Dcaches. The retry signal is here used by the protocols to force split transactions, i.e., allowing the M bus to be released between a request causing a slow remote access and its reply. The retry signal functionality has been transferred to the new MC88110 processor and its bus. The same solutions have also influenced the PowerPC and MC68060 designs, which is why the techniques described here apply to a large number of commercially available processors. Other processors implementing true split transactions, like the HP-PA, can also adapt to the techniques described here.

The Protocol Handler contains the *below protocol* (BP) and a *state memory* (SM), containing some address tag information and state, as shown in Figure C.3. There are several ways the associativity of the attraction memory can be implemented. We will leave that for later and assume for the moment that enough functionality

[1] In a COMA, they are neither very physical nor are they addresses, but rather called *item identifiers*.

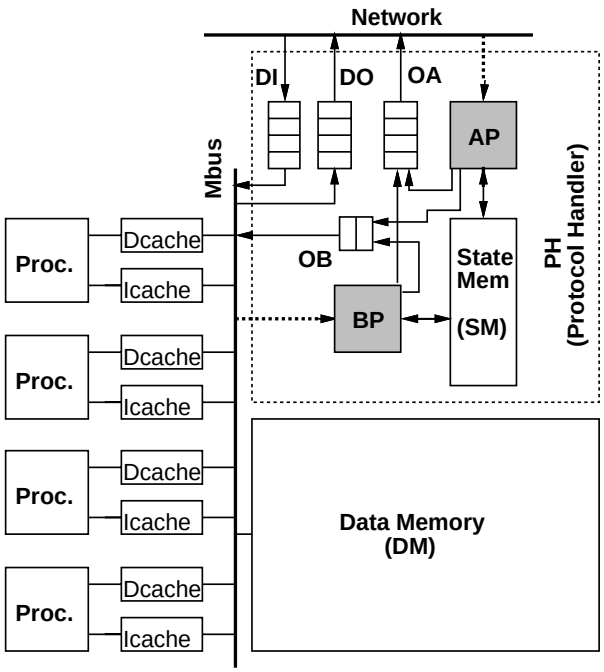


Figure C.3: DDM implementation based on the 88000 family.

exists in the *state memory* and the *below protocol* for determining if a requested item exists in the *data memory*, and if so, where it exists and in what state.

The *below protocol* performs a lookup in the *state memory* for each transaction on the bus and checks for validity. In the case of an invalid access, e.g., a *read* of an Invalid item, the *below protocol* asserts the retry signal. The retry signal makes the current bus master stop and release the bus, while the *below protocol* initiates necessary actions. While the requested item is being retrieved, the requesting Dcache will not be granted the bus. After the retrieved data have been written to the *data memory* by the *protocol handler*, the Dcache can be granted the bus and redo its transaction.

The *protocol handler* also hosts the *above protocol* (AP) and the *output above* FIFO (OA) for transactions bound for the network. The *output above FIFO* contains the transaction code and the item identifier of the transaction, but no data. The *above protocol* can access the *data memory* by putting an M bus transaction in the *output below* FIFO (OB). The *output below FIFO* only contains address and transaction code. Transactions on the M bus from the *output below FIFO* have the data FIFOs *data in* (DI) and *data out* (DO) as an implicit source or destination. Data are retrieved from the node's data memory and put in the *data out FIFO* by a *read line* in the *output below FIFO*. Data are written from *data in FIFO* to the node's memory by putting a *write line* in the *output below FIFO*.

C.3.2 Implementing the DDM Protocol

A *read* request on the M bus is snooped by the below protocol. The *below protocol* checks to see if the requested item is present in the attraction memory. If so, the state stored in the *state memory* is checked; for example, a *read* request to an item which is present and in the Shared state is approved. If the transaction is approved, the *below protocol* does not interfere with the transaction. If the transaction was not approved, for example, a *read* request to state Invalid, the *below protocol*:
1. asserts the retry signal, forcing the Dcache to release the M bus,
2. sets the address tag bits in the *state memory* to the higher order 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 above protocol:
1. puts the data part of the transaction in the *data in FIFO*,
2. puts a *write line* transaction in the *output below FIFO* containing the item identifier, and,
3. changes the item's state to Shared.

A write transaction on the M bus to an item in an inappropriate state is intercepted in a similar way by the memory below protocol, and necessary actions are taken before the Dcache is released to arbitrate for the bus again.

C.3.3 A Direct-Mapped AM

In order for the *protocol handler* to find out if an item is stored in the attraction memory, each item is associated with an address tag in the *state memory*, which is compared to the most significant bits (MSB) of the requested address. 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. The location is determined by the least significant bits (LSB) of the address. We can assume that a read transaction will succeed and start the *read line* before the approval from the *below protocol* is received. A processor cache that has already read a few words may be forced to restart before reading the last word of a cache line. The *below protocol* 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. Only one access to the *state memory* is needed while several accesses to the *data memory* might be needed to transfer the whole cache line. Thus, *state memory* may use the same memory technology as used in *data memory*,[2] and the delay of

