Prefetching Techniques for Client/Server, Object-Oriented Database Systems

Nils Knafla

Doctor of Philosophy
University of Edinburgh
1999

To Uwe, my dad,
who died on April 17, 1997.

Abstract

The performance of many object-oriented database applications suffers from the page fetch latency which is determined by the expense of disk access. In this work we suggest several prefetching techniques to avoid, or at least to reduce, page fetch latency. In practice no prediction technique is perfect and no prefetching technique can entirely eliminate delay due to page fetch latency. Therefore we are interested in the trade-off between the level of accuracy required for obtaining good results in terms of elapsed time reduction and the processing overhead needed to achieve this level of accuracy. If prefetching accuracy is high then the total elapsed time of an application can be reduced significantly otherwise if the prefetching accuracy is low, many incorrect pages are prefetched and the extra load on the client, network, server and disks decreases the whole system performance.

Access pattern of object-oriented databases are often complex and usually hard to predict accurately. The main thrust of our work therefore concentrates on analysing the structure of object relationships to obtain knowledge about page reference patterns. We designed a technique, called OSP, which prefetches pages according to a time constraint established by the duration of a page fetch. In addition, every page has an associated weight that decides about the execution of a prefetch. We implemented OSP in the EXODUS storage manager by adding multithreading to the database client. The performance of OSP is evaluated on different machines in interaction with buffer management, distributed databases and other system parameters.

For another prefetch algorithm, called PMC, we used a *Discrete-Time Markov Chain* to model object relationships. We assigned transition probabilities to object relationships and applied the *hitting times* method to compute page probabilities and the mean time to access a page. The page probability is used for the prefetch decision and for the order of the disk queue. If the probability of a page is higher than a threshold defined by cost/benefit parameters then the page is a candidate for prefetching. We developed a cost model for the benefit of a prefetch and the extra cost of an incorrect prefetch. The effectiveness of this technique was verified in a simulation in view of different degrees of clustering and buffer replacement strategies.

The granularity of a prefetch is also studied by comparing the performance of a page server and object server system that perform prefetching. The page server requests only a single page from the server and the object server always requests a group of objects. We compare both systems in a simulation in which we distinguish the case where all pages are resident at the server's buffer pool and where pages have to be read from disk first when the page is not in buffer pool. In addition, we suggest some optimisation techniques for the object server.

Acknowledgements

I would like to thank Peter Thanisch very much for his supervision and guidance. I am particularly grateful for his timely reading of all the papers and this thesis.

I would also like to thank my second supervisor David Matthews for many useful discussions at the beginning of my research; Jane Hillston, Graham Clark and Isabel Rojas-Mujica for reviewing the work on statistical prefetching; my examiners Quintin Cutts and Murray Cole for many helpful improvements and Marcio Fernandes and Thomas Zurek.

Most of all, very warm thanks to my parents who gave me financial support and encouragement for all the years. Without their support this work would not exist.

Last but not least, I thank Stephanie Main, my fiancée, for her proof-reading and loving support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The inspiration for this thesis comes from the observation that the ratio of disk access time to semiconductor memory[1] access time is increasing year-on-year. A consequence of this technology trend is that for many client/server object-based systems, the performance bottleneck will be the delay between the time at which a client application requests a page of data and the time at which the page is placed in that application's memory.

The focus of the work in this thesis is an exploration of techniques that can hide access latency by employing intelligent prefetching. The objective appraisal of prefetching techniques crucially depends on an understanding of the aforementioned technology trends. In view of this, in Section 1.1 we review hardware trends, focussing on the increasing gap in access time between successive levels in the memory hierarchy. The area of applications and typical access patterns of applications are explained in Section 1.2. A motivation for our work is given is Section 1.3. The main contribution of this work is introduced in Section 1.4. An overview of the chapters is given in Section 1.5 and in Section 1.6 we conclude this chapter.

1.1 Hardware Trends

For many applications that use object-oriented database management systems (**OODBMS**) or persistent object stores[2], the architectural component that dominates performance is the hard disk. The disk is the slowest part of the system

[1] In the rest of the thesis we refer to semiconductor memory just as memory.

[2] In this thesis we refer to applications of OODBMSs which also includes applications of persistent object stores.

and year-on-year performance improvements in disk technology are only about 5-8%. An example of a fast disk is the Seagate Cheetah 18 with an average access time of 8.19 ms. Although improvements in access time are modest, improvements in throughput rates are much higher. This is due to increasing areal density of disks (see Figure 1.1) and the movement from busses (SCSI) to fast serial lines, such as Fibre Channel Arbitrated Loop (FC-AL) and Serial Storage Architecture (SSA). The increasing areal density contributes to the rise of disk capacity by about 60% per year and improves transfer rates by about 40% per year. Areal density could reach a throughput value of 40 MB/sec in the year 2000 [Keeton et al., 1998]. The throughput is increased if bits are packed closer together, the head can read more quickly for a given rotational speed, due to more bits passing under the head.

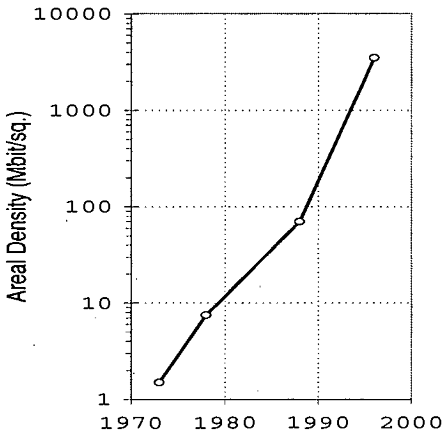


Figure 1.1: Improvements of areal density in hard disks.

The ultimate limit for hard disk capacity is a hot topic among disk engineers. As the bits, which are magnetised areas on the disk, get smaller and smaller, they will eventually reach a point where the energy required to retain magnetisation is equal to the thermal energy of the environment. In other words, if you have small enough magnets, the magnetic fields in effect will not be stable at room temperature.

The performance/price development of semiconductor memory and magnetic disk is important for prefetching. In Figure 1.2 we show the development over the last two decades. There is a two-order of magnitude gap in access time between memory and disk. Memory access is faster and the year-on-year rate of improvement has also been higher. The prices of memory are falling, thereby making database caches cheaper and buffer replacement less of a prob-

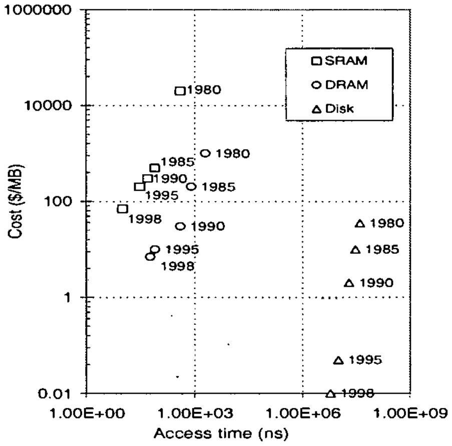


Figure 1.2: Price/performance development of semiconductor memory and disk.

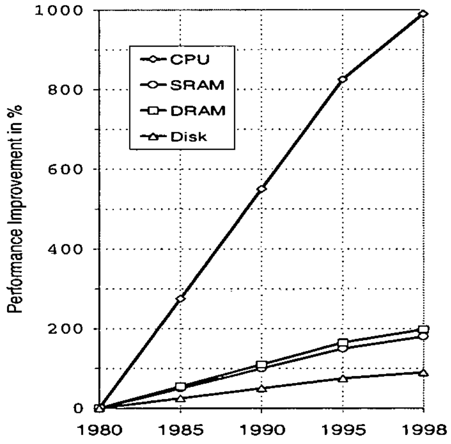


Figure 1.3: Performance improvements of CPU, memory and disk in percent.

lem. Prices of disks are also falling dramatically which is good for cheap secondary storage.

CPU performance improves at an even higher rate than memory (Figure 1.3). CPU is doubling its performance every 18 months whereas disk access times only improves at about 5-8% per year. Memory access time improves at about 10% per year. The future trend is that client workstations will become powerful multi-processor machines with high speed CPUs. Many of these CPUs will tend to be idle most of the time.

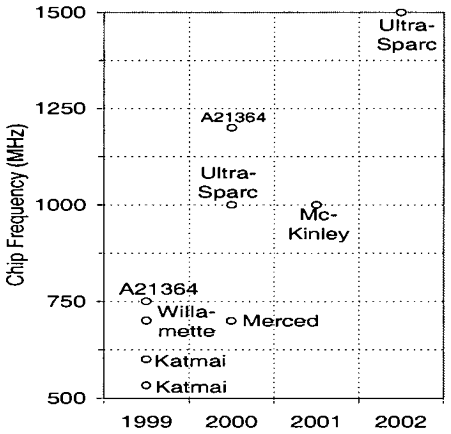


Figure 1.4: Commercial CPU roadmap.

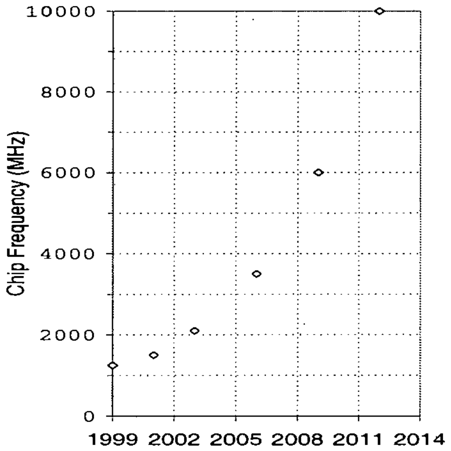


Figure 1.5: Prediction of CPU-speed according to [Semiconductor Industry Association, 1997].

Figure 1.4 shows the shipment years of future processors from Sun, Intel and Digital. Intel's Katmai and Willamette are Pentium II processors. Merced will be Intel's first IA-64 processor and McKinley will probably be a 1 GHz chip based on 0.13 micron technology and using copper interconnects. Rivals like Compaq/Digital (Alpha) and Sun (UltraSparc) have even faster processors nowadays leading to 1.5 GHz in 2001. A further prediction [Semiconductor Industry Association, 1997] for CPU development is shown in Figure 1.5. This figure shows an exponential rise in cycle speed of future processors. On the other side, disk technology is not able to make large improvements in the next years. These trends cause a widening gap in the memory hierarchy.

The speed of the network also has an impact on the performance of a client/server database system. Network transfer rates have improved dramatically over the last years but the encoding and decoding of messages remains the main bottleneck of the transfer. Network latency is lower than disk latency and therefore overcoming it is not the primary study of this work.

Prefetching techniques could hide the latency by requesting data from all levels in the hierarchy in advance. Our conclusions from this analysis of technology trends are as follows:

1. CPU capacity is unlikely to be a performance bottleneck in typical object database applications;
2. Cheaper disks mean that the trend for object bases to get bigger, as database designers get more ambitious, will continue;
3. Although cheaper memory makes it economic to have larger main memory buffers, for many enterprises it will continue to be the case that the object base will be disk-resident. Enterprises are tending to store larger amounts of data these days and there is no sign of this trend abating.
4. The high improvements in CPU processing power will reduce the I/O overlapping time for prefetching. This will mean that the primary benefit we get will be that an increased number of requests to the disk will allow the disk scheduler to do a better job of ordering requests; rather than that we are able to submit requests early enough for their cost to be hidden in processing time.

Consequently, in many client/server applications, disk technology will emerge as the performance bottleneck.

1.2 Application Areas

Considering the area of an application can give some general information about how data are stored and how data are connected. For example design applications have complex relationships; on the other side business applications have simpler relationships and are more value-based. The area of an application may also give some information about the number of users and the amount of data sharing. Some popular types of applications of OODBMSs are:

1. **Design applications (CAD/CAM/CIM/CASE).** Engineering design databases are useful in computer-aided design/manufacturing/software engineering systems. In such systems, complex objects can be recursively partitioned into smaller objects and a prefetching technique for design applications should consider that:

   - The design may be very large. For example, in an integrated circuit, there may be millions of transistors; in a Boeing 747, there are millions of individual parts.
   - There may be hundreds of engineers, managers, technicians and other workers involved. They must work in parallel on multiple versions. A design is often checked out by a designer and checked in again after hours or days. This induces a high workload for the server at check-out and check-in times.

2. **Business applications.** These applications have simpler object structures than design applications and are likely to be more value-based. The requirement from the OODBMS is to provide a very high throughput for very large numbers of users and 24x7 availability. Data sharing is problematic for business applications because data pages might be at the client for exclusive use and other applications that required that page have to wait.

3. **Web applications.** Web applications have unstructured data in the form of HTML templates and more complicated methods to manage variables, page logic and database queries. The hyper-links of HTML templates are stored as pointers in the database. The objects may be stored on fixed-size pages or as single objects if they are large. The adoption of the extensible markup language (XML) will help by adding more structure to Web page templates and other documents.

4. **Multimedia applications.** In a modern office information or other multimedia systems, data include not only text and numbers but also image, graphics and digital audio and video. Such multimedia data are typically stored as sequences of bytes with variable lengths and segments of data are linked together for easy reference. The size of the data is usually very large and for prefetching it is important to load the segments at the right time to avoid any hiccups for the user.

In this thesis we do not concentrate on any specific area of application. All our techniques for page server systems assume a fixed unit of transfer. Therefore our prefetching algorithms would not be suitable for multimedia applications which store sequences of bytes with variable lengths. The other three areas of applications would benefit from our proposed techniques. In general, prefetching will be useful to distributed applications with complex, object-oriented access patterns that:

- are data intensive, with high read/write ratios.
- use varying navigational access patterns that would not all benefit from any particular data clustering.
- create and delete medium or big granularity objects at a slow enough rate to permit tracking of changes. In our implementation we work with average object sizes of 80 bytes and 100 objects per page.
- preserve some degree of object identity.
- have application behaviour that is statistically predictable. This means that the application has a repeatable access pattern but not necessarily that every database access is identical to the previous one. Some applications with random access patterns may not be suitable for obtaining statistical information whereas other applications, like design databases, provide more repeatable access patterns. Statistically predictable object access pattern have been explored before in databases in the area of clustering [Tsangaris and Naughton, 1991; Tsangaris and Naughton, 1992] and prefetching [Palmer and Zdonik, 1991].
- provide a reasonable amount of client processing or client waiting time that can be overlapped with prefetching.

Another characteristic for identifying an OODBMS application is the use of a distributed environment. Most traditional (including relational and even some object) database systems, even though they grew up during the 1970s or 80s on minicomputers, copy the architecture of the mainframe world: all data is stored and all the processing occurs on the central server, to which the users simply send requests and get back answers. This is the same model as the mainframe, with all the computing power accessed from dumb terminals.

This first generation client/server architecture is appropriate for some applications, such as a simple airline reservation system: *a clerk at a dumb terminal requests a no-smoking aisle seat and receives 17D*. However, other applications wish to store information on many different workstations, on servers, and on mainframes, to have processing occur on all those locations, and to have users access and use objects from all those locations.

Using second generation or distributed client/server architecture, an OO-DBMS can support such distribution transparently, even over heterogeneous hardware, operating system and networks. In today's computing world the difference between the computing power of the desktop machine and the back room server is likely to be only 3-5 times, so there is much more computing power spread around the desktop, collectively, than in the back room. The OO-DBMS allows taking advantage of all the resource and incrementally adding new resources by simply wheeling in the latest greatest price/performance leader and moving some objects onto it, without changing any application.

1.3 Motivation

In Section 1.1 we described the potential bottlenecks for client/server database systems. Prefetching is one technique used to reduce this bottleneck. It is an optimisation technique to initiate the task of loading data from slow, cheap secondary storage into fast, expensive primary storage before the application requires the data.

The aim of a successful prefetching technique is to predict the future application access with sufficient accuracy to reduce the frequency of expensive demand fetches. In practice no prediction technique is perfect and no prefetching technique can remove the total demand fetch time. Therefore we are interested in the trade-off between the level of accuracy required for obtaining good results in terms of elapsed time reduction, and the processing overhead needed to achieve this level of accuracy. If prefetching accuracy is high then the total elapsed time of an application can be reduced significantly otherwise if the prefetching accuracy is low, many incorrect pages are prefetched and the extra load on the client, network, server and disks decreases the whole system performance.

We are also interested in the structure of different object relationship patterns and the consequences for predicting application access. If object access is repeatable and there are relationships with high probability then we also perform computation to identify which pages have high access probability. Another task of this thesis is to explore the trade-off between page accuracy and prefetch distance. Under complex object relationships a long prefetch distance would reduce the page fetch time completely but could result in an incorrect prefetch due to low accuracy. A prefetch from a short distance could reduce the page fetch time by only a fraction but this at least results in a correct prefetch. This trade-off is illustrated in Figure 1.6. In this figure, there is an arc from one node to another if there is a corresponding pointer in the object. The arc is labelled with the probability that this pointer will be de-referenced. Suppose a page fetch costs 3 time units and processing one object 1 unit. The aim of a prefetching technique could be total page fetch time reduction. For example, the start of prefetching page 3 at OID[3] 1 would contribute to that aim but the prefetch could turn out to be incorrect if we navigate to page 2. On other hand, if our aim is high prefetch accuracy then we would start to prefetch page 3 at OID 3, 5 or 6. The amount of savings would be less but prefetch accuracy is 100%.

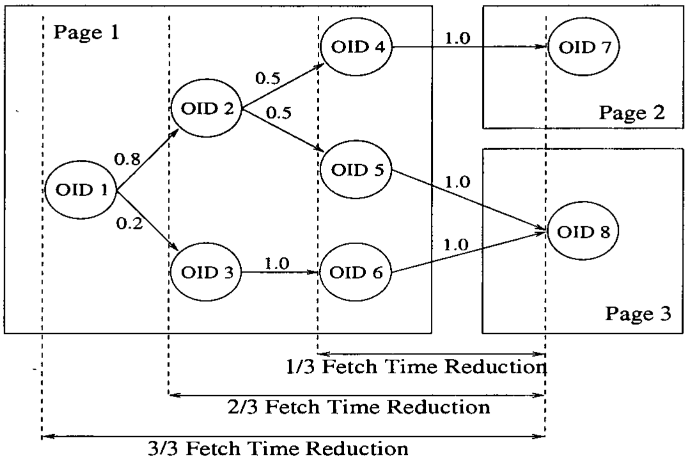


Figure 1.6: Trade-off between page fetch time reduction and prefetch accuracy.

[3] OID means Objects Identifier.

Active prefetching did not obtain much attention from commercial OODBMSs vendors. Bearing in mind the technology trends from the previous section, OODBMS vendors may have to re-think their system architectures. OODBMS vendors could employ the prefetching techniques described in this thesis and the computation-intensive algorithms would be even more valuable in the foreseeable future because of the rapid development in CPU technology. OODBMS architects could argue against prefetching with a statement such as : *"the I/O bottleneck can be reduced by larger cache sizes"*. This is true for repeated page accesses but not for the first access to the page. Another argument could be: *"average disk access time is improving year-on-year almost at the same rate as memory and there are even higher improvements in the throughput of disks"*. This is also true but there is a two-order of magnitude gap in access time between disk and memory and CPU improvements are about 8 times higher than disk improvements year-on-year.

1.4 Thesis Contribution

We started our work by implementing a prefetching environment into the EXODUS storage manager (ESM). The motivation for the ESM-implementation was firstly to get an understanding of the cost-expensive parts of a page fetch operation. Secondly, an evaluation in a real environment would give an idea about all the side-effects caused by prefetching. Thirdly, we are interested in the effectiveness of threads on multiprocessor machines. In our system model we created threads to predict and prefetch future page accesses and perform the application processing. The prefetch threads monitor the application process and make prefetch decisions on the current context. We incorporated threads only into the ESM client and left the ESM server unchanged. Every prefetch thread has an associated socket for communication with the server. Our proposed prefetching architecture differs from the MERLIN implementation [Gerlhof and Kemper, 1994a], the sole other OODBMS architecture with prefetching, because it is based on multiple prefetch threads on multiprocessor machines.

For the evaluation of the prefetching architecture we implemented a simple algorithm, called OSP, in the ESM client [Knafla, 1997a; Knafla, 1997b; Knafla, 1997d; Knafla, 1998a]. The general idea of this technique is to prefetch pages well in advance according to the context of the client navigation. The prefetch depth is determined by the time of a page fetch divided by the time of processing one object. The idea behind this design decision is not to load the pages too early, since otherwise prefetched pages could be replaced again, and also not to load the pages too late which would reduce the potential of prefetching. With regard to the prefetch depth, the prefetch thread observes the navigation of the application thread through the object graph and prefetches all objects with non-resident pages ahead. If one context has multiple pages to prefetch then a page q is assigned a weight according to the number of objects, resident on page p that have a pointer to page q, in the depth n. The pages are then prefetched according to descending weights. It also considers branches of object navigation, i.e. the prefetch is delayed until it has passed a branch of object pointers. Similar to the work of [Patterson et al., 1995] in the area of operating systems we also use a fixed prefetch distance but this distance is computed by other components and in our work we also have to take the complex relationships of object-oriented databases into account. Another difference is the consideration of weights and branch delays in our prediction algorithm.

In another approach, called PMC, we used a more computation-intensive technique to predict future page access [Knafla, 1997c; Knafla, 1998b]. The OSP approach has a relatively low prediction cost but it does not consider the probability of object pointers to be taken. The PMC technique considers the probability of object traversals and is likely to have a higher prefetch accuracy under complex object relationships. The drawback of this method is the high consumption of CPU processing time to compute page probabilities. Therefore it is more suitable for processors with high processing power and the first method is more applicable for processors with lower processing power.

The basic idea of the PMC technique is to compute the page access probability considering the structure of the relationships between persistent objects. Objects have pointers to other objects with associated transition probabilities. The object relationships are modelled using a Discrete-Time Markov Chain and a method called *hitting times* is used to compute the page access probability. From the current position of the client navigation we compute the access probability of all adjacent pages. The level of adjacent pages includes direct adjacent pages but also higher levels of indirect adjacent pages. The level is determined by the prefetch object distance which is in turn influenced by the relationship of microprocessor technology and disk advances. If the probability of a page is higher than a threshold defined by cost/benefit parameters then the page is a candidate for prefetching. There have been many studies to compute the page probability by Markov Chains [Palmer and Zdonik, 1991; Curewitz et al., 1993;

Kraiss and Weikum, 1997] but the novelty of this approach lies in the prediction mechanism of pages based on object transition probabilities. We verified the effectiveness of this technique in a simulation by considering it under different degrees of clustering and buffer replacement policies.