[2] In today's, technology probably DRAMs

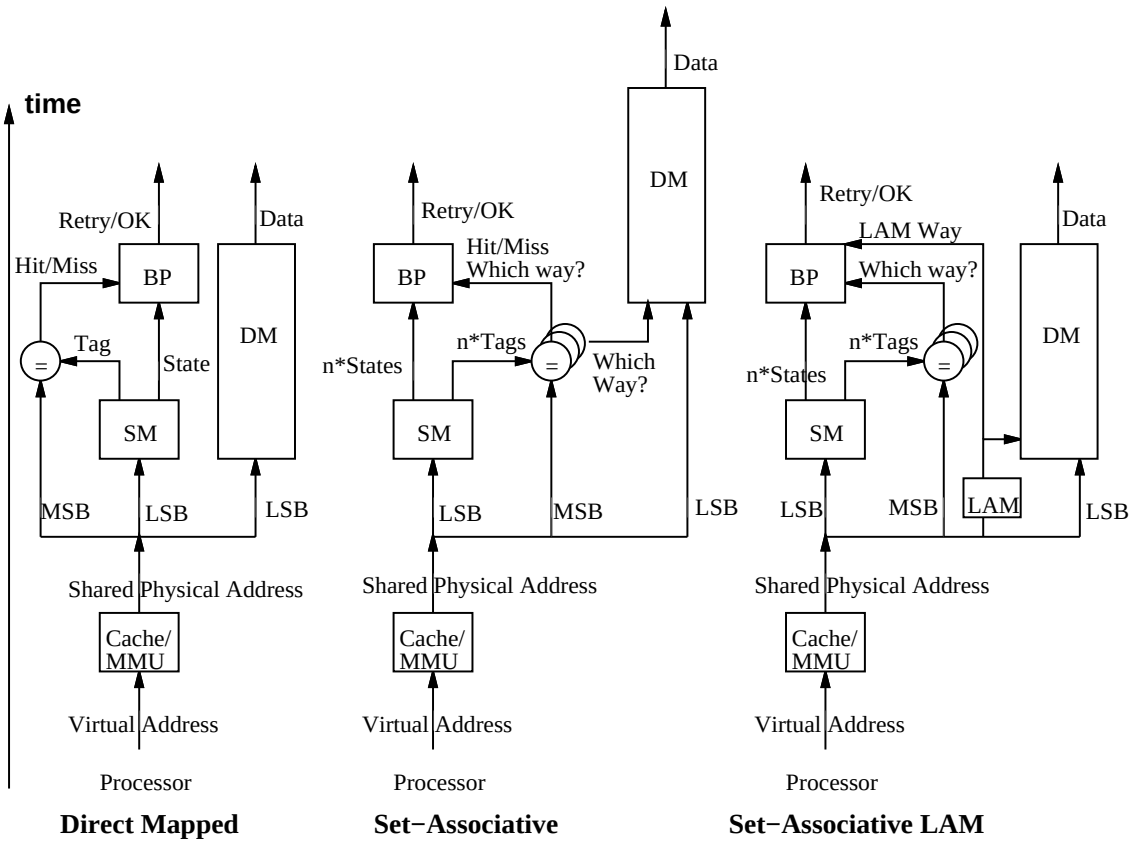


Figure C.4: Data dependency graphs for different ways of implementing associativity (BP=below protocol, SM=state memory, DM=data memory, and LAM=last accessed memory).

accessing the *state memory* will still be completely hidden, adding no extra latency caused by the functionality of the AM.

It seems that the latency for accessing the *state memory* cannot be hidden on a write, since overwriting parts of another item would be fatal. There is, however, full inclusion between a processor's (data) cache and its AM; in other words, there can be no copy of an item in the processor's cache unless there is also a copy of the item in the AM. This, together with the fact that a *write* to the attraction memory is never performed by the Dcache unless it already contains a copy of the item [Mot89], can hide the *state memory* access—even from write accesses.

As can be seen in Figure C.4, the implementation of a direct-mapped attraction memory is straightforward. The access time to data stored in *data memory* is equal to the *data memory* access time.

C.3.4 Set-Associative Attraction Memory

A direct-mapped cache is advantageous over a multi-way associative implementation for shortening access time to the AM, but it also increases conflict misses [HS89]. More associativity is expected to increase the hit rate in the AM. One can imagine situations for which a directly mapped attraction memory could be fatal for performance.

Another drawback of a directly mapped attraction memory is its limitation for replication of popular items. In order for one item to get replicated in all attraction memories, no other items for the same set of the attraction memory can be present in the machine; i.e., the item space cannot be larger than the size of one attraction memory. For two-way attraction memories, the item space can be half the sum of the AMs, and for four-way attraction memories, the item space can be three quarters of the sum of the AMs.

If the AM organization is multi-way set-associative, finding the location for the requested item is harder, since several possible locations exist for each item.[3] The address tag of all possible locations must be compared to the requested item's before the location in *data memory* can be determined. Small caches, implemented on a single chip, often access all possible data locations in parallel with the tag comparison, and select the right data at a late stage of the access. This results in only a minor overhead compared to a direct-mapped implementation. Still, direct-mapped cache implementations have been justified for large cache sizes [HS89].

Accessing all possible data locations in parallel is complicated and impractical for the implementation of a large attraction memory with several ways, which is why the whole address tag lookup and comparison must be performed before the data access is started. This means putting the *state memory* lookup and the comparison on the critical path, as shown in Figure C.4. As a result of its size, the *state memory* might be implemented with slow memory devices, resulting in a substantial overhead.

C.3.5 Last-Access Memory Read-Access Optimization

It is possible to implement a set-associative memory with almost no extra access latency for read accesses. The algorithm used is the MRU algorithm—guessing that the way last used in the set will be the one asked for next time as well. There is no fixed location in the set for the MRU entry; instead, a fast last-accessed memory (LAM) is added [Hag92]. It contains one pointer of $\log_2(ways)$ bits per set, pointing to the way of the last accessed entry in the set. The contents of the LAM are used as part of the address to start the read access immediately.

[3] Equal to the number of ways.

If the *data memory* is built of DRAMs, access to the LAM can be hidden. Half the DRAM address[4] is not needed during the first access cycle. By putting the LAM pointer address in that part of the address, no extra delay is introduced, assuming that the small LAM has a short access time. The comparisons of the address tags in the *state memory* are started in parallel with the read access. The comparison tells which set—if any—contains the item and if the item is in the correct state. If the LAM guess turns out to be the right one, and its associated state acceptable, no further action is taken. If another entry contains the right address tag, a retry signal is asserted forcing the access to be restarted while the LAM is updated. The same transaction will then be restarted, but with the correct LAM pointer. A read access according to this scheme is described in Figure C.4. The LAM optimization only works for read accesses. A write access is performed similarly to the set-associative implementation described earlier, since a write cannot be performed until the right way has been determined.

The positive effects of returning the MRU data first in large multi-way caches have been studied by Chang et al. [CCS87]. The LAM technique divides the AM into two parts, one with access time comparable to the *data memory* and one with a longer access time, similar to introducing yet another layer in the cache hierarchy. As such, the LAM strategy can potentially cut the access time for many applications, but is not expected to be successful for all applications. For a program that sequentially accesses a data set larger than the LAM part of the AM, the LAM guess consistently will turn out to be the wrong one.

C.3.6 KSR1

It is hard to get full information about the KSR1 design and the facts presented here are partly based on assumptions and guesses. KSR1 has introduced proprietary solutions to many parts of its design [WBH+93, Res92, Res91]. This is also true for its caching system. Its first-level cache[5] appears to be large, 256 kbytes, but is organized in a somewhat unorthodox way. The cache is divided into associativity units of 2 kbytes. As a whole, the cache contains 128 such units, organized in a two-way set-associative manner, i.e., 2 x 64 x 2 kbytes. Each unit contains (among other things) one address tag, one "AM-way" pointer, and has space for 32 coherence units of 64 bytes of data and a few bits of state each. The replacement strategy is random.

The second-level cache (AM) is 32 Mbytes with associativity units of 16 kbytes, called a page, organized in 16 ways, i.e. 16 x 128 x 16 kbytes. Each page contains one address tag, some random information about the page's usage, and 128 coherence units of 128 bytes data plus some state bits, as can be studied in Figure C.5.

[4]The column-access (CAS) part.
[5]Called "subcache" by KSR.

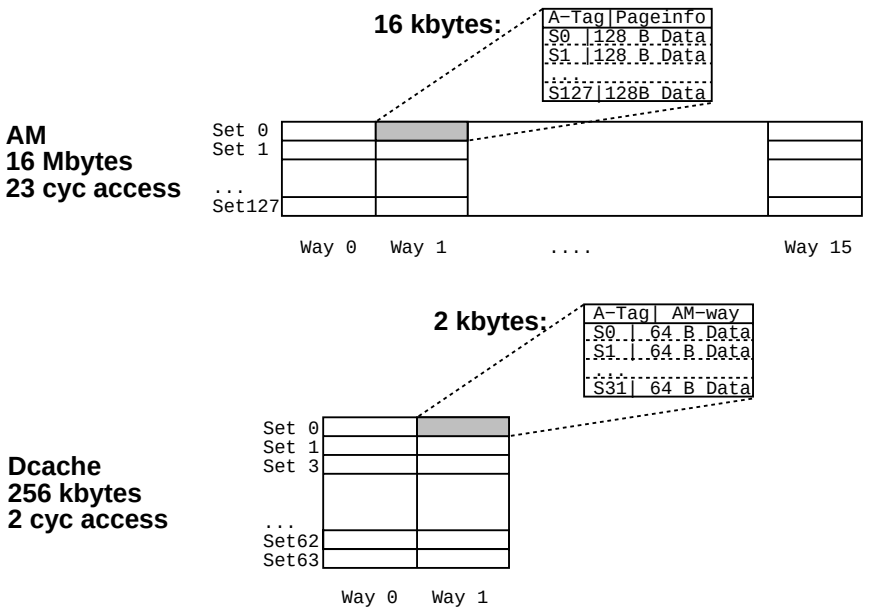


Figure C.5: The organization of the caches in the KSR1 architecture.

On an access to a new associativity unit in the first-level cache, new space in the cache must be allocated (randomly). Secondly, the 16 possible locations in the AM are checked for a matching address tag. These 16 comparisons are (probably) performed partly sequentially in a scheme similar to the set-associative implementation just described. So, the overhead for bringing the first 64 bytes of an associative unit to the first-level cache from the AM is significant. The identity of the way in the AM for which the address tag matched is stored in the first-level cache (AM-way), so that the next access to the same 2kbytes can be performed without address comparison. The data dependency of a KSR1 AM access when the AM-way information is available in the Dcache, can be found in Figure C.6.

Allocating large associative units in the caches can avoid the extra associative overhead for many accesses, as shown. It also cuts down on the memory required to store the address tags. The drawback of this scheme is a potentially low utilization of the data space in the caches. Even if only a single word is requested by the processor, 2 kbytes of the processor cache and 16 kbytes of the AM must be allocated, i.e., only 128 sparsely used words may reside in the data cache at the same time.

There is no ordinary MMU functionality found in the KSR1. Virtual addresses are used as the global addresses in the system. This creates problems with aliasing and prevents efficient implementation of copy-on-write. The lack of page fault exceptions forces the search of a requested page in the whole machine before it can be determined whether or not a disk access is necessary. This should be compared to the early page-fault exception generated by an MMU. Further more, and probably most importantly, it is not compatible with existing OS, compilers, and some ap-

plications, so a potentially large design effort is needed to rewrite portions of the software.

C.3.7 Distributed Virtual Shared Memory

The title Distributed Virtual Shared Memory (DVSM) covers a range of multiprocessor shared-memory implementations where the coherence and migration of data between processors is maintained purely by software. Traditional DVSM systems, (exemplified by [LH89, CBZ91, SSW92]), make use of processor memory management units to initiate coherence protocol actions which are implemented in software. Just as hardware distributed shared memory systems, coherence traffic between nodes is implemented as messages on a network (e.g., packets on an Ethernet).

As we shall see, most DVSM system implementations have COMA properties—the main memory of each processor is treated as a cache, with data items (pages) being allocated and invalidated, and data being moved and replicated from node to node without the notion of a fixed home which CC-NUMAs have.

C.3.7.1 Virtual Memory Operation

When a processor makes a "first-time" reference to a virtual memory address, the Memory Management Unit (MMU) has the responsibility to convert that virtual address into a physical memory address. In this case, the MMU will not have a translation in its Translation Lookaside Buffer (TLB), nor will an entry (physical-page pointer) be found in the current page table.[6] After failing to translate the virtual address, exception is taken, and the flow of the process is interrupted—a *pagefault*.

In the event of a page fault, it is the responsibility of the operating system to allocate an unused physical page for the virtual page referenced. Furthermore, the operating system must fill the allocated physical page with data corresponding to the virtual memory page accessed—either zero-filling, or retrieving a block from local disk. Further accesses by the application to the same virtual page "hit" in the TLB and are therefore completed without penalty.