Another decisive parameter for the success of prefetching is the granularity. OODBMSs can be classified as being either page or object server systems. We want to explore the circumstances under which one system can outperform the other system. For both server types the important problem to solve is how to avoid the I/O bottleneck. Here we have to distinguish two cases at the server side: (1) disk pages are resident at the server and (2) disk pages are not resident at the server. In theory, if all pages are resident the object prefetching technique has the advantage that it can put together all the relevant objects for a prefetch request independent of how these objects are dispersed on pages. If pages are not likely to be resident, a page prefetching technique has the advantage that it requests only a few high priority disk pages. All the previous research on prefetching was conducted in one of these two types but there is no research, to the best of our knowledge, which assesses a prefetching technique by comparing its performance in a page server and object server implementation [Knafla, 1998c].

In thesis we present many new prediction and prefetching techniques for object-oriented databases. The main novelty of this work is the synthesis of analysing object relationships and statistical information for predicting page accesses. The research chapters will provide a clear analysis of the following prefetching topics:

- Detailed performance analysis of the page fetch latency in a client/server environment.
- Experimental and theoretical study of the saving of a prefetch request in elapsed times under different prefetch distances.
- Impact of object relationships with associated transition probabilities on the computation of page probabilities.
- Computation of the mean access time of a page depending on the current position in the object navigation.
- Interaction of prefetching and clustering under varied cluster factors.

- Impact of buffer pool sizes and some replacement strategies on prefetching.
- Side effects of prefetching in a client/server database system, e.g. increasing synchronisation overhead at the client or increased workload at the server.
- Experimental performance study of combined prefetching and multi-threading.
- Effect of the prefetch granularity in a object server system.

We have covered a lot aspects of prefetching in all the chapters but there is much for further investigation. We would like to motivate the reader of this thesis to think about the potential extensions of this work. Here is a list of current limitations, some of which are explained in more detail in Section 7.4:

- Most of the experiments are limited to one client, connected to the server, requesting demand and prefetch pages. Multiple prefetching clients would provide a more realistic environment for a client/server system.
- The replacement strategy for prefetched pages is LRU. Using more sophisticated buffer management policies, such as the computed page probability for prefetching and buffer replacement would improve system's performance even further.
- This work is limited to databases with relatively unchanging structures and most of the prediction information is computed off-line. An efficient on-line computation would make our prefetching more useful for applications with frequent updates.

1.5 Thesis Contents

The remaining chapters of this thesis are organised as follows:

Chapter 2 (Basics of Object-Oriented Databases): the next chapter will present fundamental components of an object-oriented database management system.

Chapter 3 (Basics of Prefetching Techniques): this chapter will present a review of research literature in the area of prefetching which is relevant to our thesis. It will concentrate on topics that are associated with database prefetching in a client/server architecture but will also include prefetching algorithms in other areas of computer science.

Chapter 4 (Object Structure-Based Prefetching): the C++ implementation of a prefetching environment into the EXODUS storage manager is described in this section. We designed a simple prefetching technique and incorporated it into the database client. The results of the performance evaluation will give an insight into the merits of prefetching.

Chapter 5 (Statistical Prefetching): using the results from Chapter 4, this chapter will explain a sophisticated prefetching algorithm. At first we give a formal description of the algorithm and then present the results from a simulation evaluation.

Chapter 6 (Page versus Object Prefetching): in a simulation test we compared the performance of a prefetching page server with a prefetching object server. In addition, we will discuss some performance optimisations for an object server.

Chapter 7 (Conclusions): in the final chapter, we will summarise the result of the research programme, discuss the original contributions to knowledge which has been made, and identify further research activities which have been suggested through the current study.

The first two chapters are the basic background chapters and the following three chapters are dedicated to our research. The reader's attention is also drawn to the Index chapter at the end of the thesis. This should provide a useful and convenient resource locating concise definitions of various technical terms used within this dissertation.

The following typographic conventions are adopted throughout the thesis:

Italic is used for figure and table captions and to emphasise a keyword in a sentence.

Slanted is used for names of applications in the performance evaluation sections.

Boldface is used in the main text to introduce a new term. All of these terms can be found in the index.

`Typewriter Font` is used to present any algorithms.

1.6 Summary

Upon reading this exordial chapter, we have gained an awareness of hardware technology trends facing the design of future database architectures. The disk is the overall bottleneck and the slow average access time will remain a problem in the future. On the other hand, CPU speeds will increase drastically over the next years. Memory improvements are similar to disk improvements but there is a two-order of magnitude gap in access time between these two storage types. Unless future OODBMSs vendors employ prefetching techniques, many I/O bound applications will suffer poor response time due to the disk bottleneck. OODBMS applications vary strongly in their structure of object relationships. In the thesis we try to give an understanding of which types of object access patterns are relevant for prefetching. We also want to clarify the effect of high and low object probability relationships and the consequence for the prefetch accuracy. In practice no prefetching is able to completely avoid demand fetches. Therefore, a major contribution of this work is to find the trade-off on the level of accuracy required for obtaining good results in total elapsed time reduction. We designed several prefetching techniques that try to avoid the page miss or at least to reduce the stall time at the client. Every prefetch operation will not reduce the total page fetch time because object relationships are often complex and consequently prediction is difficult, so that the prefetch cannot be started too early. Another task of this thesis is to explore the trade-off between page accuracy and prefetch distance. A high accuracy is mostly obtained at short distances and it becomes lower at longer distances. A prefetching technique has to make the right choice between distance and accuracy to reduce total elapsed time.

All these initially described problems will be further discussed in the research chapters. The next two chapters will introduce some background material which is relevant for understanding the research chapters.

Chapter 2

Basics of Object-Oriented Databases

OODBMSs have been commercially available over a decade. Three early OODBMSs projects laid the foundation in this area – GemStone [Copeland and Maier, 1984; Maier et al., 1986], which was based on Smalltalk, V-base [Andrews and Harris, 1987], which was based on a CLU-like language, and Orion [Kim et al., 1994], which was based on CLOS. GemStone originally evolved in an effort to make the Smalltalk programming language part of a database management system. Although work on persistent programming languages [Atkinson et al., 1983] had been underway for some time in the programming language community, work on applying those ideas to object-oriented languages was just taking off. It has to be admitted that the first generation of object-oriented databases was not sufficiently mature for wide use. Using them was like writing an assembler program; one has extensive possibilities but also a great responsibility and the task was very complex.

In the early 1990's many other OODBMSs appeared on the market, e.g. ObjectStore [Lamb et al., 1991], O_2 [Bancilhon et al., 1992] and Objectivity/DB [Objectivity, 1994]. The vendors of the systems improved the quality of the system by making them more user-friendly (through graphical interfaces) and more stable. In addition, a consortium of OODBMS product vendors banded together – under the leadership of Rick Cattell of SunSoft – and formed the Object Database Management Group (**ODMG**). The ODMG standard [Cattell, 1993] was proposed as a standard interface to all OODBMSs. It includes the definition of an object model, an object definition language and an object query language. The first implementations of this standard appeared in 1995. Unfortunately, many vendors decided to support only pieces of the standard. The situation today is that there is no consensus among the vendors on the architectural issues relevant to this thesis. In this chapter, we describe the solutions adopted by the major OODBMSs vendors.

The remainder of this chapter will be reviewing basic components of OODBMSs which are important for the design of a prefetching technique. At first, Section 2.1 explains the meaning of an object. Object identifiers are relevant for identifying prefetched objects and are reviewed in Section 2.2. Section 2.3 describes the principles of persistence. Buffer management (Section 2.4) and clustering (Section 2.5) are two components that have to be closely integrated with prefetching. The two basic network computing models for OODBMSs are described in Section 2.6. Section 2.7 discusses performance issues for OODBMSs and the evaluation through benchmarks. We implemented a prefetching environment into the EXODUS Storage Manager, which is briefly described in Section 2.8. Finally, in Section 2.9 we summarise this chapter.

2.1 Objects

The term **object** has many meanings in OODBMSs [Cattell, 1994]. The following two terms are used in all systems:

1. **Object Grouping.** Objects can serve to group data that pertain to one real-world entity – a transistor, a document or a person. A weak form of object grouping is incorporated in implementations of the relational model that have primary and foreign keys.

2. **Object Identity.** Objects can have a unique identity independent of the values that they contain. A system that is identity-based allows an object to be referenced via a unique internally generated number, an object identifier.

2.2 Object Identifiers

The design of the object identifier is a fundamental decision for the database implementation. It is also important for prefetching to identify objects that have to be prefetched and identify the pages on which the objects are resident.

The various implementation alternatives are described in Section 2.2.1. In Section 2.2.2 we explain the effect of the identifier design on performance. The preservation of the identity in virtual memory is explained in Section 2.2.3.

2.2.1 Implementation of Object Identifiers

[Khoshafian and Copeland, 1986] classified the implementation of object identifiers. Two parameters are important to measure the information content: **Data independence** and **location independence**. Data independence means that identity is preserved through changes in either data values or structure. Location independence means that identity is preserved through movement of objects among physical locations or address spaces. The classification is as follows:

1. **Identity Through Physical Address.** Perhaps the simplest implementation of the identity of an object is the physical address of the object. This physical address could be the real or the virtual address of the object (if the object system is operating in a virtual memory environment). A physical address implementation does not permit an object to be moved, so that there is no location independence. A virtual address implementation allows only whole pages of objects, not individual objects, to be moved within one virtual address space, providing minimal location independence. To improve location independence a forwarder could be used to point to the new address of an object. Physical address identifiers are employed by Wiss [Chou et al., 1985], Cricket [Shekita and Zwilling, 1990], ObjectStore [Lamb et al., 1991], Texas [Singhal et al., 1992] and QuickStore [White and DeWitt, 1994].

   Identifiers that point to physical disk position are useful for prefetching because the identity of a page can be obtained from the object identifier. This reduces the amount of time for predicting pages to be prefetched.

2. **Identity Through Indirection.** In Smalltalk-80 [Goldberg and Robson, 1983], an **oop** (object-oriented pointer) is used to implement identity. Therefore, identities are implemented through a level of indirection. In LOOM [Kaehler and Krasner, 1983], it is shown that this scheme could be used to support secondary-storage resident objects, providing support for a much larger number of objects. Indirect physical or virtual address implementations allow individual objects to be moved within one address

space, providing stronger location independence than direct address implementation but allowing sharing of objects among multiple programs. Indirect address implementations provide full data independence.

3. **Identity Through Structured Identifier.** This type of identifier is very popular in relational databases and in some object-oriented databases. It contains a physical and a logical component. The physical component can address a page or a segment. The logical component addresses a slot entry in a page. Structured identifiers provide full data independence and limited location independence. It is limited because only the logical component allows movement of objects. Structured identifiers are employed by ODE [Agrawal and Gehani, 1989], Mneme [Moss, 1990], Objectivity/DB, ONTOS [ONTOS, 1995], and Bess [Biliris and Panagos, 1995].

   The CORBA approach could be also classified in this section. The specification of the CORBA/Persistent Object Service [OMG, 1997] defines an abstract identifier based on a data store handle - persistent identifier **(PID)** pair. How the implementor wishes to use these to denote some instance of state is quite arbitrary. For example, the data store handle could indicate the database type and name, or it could denote just a filename. The PID could indicate the value of some key (if used to map to a relational database) or might be the offset within some file. It is necessarily abstract so all forms of database technologies can be covered by the specification.

4. **Identity Through Identifier Keys.** The main approach for supporting identity in commercial relational database management systems is by direct implementation of user- or system-supplied identifier keys. The tuples can be physically ordered (in most cases sorted) on the identifier key and an auxiliary structure (e.g. a B-tree) is constructed on top of the set of tuples to provide fast access to objects retrieved through their identifier keys. Identifier key implementations provide full location independence. They do not provide value independence because they consist of values.

5. **Identity Through Surrogates.** The most powerful technique for supporting identity is through surrogates (purely logical component). Surrogates are system-generated globally unique identifiers, completely independent of any physical location. If surrogates are associated with every object, then they provide full data independence. The logical address has to be mapped to a physical address. Surrogates are employed by GemStone and POSTGRES [Stonebraker et al., 1990].

6. **Identity Through Typed Surrogates.** The typed surrogate is a pair comprising a type ID and an object ID. Object IDs are always local to a type ID and generated by a counter for each type. This technique is employed by ITASCA [Itasca Systems, 1991] and ORION.

It appears that no single scheme for providing object identity is emerging as the dominant one. This is partly because of the different performance/flexibility trade-offs in the designs of these DBMSs.

2.2.2 Performance Aspects of Identifiers

The design and implementation of an object identifier is critical for the database performance. A database designed on physical OIDs will generally perform better than a database based on logical OIDs. Logical OIDs on the other hand will provide more flexibility for location independence. The following two parameters are important for database performance:

1. **Disk Access.** Physical OIDs are the fastest way to retrieve an object. The object can be read by one disk access and no mapping is necessary. Surrogate OIDs have to map the logical address to a physical address by hash table or btree. At first the information from the hash table has to be made memory resident and then the object itself. This normally involves at least two disk accesses (only one if the hash information is already resident). Structured OIDs, hybrid between physical and logical OIDs, can access every object with one disk access. Only a mapping from the logical element to the physical disk position in the page is necessary. If physical or structured OIDs use forwarders, the number of disk accesses would be two.

2. **Size of OID.** Another performance factor to consider is the OID size; OIDs longer than 32 to 48 bits can have a substantial effect on the overall size of a database, particularly because many of the databases contain complex heavily interrelated objects. In theory, 32 bits is adequate for about 4 billion objects, allowing a reasonably large database. However, OIDs of 64 bits or larger may be necessary for a variety of reasons:

   - In systems where it is not practical to find all references to an object, OIDs must be unique for all time, so that dangling references can be recognised.

- If OIDs are surrogates generated by a monotonically increasing function, it is generally not practical to reuse *holes* produced in the sequence by OIDs which are no longer (or never) used.
- In a distributed environment, it may be necessary to prefix the OID with a machine or database identifier to make the OID universally unique.

Many implementations convert OIDs used for references between objects into memory addresses when objects are fetched from disk, to make reference-following fast. That is, all references to an object's OID in currently cached objects are replaced with the object's address when the object is brought into memory. ObjectStore even uses a virtual memory address form in the *disk representation* of OIDs. The replacement of OIDSs with addresses, called **pointer swizzling**, can be exercised regardless of which OID representation is used (see Section 2.2.3).

2.2.3 Pointer Swizzling

Access to objects in memory can be implemented via pointer swizzling or mapping tables. Both techniques require some housekeeping information at client; the advantage of swizzling is that this overhead is minimised. Pointer swizzling means the OID is transformed into a virtual memory pointer. The application works with the pointer like a normal pointer in a programming language. At the end of the transaction the virtual memory pointer has to be converted back in to an OID again. Pointer swizzling is advantageous for an application in which there are many operations on the objects. However, swizzling may not be appropriate when there are only a few operations on the object, since the overhead of the translation may be too expensive. In view of this, some OODBMSs use mapping tables to find the virtual memory address of an object, given its OID. This technique has the advantage that the object can be moved in memory. Also, through the indirection, the access to an object is safer, because dangling pointers cannot crash the application.

[White, 1994] classified the following pointer swizzling techniques:

1. **Hardware vs. Software-based Swizzling.** Software-based checks use bits of the OID or tables to distinguish between swizzled and OID objects. Hardware-based swizzling schemes [Lamb et al., 1991; Wilson and

Kakkad, 1992] that use virtual memory access protection violations to detect accesses of non-resident objects have been proposed. The main advantage of the hardware-based approach is that accessing memory-resident persistent objects is just as efficient as accessing transient objects because the hardware approach avoids the overhead of residency checks incurred by software approaches.

A disadvantage of the hardware-based approach is that it makes providing many useful kinds of database functionality more difficult, such as fine-granularity locking, referential integrity, crash recovery, and flexible buffer management policies. In addition, the hardware approach limits the amount of data that can be accessed during a transaction to the size of virtual memory. This limitation could conceivably be overcome by using some form of garbage collection to reclaim memory space, but this would add additional overhead and complexity to the system. The hardware approach has been used in several commercial and research systems, including Dali [Jagadish et al., 1994], Cricket, ObjectStore, QuickStore and Texas.

2. **In-place vs. Copy Swizzling** Copy and in-place strategies differ primarily where they cache persistent objects in the main memory. In-place refers to an approach that allows applications to access objects in the buffer pool of the underlying storage manager, while the copy approach copies from the buffer pool into a separate area of memory, typically called an **object cache**, and applications are only allowed to access objects in the object cache. These techniques can be used independently of whether swizzling is being done. While the copy approach incurs some cost for copying objects, it has the potential to make more efficient use of memory by only caching objects that are actually used by the application. In addition, if pointer swizzling is being done, then the copy approach can save in terms of unswizzling work since, in the worst case, only the modified objects have to be unswizzled. Depending on the type of swizzling used, an in-place scheme may have to unswizzle an entire page of objects in the buffer pool whenever any object on the page is updated.

3. **Eager vs. Lazy Swizzling.**

   A swizzling technique is said to be *eager* if it swizzles all the pointers of an object collection on the first access of an object [Moss, 1992]. If pages are not resident they have to be prefetched from disk. The advantage of this technique is that no residency checks are needed at run-time to distinguish swizzled and unswizzled pointers. [Kemper and Kossmann, 1993] and [McAuliffe and Solomon, 1995] define eagerness to be swizzling all pointers in a page or object.

Lazy swizzling uses a more conservative approach to swizzling. The decision is made at run-time as to what and when to swizzle. The granularity of what to swizzle can be a page, an object or just a pointer. Pointer swizzling can be initiated on a dereference, a comparison of a pointer or an object fetch. The advantage of this approach is that less unnecessary objects are swizzled or even fetched into memory.

4. **Direct vs. Indirect Swizzling.**

   Direct swizzling techniques place the in-memory address of the referenced persistent object directly into the swizzled pointer itself. By contrast, under indirect swizzling a swizzled pointer points to some intermediate data object (usually termed a **fault block** in [Hosking and Moss, 1993]), which in turn, points to the target object when it is in memory. The advantage of indirect swizzling is that it provides more flexibility to uncache objects. If the object is not resident anymore the pointer to the target object in the fault block is set to zero. It also provides a higher level of safe object access. The obvious disadvantage is that the overhead of accessing the object via the fault block is more time-consuming.

2.3 Persistence

Transient data last only for the invocation of a program. Persistent data survive the program termination and are stored in a persistent object store. Persistence was first defined by [Atkinson et al., 1983]. To achieve orthogonal persistence they defined three principles:

1. **The Principle of Persistence Independence.** The form of a program is independent of the longevity of the data that it manipulates. Programs look the same whether they manipulate short-term or long-term data.

2. **The Principle of Data Type Orthogonality.** All data objects should be allowed the full range of persistence irrespective of their type. There are no special cases where objects are not allowed to be long-lived or are not allowed to be transient.

3. **The Principle of Persistence Identification.** The choice of how to identify and provide persistent objects is orthogonal to the universe of discourse of the system. The mechanism for identifying persistent objects is not related to the type system. Objects are identified by reachability. In this, the identification of persistent objects is performed by the system automatically by computing the transitive closure of all objects reachable (by following pointers) from some persistent root or roots.

The third Principle of Persistence Identification was implemented in PS Algol [Atkinson et al., 1983] and GemStone. The advantage of this approach is that the programmer is free of using persistent type or function calls. This means the application code is very portable. The disadvantage is that it involves an overhead for moving objects from the transient to the persistent root. Persistence can also be implemented using:

1. **The Type of an Object.** An object is made persistent by using a persistent data type. The type might be declared persistent or made persistent by inheritance from a persistent class. This approach is used by Objectivity/DB, ONTOS and POET [Vigna, 1997].
2. **An Explicit Function Call.** There exists a method or function to make an object persistent. For example ObjectStore provides an overloaded new operator to create persistent objects.

Both approaches are less convenient for the programmer but easier to manage for the database system and therefore used in most commercial OODBMSs.

2.4 Buffer Management

An efficient object caching technique is very important for database performance. The unit of transfer between client and server can be an object, a page, a segment or a query result. At the client there exists three alternatives to cache objects or pages:

1. **An Object Server Architecture.** Figure 2.1 shows the architecture of an object server. The client has one object buffer whereas the server has an object buffer and a page buffer. The objects at the server are copied from

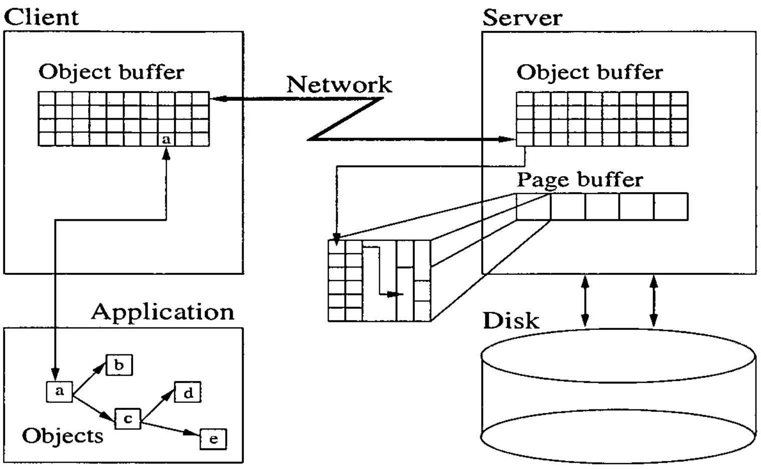


Figure 2.1: Object server architecture.

the page buffer into the object buffer. The transfer between client and server is an object or a group of objects. The object server stores objects in units of pages or segments on disk. The advantage of this design is that there are no unneeded objects in the client buffer pool and it is a finer granularity for locking. This technique is employed by Versant [Versant, 1992], UniSQL [Cattell, 1994], Thor [Liskov et al., 1996], ORION [Kim et al., 1994] and ITASCA.

2. **A Page Server Architecture.** A page server system transfers pages between client and server and has a page buffer pool at client and server. A database page is divided into slots and data. The slots have information about the data and a pointer to the offset of the data. An application pointer points directly to the slot of a page. This architecture normally outperforms the object-base approach and is incorporated in most commercial object-oriented databases: e.g. EXODUS [Carey et al., 1986a], ObjectStore, Objectivity/DB.

3. **A Dual-Buffer Architecture.** This architecture has both a page buffer and an object buffer and is depicted in Figure 2.2. In [Kemper and Kossmann, 1994] showed that dual-buffering very often outperforms page-buffering in the OO7 benchmark [Carey et al., 1993]. Dual-buffering is also used in [Cheng and Hurson, 1991a], DASDBS [Schek et al., 1990], ORION, O_2 [Bancilhon et al., 1992] and Shore [Carey et al., 1994a].

There has been a long discussion in the database literature about the performance advantages of these architectures [DeWitt et al., 1990; Hohenstein

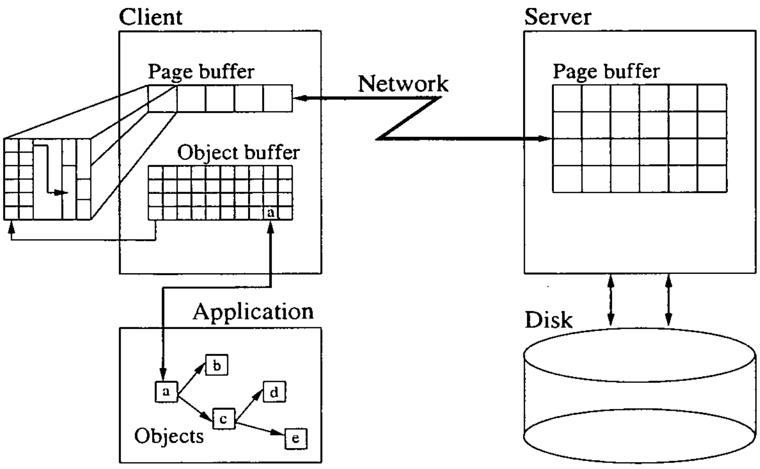


Figure 2.2: Dual-buffer architecture.

et al., 1997]. In cases where network overhead is high and objects cannot be clustered on pages, the object server approach could conceivably be faster. The page server approach might provide less performance advantage for applications that can encapsulate their operations on data in a query language, rather than operating directly on objects. In a network environment, some applications could be considerably slower using this approach, because it is more efficient to ship the operation (query) to where the data is stored, rather than shipping all the data to a workstation to perform the operation. On the other hand, if many objects from a page are accessed then the page server will outperform the object server.

The network cost is an important criteria [Delobel et al., 1995]. The cost of sending a message on a network is not proportionate to the length of the character string transmitted; the costs of sending a 100-byte object and a 4 K-byte page are almost the same. The cost of initialising a message transfer will remain high in relation to the transmission time. This fact is an important argument for a page server. OODBMSs have the possibility to execute a query on the server or on the client. For example executing a single query would perform better on the server whereas repeated queries or operations on the same data would perform better on the client.

In a client/server database system the disk latency is in general higher than the network latency. Due to the fact that both object and page servers store objects on disk pages, both systems retrieve the same number of demand requests from the disk. Therefore the high disk latency has no impact on the performance comparison of object and page servers.

The discussion about the performance advantages of these architectures can also be extended for prefetching. A client can either prefetch objects, pages or a group of objects. Prefetching objects would induce a high overhead for the transferring each object and is therefore not appropriate. The trade-off between object group prefetching and page prefetching will be evaluated in Chapter 6.

2.5 Clustering Techniques

Clustering techniques for OODBMSs have a strong impact on the database performance. Having objects clustered together on a page reduces the number of disk I/Os. Clustering is especially important for page server systems because their intention is to ship related objects to the client. [Bertino et al., 1994] surveyed clustering techniques used in commercial and research systems:

1. **Type-based Clustering.** Objects can be clustered according to their type. All instances of a class are clustered in one or more segments. This technique is employed by ORION, ENCORE [Hornick and Zdonik, 1987] and the ODE Object Manager [Agrawal et al., 1993]. ORION also allows a graph of the type to be clustered in the same segment. Most relational databases use this type of clustering to store tuples together. In OODBMSs it would be useful for collection classes.

2. **Value-based Clustering.** Objects are clustered according to their values. A special attribute of the object is the clustering criteria. A special kind of value-based clustering is index-clustering: it consists of performing value-based clustering and imposing an index (usually a B^+-tree) on the clustering attributes. This clustering strategy can be used to optimise frequent queries.

3. **Composite Object Clustering.** In this case an object is clustered with some or all of its sub-components. This cluster aggregation relationship can be defined at run-time or in the schema definition. This technique is beneficial for following the pointer references of an object. All subcomponents would be already resident after the first access to the object. The DASDBS database and ORION uses this type of clustering.

4. **Clustering based on Greedy Graph Partitioning.** The greedy graph algorithm partitions the object composition graph into a set of sub-graphs

such that for each sub-graph all objects fit in one page [Gerlhof et al., 1993]. The clustering is performed according to the weights of the object relationships. A list sorts the relationships according to their weights in descending order. The algorithm then clusters the objects with the highest weights together.

5. **Stochastic Clustering.** By representing access patterns as stochastic processes, clustering can be formulated as an optimisation problem [Tsangaris and Naughton, 1991]. Given a statistical description of the client access request stream and a frame (physical page) access cost formula, the problem is to find a mapping from the set of all objects to the set of frames such that the average cost of access is minimised, while there are at most n objects per frame and other additional constraints are satisfied. It was shown [Tsangaris and Naughton, 1991] that solving this problem while considering two consecutive requests is a weighted graph partitioning problem. Consequently, optimising while considering more than 2 consecutive requests can be construed as hyper-graph partitioning problem [Tsangaris and Naughton, 1992].

6. **Custom Clustering.** The user of the database system can specify a segment in which he/she wants the objects to be placed. ObjectStore for example employs this technique. Another possibility is to specify an object to be placed near another object (EXODUS, Objectivity/DB).

Most commercial systems use type-based or composite-based clustering. Applications with complex object structures would benefit from the composite-based clustering. Type-based clustering is more useful for simple collection classes, that spawn over many pages. A query over the collection would make all objects resident. However, some studies devoted primarily to clustering in object bases ([Tsangaris and Naughton, 1991], [Gerlhof et al., 1993]) have pointed out the need of exploiting behavioural information, which gives hints about access patterns of objects, in order to approach optimal clustering. Using this information could be more specific for clustering but also induces overhead for updating this information.

Another classification of clustering techniques is dependent on the time of clustering. **Static clustering** is done when the objects are created. Frequent access to the object can destroy the object cluster and off-line clustering is necessary to rebuild the object cluster. **Dynamic Clustering** changes the objects belonging to a cluster at run time. Dynamic clustering techniques [Cheng and Hurson, 1991a] usually improve the overall system performance but they have a great runtime overhead which has to be justified.

Objects can be clustered on pages or segments. Page clustering is useful when the **working set** of objects cannot determined precisely. Value-based and composite object clustering is especially applicable for page clustering. If it is possible to specify a logical group, a larger conglomerate like a segment could perform better. Typed-based clustering is applicable for segment clustering.

[Day, 1993] argues that clustering is a *zero-sum game*: making the clustering better for one application makes it worse for another. Applications have different access pattern (e.g. depth-first vs. breath-first) which can differ in many cases to the actual clustering structure on disk. Day proposed an idea called **crystals** in which each user can specify the desired access pattern. The object server then reads all related objects from disk and sends this group of objects to the client. The problem with this approach is that the server becomes an even higher bottleneck because of additional computation and fetching the set of objects.

The conclusion concerning clustering techniques is that no technique offers a perfect solution. The choice of a clustering technique is dependent of the application data structures and access patterns.

2.6 Network Computing Models

There are two basic network computing models for OODBMSs: **client/server** and **peer-to-peer communication.** The client/server model dictates that one centralised server is responsible for serving all clients requests and sharing the resources on the network. This approach is used in most commercial OODBMSs like ObjectStore, O_2, Ontos and Versant. The advantage is that all data are centralised and query processing is very efficient. Also there is no identification overhead for addressing an object on a server.

In a peer-to-peer architecture every workstation can act as a server. The server manages the requests from the local client and client requests from other workstations. The advantage is that there is no bottleneck server and clients can store their data locally. The use of resources on the network is far greater and the sharing of information is easier to accomplish. There is a great deal more flexibility in this model. The disadvantage of such systems is that they are hard to manage. Security is also a problem, and performance cannot compete with the client server model. Bess, ITASCA, Objectivity/DB and Shore are peer-to-peer systems.

The choice of the network computing model has also relevance for prefetching. The advantage of a peer-to-peer architecture is that prefetch requests could be executed in parallel by the servers and therefore unload a centralised server.

2.7 Performance Issues

The efficient implementation of all the aforementioned database components in the previous sections have an effect on the overall system performance. The implementor of an OODBMS has to be clear about the importance of performance. The design of the database is a compromise between security of data, location independence and high performance, which is important to Objectivity/DB, Versant and ObjectStore respectively.

A major performance problem for all OODBMSs is the I/O bottleneck. Many optimisation techniques tried to reduce this bottleneck, e.g. caching (see Section 2.4), clustering (see Section 2.5), prefetching (see Chapter 3), main memory databases [Garcia-Molina and Salem, 1992] or disk striping [Salem and Garcia-Molina, 1986]. For example Objectivity/DB addresses this problem by distributing objects within an object database across multiple servers and cluster data into partitions.

To evaluate the performance of an OODBMS there exists two widely used benchmarks: OO1 [Cattell, 1992] and OO7 [Carey et al., 1993]. The OO1 benchmark has a simple object graph in which every object has three pointers to other objects. It defines a clustering level of 90%, i.e. 90% of the references are local to objects in the page. The performance evaluation showed an OODBMS outperforming a DBMS in lookup, object traversal and insert operations. The OO7 benchmark is much more complex in its object graph structure and has more operations. In contrast to the OO1 benchmark, it uses a dense and a sparse traversal and update and query operations. The benchmark was implemented to compare the performance of four OODBMSs: Versant, Ontos, Objectivity/DB and EXODUS; the system with the best results was EXODUS.

2.8 The EXODUS Storage Manager

The EXODUS client/server database system [Carey et al., 1986a; Carey et al., 1986b; Carey et al., 1990] was developed at the University of Wisconsin. It aids a database implementor in the task of generating a DBMS by providing a storage manager, a programming language E (an extension of C++), a library of access-method implementations, a rule-based query optimiser generator, and tools for constructing query-language front ends.

The basic representation for data in the storage manager is a variable-length byte sequence of arbitrary size, incorporating the capability to insert or delete bytes in the middle of the sequence. In the simplest case, these basic storage objects are implemented as contiguous sequences of bytes. As the objects become large, or when they are broken into non-contiguous sequences by editing operations, they are represented using a B-tree of leaf blocks, each containing a portion of the sequence. Objects are referenced using structured OIDs. The OID has the form (volume#, page#, slot#, unique#), with the unique# being used to make OID's unique over time (and thus usable as surrogates). The OID of a small object points to the object on disk; for a large object, the OID points to its large object header.

On these basic storage objects, the storage manager performs buffer management (LRU or MRU), concurrency control, recovery, and a versioning mechanism that can be used to provide a variety of application-specific versioning schemes. The database client and server communicate via sockets. The client specifies the requested data in a message structure and sends it to the server. The server retrieves the requested page and sends it to the client.

2.9 Summary

In this chapter we briefly mentioned the components of an OODBMS architecture relevant to this thesis. The right choice of the object identifier design, the pointer swizzling technique, the buffer replacement strategy and the clustering technique and the close integration of the components have a big impact on the performance of an OODBMS. To alleviate the high cost of disk access, all commercially available systems cluster objects onto pages and equip client and server with large cache sizes but no system makes use of intelligent prefetching techniques to reduce I/O costs. The increasing gap in access time between memory and disk and memory and CPU induces new challenges for future high-performance object stores. In the next chapter we review previous work on prefetching in the area of OODBMSs and other subjects.

Chapter 3

Basics of Prefetching Techniques

The aim of a prefetching technique is to diminish applications' elapsed times. There are several aspects of optimisation to reduce response time:

1. **Reduction of Seek Times.** In a client/server database system the disk request queue is an important performance component. Prefetching tries to insert many requests into the queue to give the queue manager the opportunity to sort disk accesses [Patterson et al., 1993]. The queue manager sorts the disk requests according to the position of the pages on disk to reduce the seek costs, which is the expensive part of the disk access.

2. **Reduction of Idle Times.** Year-on-year, improvement in CPU performance outstrips performance improvement in other aspects of computer technology, such as disks. Consequently, in some applications, processor power may be increasingly under-utilised. Idle system resource could be used to predict which pages should be prefetched.

3. **Global Resource Optimisation.** A carefully designed prefetching technique should consider the adequate consumption of resources [Voelker et al., 1998; Kraiss and Weikum, 1998]. For example one client might use prefetching extensively and consequently acquires a high percentage of system resources, like network bandwidth, server processing times and disk retrieval times. This particular client would probably improve its performance but overall system performance would decrease.

4. **Group Requests.** A group request has the advantage over a single request that the average cost per unit is smaller. The origin of a group request may come from a single user, i.e. one user requests many pages at once or it

comes from multiple users, i.e. multiple users request closely-clustered pages. Applications of group requests can be found in the management of complex objects [Weikum, 1989; Keller et al., 1991; Maier et al., 1994], file systems [McVoy and Kleiman, 1991] and big objects [IBM, 1994]. Obviously, both the client and server need a group-oriented request interface [Weikum et al., 1987].

5. **Overlapping of CPU and I/O.** Prefetch operations are overlapped with the client application processing. Processing objects must be overlapped with the time during which the prefetch takes place. If the processing time is high, the amount of saving will also be high. Otherwise only a small saving can be achieved. If the application runs on a multiprocessor, the client processing can be done in parallel with the prediction computation. Otherwise in the uniprocessor system the prediction information can be computed at stall times for demand fetches.

6. **Parallel Disk Access.** Nowadays many database systems share data on multiple I/O-subsystems to avoid the I/O-bottleneck. This technique is called **de-clustering** or **disk-striping** [Salem and Garcia-Molina, 1986; Treiber and Menon, 1995]. The idea is to balance the load on many disks so that many I/O-jobs can be served in parallel. This is supported through the popular **RAID** technology[1] [Chen et al., 1994] which replicates data on many cheap disks. Disk-striping might help an individual application directly but this does not directly impact many parallel I/O jobs. Other aspects of disk technology, such as replication, could benefit many parallel I/O jobs. Prefetching a group of pages is an ideal application for the parallel execution of disk requests [Pai and Varman, 1992; Wu et al., 1994]. At the same time it reduces the cost of prefetching, i.e. blocking other demand requests. [Patterson, 1997] sees the strength of prefetching in the parallel execution of the requests.

In this chapter we survey previous work in prefetching. This subject has been studied extensively over the last two decades so that we can concentrate on material that is relevant to our work. An important aspect is the collection of prediction information. Several techniques are classified in Section 3.1. The interdependency of clustering and prefetching is subject of Section 3.2. Section 3.3 surveys ideas about the integration of clustering and buffer manage-

[1] Redundant Array of Inexpensive Disks.

ment. Implementations issues of prefetching in a client/server architecture are described in Section 3.4 and other implementation issues in Section 3.5.

3.1 Prediction Techniques

The condition for the use of a prefetching technique is that there exists sufficient knowledge about the access pattern of an application; this means the probable time of access and the knowledge of which object will be accessed. Therefore a successful prefetching technique is very dependent on the accuracy of the prediction technique.

This section classifies all proposed prefetching techniques into four categories. **Prediction Engines** in Section 3.1.1 using an internal oracle to decide which objects or pages should be prefetched. In Section 3.1.2 we describe **program-based techniques**, which perform code analysis to obtain prefetch information. **Hint-based techniques**, that obtain the prediction information from a user, are discussed in Section 3.1.4. Other classifications of prefetching techniques can be found in Section 3.1.5.

3.1.1 Prediction Engines

Most prefetching techniques use a separate prediction engine that exploits knowledge of future application accesses. Prediction engines have many realisation possibilities. A **deterministic prediction** technique determines a strategy, e.g. load one block ahead. **Statistical prediction** techniques generate probabilistic information about future access by analysing past accesses. **Object structure-based prefetching** techniques predict the access via pointers from objects to other objects, which are mostly used in OODBMSs.

3.1.1.1 Deterministic Prediction

Deterministic prefetching determines a strategy concerning when and what to prefetch. The simplest algorithm is to load on every demand fetch the next adjacent block on disk: **One Block Lookahead** (OBL) [Joseph, 1970]. The OBL-prediction engine was also used in databases [Smith, 1978b; Smith, 1978a; Smith, 1982; Smith, 1985] and similarly in parallel file systems [Arunachalam and Choudhary, 1995]. Two hardware solutions can be found in [Fu and Patel, 1991; Palacharla and Kessler, 1994]. A strategy for a parallel merge-sort algorithm was developed in [Pai and Varman, 1992; Pai et al., 1994; Wu et al., 1994]. Simple strategies were already used in vector programs [Fu and Patel, 1991]. Another strategy loads either sequential parts of the file or the whole file according to the access patterns of parallel applications [Kotz and Ellis, 1990; Kotz and Ellis, 1991]. Simple deterministic strategies work well for special applications but cannot be applied to the complex relationships between objects that typify OODBMSs applications.

Prefetching in Commercial DBMSs. Many commercial DBMSs employ deterministic prefetching techniques to load pages from the disk into the server's buffer pool. IBM's DB2 distinguishes between **sequential prefetch** and **list prefetch** [Teng and Gumaer, 1984; Mohan et al., 1993; Gassner et al., 1994; IBM, 1994; IBM, 1997]. Sequential prefetch reads several consecutive pages into the buffer pool using a single I/O operation. The list prefetch is used when the required pages are not in sequential order. The list of prefetch pages is obtained from the index structure and all the pages are retrieved in parallel. The ORACLE database system optimises full table scan operations by data prefetching [Gokhale, 1997] and performs **row-prefetching** for tuples for Oracle's *JDBC* driver [Oracle, 1997]. The query optimiser in Sybase's SQLServer also performs sequential prefetch for consecutive physical pages [Agarwal, 1995].

Broadcast Disks. A hybrid prediction technique with a deterministic component and a statistical component was developed for so-called broadcast disks [Acharya et al., 1996a; Acharya et al., 1996b; Acharya et al., 1997; Acharya, 1998]. The idea behind broadcast disks is that the server sends pages in a cycle. The broadcast program determines the broadcast frequency and cycle lengths. The client can use this knowledge to replace pages that will be broadcast again soon under low costs (**tag-team caching**). This technique was extended for a wireless environment in which mobile clients may have only a low-bandwidth channel for sending messages [Zdonik et al., 1994]. The conventional cache and prefetching management techniques which are mainly designed for fixed networks are inefficient in the radio environment where the communication channels are unpredictable and highly variable with time and location. To reduce this high latency another prefetching technique fetches adjacent pages from a faulted page to the mobile client [Liu, 1994].

3.1.1.2 Statistical Prediction

The deterministic prediction techniques from the previous section use simple strategies to prefetch pages with low prediction overhead. Statistical prediction methods monitor the access pattern of an application. After a sufficient gathering of knowledge about the access pattern the information is used for prefetching. This information gathering process is far more expensive than the one associated with deterministic techniques. We define our own statistical prediction technique in Chapter 5.

Table method. To keep track of the inter-dependency of block accesses a table was used to register blocks which will be accessed after a current block [Grimsrud et al., 1993]. Each referenced block has an associated weight which gives information about the probability of accessing the next block. Two other studies use the same algorithm [Shah and Kumar, 1995; Chee et al., 1997]. In the area of CPU prefetching a so-called **reference-prediction-table** was introduced to recognise memory distances with repeatable strides [Chen and Baer, 1992; Chen, 1993; Dahlgren and Stenström, 1995; Chen and Baer, 1995].

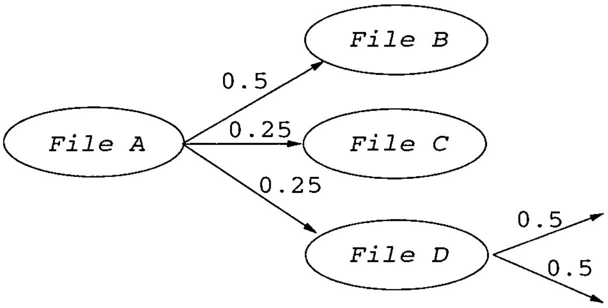


Figure 3.1: Dependency graph that depicts the frequency of inter-dependent file accesses.

Stochastic process. The probability of future file accesses were established by a **dependency graph** that depicts the pattern of accesses to different files stored at the server [Griffioen and Appleton, 1993; Griffioen and Appleton, 1994; Griffioen and Appleton, 1995a; Griffioen and Appleton, 1995b]. Figure 3.1 gives an example of a graph that has a node for every file and the arc from a node gives information about previous file accesses from the node. Every arc has also an associated weight. A similar approach [Kroeger and Long, 1996] also creates a tree of file dependencies providing information about previous file system events through a finite multi-order context modelling technique adapted from the data compression technique **Prediction by Partial Match (PPM)** [Bell et al., 1990] (see below). Another similar technique keeps track of the file access pattern of processes and forked child processes [Tait and Duchamp, 1990; Lei and Duchamp, 1997].

Prefetching techniques for the World Wide Web use mostly probabilistic models to compute the page probability. The prediction algorithm of [Padmanabhan, 1995; Padmanabhan and Mogul, 1996] is based on the idea of constructing a dependency graph [Griffioen and Appleton, 1994] to keep track of client access pattern. A very similar approach uses the conditional probability of inter-related document accesses [Bestavros, 1995; Bestavros, 1996; Jiang and Kleinrock, 1997; Jiang and Kleinrock, 1998]. Changes of the user's access pattern of web pages were also studied [Cunha and Jaccoud, 1997; Cunha, 1997; Banatre et al., 1997; Loon and Bharghavan, 1997; Yamaguchi et al., 1997].

Markov-Chains. For a discrete-time stochastic process with a discrete state space, if the future of the process depends only on the current state of the system, the process is called a **Markov chain**. Most Markov chains prefetching techniques use a **discrete-time Markov chain**[2]. The first object-oriented prediction engine to use a neural net was introduced in [Palmer and Zdonik, 1990; Palmer and Zdonik, 1991]. The prediction engine learns object access patterns over time. Unfortunately this work does not mention the CPU and storage overhead of the proposed technique. A Markov chain was also used to prefetch neighbours of an object in the area of hyper-media systems [Ghandeharizadeh et al., 1991]. The frequency of access to the objects of a hyper-media application is based on the frequency of access to its hyper-links. A similar technique learns about relationships among the blocks by looking for patterns in the stream of block requests [Salem, 1991]. If one block, say y was frequently requested shortly after past requests to x, then y is included in the prediction and prefetched.