C.3.7.2 Distributed Virtual Shared Memory

We can begin to see how a shared memory system can be implemented in software.

[6] By either a software or hardware table lookup—depending whether the CPU's TLB is software or hardware loaded.

Consider a "first time" access to data in a Distributed Virtual Shared Memory region. The MMU cannot perform a virtual access to physical address translation, so a pagefault exception is generated. The operating system (or DVSM system) then allocates a new physical page. Data for the page is retrieved from **another processor node** which already has a copy of the data corresponding to the virtual page being accessed. This is done by sending a request message to the other node, and then receiving the reply, which includes a copy of the data—much the same as one cache requesting a copy of data from another in a COMA. [7]

We have seen how data can be replicated when read in DVSM, but not how it is kept coherent when written. Any coherency protocol of choice is implementable for DVSM, indeed Munin [CBZ91] offered quite a selection.

Aside from detecting the validity of an item (page), to maintain coherency we need to detect write accesses to an item (page) — for example when the item is shared across several nodes. To to do this the page *write-protect* functionality of the processor MMU is used. By write-protecting a virtual page, read accesses proceed as normal, but write accesses cause the MMU to generate a *write-protect pagefault*[8]— even though there is a valid virtual to physical mapping for the page.

C.3.7.3 Pros and Cons of DVSM

Software (DVSM) COMA implementations have a number of advantages over hardware COMAs.

The MMU functionality of the processor enables a virtual memory page to be mapped to **any** physical memory page on the local node. This enables a DVSM COMA to be built with a fully associative attraction memory, while hardware COMAs are restricted to either a direct mapping or limited associativity. Furthermore, as such hardware COMAs use processors with some MMU functionality, their attraction memory access cost still includes the cost of the MMU address translation. Therefore, software COMAs utilize the full associativity properties of the MMU for free, and may even provide faster access to data than hardware COMAs which have attraction memories with limited associativity.

Being implemented in software means that DVSM systems can have more complex replacement and prefetching algorithms than would be reasonable to implement in hardware. The simplicity of a DVSM memory access can be studied in Figure C.6.

[7] As with hardware COMA implementation, finding the node which has a copy of the data of interest is a problem orthogonal to that of maintaining coherency. A number of schemes to find a node with the correct data are presented by Li and Hudak in [LH89].

[8] This is normally used in Unix systems (for example) to efficiently implement `fork(2v)` semantics, i.e. copy-on-write.

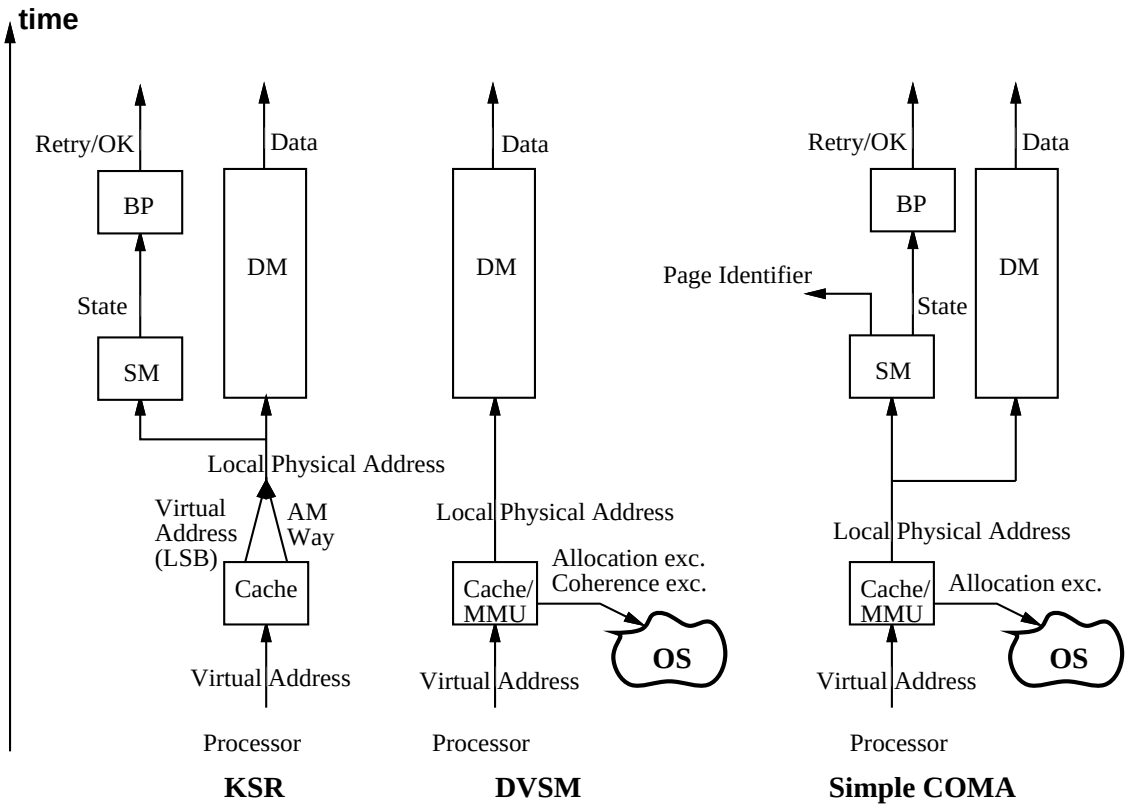


Figure C.6: The data access route for different attraction memory implementations.

DVSM systems do, however, generally suffer from three problems. The first is the large item size—typically page sizes for today's microprocessors are 4 kbytes and growing. This large item size results in a potentially large amount of false sharing.

The second problem is the long latency associated with a cache miss. This perhaps surprisingly is **not** due to the cost of taking a page fault—processors with software loaded TLBs [MIP92] illustrate that page faults can be dispatched and dealt with in only a few tens of cycles. The costs are associated with the creation and dispatch of messages on more traditional networks such as Ethernet.