Data compression techniques have been used to predict future page access in databases [Curewitz et al., 1993; Krishnan, 1995]. A PPM data compressor was compared with Lempel-Ziv (LZ) algorithm [Ziv and Lempel, 1978]. PPM has a **jth-order Markov predictor** on a page access sequence that uses statistics of context length j from the sequence to make its predictions for the next page. The applied LZ predictor creates a parse tree with probabilities for each page access. The PPM technique showed the better performance for data compression and prefetching. An earlier theoretical study had already demonstrated a close relationship between prediction and compression techniques [Vitter and

[2] The time interval between two object accesses is discrete.

Krishnan, 1991].

Markov chains are sufficient to model many scientific input/output file access patterns; however, if a single file has several predictable patterns the use of a **Hidden Markov Model** is more appropriate [Madhyastha and Reed, 1997]. Recently Markov-predictors were also designed for CPU prefetching [Phalke and Gopinath, 1995; Joseph and Grunwald, 1997]. Joseph and Grunwald used the **miss address stream** as a prediction source. To incorporate document-specific client interaction times between successive document requests a **continuous-time Markov chain** model was proposed [Kraiss and Weikum, 1997; Kraiss and Weikum, 1998]. The prefetching techniques which were designed for multimedia objects on tertiary storage, which fetches the documents with the highest number of expected accesses within a specified time horizon into secondary storage.

Branch prediction. Branch prediction reduces performance degradation due to branch instructions. A pipeline with branch prediction uses some additional logic to predict the outcome of a branch decision before it is determined. The pipeline then begins prefetching the instruction stream from the predicted path. If the branch is predicted to be taken, the branch target address also must be predicted. The processor may predict the branch either by a **static branch prediction** [Calder et al., 1997] or **dynamic branch prediction** [Lee and Smith, 1984; McFarlin and Hennessy, 1986; Kaeli and Emma, 1991; Baer and Chen, 1991; Fischer and Freudenberger, 1992; Calder and Grunwald, 1994; Talcott et al., 1994] technique. In the static case, (also called software-directed), the compiler performs static program analysis and selectively inserts prefetch instructions. Whenever the instruction executes, the processor prefetches from the same predicted path. In the dynamic case (also called hardware-based), the hardware maintains some past information on the branch instruction being executed. The processor uses this information to predict the branch decision according to some prediction algorithm. A good survey over branch prediction techniques can be found in [Lilja, 1988; VanderWiel and Lilja, 1997].

Prefetch Threshold. Some statistical prefetching techniques compare the document probability with a **static threshold** [Salem, 1991; Griffioen and Appleton, 1994; Padmanabhan and Mogul, 1996] or a **dynamic threshold**. The advantage of a dynamic threshold is that it can consider factors like system load and capacity, but it involves a higher computation overhead. One dynamic threshold algorithm addresses the trade-off between bandwidth usage and latency by considering the delay cost per time unit and the system resource cost per packet [Jiang and Kleinrock, 1997; Jiang and Kleinrock, 1998]. In multimedia systems the size of a document is not uniform [Kraiss and Weikum, 1997]. To avoid unnecessary space consumption every document has an associated weight which is computed by dividing the number of requests to that document by the size of the document. A prefetch to a document is only issued when it has a higher weight than a replacement victim. In addition to the prefetch decision the benefit of the prefetch must exceed the penalty.

3.1.1.3 Object Structure-Based Prefetching

Object structure-based prefetching techniques predict the access via pointer references from objects. In our work we present a structure-based technique in Chapter 4 and combine it with a statistical prediction technique in Chapter 5. The prefetching technique from [Chang and Katz, 1989; Chang, 1989] is a hybrid between an object structure-based and hint-based technique. The user provides hints about the data access, such as "*my primary access is via configuration relationships*". The hint is used by the client to load then the next immediate object in advance. Considering the high cost of a page fetch this technique can achieve only small savings in elapsed time. This study was extended to take into account: *multiple hints, a prefetch depth* and *physical storage considerations* [Cheng and Hurson, 1991b].

A more general technique was implemented on top of the Volcano query systems [Keller et al., 1991; Maier et al., 1994]. The client object buffer with an **assembly-operator** which makes a breadth-first search on all object references and loads successive non-resident objects. The assembly-operator obtains its object information (object structure, object semantics) from a *template*. In addition, the assembly operator considered information about the positioning of the disk head[3] and statistics about the degree of object connectivity. The disadvantage of this technique is, that it fetches many unnecessary objects and pages because it does not consider the probability of object accesses.

In the commercial OODBMS GemStone [GemStone, 1991] the application programmer has the choice of three functions for loading objects: *full-traversal, traversal-to-level-n and path-traversal*. The first function loads all the objects

[3] This is only possible with physical OIDs.

in the transitive closure, the second function loads all the objects to a level n. The path-traversal is similar to the technique of [Chang and Katz, 1989]. The Thor database system prefetches a group of objects from the server [Day, 1995; Liskov et al., 1996]. Several prefetching variants were tested in the experiments: *breadth-first*, *depth-first* and *class-hint*[4]. The breadth-first techniques included variants that check if the object is already resident at the client and variants that do not check this. The breadth-first method with the client check emerged as the best prefetching variant.

There has been some other work on prefetching pointers in advance which is not relevant to our work [Butler, 1987; Burdorf and Cammarata, 1990; Cleal, 1996].

3.1.2 Program-Based Techniques

Program-based prefetching techniques provide functions to prefetch and free pages from the buffer pool [Trivedi, 1977]. Non-linear data structures like binary trees are a big challenge for program-based prediction techniques because the reference order is highly data-dependent and prediction is only possible for a short lookahead [Klaiber and Levy, 1991; Chen and Baer, 1992]. This is similar to object-oriented databases which are navigating through object references.

[Mowry et al., 1992] suggested to decompose a loop into three parts. The first part generates the pre-run which loads an object k iterations before access. In the second part the prefetching of objects continues and objects are accessed by the application. In the third part all objects are already resident for access. This transformation is done automatically by an algorithm. The problem is that the program code size is increased. An alternative would be to use prefetch predicates. [Mowry et al., 1992] did not consider this approach because of the relatively high costs at run-time. In database systems these costs are low compared to I/O-costs.

In real-time databases every database access operation was transformed into an external procedure call with the known and unknown arguments at compilation time [Wedekind and Zoerntlein, 1986; Kratzer et al., 1990]. Such a procedure determines the set of database pages which will be accessed. This computation is executed in parallel with the dual database application. The

[4] Hints to prefetch relevant pointers of the class.

technical synchronisation of both the application and the page computation is not an easy task.

Deferment of Prefetch Operations. The aim of deferred prefetch operations, also called scheduling in program optimisations, is to increase the overlapping time between I/O and CPU. All the discussed work in this section comes from the area of compiler construction and microprocessors. The foundations of scheduling of a prefetch-operation was produced in [Gornish et al., 1990]. They developed a conservative algorithm that computes the earliest time at which a prefetch-operation could be started. The deferment of prefetch operations was limited to data dependencies and the control flow. A measure to reduce wrong prefetches is to defer the start of a prefetch operation behind a loop or branch [Rogers and Li, 1992]. Furthermore, it is not worthwhile to start the prefetch operation too early because the data could be invalidated or already replaced from the buffer [Tullsen and Eggers, 1993; Tullsen and Eggers, 1995]. A solution to this problem is to limit the deferment of a prefetch to a maximal distance [Chen and Baer, 1992; Chen and Baer, 1994].

Program-based techniques are not important for databases because they do not consider the content of the buffer pool for a prefetch decision. Nevertheless the deferment of a prefetch operation to a maximal distance had an influence on our work in Chapter 4.

3.1.3 Off-line Techniques

Off-line techniques have a perfect knowledge about the future reference pattern. Assuming a perfect knowledge is in general not practical for database accesses. One exception is an off-line technique, called **Prefetch Support Relation** [Gerlhof and Kemper, 1994b], stores the precomputed page answer of a database operation, i.e. all pages which are referenced by a particular operation invocation. In addition, it stores the frequency of page access and the ordering of the page answer according to the first reference of a page during the execution of an operation.

Cao et al. [Cao et al., 1995b; Cao et al., 1995a; Cao et al., 1996; Cao, 1996] also used off-line techniques to develop an optimal combined prefetching and caching technique (see Section 3.3.2). They proposed three simple prefetching strategies:

1. **The Conservative Strategy.** The conservative prefetching strategy tries to minimise the elapsed time while performing the minimum number of fetches. The conservative prefetching strategy performs exactly the same replacements as the optimal off-line demand paging strategy MIN [Belady, 1966], except that each fetch is performed at the earliest opportunity.

2. **The Aggressive Strategy.** The aggressive prefetching strategy is the strategy that always prefetches the next missing block at the earliest opportunity. In order to bring in this next missing block it replaces the block whose next reference is furthest in the future. Notice that aggressive is not mindlessly greedy – it at least waits until there is a block to replace whose next reference is after the request to the missing block.

3. **The Controlled-Aggressive Strategy.** The controlled-aggressive strategy behaves like the aggressive strategy but also considers the disk workload. A prefetch is issued only when the disk is idle.

Kimbrel and Karlin [Kimbrel et al., 1996a; Kimbrel and Karlin, 1996; Kimbrel et al., 1996b; Kimbrel, 1997] extended the work of Cao et al. for parallel disk access with two prefetch algorithms:

1. **The Reverse Aggressive Algorithm.** At first the algorithm transforms the reference string into the reverse reference string. Switching between forward and reverse sequences, i.e. fetches become eviction and vice versa. In the reverse direction the block to evict is the one not needed for the longest time which is on the same disk as the free disk. This reverse sequence is then derived to the forward sequence in which evictions become prefetches. The advantage of reverse aggressive over aggressive is that aggressive chooses evictions without considering the relative loads on the disk, whereas reverse aggressive greedily evicts to as many disks as possible on the reserve sequence. In the forward direction, this translates to performing a maximal set of fetches in parallel.

2. **The Forestall Algorithm.** This algorithm is a hybrid between the **fixed horizon algorithm**[5] [Patterson et al., 1995] and the aggressive algorithm. Depending on the number of blocks and remaining time to fetch these blocks it behaves like fixed horizon or aggressive.

[5] This algorithm uses the average disk access time divided by a system computation time to determine the start of a prefetch.

All the presented techniques in this section assume perfect knowledge. Having this knowledge makes it easy to develop an integrated caching and prefetching technique. Unfortunately most applications do not have this knowledge which makes these techniques of marginal use. However, an understanding of techniques that work with perfect knowledge gives us some insight into what is possible in more realistic situations.

3.1.4 Hint-Based Techniques

The knowledge about future I/O accesses could be obtained from a programmer or user of the system. The programmers could give hints about their program's accesses to the file system. Thus informed, the file system could transparently prefetch needed data and optimise resource utilisation. One example of hint-based prefetching is **Transparent Informed Prefetching (TIP)** [Gibson et al., 1992; Patterson et al., 1993; Patterson and Gibson, 1994; Patterson et al., 1995; Rochberg and Gibson, 1997; Tomkins et al., 1997; Tomkins, 1997]. Hints can be divided into hints that disclose (e.g. *I will read these 50 files serially and sequentially*) and hints that advise (*cache file F*).

Hints that advise do not give a lot of usable knowledge to the file system because it might not be able to accommodate the file access pattern given its current resource constraints. An implementation into the Mach operating system uses the advice-approach [Song and Cho, 1993; Cho and Cho, 1996]. Hints that disclose are more valuable for portability and flexibility to support global system resources. The **ELFS** file system [Grimshaw and Loyot, 1991; Karpovich et al., 1994] employs the disclosure approach.

Hints could be also be provided by a user of an object-oriented database [Chang and Katz, 1989; Chang, 1989; Cheng and Hurson, 1991b]. A user indicates the object access, e.g. access via configuration relationship or version history.

3.1.5 Other Classifications

[Vitter and Krishnan, 1991] distinguish prediction-based prefetching techniques according to the training time: **On-line** techniques make prefetch decisions based on the past history and **off-line** techniques use the perfect knowledge of the future access. [Staehli and Walpole, 1993] classify prefetching techniques for multimedia-applications as follows: **periodic**, **scripted** and **probabilistic**. Periodic repeatable events (e.g. video-playback and movie sequences) are the easiest candidates for prediction. In scripted-prefetching, the pattern of multimedia applications is known. It maintains a list of events with associated request times. All other types of access pattern fall into the last category.

[Kroeger et al., 1997] categorise prefetching techniques for the World Wide Web into two categories, **local-hint** and **server-hint**, based on where the information for determining which objects to prefetch is generated. In the local-hint prefetching technique, the agent doing the prefetching (e.g. a browser-client or a proxy) uses local information (e.g. reference patterns) to determine which objects to prefetch. In server-hint based prefetching, the server is able to use its content specific knowledge of the objects requested, as well as the reference patterns from a far greater number of clients to determine which objects should be prefetched.

3.2 Clustering Techniques

This section discusses the inter-dependency of prefetching and clustering. In Section 3.2.1 we consider this relationship in detail and in Section 3.2.2 we look at previous evaluations of prefetching techniques under clustering.

3.2.1 Combined Clustering and Prefetching

The different techniques for clustering are mentioned in Section 2.5. This section illustrates the relationship between clustering and prefetching:

1. **Quality of Clustering.** If the clustering of objects onto pages is very good then prefetching is limited to prefetch only pointers over the page boundaries. The high-probability pointers between objects would be clustered together on a page and pointers with low probabilities would cross the page border. Prefetching the low-probability pointer could often result in an incorrect prefetch and would not be very efficient. If the quality of clustering is low then prefetching has a greater success in saving elapsed times.

2. **Different Access Pattern.** Applications access the database according to different patterns but clustering can be only performed according to one

access pattern. Therefore clustering is a compromise of access patterns between the needs of several applications. Individual clients can exploit the navigation through the database differently and clustering can only be done according to an average access pattern from all the clients or only according to one client.

3. **Clustering Granularity.** Objects can be clustered onto pages or segments. If all the objects of a database are stored in one segment and the segment fits into memory then the whole segment is loaded into memory on an object fault. Clustering objects into a segment can reduce the total number of faulting pointer traversals since a larger proportion of pointers point to objects within the same fetch unit. However, prefetching segments can be less effective because the prefetch of a segment might mean that a larger proportion of unnecessary objects is fetched. Prefetching pages can be easier to implement because it avoids a mismatch between the size of a page and the size of a prefetch unit. No research so far has compared clustering granularities for prefetching.

3.2.2 Evaluation of Prefetching under Clustering

In the literature prefetching techniques were evaluated under different clustering algorithms. [Chang and Katz, 1989; Chang, 1989] proposed a clustering technique which considers I/O operations to perform clustering. Since the information used by the clustering algorithm is on an instance basis, the clustering algorithm needs to retrieve the physical page in order to get the corresponding information for the clustering decision. When looking for a candidate page for placement, the clustering algorithm may use only the pages available in the buffer pool, avoiding any I/O. Or, the algorithm may search a limited number of pages on disk. Alternatively, if the number of I/O requests is unbounded, then the algorithm may use the entire database as a candidate page pool.

[Ahn and Kim, 1997] used two alternative clustering factors, 90%-1% and 80%-5%, from the OO1 benchmark to evaluate the goodness of clustering. [Cheng and Hurson, 1991a] proposed a levelled clustering scheme under prefetching in which objects connected by one specific relationship are tightly (primarily) clustered while objects connected by other relationships are loosely (secondarily) clustered. [Gerlhof and Kemper, 1994a] tested their prefetching technique under the greedy graph partitioning algorithm [Gerlhof et al., 1993] and random clustering technique. [Keller et al., 1991] also clustered data randomly, or according to types or according to the structure of a component object.

3.3 Buffer Management

This section discusses the interaction between prefetching and buffer management. The replacement strategy and the size of the buffer pool for prefetching are the most important components. In the past, replacement strategies were a major research area [Effelsberg and Härder, 1984; Robinson and Devarakonda, 1990; Jauhari et al., 1990; Chan et al., 1992; O'Neil et al., 1993; Johnson and Shasha, 1994; Weikum et al., 1994]. The problem that prefetched data will be already replaced again on access was also considered [Patterson et al., 1993; Tullsen and Eggers, 1993; Lee et al., 1994]. This problem is even bigger in microprocessors because of the small on-chip cache.

An unsolved problem is the prefetching quantity, i.e. how many objects or pages should be requested by prefetching. The optimal quantity changes at run-time and its size must be adapted to the buffer management strategy. For prefetch group requests, the timing of the buffer allocation is another factor. The key question is whether the allocation should take place before or after the request to the server. A discussion on the interaction of prefetch quantity and allocation time can be found in [Gerlhof, 1996].

In Section 3.3.1 we discuss replacement decisions in buffer management. A description of an integrated prefetching and caching is given in Section 3.3.2. Issues like buffer frame allocation and allocation time are illustrated in Section 3.3.3.

3.3.1 Buffer Replacement Strategies

Although it would make sense to treat prefetched pages and demand pages differently few researchers have made this distinction. Both types of requests shared the same buffer pool with the **LRU-replacement** strategy [Palmer and Zdonik, 1990; Fu and Patel, 1991; Palmer and Zdonik, 1991; Grimsrud et al., 1993; Pai and Varman, 1992; Curewitz et al., 1993]. Independent of the type of request, all new pages get the highest priority. The same is true for pages that are predicted but already resident [Horspool and Huberman, 1987; Wilson et al., 1994]. [Chang and Katz, 1989; Chang, 1989] used two priority levels: *high* and *low*. It is unclear in this work when high priority pages become low priority pages. The authors distinguish between two types of prefetching: *prefetching within buffer* and *prefetching within database*. The former type only increases the priority of pages that are already resident in the buffer and the latter performs requests to the server.

Prefetched pages could also be assigned a higher ageing policy than demand pages [Horspool and Huberman, 1987]. A speed-up factor k, which determines the ageing factor, had only a low significance to their strategy towards elapsed time. In a big buffer the number of page faults were reduced slightly but increased slightly with a small buffer.

3.3.1.1 Inclusion Property

[Horspool and Huberman, 1987] developed a prefetching technique with a so-called **inclusion-property**. The inclusion-property means that the replacement strategy is independent of the buffer pool size. Let $MEM(g(s), t)$ be the set of resident pages at time t with replacement strategy g. Now, all paging policies have a control parameter of some kind; e.g. the number of frames of main memory that are allocated by the operating system. g(x) means the replacement policy g operating with control parameter x. The memory inclusion property can now be stated as:

$$\forall : x \leq y \rightarrow MEM(g(x), t) \subseteq MEM(g(y), t)$$

Page fault rate anomalies cannot occur with any replacement policy that possesses the memory inclusion property. That is, if a fault occurs at time t when using control parameter value x, the fault would have occured for any smaller value of the control parameter too. Conversely, if no fault occurs for value x, there could not be a fault for any larger values of the control parameter. Some simple prefetching techniques which, on every reference load the next page into the buffer pool, e.g. OBL [Joseph, 1970; Smith, 1978b], possess the inclusion-property. For clarification consider the following situation in a LRU buffer:

On reference to page P_3 a buffer miss occurs only in a buffer pool with less than four pages. This means that the demand prefetching technique is dependent on the buffer size and does not have the inclusion-property. Horspool

Page	P_1	P_7	P_2	P_3	P_6	P_5	P_4
Position	1	2	3	4	5	6	7

Table 3.1: Example LRU buffer.

and Huberman developed then a demand prefetching technique that suppresses prefetches if they do not occur in all buffer sizes. The physically adjacent page on disk is only loaded if it is in the LRU queue before the demand page. In our example the reference of page P_3 would not produce a prefetch of page P_4 but the reference of page P_5 would generate of request for P_6. [Wilson et al., 1994] criticised this algorithm on the grounds that it allows only very simple demand prefetching techniques and suggested a modification of the algorithm for more complex techniques.

3.3.2 Integrated Prefetching and Caching

The first integrated study of the interaction of prefetching and caching strategies was performed by [Cao et al., 1995b; Cao et al., 1995a; Cao et al., 1996; Cao, 1996]. Although prefetching and caching have been studied extensively, most studies on prefetching have been conducted in the absence of caching or for a fixed caching strategy. Cao et al. argued that the main complication is that prefetching blocks into a cache can be harmful, even if the blocks will be accessed in the near future. This is because a cache block could be reserved for the block being prefetched at the time the prefetch is initiated (see Section 3.3.3.2)[6]. The reservation of a cache block requires performing a cache block replacement earlier than it would otherwise have been done. Making the decision earlier may hurt performance because new and possibly better replacement opportunities open up as the program proceeds.

An Example: Consider a program that references blocks according to the pattern "ABCA". Assume that the cache holds two blocks, that each reference takes one time unit, that fetching a block takes four time units, and that blocks A and B are initially in the cache.

The top half of Figure 3.2 shows a no-prefetching policy using the optimal replacement algorithm. The first two references hit in the cache. The third reference (to C) misses the cache, thus triggers a fetch of C, replacing B at time

[6] This is not true in general because a buffer could be reserved on receipt of a prefetched page.

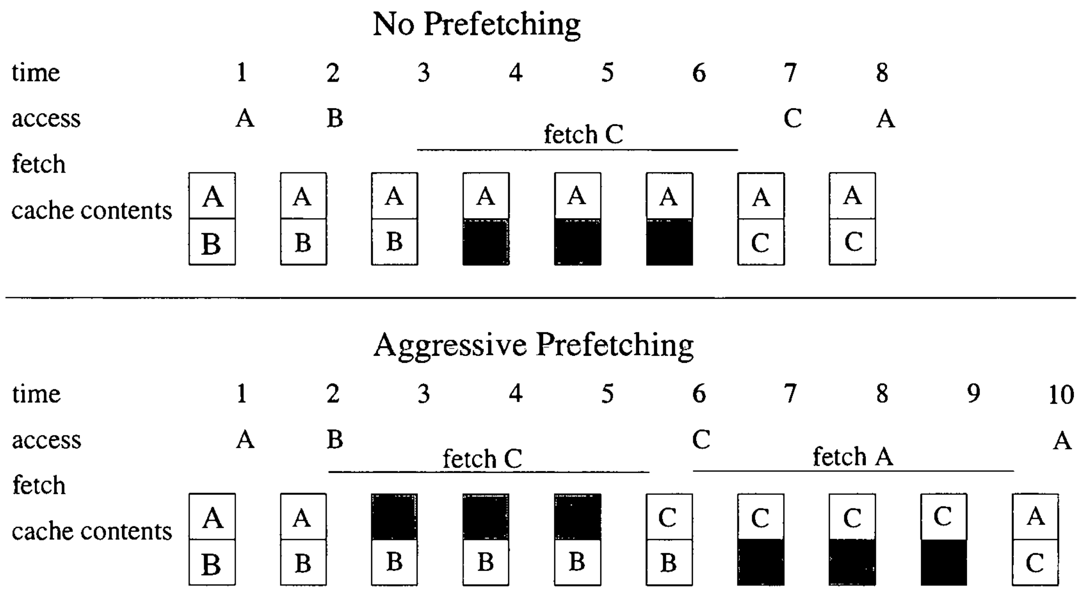


Figure 3.2: Effect of access streams on buffer replacement. A file access stream for which prefetching hurts performance. Eight time units are required in the absence of prefetching; ten time units are required when prefetching is done; optimal replacement is assumed in both cases.

3. Finally, the fourth reference hits in the cache. The execution time of the no-prefetch policy would therefore be eight time units (one for each of the four references, plus four units for the miss).

By contrast, the bottom half of Figure 3.2 shows that a policy that prefetches whenever possible (while making optimal replacement choices) takes *ten* time units to execute this sequence. The policy decides to prefetch C at time 2, resulting in the replacement of A because B is in use at the time; thus the fourth reference (to A) misses in the cache.

This example illustrates that aggressive prefetching is not always beneficial. The no-prefetch policy fetched one block, while the aggressive prefetching algorithm fetched two. The price of performing an extra fetch outweighs the latency-hiding benefit of prefetching in this case. On the other hand, prefetching might have been beneficial under slightly different circumstances. If the reference stream had been "ABCB" instead of "ABCA", then aggressive prefetching would have outperformed the no-prefetch policy. Thus we see that aggressive prefetching is a double-edged sword: it hides fetch latency but it may increase the number of fetches.

Cao et al. defined four rules that an optimal prefetching and caching strategy must follow:

1. **Optimal Prefetching.** *Every prefetch should bring into the cache the next block in the reference stream that is not in the cache.*

2. **Optimal Replacement.** *Every prefetch should discard the block whose next reference is furthest in the future.*

3. **Do Not Harm.** *Never discard block A to prefetch block B when A will be referenced before B.*

4. **First Opportunity.** *Never perform a prefetch-and-replace operation when the same operations (fetching the same block and replacing the same block) could have been performed previously.*

These rules are mandatory, in the sense that any algorithm can easily be transformed into another algorithm, with performance at least as good, that follows the rules. Cao et al. assume the knowledge of a complete reference string of blocks. In practice, this knowledge is in most cases not available which makes it difficult to follow the rules. This study was extended by a polynomial time algorithm for an optimal prefetching/caching on a single disk [Albers et al., 1996].

The study of integrated prefetching was recently extended to a network-wide global memory system [Voelker et al., 1998]. Co-operative prefetching permits multiple network nodes with idle CPU cycles and memory pages to co-operate in prefetching on behalf of active nodes. This prefetching to global memory can reduce stall time without the risks of aggressive prefetching on the active nodes. The prefetching algorithm is a hybrid that combines aggressive prefetching into global memory with more conservative prefetching into local memory.

3.3.3 Buffer Allocation

Buffer allocation can be distinguished in the number of frames that we allocate for prefetching and the time when we allocate frames.

3.3.3.1 Buffer Frame Allocation

Prefetched pages can be placed in the demand buffer pool or in a separate buffer pool for prefetching. The advantage of two separate buffer pools is:

1. A prefetch never replaces a valuable demand page.
2. Prefetched pages that are never referenced by the program are easier to manage.
3. Even if a prefetched page is referenced by the program it cannot replace a demand page too early.

Separate buffers for demand fetches and prefetches were introduced for the first time in the IBM DB2 system [Teng and Gumaer, 1984]. DB2 manages two buffer pools for two types of access pattern: all pages come at first into the buffer pool for sequential access. If a page is referenced again it is moved into the buffer pool for multiple accessed pages.[7] This arrangement was made to avoid problems with both access patterns in one buffer pool [Sacco and Schkolnick, 1986]. Due to the fact that DB2 prefetches only sequential access pages the different treatment of demand and prefetch pages is an artefact of the DB2 buffer philosophy.

The work of [Freedman and DeWitt, 1995] also separated demand and prefetched pages in two buffer pools. In contrast to [Teng and Gumaer, 1984] they gave repeatedly accessed pages a lower priority. The reason was that video access pattern are mostly sequential and are low in locality.

In the area of operating systems [Cao et al., 1996] used a **two-level cache management strategy**. The kernel decides how many cache blocks each process may use. Each process decides how to use its own blocks for caching and prefetching.

3.3.3.2 Buffer Allocation Time

The time for the buffer allocation can be classified into two possibilities:

- **Binding load.** The client allocates a buffer before the request to the server and fixes it.
- **Non-binding load.** On receipt of the data the client checks the buffer pool. The advantage is that until receipt no data will be replaced that

[7] This is similar to the 2Q-buffer [Johnson and Shasha, 1994] which is in turn a simple implementation of the LRU-2-buffer [O'Neil et al., 1993] and similar to the W^2R algorithm [Jeon and Noh, 1997; Jeon and Noh, 1998].

could be still in use. The disadvantage is that if many pages are fixed in the buffer pool the client cannot allocate enough space.

In the microprocessor area the distinction between binding load and non-binding load is even more significant. In the case of the binding load the target address (in most cases in a register) must be known [Farkas et al., 1995] which is not the case with a non-bind load. The idea of the non-bind load is only to bring the data closer to the microprocessor cache [Chen and Baer, 1992]. In the database area, the IBM DB2 system [Teng and Gumaer, 1984] is one of the few known systems that reserves buffer space before the request.

Overruns. Smith [Smith, 1982] reported that prefetching can produce so-called overruns if the client is not able to service the incoming data. The consequence is that the client has to request the data again. Buffer replacement is an expensive operation in databases. At the time when the client is looking for a free buffer this overrun effect can happen. Data packets have to be dropped because the client is otherwise busy with the buffer management. Nowadays thread technology could alleviate this problem.

3.4 Client/Server Architecture

The efficient implementation of a prefetching technique is important for improving performance. One consideration is the prefetch unit of I/O between client and server, see Section 3.4.1. In Section 3.4.2 we discuss implementation issues of prefetching in multithreaded systems. The location of the prefetch engine, at the client or server, is explained in Section 3.4.3. Aggressive prefetching can also have negative effects on performance. These issues are mentioned in Section 3.4.4.

3.4.1 Prefetch Granularity

A further classification characteristic for prefetching techniques is the unit of prefetching. The oldest prefetching technique are from the area of operating systems and are based on pages [Joseph, 1970; Baier and Sager, 1976]. A cost model for the optimal prefetching quantity was developed [Smith, 1978b]. The model is based on sequential access and uses the probability values to estimate the access of the next k pages. An object-based prediction technique can be adopted to load either segments or a group of objects from the server [Palmer and Zdonik, 1991]. The disadvantage of an object-based prediction technique is that it involves a very high cost of predicting object access.

Many techniques restrict the quantity on a system constant, e.g. a page size [Teng and Gumaer, 1984; Fu and Patel, 1991; Palmer and Zdonik, 1990; Palmer and Zdonik, 1991; Pai and Varman, 1992; Pai and Varman, 1992; Curewitz et al., 1993; Mohan et al., 1993; Wu et al., 1994]. In [Bianchini and LeBlanc, 1994] different prefetch quantities were investigated through profiling and compiled into the application. For **news-on-demand** applications a special prefetching technique was developed to increase the throughput by using unused buffer space. Other systems limit the number of concurrent pending requests through a system constant [Klaiber and Levy, 1991; Mowry et al., 1992; Chen and Baer, 1994; Patterson et al., 1995].

For the reason of completeness we also want to mention that there exists prefetching techniques for files [Griffioen and Appleton, 1993; Griffioen and Appleton, 1994; Griffioen and Appleton, 1995a; Griffioen and Appleton, 1995b; Patterson and Gibson, 1994; Cortes et al., 1997] and multimedia objects [Staehli and Walpole, 1993; Ng and Yang, 1994; Rubine et al., 1994; Chaudhuri et al., 1995; Jeong et al., 1997; Gollapudi and Zhang, 1998] but they are not relevant for the database area. File systems save data on blocks which are similar to database pages and most of the work in file systems concentrates on inter-block access patterns. In constrast, we are interested in obtaining inter-page access patterns by analysing object relationships.

3.4.2 Prefetching and Multithreading

The first implementation of a multithreaded architecture in a client/server database system was conducted by [Gerlhof and Kemper, 1994a] in the MERLIN system. The implementation was a motivation for our implementation of a multithreaded prefetching architecture into EXODUS. In the MERLIN system each client creates three threads: one for application processing, one for prefetching and a third for receiving pages from the server. The server creates two threads for each client: one for demand requests and one for prefetch requests. It seems that the multithreaded software was only evaluated on a uniprocessor; a multi-processor platform would even ensure higher potential in the reduction of elapsed times.

Combined multithreading and prefetching was also the subject of a study in the area of microprocessors [Lim and Bianchini, 1996]. The implementation on the MIT Alewife Machine showed that only a few of the applications can benefit significantly from multithreading (up to 10% improvement), while some but not all applications can benefit from prefetching (20-50% improvement). The main reason behind this is the relatively short remote cache miss latencies (< 150 cycles). With short latencies, prefetching has an advantage over multithreading because a context-switch usually consumes more processor cycles than a prefetch instruction.

Another study evaluated prefetching and multithreading for a bus-based shared memory multiprocessor [Moreno et al., 1997]. The result of the tests was that sequential-prefetching can reduce the influence of medium latencies. The traffic on the bus is the key bottleneck and limits the effect of prefetching when access probabilities are close to 0.5 percent. Multithreaded architectures with 2 to 8 contexts[8] have better processor utilisation than single-threaded architectures with the same overall number of contexts. This increase in the processor utilisation can be exploited to speed up parallel applications.

3.4.3 Location of the Prediction Engine

The most important information that is exchanged between application and prediction engine is the actual navigation context, i.e. on which page or object the application is currently working. This information exchange should not be expensive because it is needed frequently. That is why the location of the prediction engine is important for the success of prefetching. In the following sections we discuss the advantages and disadvantages of having the prediction engine at the client or at the server. Please note that this discussion is only important for the prediction-based prefetching techniques.

3.4.3.1 Prediction Engine at the Client

If the prediction engine is at the client the request for prediction information is faster and more efficient than requesting this information from the server. Another big advantage is that the prediction component is able to check whether objects or pages are resident in the local buffer pool. In this case the prediction

[8] A context contains information about the page tables of a process.

engine can make prefetches that are worthwhile. In addition, the communication overhead and buffer management will be reduced and the server load decreases as well.

The problem of training-based prediction techniques at the client is that the loading of the prediction information is expensive. Either the prediction information is built into the application program (at compile time) or the client requests these pages from the server. The last approach is more efficient in memory management but involves high run-time costs. In parallel database applications it was discovered [DeWitt, David; Gray, Jim, 1992] that long start and initial set-up costs for achieving parallelism consumed a major part of the application elapsed time. This is probably also true for the prediction engine at the client side.

In [Palmer and Zdonik, 1990; Palmer and Zdonik, 1991] a training-based object-prediction engine at the client side was used; in [Curewitz et al., 1993] a similar prediction engine was developed, this time page-based, in which they concluded that in general the prediction engine cannot be build up to its full size because of memory restrictions. In our architecture a client side prediction engine is also employed.

3.4.3.2 Prediction Engine at the Server

At the first glance it seems to be a good idea to have the prediction engine at the server because that is where the database pages are stored and the prediction information that is associated with them. Unfortunately there are some new problems that did not occur when the information was located at the client:

- The prediction engine has no access to the buffer pool of the client which makes the prediction more difficult. The server does not recognise when the client stalls for an object. To make the prediction work successfully the server has to keep track of all the objects or pages that are in the client buffer pool and the client has to update the server when these items of information are changed. For example this is a necessary assumption in buffer-coherence protocols with a **call-back mechanism**, which caches locks after the transaction end [Franklin et al., 1993]. This information can be piggy-backed with the data transfer.

- The client does not know the currently prefetched data from the server. Therefore on a cache miss the client has to send a request to the server which could be already on its way.

- The server is already the bottleneck of the system and prefetching would increase the server workload.

In the Thor database [Liskov et al., 1996; Day, 1995] a prediction engine was implemented at the server. Various prefetching techniques were tested on the transitive closure of an object graph to the client. The transfer to the client was always a group of objects. [Gerlhof, 1996] also implemented a server-side prediction engine.

Rather than speculatively serving document to clients, servers could assist clients in prefetching decisions [Bestavros, 1996]. In particular, servers could attach to each document they serve a list of document identifiers that are highly likely to be accessed in the near future, leaving it to clients what to prefetch.

Consumer-oriented (client-side) prefetching and producer-oriented (server-side) prefetching has also been studied in the area of shared-memory multiprocessors [Ohara, 1996]. The simulation result showed that the qualitative advantage of producer-oriented prefetching can yield a slight performance advantage when the cache size and the memory latency are very large. Overall, however, deliver turns out to be less effective than prefetch for two reasons. First, prefetch benefits from a **filtering effect**[9] and thus generates less traffic than deliver. Second, deliver suffers more from cache interference than prefetch.

3.4.4 System Workload Considerations

Incorrect prefetches have a negative effect on the network bandwidth [Wang and Crowcroft, 1996]. There is a general trade-off between bandwidth and latency. As we reduce the threshold for statistical prefetching, the latency may improve but at the price of increased bandwidth consumption. A study on FTP shows that the latency can be reduced by 67% with a 7-fold increase in bandwidth [Touch and Farber, 1994].

Prefetching also affects the queuing behaviour of the network [Crovella and Barford, 1997]. Even when prefetching adds no useless traffic to the network, it can have serious performance effects. This occurs because prefetching changes

[9] Prefetch only cache lines that are not in the cache.

the pattern of demands that the application places on the network, leading to increased variability in the demands placed by individual sources and the network traffic as a whole. Increases in traffic variability directly results in increased average packet delays due to queuing effects.

3.5 Other Issues

This section discusses other issues that do not fit in the previous categories. The efficient scheduling of disk requests is the subject of Section 3.5.1. In Section 3.5.2 we briefly mention the prefetch from multiple disks or multiple processors in parallel. Prefetching multiple levels in memory hierarchy is explained in Section 3.5.3 and finally Section 3.5.4 lists some performance metrics to measure the success of prefetching.

3.5.1 Disk Scheduling

Because of the physical attributes of disks, careful scheduling of disk accesses can provide significant improvement in performance [Seltzer et al., 1990]. Without prefetching, scheduling opportunities only come from asynchronous I/O activities or multiple processes. Prefetching provides new opportunities for disk scheduling because prefetch requests can be generated in group. A simple heuristic, *limited batch scheduling*, considers the workload of disks [Cao et al., 1996]. Every time the disk becomes idle, the prefetcher tries to issue a batch of prefetch requests, instead of just one request. There is a batch size limit on B to ensure that the prefetcher will not issue more than B requests. These requests are issued to the disk driver, which then sorts them and all other requests into order of increasing logical block number, so that disk fetches are performed in sorted order.

In addition to sorting disk requests, throughput can be increased even more when page requests are in **exactly sequential order** (e.g. 1,2,3) [Tan et al., 1995]. A test on the SP2 showed that when requests are read from disk the exact order in which they arrive (**roughly sequential order**), the filesystem and disk throughput is about 1 MB/s. However, measurements also showed that an SP2 filesystem and disk are capable of delivering about 3.5 MB/s when the access is in exactly sequential order. Therefore the prefetching technique fetches disk pages always in exactly sequential order and also fetches unnecessary pages to maintain this order. For example, the last fetched page from the disk was p_i. When the next request for page arrives (page p_j) then the server reads pages p_{i+1} through p_{j+k} (k is some arbitrary constant) into the buffer.

In the database literature some authors recommend giving prefetch operations low priority [Palmer and Zdonik, 1990; Palmer and Zdonik, 1991; Patterson et al., 1993; Curewitz et al., 1993; Datta et al., 1995] and similar in microprocessors [Chen and Baer, 1992; Rogers and Li, 1992]. Some authors even suggested to start prefetches only when the disk is idle, i.e. not serving a demand page request [Datta et al., 1995; Cao, 1996; Kimbrel, 1997]. Implementing priorities into queues induces a higher CPU overhead but it is small cost in comparison to the disk retrieval time.

3.5.2 Parallel Prefetching

The effectiveness of caching and prefetching in the parallel environment in MIMD multiprocessors has been studied in [Kotz and Ellis, 1990; Kotz and Ellis, 1991]. For the efficient use of multiple disks there are theoretical [Varman and Verma, 1996; Barve et al., 1997] and practical [Lee et al., 1997; Kallahalla and Varman, 1998] studies on parallel disk prefetching. The special problem of improving the performance of external merging in a parallel I/O system using read-ahead prefetching and disk scheduling was studied by Lee and Varman [Lee and Varman, 1995b; Lee and Varman, 1995a].

3.5.3 Memory Hierarchy Prefetching

Prefetching techniques in databases can be basically applied to two levels in the client/server architecture:

1. **Disk-Server.** The server prefetches disk pages from its local disk into memory to avoid the most expensive cost: seek time and rotational latency [Ghandeharizadeh et al., 1991].

2. **Disk-Client.** The prefetched page is inserted into the server's memory and client's memory. This type of prefetch has the highest savings in elapsed time because it masks disk and network access, and it used in most systems.

3. **Server-Client.** The client prefetches only pages that are resident at the server's buffer pool. The advantage of this approach is that prefetch requests do not block demand requests at the disk but the full potential of latency reduction cannot be achieved.

To improve CPU performance some researchers suggested to prefetch into a disk cache. A disk cache is part of a computer system's memory hierarchy between the disk device and the CPU of the computer. Disk caches typically have 5%-30% miss ratios and prefetching into the disk cache can reduce the miss ratio [Smith, 1985; Grimsrud et al., 1993; Zivkov and Smith, 1996].

Proxy servers are a conduit between a world wide web browser and the internet. A proxy server is usually installed beside a firewall gateway on a border network segment where an enterprise network and the internet are connected. Each client sends its HTTP request to the proxy server instead of sending it directly to the servers. To avoid request stall times, data could be forwarded by a **pre-push technique** from the proxy to the client browser [Fleming et al., 1997; Jacobsen and Cao, 1998].

An integrated approach was proposed by Kraiss and Weikum [Kraiss and Weikum, 1998] to the vertical data migration between the tertiary, secondary and primary storage. This approach reconciles speculative prefetching to mask the high latency of the tertiary storage, with the replacement policy of the document caches at the secondary and primary storage level. It also considers the interaction of these policies with the tertiary and secondary storage request scheduling. Tertiary storage provides huge and cheap storage capacities but the transfer rate and the robot arm of the storage library are potential bottlenecks. The transfer rate is fairly limited and so prefetching can lead to substantial queuing delays in serving other pending document requests. Also, the robot arm incurs a high latency in every volume exchange.

3.5.4 Performance Metrics

The obvious aim of a prefetching technique is the reduction of latency but there are several important performance metrics:

1. **Latency reduction.** In most cases, latency reduction is expressed as the amount of savings of the prefetch application in proportion to the demand application. Latency reduction can be further classified into:

   - **Hidden latency.** The user-visible latency that is avoided because the page has been prefetched completely before access.
   - **Reduced latency.** At the time of a page fault the page is currently prefetched and the client stalls for a period that is less than the whole page fetch time.

2. **Number of page faults.** The accuracy of a prefetching technique could also be measured by the number of page faults at the client. The problem of this metric is that it does not consider reduced latency, which is important for object-oriented databases.

3. **Wasted system resources.** Idle times of the client and server CPU, the network and the disk could be used for prefetching. This metric measures the wasted system time in percent.

4. **Time and space consumptions.** The amount of CPU consumption for predicting and prefetching pages. The space consumption considers the size of prediction information on disk.

3.6 Summary

Prefetching techniques have been studied extensively in many areas of computer science. Even in OODBMSs, prefetching has been the subject of some studies. Most of these concentrated on showing the benefits of prefetching. In this thesis we want to give a better understanding under which circumstances prefetching is more and less successful.

A major difference to previous work in OODBMS prefetching is that our work considers the probability of navigation from one object to another object. We study prefetching under different object relationship structures with high and low object relationship probabilities. By doing so we get an idea which access patterns are most suitable for prefetching.

In the next research chapter we describe our implementation into the EXODUS storage manager.

Chapter 4

Object Structure-Based Prefetching

4.1 Introduction

A serious evaluation of a prefetching technique requires a real client/server system environment. For that reason we decided at the beginning of our research to implement a prefetching environment into an existing storage manager. The requirements for the selection of a storage manager were:

1. Client/Server Architecture;
2. Source Code Availability;
3. Widespread Use of the System;
4. Support for Sun C++ and Solaris.

All these requirements were met by ESM. We implemented a prefetching environment into ESM by using Solaris threads. The design of the prefetching client and the interaction with the server is described in Section 4.2. We also implemented a simple prefetching technique into ESM which is explained in Section 4.3. The basic idea is to prefetch all the objects to a depth which is influenced by the current client navigation. The depth is determined by the page fetch latency divided by the time of object processing. The technique also considers the branch of object relationships and the frequency of non-resident objects in the transitive closure. We created two benchmarks to study the general benefit of a prefetch and our proposed technique. All the results are presented in Section 4.4. Finally, in Section 4.5 we summarise our experience from the ESM implementation.

4.2 Prefetching Architecture

In this section we describe our implementation of a prefetching environment into the ESM client. At the beginning, in Section 4.2.1, we mention the goals for the implementation. The changes to the ESM client are explained in Section 4.2.2 and to the ESM server in Section 4.2.3. In Section 4.2.4 we discuss other detailed implementation issues.

4.2.1 Implementation Goals

To achieve the main goal of a prefetching technique, i.e. saving elapsed time, an efficient implementation is essential. As a consequence, we defined the following design goals for the ESM implementation:

1. **Minimal Synchronisation Cost.** Access to global data is protected by mutexes. Mutexes allow only one thread at a time to access global data; other threads have to wait until the mutex is released. If the global datum is frequently used then the waiting time of other threads is increased. Special care must be taken to reduce the waiting time of the application thread because this has a direct effect on the total elapsed time.

2. **Concurrent Thread Execution.** The concurrent execution of threads can be achieved through multiprocessor machines. This is especially important for the CPU-bound threads, e.g. application processing and prediction.

3. **Minimal Prediction Cost.** The amount of storage space and computation time is obviously dependent on the prediction algorithm. For the implementation we have to use the adequate data structures to compromise between time and space.