Finally, and most importantly, the processor must deal *also* with the coherency traffic from other nodes, as well as its own. Hardware COMAs such as the DDM and KSR1 can deal with coherency traffic from other nodes on the network without disturbing the local processor.

C.4 The Simple COMA

We firmly believe that COMAs will be seen to be the right architecture for general multiprocessors. However, looking at the only two COMAs being built in the world today (the DDM and the KSR1), one might be forgiven for believing that COMAs are more complex than more traditional NUMA distributed shared memory systems.

We have seen in earlier sections how a COMA can be built on an arbitrary communication network, and how the coherence protocol of a COMA is very similar to CC-NUMA coherency protocols. So where is the complexity in a COMA ?

Taking the DDM and the KSR1 as COMA examples, it would appear to be the attraction memory implementation that is overly complex. This would not be an accurate conclusion, as both the DDM and KSR1 are early implementations of COMAs.

We believe the right implementation solution for a COMA results from combining the best features of the complex all-hardware COMA approach, and the simple-but-poor-performance software DVSM approach.

C.4.1 A Better COMA Implementation

Here we describe a solution to the problem of attraction-memory implementation, which we believe is optimal compared to the previously described solutions.

Our proposed implementation has a fully associative attraction memory *with* a short access time. It is far simpler than the KSR1 implementation, and *still* reduces some of the disadvantages found in the KSR1 solution.

COMAs differ from DVSMs mainly by their smaller coherence units (items), and by the coherency protocol being implemented in hardware, rather than by software. We propose to retain some of the DVSM software functionality, while additional hardware support, similar to the DDM implementation, is added to decrease the size of the coherence units and also make the coherency protocol implementation more efficient.

Our proposal is a combination of fully associative mapping using the MMU, as seen in DVSM, in combination with a coherence protocol implementation similar to the protocol handler of the baseline architecture.

C.4.2 The Proposed COMA Node

Ignoring issues of network and protocol as orthogonal, we propose a COMA node designed as follows.

Just as with DVSM implementations, the allocation and replacement of items within the attraction memory is handled by software—performed necessarily at page-size granularity. This enables our COMA system to have a fully associative attraction memory as mentioned earlier. Unlike DVSM, coherence actions will not cause MMU exceptions, the addition of state memory and a simple protocol handler (PH) enables coherence checks to be performed on a per-item granularity - typically a first- or second-level cache line size.

Therefore the *state memory* holds protocol state bits per *attraction memory* item. Unlike the DDM implementation, however, there is no address tag stored with each item. We do not need to validate access to an item with an address tag, since we performed the item identification validation effectively with the MMU.

An additional state memory (or part of the *state memory*) holds a *page identifier* for each page (of items) in the *attraction memory*. This page identifier (PI) is assigned by the software which allocates the virtual-to-physical attraction memory page mapping. This page identifier is used by the protocol handler to identify a shared page when communicating with another node. In order to uniquely identify a page 20 bits of page identifier is enough for a machine with up to 4 Gbytes held in 4 kbytes pages.

Note that the number of bits for the page identifier in no way affects the virtual addressing capabilities of the machine's processors, just the total number of physical memory pages in the machine. Figure C.7 compares the *state memory* in the proposed solution to that in the DDM.

Figure C.6 illustrates the memory access paths of the proposed simple COMA alongside those of a basic DVSM system, and the KSR1 architecture. Access to the attraction memory may be started simultaneously with the state memory lookup. This is because the MMU performs the associative cache lookup for the item position. When the physical memory address appears, for both the *state memory* and *data memory* lookup, there is a **direct** mapping for an item within the selected page. This functionality enables the state memory to be implemented in the same speed devices as the attraction memory - for example conventional DRAM. The MMU has already tested the validity of the mapping (item identification) in the attraction memory, and so we can start either a read or a **write** knowing that the item either exists or will exist at that location. A miss caused by an unfavorable item state can simply abort the operation as part of the coherence actions taken.

C.4.3 Implementing Remote Associativity

So far we have described the mechanism by which local processor memory accesses are directed to the correct data item, and access is validated by the *state mem-*

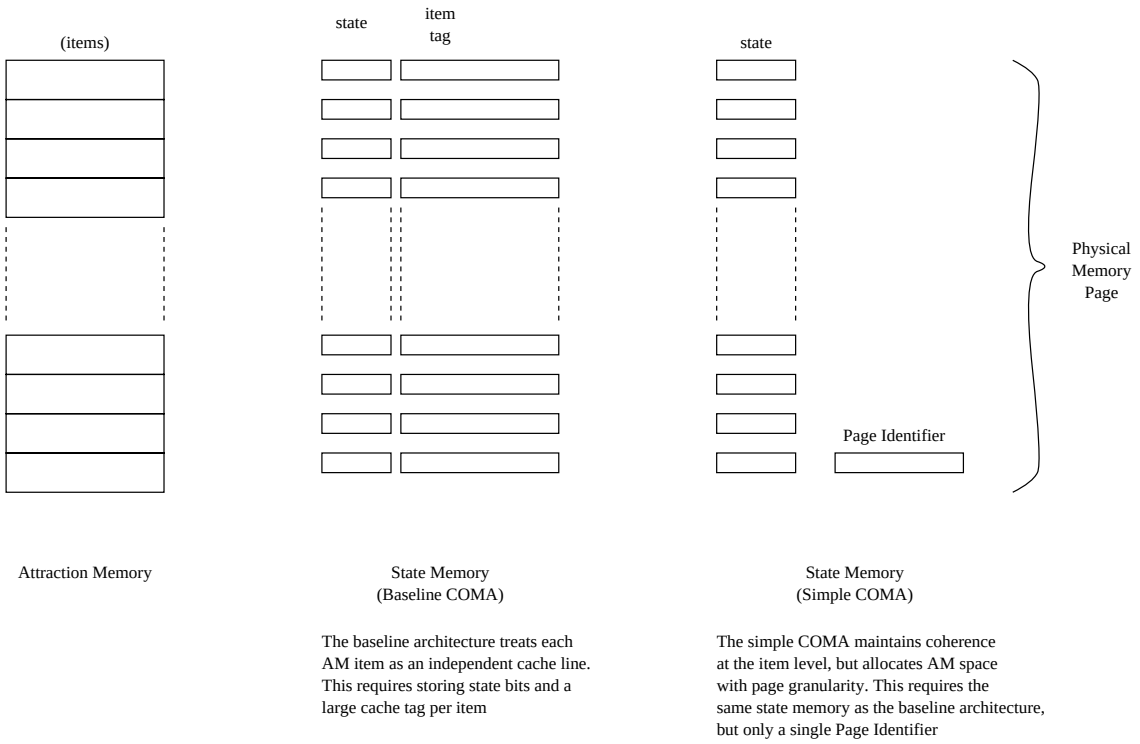


Figure C.7: Comparison of the DDM and Simple-COMA state memories

ory. We have not discussed how the coherency protocol handler on one node can communicate with its counterpart on another processor node.