For the concurrent execution of the application and the prefetch system we used the Solaris thread interface. Multithreading on its own has the following benefits:

1. Increased application throughput and responsiveness;

2. Performance gains from multiprocessing hardware (parallelism);

3. Efficient use of system resources.

We also considered the POSIX thread interface for our implementation but abandoned this concept by reason of its higher implementation overhead.

4.2.2 ESM Client

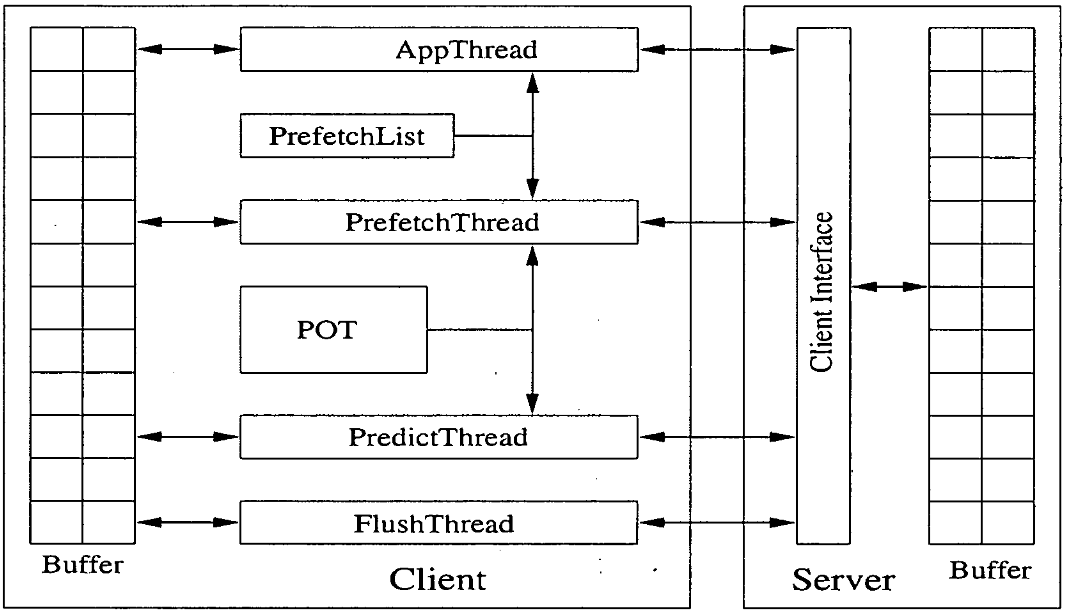


Figure 4.1: Prefetching architecture.

As depicted in Figure 4.1, the database client is multithreaded. The **AppThread** is responsible for the processing of the application program and the **PrefetchThread** is responsible for fetching pages in advance into the buffer pool. There can be one or multiple PrefetchThreads. Each thread has one associated socket-connection to the server. The task of the **PredictThread** is to compute the relevant pages for prefetching. This computation can be done either on-line or off-line. The **FlushThread** pre-flushed dirty pages out of the buffer pool to the server and disk. The **Prefetch Object Table (POT)** informs the PrefetchThread which pages are candidates for prefetching from the current processing of the application. The **PrefetchList** is a list of pages which are currently prefetched.

At the beginning of a transaction the AppThread requests the first page from the server by a demand read. The PrefetchThread always checks which objects the AppThread is processing. Having obtained this information, it consults the POT for a page to prefetch and checks if this page is already resident. If there is no page to be prefetched the PrefetchThread waits on a semaphore, otherwise the page is inserted in the PrefetchList and the request is sent to the server. The PrefetchThread goes into the sleep state until the server responds with the required page and the client inserts the page into its buffer pool. Eventually the page is removed from the PrefetchList and inserted into the hash table of the buffer pool.

When the AppThread requests a new page, it first checks if the page is in the buffer pool. If the page is not resident then it checks the PrefetchList. In case the page has been prefetched the AppThread waits on a semaphore until the page arrives, otherwise it sends a demand request to the server and also inserts the page into the PrefetchList to avoid a double request for a page.

One prefetch thread can request multiple pages by one request but when a new context requires another prefetch then multiple prefetch threads are required. The prefetch threads are allocated to a **prefetch thread pool** in which prefetch threads are either idle or busy. The number of simultaneous prefetch requests determines the number of threads. Most of our prefetch algorithms analyse past behaviour which also gives us information about how many threads are required at each time interval.

Prefetch threads are mostly idle as they await the completion of I/O. This means that several threads can be allocated to a single processor and the threads will not have to wait for an operating system time-slice to complete before they can execute. Each prefetch thread runs on its own LWP[1] and while one prefetch thread blocks on I/O, its LWP gives up the control of the processor and another LWP with its prefetch thread can work on the processor. This concept of LWPs is valuable for prefetching in view of the fact that pending requests never block the processors. The Solaris operating system interface offers a function to bind one LWP to one processor to ensure the parallel execution of threads. However, this facility is not very flexible and did not show any improvements in our performance measurements.

In the asynchronous prefetching architecture of [Gerlhof and Kemper, 1994a] the client has one thread for application processing and two prefetch threads, one for predicting and prefetching pages and one for receiving pages from the server. Presumably this architecture was designed for a uniprocessor machine. The general advantage is that the synchronisation cost between the threads is minimal. On the other hand, the major disadvantage is that if the prediction cost is high it unnecessarily delays the prefetching operation which is contradictory to the aim of a prefetching technique. Another disadvantage is

[1] Lightweight process (LWP) can be thought of as a virtual CPU that is available for executing code.

the neglect of parallelism. In Figure 4.6 we have shown that receiving a page from the network is an expensive operation. If this operation is not done in parallel, it has the following two implications (1) elapsed time could be increased (if applications stall) and (2) the network receive buffer could get exhausted so that incoming data cannot be stored anymore and consequently have to be requested again.

4.2.3 ESM Server

The multithreaded software was not incorporated into the ESM server because of its complexity. Although it is not multithreaded it can run many tasks as concurrent processes on one processor. If one task stalls for I/O another task is scheduled on the processor. The server also forks a new process, the disk manager, for every disk volume. Communication between server and disk manager is achieved by shared memory. The server puts a request for a new page in a disk queue and the disk manager retrieves the page from the volume and copies it into the buffer pool of the server. These interprocess communication costs could be drastically reduced by using lightweight threads. The efficient support of parallelism by threads is another performance improvement and is part of future work. Nowadays most OODBMS servers employ multithreading.

One change that we had to make to the server is the collection of object relationship information. This information is essential for the client to make prefetch decisions. Every time a user updates an object relationship all prefetching clients have to be informed about the change. The content of this information simply comprises the OID with the list of referenced OIDs. The size of an ESM OID is 12 bytes. If we let nop be the number of pointers of an object and nco be the number of updated objects then the occupied space can be computed as follows:

$$\textit{space in bytes} = 12 \cdot (nop + 1) \cdot nco \tag{4.1}$$

This information can be piggybacked to the data transfer to the client.

4.2.4 Implementation Issues

For the parallel execution of threads on the client, synchronisation mechanisms are required. The access to the buffer pool is protected by mutexes, which means that only one thread at a time is able to make a residency check or manipulation. Mutexes are also used for access to the PrefetchList, the hash table for resident pages and some other global variables. When either the AppThread or the PrefetchThread is idle they wait on a semaphore.

The Solaris thread interface provides a function to give threads priorities[2]. The AppThread has the highest priority to make sure that the application processing always gets scheduling priority on one of the CPUs before the prefetch threads. PrefetchThreads have lower priorities but higher than the PredictThread and the FlushThread. The FlushThread has the lowest priority because prefetching guarantees higher savings in elapsed time than flushing. The assumption of this design is that prefetch threads get scheduled on other processors; otherwise the AppThread gets a lower priority than the prefetch threads. On a uniprocessor, a subtler approach to allocating priorities would be needed in order to strike a balance between application processing and prefetching. The AppThread is a CPU-bound thread which runs until an object fault occurs. On the other hand, the PrefetchThread is I/O-bound which means it needs only a short CPU processing time and spends most of its time waiting for the completion of I/O. Now if the AppThread has a high priority on a uniprocessor, the PrefetchThread would be only be scheduled when the AppThread gives up control, i.e. on a object fault. By that time the prefetch would unnecessary. Therefore on a uniprocessor the PrefetchThread gets a higher priority than the AppThread. To avoid the endless running of the AppThread, it has to give up control after a change of context.

4.3 The OSP Prefetch Algorithm

The basic idea of the object structure-based prefetch algorithm **OSP** is explained in Section 4.3.1 and the buffer replacement strategy is described in Section 4.3.2.

[2] Priorities in Solaris are integer values from 0 to 127.

4.3.1 Prefetch Algorithm

4.3.1.1 Description

The general idea of this technique is to prefetch pages well in advance according to the context of the client navigation. The depth of the number of objects to be prefetched is determined by the time of a page fetch divided by the time of processing one object. The PrefetchThread observes the navigation of the AppThread through the object graph and prefetches all objects with non-resident pages ahead. The PrefetchThread operates like a moving window in front of the AppThread.

We obtain the prediction information from the object references without knowledge of the object semantics. Considering the object structure in a page, we identify the objects which have references to other pages (**out-refs**). One page could possibly have many out-refs but sometimes it is not possible to prefetch all pages due to time and resource limitations. Instead, we observe the client navigation through the object net. We define an object that has a reference to an object in another page as an **Out-Ref-Object (ORO)** and the object in the other page as a **Page-Border-Object (PBO)**. We know which objects have out-refs and when we identify that the application is processing towards such an PBO, the out-ref page becomes a candidate for prefetching.

The prefetch starts when the application encounters a so-called **Prefetch Start Object (PSO)**. Although the determination of OROs and PBOs is easy, determining PSOs is slightly more complicated. There are two factors that complicate finding PSOs:

1. Prefetch Object Distance (POD)

   For prefetching a page it is important that the prefetch request arrives at the client before application access to achieve a maximum saving. The POD defines the distance of n objects from the PSO to the PBO object which is necessary to provide enough processing to overlap with prefetching. If there are several paths to an object, then we compute a mean distance from the PSO to the PBO.

   Let C_{pf} denote the cost of a page fetch and let C_{op} denote the cost of object processing, i.e. the client processing time required before the object can be used by the ongoing computation.[3] We distinguish between a page

[3] Also called inter reference time.

fetch from server memory and one from the server's disk. The cost of object processing is the ESM client processing time before the application can work on the object. Additionally we could use the expected amount of processing from the application plus user waiting time[4]. We assume both parameters, C_{pf} and C_{op}, to be constant. In practice, the page fetch costs depends on the server and disk workload which could be taken into account in our model by using different page fetch cost values according to the systems workload. Then POD is computed as follows:

$$POD = \frac{C_{pf}}{C_{op}}$$

If the prefetch starts before the POD, a maximum saving is ensured, however, if it starts after the POD, but before access, some saving can still be achieved (see Section 4.4.3).

2. Branch Objects

   A complex object has references to other objects. The user of the application decides at a higher level the sequence of references with which to navigate through the object net. We define a **branch object** as an object which has at least two references to other objects. Objects that are referenced by a branch object are defined as a **post-branch object**. For example in Figure 4.2 we have a object hierarchy. The object with the OID 1 would be defined as a branch object as it contains a branch in the tree of objects. Objects with OID 2, OID 6 and OID 10 would be defined as post-branch objects because they are the first objects de-referenced by a branch object.

For every identified ORO in the page we compute the PSO by the following reverse algorithm:

1. Identify the referenced PBO by the ORO.

2. Compute the POD to get the distance of n objects from the PSO to the PBO.

3. Determine the PSO by following the object reference n objects in reverse order from the PBO. If there are not enough objects in the page before the

[4] User waiting time means the time the user is not entering new commands, i.e. watching results at the screen.

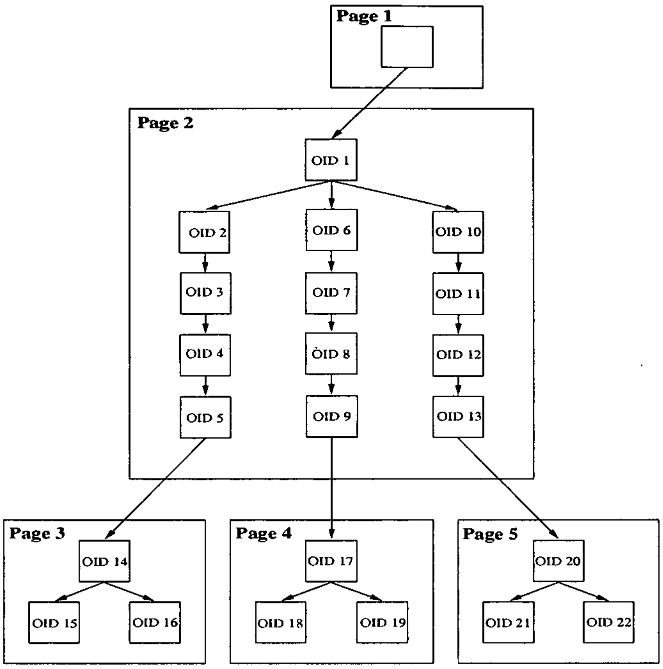


Figure 4.2: Object relationship example.

PBO, then we would identify an object of a previous page in the object graph as a PSO.

4. If the object is already identified as a PSO and the previously identified PSO has different post-branch objects and a different out-ref page then we would identify the post-branch objects of the object as PSOs. This modification makes sense because many object-oriented relationships are organised like tree structures. This step is executed after we have defined all PSOs from the PBOs in a page.

5. If a PSO has multiple pages to prefetch then a page q is assigned a weight according to the number of objects, resident on current page p that have a pointer to page q, in the forward sequence of depth n from the PSO. The pages are then prefetched according to descending weights.

4.3.1.2 An Example Identification Process

Defining post-branch objects as PSOs can improve the accuracy for the prediction and reduces the number of adjacent pages to prefetch. For example in Figure 4.2 we would identify OIDs 5, 9 and 13 as OROs and OIDs 14, 17 and 20 as PBOs. In this example we assume a POD of 5 objects. From OID 14 we would go through the chain backwards by 5 objects and identify OID 1 as a

PSO. Then we would do the same for the OIDs 17 and 20 and identify OID 1 as the PSO for both. After analysing the whole page we would find out that OID 1 has three PSOs with different post-branch objects and different out-ref pages. In this case we would identify the post-branch objects of OID 1 (OID 2, 6 and 10) as PSOs instead of OID 1.

4.3.1.3 Design Issues

The novel idea about our technique is to make prefetching adaptable to the client processing on the object net. Because the cost of a page fetch is high we try to start the prefetch early enough to achieve a high saving but not too early to prefetch inaccurately. In contrast to the work of [Keller et al., 1991], we do not prefetch all references recursively; instead we select the pages to prefetch, dependent on the client processing. Recursive object prefetching also has the problem that prefetched pages can be replaced again before access. Adaptive object prefetching limits the number of prefetch pages to the adjacent pages.

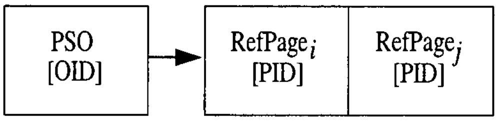


Figure 4.3: One entry in the POT.

Each page of the database is analysed off-line. The Analyser stores this information in the POT for every database root[5]. Figure 4.3 depicts the layout of one entry in the POT. A PSO has one or more associated RefPages to prefetch. Entries for one page are clustered together on disk. The overhead for this table is quite low as it only contains a few objects of the page. At run time, the information from the POT is used to start the prefetch requests. Using the defined PSOs by the reverse algorithm in the forward sequence means that we prefetch all the objects in the time depth of POD_{pf}. Let O be a set of persistent objects. Then we define o_i to be the current object in the navigation process and $o_j..o_{j+n}$ be the set of objects in the depth of POD_{pf} to be prefetched.

We assume that the run time system allocates enough threads for prefetching. If essential pointers for the navigation are updated in a transaction we would invalidate the POT for this page and modify it after the completion of the transaction. This prefetching technique is not only useful for complex objects, it can also be used for collection classes (linked list, bag, set or array) in

[5] This is important because objects on the same page could belong to different roots.

OODBMSs. Applications traverse an object collection with a cursor. With PSO and PBO it would be possible to prefetch the next page from a cursor position. In the description of our technique, the object size is assumed to be smaller than the page size. If the object is larger than a page, prefetching can be used to bring the whole object into memory.

The prediction information can be gathered either by an online-forward or by an offline-reverse algorithm (described above). An online-forward algorithm seeks non-resident objects in the forward direction of the object graph. Both algorithms have advantages and disadvantages. The advantages of the offline-reverse algorithm are:

1. It has no online computation for every object which delays the prefetch unnecessarily and reduces the saving times.
2. If complex relationships exist, we could use a method called *hitting times* (see Section 5.2.3) to compute the mean access time.
3. There is no PredictThread which could interfere and delay the AppThread or PrefetchThread.

The online-forward has the following advantages:

1. Object pointers may change their values dynamically during execution. An online approach is therefore more up-to-date.
2. It consumes no disk space for prefetch information.
3. It can more easily consider variable parameters, e.g. the page fetch time varies according to the workload of the system.
4. If object relationships are complex, identifying objects with the reverse algorithm may be difficult and results in many PSOs.

We opted for the offline-reverse approach in our implementation because of its higher savings in elapsed time. OODBMSs often do not have high processing times on objects which makes an offline approach more pertinent.

4.3.2 Replacement Policy

In the ESM client it is possible to open buffer groups with different replacement policies (LRU and MRU). Freedman and DeWitt [Freedman and DeWitt, 1995] proposed a LRU replacement strategy with one chain for demand reads and one chain for prefetching. We also plan to use two chains with the difference that when a page in the demand chain is moved to the top of the chain, the prefetched pages for this page are also moved to the top. The idea of this algorithm is that when the demand page is accessed, it is likely that the prefetched pages are accessed too. If a page from the prefetch chain is requested it is moved into the demand chain.

4.4 Implementation Results

To understand the results of our implementation we must first explain the system environment in Section 4.4.1 and the benchmark description in Section 4.4.2. Then we present some theoretical results in Section 4.4.3 and the implementation results in Section 4.4.4.

4.4.1 System Environment

For the ESM server we need a machine (called Dual-I[6]) configured with a large quantity of shared memory and enough main memory to hold pages in the buffer pool. To take full advantage of multithreading we chose a four-processor machine (called Quad) for the client. Table 4.1 presents the performance parameters of the machines. Dual-II and Uni are also used as database clients. The network is Ethernet running at 10Mb/s. The disk controller is a Seagate ST15150W and its performance parameters are explained in Table 4.2.

The estimation of the average disk access time is very important for the correct timing of prefetch requests. The seek latency depends on the physical distance that the disk arm has to move and latency of the read depends on the amount of data to be transferred. In an experiment we tried to measure average disk access times. We created a big database and measured the time for a disk seek and a disk read. The disk seek operation was positioned to a list of file offsets and we always read an 8 KB page. All tests were repeated several times.

[6] The names of the machines indicate the number of processors

Unfortunately, we observed very high variations in the measurements which made an approximation of an average value for the seek and read operation impossible. Therefore we used the performance specification of the Seagate ST15150W for our experiments. Using an average value will improve system's performance but in some cases the prefetch will be incorrectly timed, i.e. either it arrives too early or too late.

Parameter	Dual-I	Quad	Dual-II	Uni
SPARCstation	20/612	10/514	20/502	ELC(4/25)
Main Memory	192 MB	224 MB	512 MB	24 MB
Virtual Memory	624 MB	515 MB	491 MB	60 MB
Number of CPUs	2	4	2	1
Cycle speed	60 MHz	50 MHz	50 MHz	33 MHz

Table 4.1: Computer performance specification.

Parameter	Disk controller
External Transfer Rate	9 Mbytes/s
Average Seek (Read/Write)	8 ms
Average Latency	4.17 ms

Table 4.2: Disk controller performance.

4.4.2 Benchmark Description

In all our experiments we used synthetic benchmarks to evaluate our prefetching techniques. Using synthetic benchmarks was especially important for this chapter to get a clear understanding of how much elapsed time can be reduced by prefetching. In contrast, real benchmarks provide a more realistic workload environment but the timing results might be more difficult to understand. We constructed two benchmarks: one simple benchmark to get a first impression about the benefits of prefetching and a more complex benchmark.

In the simple benchmark every object in the data structure has two pointers to other objects. Most of the objects point to another object in the same page; only one object in a page has two pointers to objects that are resident in two other pages, e.g. the current object is in page 1 and it has one pointer to an object in page 2 and one pointer to an object in page 3. Having this object structure, the pages are connected like a tree. The size of one object is 64 bytes which gives space for 101 objects in one 8K page. In one run 200 pages are accessed, i.e. equal to the size of the buffer pool at the client and server. The application reads only one object from the first faulted page and then all objects from the second faulted page. Every object is fetched into memory with no computation or waiting time on the object.

The requirements for the complex benchmark were:

- The application access pattern should be dynamic and different for every run;
- The sizes of the objects should be fairly uniform;
- Object references should be complex;
- The number of pages accessed in one run should be equal to, or less than, the number of pages in the buffer pool at the client and server.

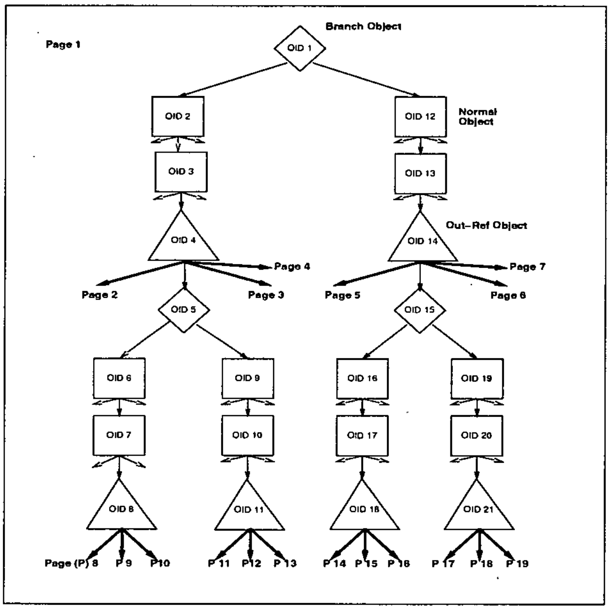


Figure 4.4: Benchmark structure of one page.

In Figure 4.4 we depict the design of one page from the complex benchmark[7]. There are three types of objects: branch objects, OROs and normal objects. A branch object decides by a random operator which object reference to follow in the tree. An ORO has pointers to objects in other pages which are

[7] Every page has the same structure.

all accessed when encountered. A normal object points to three other objects in the same page. The type for all objects has four pointers and a size of 72 bytes. In one run 195 pages are accessed and each page contains 112 objects.

The application starts with one root object from the first page. The branch objects decide the navigation in the page. When a reference to another page from the upper level (e.g. pages 2 to 7) is encountered only the first object from the other page is de-referenced and then the application continues in the current page. At the lower levels (e.g. pages 8 to 19) two pages are de-referenced with 1 object (the same as at the upper level) and in one page the application continues the navigation. Having two or three references to other pages gives us the possibility to test prefetching under strict time conditions. It also means that the program is quite I/O intensive and the savings in percentage terms are potentially high.

Figure 4.4 needs some explanation concerning the number of normal objects. The number of normal objects before a PBO is 15. The cost of processing 20 objects is equal to the cost of one page fetch in our system environment. Every object is fetched into memory with no computation or waiting time on the object which would clearly reduce the prefetch distance.

4.4.3 Theoretical Results

4.4.3.1 Performance Improvements

Given the huge gap between disk access time and main memory access time, it might appear that orders of magnitude improvements in elapsed time might be possible with prefetching. In this section, we explain why only relatively modest improvements are possible in practical situations.

Figure 4.5 shows a multilevel memory hierarchy, including typical sizes and speeds of access. In this thesis we are only interested in avoiding the disk latency and not memory latency because the cache is often too small for database applications. Assume we have a prefetching technique that has the following characteristics:

- Prefetching accuracy is always 100%;
- Every prefetch arrives in memory before access;

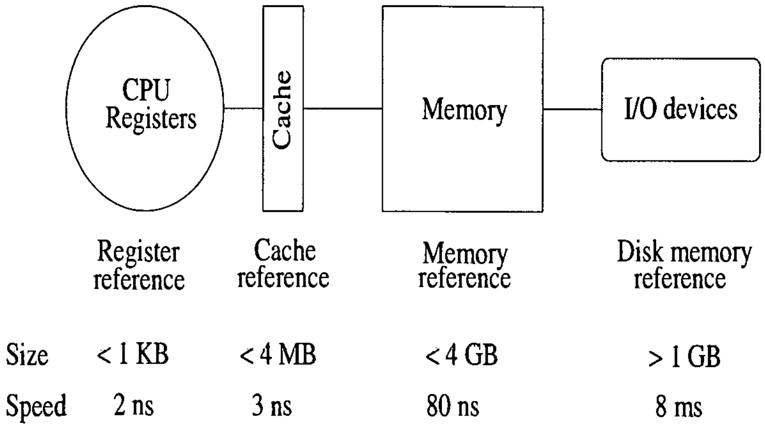


Figure 4.5: Levels in a typical memory hierarchy [Hennessy and Patterson, 1996].

- Memory size is infinite.

This perfect prefetching technique would find all objects resident in memory compared with a demand application that would often stall for disk requests. According to Figure 4.5 memory access is 100,000 times faster than disk access. Would this mean that the prefetching application is also 100,000 times faster than the demand application?

To answer this question we have to consider at first the number of objects accessed in a page. If we would access always one object of a page then a speedup of 100,000 is feasible. On the other hand, if we access multiple objects in each page then we have to use the following formula to compute the speedup in access time:

$$Speedup_{fetchtime} = \frac{A_D + (n-1) \cdot A_M}{n \cdot A_M} \qquad (4.2)$$

where n is the number of accessed objects in each page, A_D is the access cost for disk and A_M the access cost for memory. For our client/server environment we could use the cost of a page fetch instead of A_D. The result of Equation 4.2 is to reduce the speedup factor of 100,000 according to n.

For the overall computation of the speedup of a prefetch application we also have to consider the fraction of the fetch time in comparison to computation time of an application. We therefore apply Amdahl's Law [Amdahl, 1967] to compute the overall speedup factor:

$$Speedup_{overall} = \frac{1}{(1 - Fraction_{fetchtime}) + \frac{Fraction_{fetchtime}}{Speedup_{fetchtime}}} \tag{4.3}$$

$Fraction_{fetchtime}$ is the proportion of elapsed time during which the application is idle, awaiting the arrival of a fetched page.

An Example: Let the number of accessed objects per page (n) be 100. We use the access times for memory and disk from Figure 4.5.

$$Speedup_{fetchtime} = \frac{8ms + (100 - 1) \cdot 80ns}{100 \cdot 80ns} = 1000.99 \tag{4.4}$$

Now the speedup in fetch time is reduced from 100,000 to 1,000. Suppose the fetch time is 50% of the whole application then the overall speedup is:

$$Speedup_{overall} = \frac{1}{(1 - 0.5) + \frac{0.5}{1000.99}} = 1.998 \tag{4.5}$$

Finally, the total speed up of the prefetch application is only about a factor of 2. The percentage of page fetch time is very application dependent. Lower percentages than 50 would reduce the savings potential even further.

4.4.3.2 Savings in Elapsed Times

The success of prefetching is dependent on the accuracy of the prediction and the completion of the prefetch before access. We define the cost of object processing to be C_{op}. Let C_{oe} denote the cost/elapsed time of processing an object by ESM and let C_{oa} denote the cost of processing an object by the application plus waiting time. C_{op} is calculated by:

$$C_{op} = C_{oe} + C_{oa} \tag{4.6}$$

The cost of a page fetch, C_p, is dependent on client and server processing, the network and the disk. C_{cp} denotes the cost of client processing; C_{nt} denotes the cost of network transfer; C_{sp} is the cost of server processing; C_{sq} is the server queuing cost, C_{dr} is cost for the disk retrieval. C_p is then calculated by:

$$C_p = C_{cp} + C_{nt} + C_{sp} + C_{sq} + C_{dr} \tag{4.7}$$

The saving for one out-going reference to a non-resident page S_{or} is dependent on the number of objects between the start of the prefetch and application access to the prefetched object (n) and C_p:

$$S_{or} = \begin{cases} C_p & if(C_{op} \cdot n \geq C_p) \\ C_{op} \cdot n & otherwise \end{cases} \tag{4.8}$$

If there is enough processing to overlap then the saving is the cost of a page fetch. If not, there is also a saving, albeit lower, of the amount of processing from prefetch start to access $(C_{op} \cdot n)$. Pages normally have many out-going references. The number of references to different pages is denoted by p. S_{page}, the saving for a whole page, is given by:

$$S_{page} = \sum_{i=1}^{p} S_{or}(i) \tag{4.9}$$

Finally, the saving of the total run is defined by S_{run} which is influenced by the cost of the thread management (C_t), by the cost of the socket management (C_s) and by the number of pages in the run (q):

$$S_{run} = (\sum_{j=1}^{q} S_p(j)) - C_t - C_s \tag{4.10}$$

4.4.4 Performance Measurements

Although the tests were made in a multi-user environment the workload of the machines, disk and network was low. The results of the benchmark are dependent on the workload of the machines: using busy machines and networks would increase the page fetch latency. Since there were different workloads during the tests, it is not possible to compare the absolute times in multifarious tests. Savings in percent mean the percentage saving of a prefetching version compared with a version without prefetching and multithreading, i.e. Demand version.

In repeated tests, we at first measured the cost of a page fetch and the cost of the ESM client processing for one object. The average result for the page fetch was 11.577 ms and for the ESM processing 604 μs. The page fetch cost does not comprise the expensive disk seek and rotational latency cost since this is conditional on the current position of the disk arm. Most of the ESM client processing is due to an audit function that calculates the slot space of the page.

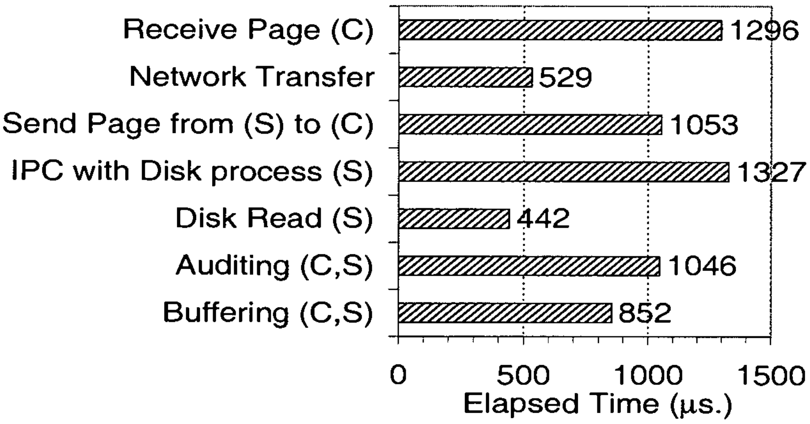


Figure 4.6: Expensive components of a page fetch.

A major cost factor of the page fetch is the cost for sending and receiving a page (setup costs) via the network. The network transfer cost is low, compared with the setup costs. All elapsed times of the cost components (apart from network transfer and disk read) are dependent on the speed of the processor. Thus these costs could be reduced using up-to-date processors and network transfer could be reduced by higher bandwidths in which case the disk access would emerge as the major bottleneck. The seek cost is the most expensive part of the disk access but does not appear in Figure 4.6 because we read all pages sequentially from disk. The IPC cost could be reduced by using a disk thread instead of a disk process.

4.4.4.1 Results of the Simple Benchmark

In Figure 4.7 we present the results of our benchmark. The prefetching version is always faster than the *Demand* version. The best result was made on the slow Uni machine because of its low cycle speed and slower access to the

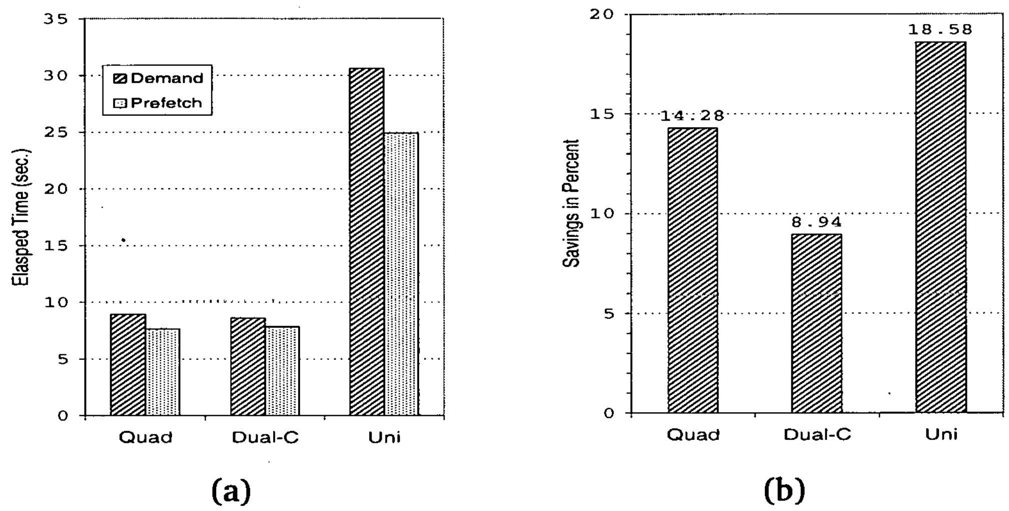


Figure 4.7: The result of the simple benchmark on different machines: (a) shows results in total elapsed time and (b) depicts only the savings of the prefetch versions in percent.

socket. Quad has the same cycle speed as Dual-II but a higher saving by virtue of the larger number of processors. Dual-II and Quad have, in contrast to Uni, two processors or more, allowing threads to run on different processors concurrently. This would be more beneficial with more prefetch requests at the same time. In this test every prefetch is done with 100% accuracy to give an idea of the maximum speedup that can be achieved with prefetching. All the pages in this test are read in sequential order. Prefetching could achieve even higher savings with access pattern that produce higher disk seek times.

As mentioned in Section 4.4.3 the saving of prefetching is dependent on the amount of processing from the application. Having 101 objects on one page, we compared the elapsed-time savings under varying object access rates from the application (from 10 objects to 100 objects accessed). Figure 4.8 shows that the highest saving is with an object access of 20 because the object processing cost is almost equal to the page fetch cost. For the access of 10 objects there is not enough CPU overlap for prefetching to reduce complete page fetch latency. Increasing the number of objects gradually decreases the savings because the application gets more CPU dominated.

The amount of savings at the object access of 20 should be closer to 100% than to 45% but there are two possible reasons why the savings are not higher:

- The prefetch might arrive late because of delays in the client/server architecture.

- We have no control over any caching in the operating system or hardware. For example, data items could be resident in the disk cache.

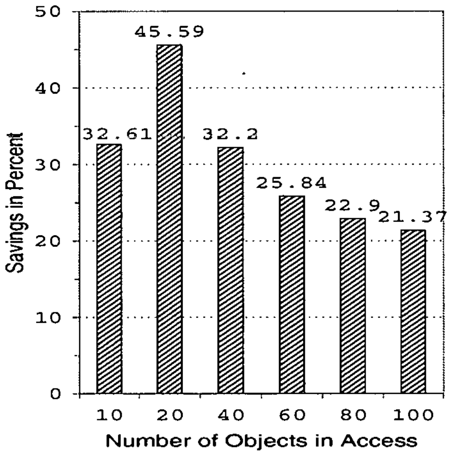


Figure 4.8: Savings of prefetch applications depending on the number of object accesses.

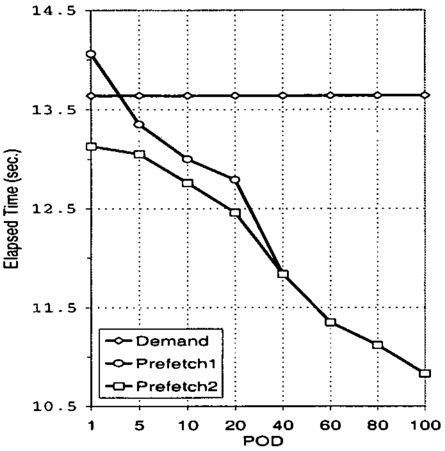


Figure 4.9: Prefetching with multiple threads.

When two pages have to be prefetched under strong time restrictions such that there would only be enough time to prefetch one page successfully, we use multiple prefetch threads simultaneously. This is especially appropriate under fast changing application contexts; otherwise we could request multiple pages by a single message to the server. We compared different prefetch object distance parameters to see under which conditions more prefetch threads are useful. In Figure 4.9 *Prefetch1* means a prefetching version with just one prefetch thread and *Prefetch2* means a version with two prefetch threads. Above the distance of 40, both prefetching versions perform equally well. Then *Prefetch2* can improve performance and, even at a distance of 1, is better than *Demand* (*Prefetch1* is worse than *Demand* at a POD of 1).

The application fetches all objects by OID into memory without any processing on the objects or any waiting time. In addition, a pointer swizzling technique is necessary for real applications to translate the OID into a virtual memory pointer. All this would produce more processing overhead for the client. We simulate this overhead with a loop after every object fetch. The results in Figure 4.10 show that with more processing the savings in percent get smaller. The reason for this outcome is that the application is increasingly dominated by CPU processing and the prefetch engine had enough time for completing the request before access. In other circumstances, when there is not enough overlapping time for prefetching, an expansion of client processing would culminate in a better result for prefetching.

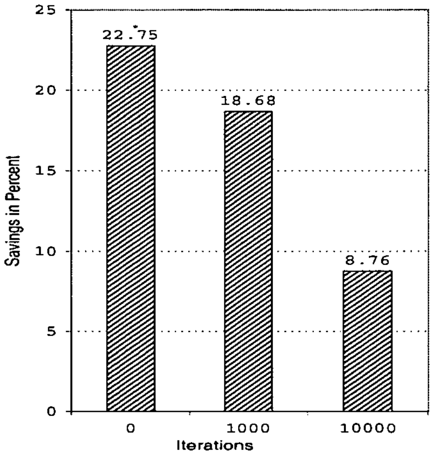


Figure 4.10: Benefits of prefetching with varied client processing.

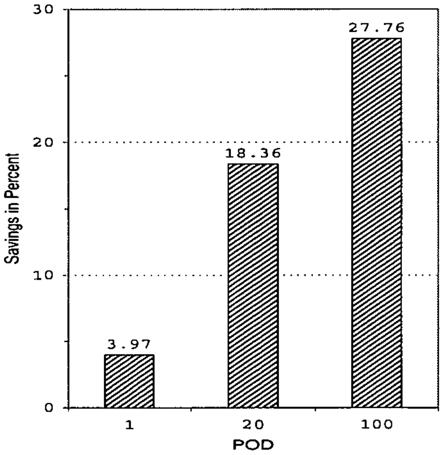


Figure 4.11: Effect of incorrect prefetching in the simple benchmark.

Most previous work in prefetching focussed on the reduction in elapsed times but neglected the impact of incorrect prefetches. We always fetched one incorrect page and one correct page at the same time; 100 each in total. The other important parameter is the prefetch object distance. We used the distances of 1, 20 and 100. The distance of 100 is enough to do an incorrect prefetch, the distance of 20 makes it critical to do one prefetch right on time and with the distance of 1, the prefetch is always late. Figure 4.11 shows the best result of 27 percent savings with a distance of 100, but even with a distance of 1 there is still a saving albeit of only 4 percent. The outcome of this test is also crucial to the scheduling of the requests at the disk. In this test the correct prefetch is scheduled before the incorrect prefetch in most cases.

In a client/server environment with multiple clients, prefetching has benefits and drawbacks. On one hand, the page fetch latency is increased due to the higher workload of all system components which yields in higher savings. On the other hand, prefetch requests can seriously delay demand requests from other clients. This is particularly true for the slow disk. In our test the requests from other clients access different data pages. Figure 4.12 shows that *Demand* decreases performance significantly with 4 clients and the prefetching versions decline with 7 clients. Prefetching does not show any negative effects here as a consequence of the good prediction possibilities and high prefetch distance. The general conclusion of this test result is that every additional client slows down the *Demand* application but as long as the prefetch is started in compliance with the POD_{pf} the prefetch application attains consistently good results.

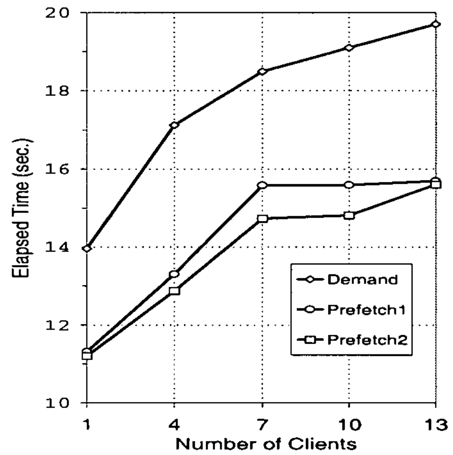


Figure 4.12: Effect of an increased server workload due to additional clients.

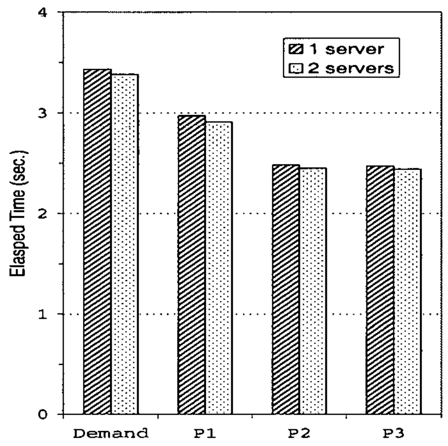


Figure 4.13: Distributed database test. For example, P1 represents a prefetch application with 1 thread.

4.4.4.2 Results of the Complex Benchmark

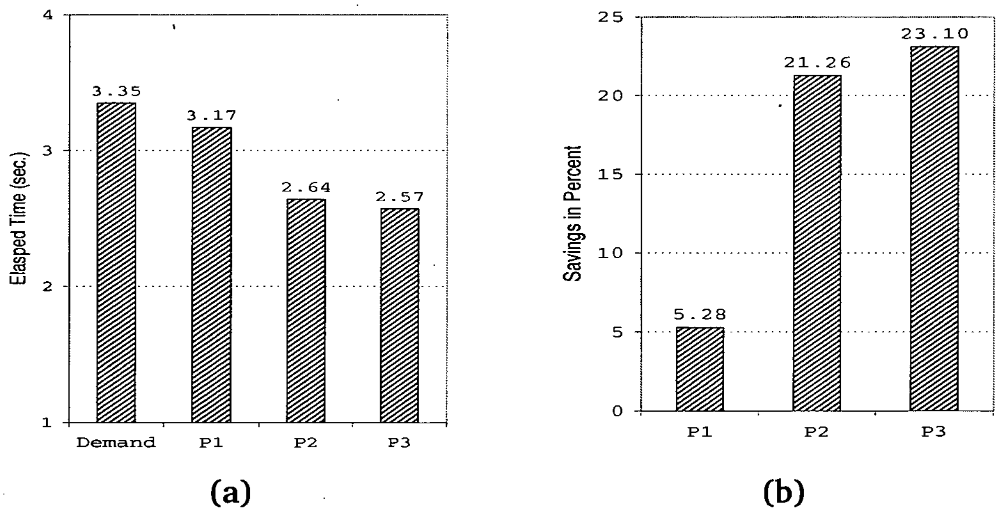


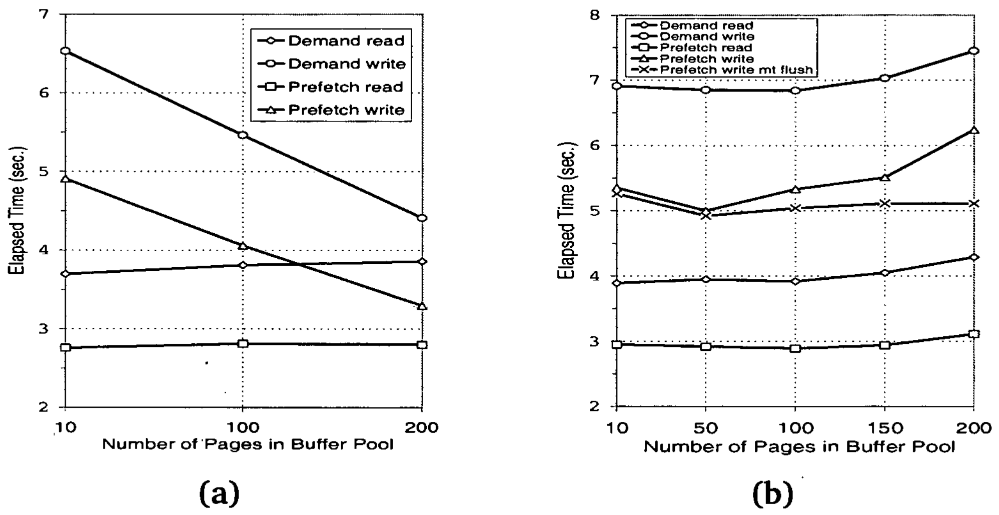
Figure 4.14: The result of the complex benchmark is presented in (a) total elapsed times and (b) in relative saving of the prefetch versions.