When a local memory access fails, for example because of a read to an invalid data item in the *attraction memory*, the page identifier is produced by the smaller *state memory*, or possibly even from a table in main memory. This PI is used in a message to another node to retrieve a copy or exclusive ownership of the missing item.

When the protocol message arrives at the destination node, that node must, from the page identifier, be able to lookup **its local** physical page mapping in order to find its copy of the item and state. There are several methods of performing this reverse PI to physical page translation.

C.4.3.1 Page Identifier CAM

The PI CAM is a Content Addressable Memory - when an incoming PI is applied, the PI CAM can return the physical page number corresponding to that page on the local node. When a physical page is allocated on a node, the entry in that node's PI CAM allocated for the physical page chosen is filled with the PI given to the page.

For a processor node with say 64 Mbytes of RAM organized in 4 kbytes pages, a PI CAM with 16384 entries is required, something which is practicable with today's technology.

C.4.3.2 Page Identifier MMU

A simpler solution to the fully associative memory as described above, a kind-of MMU functionality, can be implemented. A small associative cache (like a TLB) holds the most recent PI conversions, and a PI table is walked when an entry is not found in the PI-TLB.

It is not clear how effective this will be in practice compared to the PI CAM, since the PI MMU must field coherency traffic from potentially all other nodes in the system. Such traffic will not have as much locality as the memory accesses from the single local processor (the latter fact exploited by the processor's own TLB). The PI-TLB may have to be several times larger than one might allocate for the CPU.

A similar functionality to this is already implemented for the "DMA Engine" (Elan chips) in the Meiko CS-2 network communication interface. This is used under direct processor control to transfer messages from one processor's memory to another's.

C.4.3.3 Physical Pointers

This scheme effectively moves the hardware complexity of a PI-MMU or PI-CAM into software. In addition it reduces the latency of a remote access to data, at some extra cost to page replacement and the protocol implementation.

The idea of this scheme is that the page identifier should be the physical page number of the corresponding page on another node (and possibly also some form of node identifier). For example if node A shares a page copy with the owner of the page—node O, then node A will have the physical page number of the page on node O as its page identifier.

When node A wishes to perform some protocol action (such as item invalidation), it sends a message to node O. As the physical page number is given with the message access to the correct state memory slot can be started immediately. There is no need for the delay associated with the PI-CAM or PI-MMU.

To complete a protocol implementation, there must be some way of identifying the physical pages allocated on the sharing nodes (node A in the example above). This is achieved by storing their respective physical page numbers with the item copy set information.

As with other hardware COMAs or NUMAs, the copy set (and reverse page identifiers) may be held in any form: as linked lists such as SCI [JLGS90] or SDD [TD90], or even in hierarchical directories as in the DDM.

Note, we introduced the notion of an "owner" node merely as a focal point for identifying the copy list. Since pointers are allocated and reclaimed by software, ownership of a page may be easily made to move.

While providing faster access when a coherence message arrives at a node, this scheme requires extremely careful pointer allocation and reclaimation. A physical page may not be reallocated until all other nodes which hold pointers (node and physical page number) to it have this mapping invalidated. This may require expensive interprocessor interrupts and synchronizations. All said and done, the one or two cycle extra cost of the PI-CAM or PI-MMU may be of no significance in the face of a typical 100 cycle network latency.

Experiments will indicate which of the above three schemes is likely to be the most effective. An attractive solution might, once more, be to not choose either the simple (MMU) solution or the efficient (physical pointer) solution, but rather a combination of all three solutions and thus combining simplicity and efficiency.

C.4.4 Potential Drawbacks

Using the MMU to perform the attraction memory item lookup has one potential drawback. It is possible for an application to behave so as to access only one item on each memory page. This commits the other unused items on the same page to particular virtual memory addresses; the physical memory cannot be used for other virtual items.[9].

In the worst case, an example machine with 64 Mbytes of memory could only share 4096 items (if using a 4 kbytes page size). With an item size of, say, 128 bits (16 bytes), this enables a maximum shared data space of 64 kbytes before thrashing starts taking place—such thrashing involves full page faults.

We do not believe that this is a problem for a couple of reasons. Firstly the KSR 1 allocates 16 thousand blocks with similar restrictions in its second-level cache, comparatively we have one quarter of the granularity with a 4 kbytes page size. Secondly, most programs tend to exhibit locality with adjacent items in order to get good performance with conventional caches.

[9] This is not quite true. With care, two virtual pages can be mapped onto the same physical page, provided it can be guaranteed that the accessed items in each of the respective virtual pages do not overlap in the physical page. This requires some knowledge of the application by the OS.

C.5 Performance and Complexity

When comparing different implementation proposals it is important to deal with both the performance and the implementation complexity. Here we will qualitatively discuss the performance and complexity of the direct mapped, the set associative, the KSR1 and the simple approaches to building a COMA.

C.5.1 Performance

The performance of a COMA implementation is characterized by the hit rate in the attraction memory and first-level caches, and by the latency for the different types of accesses. This study focuses on the node implementation techniques and does not cover in detail the differences in latency for remote accesses.

Typical applications (for example those from the SPLASH suite [SWG91]) generally have good hit rate figures for attraction memory accesses [HGL+93]. To achieve good performance in a COMA machine it is essential to minimize the latency for these hits. In the case of an attraction memory miss, the latency differences due to these node implementation considerations are not so significant since they are part of a much larger latency associated with the remote access. Other implementations such as network structure and COMA directory policies are, of course, also highly important but lie outside the scope of this study.

The Table in Figure C.8 summarizes the main characteristic differences for the implementation alternatives considered in this study.

Note that direct mapped has a lower hit rate for the same size of attraction memory since it might suffer from an increased number of conflict misses. It also has limitations on the degree of replication that can be utilized. We also assume that there is full inclusion between the primary cache and the attraction memory.

The KSR1 can be expected to have a lower hit rate in its Pcache since it has very large allocation blocks and will suffer if the accesses are sparse and spread in the address space. On the other hand it is fair to assume that the caches can be made larger using KSR1's technique since the associative part is comparably small.

The fast access times for the KSR1 applies for blocks that have already been allocated in the primary cache. For the first access after a block has been replaced from the Pcache, the longer access time applies.

The Simple and the KSR1 alternatives might suffer from reduced AM hit rate since they both have large allocation units. If accesses are sparse the effective size of the AM can be significantly smaller than it nominally appears to be.

Implementation	*Access time*		*Comment*
	Pcache hit	AM hit	
Direct mapped	fast (1 cycle)	fast (DM access)	Lower AM hit rate. Limited replication.
Set assoc.	fast (1 cycle)	slow (SM + DM access)	
KSR1	fast (1 cycle)	fast (DM access) or very slow (SM search + DM access)	Proprietary Pcache with lower hit rate.
Simple	fast (1 cycle)	fast (DM access) or very slow (TLB fill + DM access)	Fully associative AM.

Figure C.8: Main performance characteristics for the different implementation strategies.

C.5.2 Complexity

All the hardware-based implementation proposals differ from the DVSM approach in that hardware complexity is added to improve the performance of the system. The central problem is of course to achieve the highest performance with the least hardware. While this is normally hard to do, it is desirable to find solutions that with a limited hardware complexity give a high performance.

To make sense of the complexity of these strategies, we have studied the hardware needed to implement an each scheme. We assume a machine of 500 nodes each with an attraction memory of 64 Mbytes. We assume a word size of 64 bits. The size of a page is 4 kbytes and the coherence unit is 4 words (32 bytes).

Note that we do neither cover the hardware needed for the interconnection network nor the directory information needed to find remote copies in the machine. These are properties that are orthogonal to the node implementation which is the focus of this paper.

C.5.2.1 Direct Mapped

The direct mapped approach is probably the most straightforward method to implement a COMA. Each node needs a *state memory* that contains state and tag information for all 4 word items in the node. 64 Mbytes of data makes 2 Mitems.

The total memory space in the machine is $64\ Mbytes * 500 = 32\ Gbytes$. To address this the item identifiers need to be 35 bits. The lower 26 bits are needed to address within the attraction memory. This leaves 9 bits for tags associated with each item entry. If we reserve 4 bits for state information the net result is 13 bits per item ($32 * 8\ bits = 256\ bits$). This leaves us with an overhead of $13/256 = 5\ percent$.

C.5.2.2 Set Associative

If we assume an associativity of 16 ways for the set associative proposal we get the following overhead calculations:

The 64 Mbytes attraction memory contains $2\ Mitems/16 = 128\ ksets$. These are addressed with $17 + 5 = 22\ bits$, leaving 13 bits for tag. The memory overhead evaluates to: $(13 + 4)/256 = 6.6\ percent$.

In addition to the memory overhead we also need hardware for the associative comparison. In this case sixteen 13-bit comparators are needed for the attraction memory. The state memory also requires to be able to support all the $16 * 13 = 208$ bits simultaneously. The alternative is to make the comparison in serial. This requires less hardware but further increases the access latency.

C.5.2.3 KSR1

Unlike the other proposals, the KSR1 approach requires special hardware at the primary cache level. This rules out the use of off-the-shelf processors with on-chip caches. Although the actual overhead is very small (0.2 percent) the inconvenience of a proprietary design might be substantial.

The overhead of the attraction memory state and tag is reduced in the KSR1 approach since the AM is divided into 16 kbytes pages. Tag information is only stored with each page while state still must be supplied with every item. Similarly to the set-associative case, the tag associated with a page is 13 bits. This is however negligible compared to the 16 Kbytes data of that page. The state overhead is $4/256 = 1.5\ percent$.

C.5.2.4 Simple

The Simple approach uses the standard MMU for associativity. State information still has to be supplied with each item as in the KSR1 case. The overhead is also 1.5 percent.

C.5.2.5 Complexity Summary

The complexity properties for the proposals are summarized in the following table:

Implementation	*Overhead*			*Standard components usable ?*
	AM Tag	AM State	Other	
Direct mapped	3.5%	1.5%		YES
Set assoc.	5.1%	1.5%	Wide memory, comparators	YES
KSR1	0%	1.5%		NO
Simple	0%	1.5%		YES

Figure C.9: Complexity summary.

As can be seen, both the KSR1 and the simple proposals have very small overhead, while the simple proposal can be implemented with standard hardware and also gives a fully associative attraction memory.

C.6 Related Work

C.6.1 Wind tunnel

In an attempt to create an efficient parallel simulator on the CM-5 for simulation of shared memory architectures, a research group at the University of Wisconsin came up with a solution similar to the one described here [RHL+93]. That solution combines the DVSM coherence mechanism with support for coherence units smaller than a page. Both allocation exceptions and coherence exceptions are handled by software, similarly to the DVSM, but validation checks of access right can be made with a finer granularity than a page.

A bogus ECC code is simply set for the memory of all "invalidated" cache lines of a page. Only accesses to these cache lines will generate exceptions. A problem arises if cache-lines with different access priority co-exist on a page, sine the ECC code cannot differentiate between read and write accesses. Instead, the whole page will be write-protected, causing some valid writes to cache lines to create exceptions, a potential drawback of this scheme. The advantage is that it makes use of an existing architecture and its ECC implementation.

C.6.2 The Sun S3.MP Project

The S3 MP project [NMP+93] aims to build a distributed shared memory architecture by adding a relatively small amount of electronics to an exisiting workstation's memory bus (typically a SPARC-10).

The S3 operates by partitioning the main memory of the local processor into a local main memory partition, to be used as usual, and a cache partition for remote memory accesses. This cache size is programmable—a distinct advantange over conventional NUMAs which have their cache size fixed by hardware; however, it does not allow a dynamic trade between replication and problem size as COMAs do, since the cache size is determined when the machine is booted.

The S3 MP includes hardware to convert local memory addresses into a "global 64 bit address." Similarly, hardware converts such global addresses from the network into local memory addresses.

The design philosophy has much in common with the simple COMA we propose: use exisiting workstation technology, simple coherence, and communication hardware as an extension to the main memory operating as the cache. However the result of the S3 MP is still a cache-coherent NUMA; there is still the fixed cache size; and fixed home.

C.7 Conclusion

A COMA node has been believed to be slower and more complex to implement than alternatives. In this paper we propose a simple and efficient COMA implementation based on existing commercially available components. Alternative architectures (NUMA) can be expected to replace some of the functionality achieved here by a large (second-level) cache and hardware support for page migration. By taking the extra implementation cost of such into account, the simple COMA solution might actually come out ahead in a simplicity comparison. COMA also offers unique dynamical properties that further increase its attraction.

In spite of its new structure and behaviour, the implementation presented conforms to existing assumptions about shared memory architectures made by operating systems, compilers, and applications, and actually adapts to their behaviour.

C.8 Acknowledgements

SICS is sponsored by Asea Brown Boveri AB, Ericsson AB, IBM Svenska AB, Televerket (Swedish Telecom), Försvarets Materielverk FMV (Defense Material Adminis-

tration), and the Swedish National Board for Industrial and Technical Development (Nutek).

References for Appendix C

[CBZ91] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Thirteenth Symposium on Operating System Priciples*, October 1991.

[CCS87] J.H. Chang, H. Chao, and K. So. Cache Design of A Sub-Micron CMOS System/370. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 208–213, 1987.

[CFKA90] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-Based Cache Coherence in Large-Scale Multiprocessors. *IEEE Computer*, 23(6):49–58, June 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.

[GJS92] A. Gupta, T. Joe, and P. Stenström. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. TR #CSL-TR-92-524 Stanford University, 1992.

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

[HGL+93] E. Hagersten, M. Grindal, A. Landin, A. Saulsbury, B. Werner, and S. Haridi. Simulating the Data Diffusion Machine. In *Proceedings of Parallel Architecture and Languages Europe*. Springer-Verlag, 1993.

[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.

[HL91] E. Hagersten and A. Landin. An Initial Attempt to a General Network COMA. DDM-memo, Swedish Institute of Computer Science, August 1991.

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

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

[JLGS90] D. James, A.T. Laundrie, S. Gjessing, and G.S. Sohi. Scalable Coherence Interface. *IEEE Computer*, 23(6):74–77, June 1990.

[JSGH93] T. Joe, J. P. Singh, A. Gupta, and J Hennessy. An Empirical Comparison of the Kendall Square Research KSR1 and the Stanford DASH Multiprocessor. Presented at the Third Workshop on Scalable Shared Memory Multiprocessors (in connection with ISCA), May 1993.

[LH89] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[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.

[MIP92] MIPS. R4000 Users Reference Manual, 1992.

[Mot89] Motorola. *MC88200–Cache/Memory Management Unit, User's Manual.* Prentice Hall, New Jersey, 1989.

[NMP+93] A. Nowatzyk, M. Monger, M. Parkin, E. Kelly, M. Browne, G. Aybay, and D. Lee. S3.mp: A multiprocessor in a matchbox. Technical Report parcftp.xerox.com:/pub/dlee/PASA_proc.ps, Sun Microsystems Computer Corporation, 1993.

[Res91] Kendall Square Research. U.S. patent 5,005,999 Multiprocessor Digital Data Processing System. October 1991.

[Res92] Kendall Square Research. Technical Summary. 1992.

[RHL+93] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of ACM SIGMETRICS Conference*, May 1993.

[SSW92] T. Stiemerling, A. Saulsbury, and T. Wilkinson. A DVSM server for Meshix. In *Symposium on Experiences with Distributed and Multiprocessor Systems III*, March 1992.

[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.

[TD90] M. Thapar and B. Delagi. Stanford Distributed-Directory Protocol. *IEEE Computer*, 23(6):78–80, June 1990.

[Wal90] D. Wallach. A Scalable Hierarchical Cache Coherence Protocol. SB Thesis. MIT AI lab, May 1990.

[WBH+93] D. Windheiser, E. L. Boyd, E. Hao, S. C. Abraham, and E. S. Davison. Analysis of Latency Hiding Techniqueus in a Sparse Solver. In *Proceedings of IPPS*, 1993.

Swedish Institute of Computer Science

SICS Dissertation Series

01. Bogumił Hausman, *Pruning and Speculative Work in OR-parallel PROLOG*
02. Mats Carlsson, *Design and Implementation of an OR-parallel Prolog Engine*
03. Nabiel A. Elshiewy, *Robust Coordinated Reactive Computing in SANDRA*
04. Dan Sahlin, *An Automatic Partial Evaluator for Full Prolog*
05. Hans A. Hansson, *Time and Probability in Formal Design of Distributed Systems*
06. Peter Sjödin, *From LOTOS Specifications to Distributed Implementations*
07. Roland Karlsson, *A High Performance OR-parallel Prolog System*
08. Erik Hagersten, *Toward Scalable Cache Only Memory Architectures*
09. Lars-Henrik Eriksson, *Finitary Partial Inductive Definitions and General Logic*

Printed in Malmö, Sweden by Graphics Systems using a camera-ready copy typeset in Computer Modern by the author, using a LaserJet III Si, the LaTeX document preparation system, the emacs editor, and a SPARCstation 1 workstation.