In Figure 4.14 we present the results of our complex benchmark. The number after the prefetch applications indicates the number of prefetch threads. Figure 4.14(a) shows that with an increased number of prefetch threads the elapsed time of the applications is reduced. Recall from the benchmark structure that an ORO has three references to other pages, therefore *P3* has the best performance because it achieves the optimal number of prefetch threads for page requests. Figure 4.14(b) shows the savings of the prefetching versions in percent. *P1* only provides a 5% improvement, compared with *P3* which achieves a saving of 23%.

In Figure 4.13 we present the upshot of our distributed database test. Prefetching always generates additional workload for the server, so that a multi-server environment is more suitable for prefetching. For this test we split the database into two databases, each managed by one server. The servers both run on the same machine so as to have the same circumstances. Figure 4.13 shows that all versions improve slightly performance in the distributed environment. This result does not show the full speed-up potential of multiple servers. Firstly, every ORO has three pointers to non-resident pages which means that one server has still to fetch two pages and consequently slows down the client. Secondly, the two servers run on the same machine and each server also forks a disk process. All four processes interfere with each other on the two processors of the server machine.

The size of the buffer pool has an important impact on the performance of the prefetch technique. We balance the difference between 10, 100 and 200 frames in the buffer pool. The update versions write just one object on the page, which causes the page to be marked dirty. The time for this test was stopped just before the commit of the transaction. Comparing both read versions in Figure 4.15, the prefetch version can slightly increase the amount of saving with increased buffer size. The elapsed time of the demand version increases whereas the elapsed time of the prefetch version stays almost constant. The prefetch version performs better with a larger number of buffer frames because this reduces locking of synchronisation variables. The extended buffer pool size has an enormous impact on the write applications. A larger number of available frames reduces the number of server flushes at transaction time, which has a direct effect on the response time.

In the next test, Figure 4.15(b), we stopped the time after the commit of the transaction. For the read versions the result are the same as in Figure 4.15(a), the demand version increases slightly and the prefetch version stays almost constant. For the write versions we created one version, called *Prefetch write*, which flushes all dirty pages at the end of the transaction sequentially and an-



(a) (b)

Figure 4.15: Demand and prefetching applications under different buffer pool sizes at the client. The result of figure (a) shows the elapsed time before transaction commit and (b) after the commit.

other version, called *Prefetch write mt flush* which has two FlushThreads to do the flushing in parallel. All write versions reduce elapsed time with a buffer size of 50 compared with 10, but they deteriorate after 50 buffer frames because more pages have to be flushed sequentially at the end of the transaction. Over a buffer size of 100 the multithreaded flush version outperforms the sequential flush version; at a buffer size of 200, the advantage of the multithreaded version is 1.23 seconds. This outcome makes clear that multithreading is not only useful for prefetching; flushing dirty pages to the server is an ideal application for multithreading.

With this benchmark we made another test to evaluate the influence of incorrect prefetches. From the three references to other pages we used one reference for the application navigation and the other two pages for incorrect prefetches. In Figure 4.16 *Two incorrect* means prefetching two pages incorrectly from an ORO; *One incorrect* means prefetching one incorrectly and *Correct* means optimal prefetching. We produced some application processing after every object access by using a loop iteration. For example, the elapsed time of 3560 iterations is equal to the ESM processing time of one object, i.e. with 3560 iterations the total object processing time is doubled. One incorrect and Correct always perform better than *Demand.* After an IRT value of 850, Two incorrect also performs better than Demand.

The workload of the database client is important for the scheduling of the prefetch threads. If the prefetch thread is scheduled at the time of encoun-

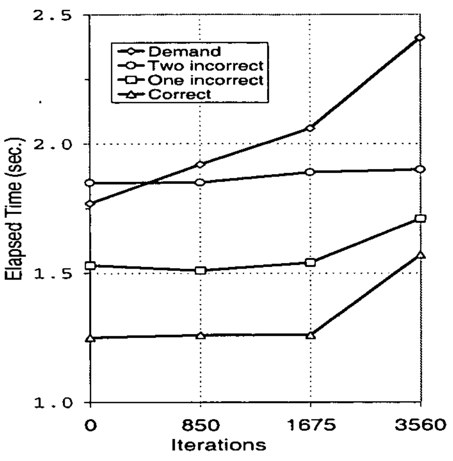


Figure 4.16: Effect of incorrect prefetching in the complex benchmark.

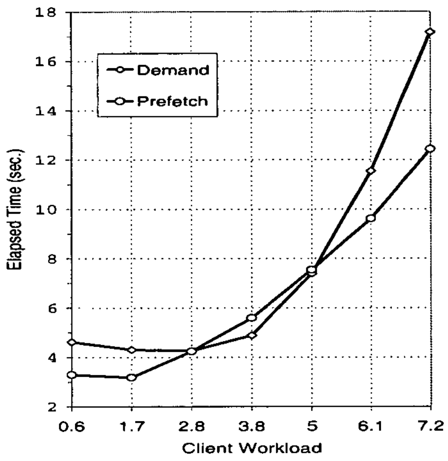


Figure 4.17: Prefetching under varied client workload levels.

tering a PSO and the operating system time slice ends after sending the prefetch request to the server, prefetching can be even more successful under a high workload. Otherwise, if the prefetch thread is not scheduled before application accessing the prefetch request is unnecessary and produces processing overhead. In Figure 4.17 we varied the workload on the client workstation. A workload of 4 means that all four processors are fully utilised and the idle time is almost 0%. The *Prefetch* version performs well under a workload of 2.8 and even better above the workload of 5, i.e. where there is queuing for CPU resources. At the workload level around 4, i.e. just at the point where all processors are busy, the performance of the prefetch threads suffers as a result of operating system scheduling and therefore prefetch requests are arriving late or after the object fault.

Multithreading on the database client side can be used not only for I/O but also for very expensive CPU functions. On analysing the client code we found out that there is an expensive function to calculate the free slot space in the ESM client software. This function is called on every object access and then calculates the free slot space of the whole page. We created another thread for this function (called *Audit* application) and the result of this test can be found in Figure 4.18. In the examination of Figure 4.18 we varied the number of objects accessed in a **leaf page**[8]. With an increasing number of objects in access the *Audit* application speeds up. All the pages that have to be checked by the

[8] Recall the structure of the benchmark in which we accessed only one object in a page and then followed the navigation through other pages. We define such a page as a leaf page.

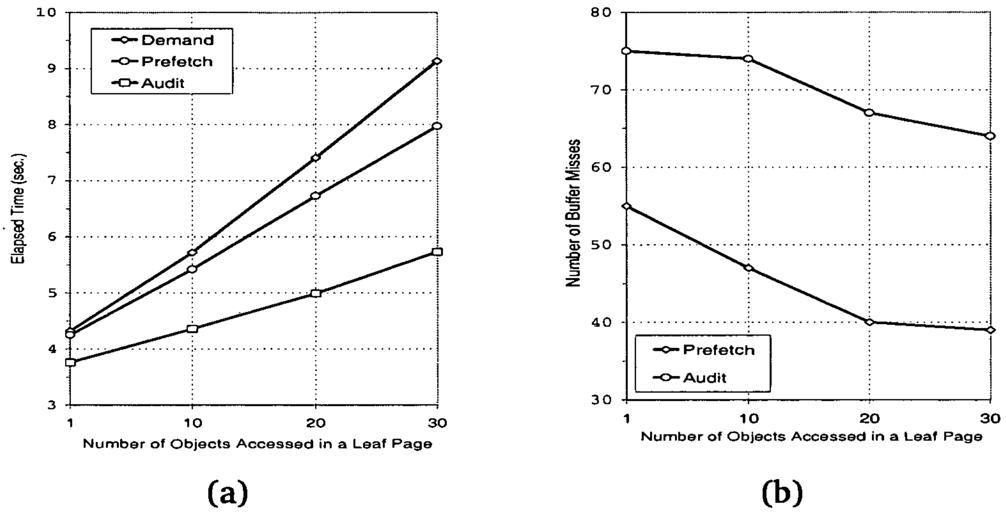


Figure 4.18: Multithreading for CPU-intensive functions, like auditing. Figure (a) depicts the results in elapsed times and figure (b) in the number of buffer misses.

AuditThread are put into a queue and then the application thread continues with processing. If the audit page is already in the queue it is not inserted again which reduces the amount of client processing and biases the success of the AuditThread slightly[9]. Figure 4.18(b) shows that the AuditThread has actually a higher number of buffer misses because we used the same POD but the total amount of overhead is less.

	Pref.Thr. 1	Pref.Thr. 2	App.Thr.
file reads (bytes)	956,768	643,344	8,248
file reads (ops)	627	412	5
file write (bytes)	6,496	4,368	56
file write (ops)	116	78	1
CPU time	34.01	34.24	31.23
CV[10]wait time	163.23	0.00	0.00
mutex wait time	3.39	1.61	0.02
read wait time	0.56	0.40	0.00
semaphore wait time	0.70	184.89	0.00
total sync wait time	167.32	186.50	0.02

Table 4.3: Performance characteristics of a 2 prefetch threads application.

Increasing the number of prefetch threads also intensifies the total synchronisation time of the application. We analysed an application with two prefetch

[9] To ensure the integrity of the database pages the AuditThread must be finished before the commit of the transaction.

[10] CV means condition variable.

	Pref.Thr. 1	Pref.Thr. 2	Pref.Thr. 3	App.Thr.
file reads (bytes)	643,344	486,632	470,136	8,248
file reads (ops)	385	302	297	6
file write (bytes)	4,368	3,304	3,192	56
file write (ops)	78	59	57	1
CPU time	35.75	35.99	35.94	32.67
CV wait time	170.88	0.00	0.00	0.00
mutex wait time	2.57	2.29	2.17	0.02
read wait time	0.28	0.19	0.20	0.00
semaphore wait time	0.74	182.33	180.53	0.00
total sync wait time	174.19	184.61	182.70	0.02

Table 4.4: Performance characteristics of a 3 prefetch threads application.

threads (Table 4.3) and an application with three prefetch threads (Table 4.4) using the Solaris Thread Analyser [SPARCworks, 1995]. The first four rows show the I/O per second. The prefetch threads do most of the I/O. The 3 prefetch threads distribute the I/O work more evenly. The synchronisation costs are higher with 3 prefetch threads. Three prefetch threads have a total mutex wait time of 7.05 seconds whereas the two prefetch threads only require 5.02 seconds. Moreover 3 prefetch threads consume more condition variable and semaphore time. The semaphore time is mostly waiting time for a prefetch. The conclusion of this test is that more prefetch threads increase synchronisation time for the prefetch threads but not for the AppThread.

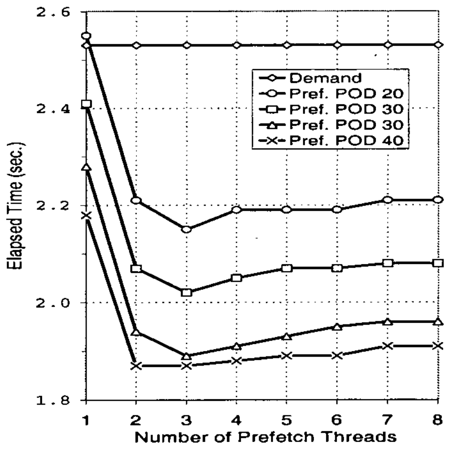


Figure 4.19: Effect of the number of prefetch threads at the database client.

For the last test we created a benchmark in which every ORO has 7 references to other pages. This benchmark was designed to test the scalability of prefetch threads. We diversified the number of prefetch threads from 1 to 8. In theory, if the thread overhead is low and all prefetch threads are scheduled in time by the operating system, the best result could be achieved with 8 threads. On the other hand, if the synchronisation cost of the threads is high and the prefetch threads are scheduled late then best performance is achieved by about 3 prefetch threads, i.e. 4 threads run on 4 processors. We established 4 prefetching versions which start the prefetch operation with different PODs. We used the values of 20, 30, 40 and 50 as a POD. Figure 4.19 shows the result of this benchmark. All prefetching applications show the highest decrease from 1 to 2 threads. The best result is achieved at the level of 3 threads since all threads are executed on the same processor without any context switching. After the level of 3 all applications deteriorate.

4.5 Summary

In this chapter we presented a new architecture for prefetching. The implementation results demonstrated under which circumstances prefetching is advantageous. The key findings of this chapter are:

- The total reduction of elapsed time is dependent on CPU-I/O ratio. We achieved a reduction of up to 23%.
- Multiple prefetch threads improve performance as a result of intensified parallelism at the client.
- The demand application slows down with every additional client connected to the server while a prefetch application can achieve a constant, lower elapsed time as long as the prefetch is started according to POD_{pf}.
- A multiple-server architecture is more attractive for prefetching than a single server architecture, even where the single server has a power that is comparable to the combined power of the multiple servers.
- The buffer pool size has a prodigious impact on update applications. The results measured before the commit showed that an increased buffer pool size improves the applications performance. Stopping the time after the commit showed the opposite result, i.e. a higher buffer pool size decreases the performance of the update applications. Using one additional flush thread can improve performance by 22% at the level of 200 buffer frames.

- The percentage of incorrect prefetches is vital for the success of prefetching. The complex benchmark result substantiated that one incorrect prefetch was acceptable but two incorrect prefetches without additional client processing were unacceptable.

- Using multithreading for CPU-intensive functions also reduces elapsed time.

- The number of all the threads for prefetching and processing should not be higher than the number of processors available.

In this chapter we also described a new structure-based prefetching technique. This technique has a low overhead and works effectively when the number of adjacent pages is small. If the number of non-resident pages is high or the prediction computation is becoming less of a problem, owing to faster processors, we investigated more intelligent prediction techniques in the next chapter.

Chapter 5

Statistical Prefetching

5.1 Introduction

The idea of the prefetching technique presented in this chapter, called **PMC**, is to compute the page access probability considering the structure of the relationships between persistent objects. We assume that there are transition probabilities associated with inter-object pointers. Furthermore, we assume that every object belongs to exactly one page. From the current position of the client navigation we compute the access probability of all adjacent pages. The depth of adjacent pages (for explanation see Figure 5.1) includes directly adjacent pages but can also include higher depths of indirectly adjacent pages. The prefetch object distance determines the depth. If the distance is short then the depth is limited to the directly adjacent pages otherwise multiple depths could be prefetched. Recent and future developments in microprocessor technology suggest to prefetch higher depths in the future.

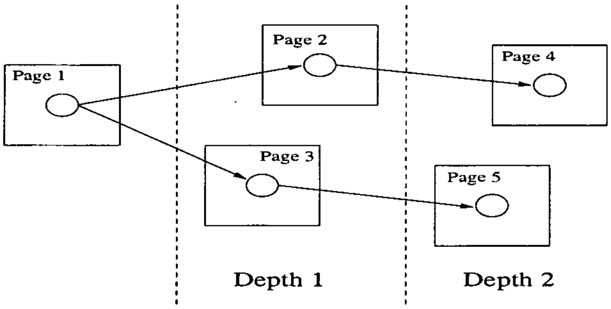


Figure 5.1: Example of page dependencies. Page 2 and 3 are directly adjacent to page 1 and page 4 and 5 are indirectly adjacent.

We compute the page probability by evaluating all paths from the current object to objects in the adjacent pages. The object relationships are modeled using a Discrete-Time Markov Chain (DTMC) and a method called *hitting times* is used to compute the page access probability. If the probability of a page is higher than a threshold defined by cost/benefit parameters then the page is a candidate for prefetching. In Section 5.2 we will give an introduction to the model definitions, explain the decision process for prefetching and the computation of a page probability. To determine the prefetching threshold we consider various cost parameters to compare the benefit of a correct prefetch with the cost of an incorrect prefetch which are presented in Section 5.3. The results from the implementation of the prefetching technique and simulation results of the OO1 benchmark are presented in Section 5.4. Finally, in Section 5.5 we conclude this chapter.

5.2 Prediction Model

At the beginning we give some formal definition of objects and object relationships in Section 5.2.1. The decision process for carrying out a prefetch is explained in Section 5.2.2. In Section 5.2.3 we describe how we compute the access probability of a page and the mean time to access the page.

5.2.1 Model Definitions

In OODBMSs objects have relationships with other objects. Let O denote the set of objects in the store and let $R \subseteq O \times [0, 1] \times O$ denote the set of object relationships between objects, along with a weight for each such relationship.[1] The weight denotes the probability that we traverse from one object to another. Further, let $o_i \in O$ be the current object that the database client is processing. Let $o_j \in O$ and $x \in [0, 1]$. If $(o_i, x, o_j) \in R$ then we let $\mathrm{o}_i \xrightarrow{x} \mathrm{o}_j$ denote that we go from o_i to o_j with probability x.

Let PG be the set of database pages and $pg_i \in PG$ the page that contains the object o_i, i.e. the page on which the client is currently processing. A page pg_j is said not to be resident in the client buffer pool BP, with $BP \subseteq PG$, if $pg_j \in PG \backslash BP$. The condition for an object relationship is $\forall o_i \in O, \sum\{x :$

[1] We assume that these relationships are invariant to the history of the computation. Where this assumption does not hold, the probability associated with a particular relationship will have to be obtained from behaviour observed in several different computations exemplifying these different histories. More information about profiling graph access pattern can be found in [Banatre et al., 1997].

$\exists o_j \in O : (o_i, x, o_j) \in R\} = 1$, i.e. the sum of the probabilities associated with the emerging arcs from o_i must add up to 1. For the case when the traversal terminates at an object we introduce a self-loop for an object o_i such that $o_i \xrightarrow{x} o_i$ denotes the probability x that the traversal will be terminated at object o_i.

5.2.2 Prefetch Decision Model

In Chapter 4 we used a *Prefetch Object Distance* (POD_{pf}) to start the prefetch operation d object processing units (steps) before application access. Recall that the prefetch object distance is the amount of client processing to be overlapped with the prefetch to receive the page before application access. The advantage of this approach is that the savings in elapsed time are high but the probability that the traversal will be from the PSO (Prefetch Start Object) to an object in a non-resident page could be low. Prefetching a page less than d objects before access has certainly a lower saving but the probability that we traverse from the current object to an object in a non-resident page could be higher.

In this chapter we introduce a *Prefetch Distance Range* (PDR) with a *minimal POD* (POD_{min}) and *maximal POD* (POD_{max}) in which we would identify a PSO. POD_{min} is defined to be the break-even-point when the prefetch benefit starts to outweigh the prefetch costs. POD_{max} has a higher value than POD_{pf} because its value takes into account possible delays of the page fetch. A prefetch started earlier than POD_{max} would result in the same benefit. Starting a prefetch too early could result in a bad replacement decision which is considered in our model and explained later.

Firstly we explain when we prefetch pages and in the next sections we describe the components that influence this decision process. Suppose $i \in O$ is the current object and $\alpha \in PG \backslash BP$ is a page then we will denote by $\mathbb{P}_{i,\alpha,}$ the probability that starting in i, we hit[2] page α (definition in Section 5.2.3). Also let CIP be the Cost of an Incorrect Prefetch (definition in Section 5.3.1) and BCP$_{(d)}$ the Benefit of a Correct Prefetch (definition in Section 5.3.2) which is dependent on the POD parameter d. The decision whether to prefetch a page is made by the following constraint:

$$\mathbb{P}_{i,\alpha} > \frac{CIP}{BCP_{(d)}+CIP} \tag{5.1}$$

[2] To hit a page means the traversal from a current object to an object in that page.

To explain this inequality it was derived from:

$$\mathbb{P}_{i,\alpha} \cdot BCP_{(d)} > (1 - \mathbb{P}_{i,\alpha}) \cdot CIP \tag{5.2}$$

If the probability that the page will be accessed, multiplied by the benefit of the page, is greater than the probability that the page is not accessed, multiplied by the cost of an incorrect prefetch, then we will prefetch the page.

Let O_{NR} ($O_{NR} \subseteq O$ and $O_{NR} \nsubseteq PG \backslash BP$) be the set of objects which are not in the buffer pool where there are paths from the current page pg_i. For the purpose of our model, for every element o_k ($o_k \in O_{NR}$) we check constraint (5.1) for every object o_i which has path to o_k in the distance range ($POD_{Min} \leq d \leq POD_{max}$). If constraint (5.1) is fulfilled then we define object o_i as a PSO. There may be a number of paths from o_i to o_k that is exponential in d. However, as we shall see, we do not have to examine each path individually.

Let O_{PDR} ($O_{PDR} \subseteq O$) be the objects in the PDR which have a path to a page α. Then we compute the *heat* of an object $o_i \in O_{PDR}$ to access page α by:

$$heat(o_i, \alpha) = \mathbb{P}_{i,\alpha} \cdot BCP_{(d)} - (1 - \mathbb{P}_{i,\alpha}) \cdot CIP \tag{5.3}$$

For objects, like o_i, we compare the $heat(o_i, \alpha)$ value of o_i with objects that are referenced by o_i and other objects in the forward direction of the object graph up to a depth dp. The object with the highest heat value executes the prefetch. This process could involve a comparison over multiple objects. The identified PSO has then the theoretical optimal distance to prefetch a page (POD_{opt}).

After the analysing process we decide whether to prefetch from the persistent store. If the estimated benefits outweigh the fixed costs (thread and socket creation) then we will use prefetching.

5.2.3 Computation of the Page Access Probability

A DTMC is a stochastic process which is the simplest generalisation of a sequence of independent random variables. A Markov Chain is a random sequence in which the dependency of the successive events goes back only one unit in time.[3] In other words, the future probabilistic behaviour of the process depends only on the present state of the process and is not influenced by its past history. This assumption is valid for OODBMSs due to the fact that the object traversal is not concerned with how we navigated to an object instead it is interested in the navigation from the current object.

Let $(X_n)_{n\geq 0}$ be a DTMC with transition matrix P. We associate one state in the DTMC with one object and the current state is associated with the current object. The hitting time of a page α is the random variable $H^\alpha : \Omega \rightarrow \{0, 1, 2, ...\} \cup \{\infty\}$ given by

$$H^\alpha(\omega) = \inf\{n \geq 0 : X_n(\omega) \in \alpha\} \tag{5.4}$$

$H^\alpha(\omega)$ is one state of α (one object in page α) to be hit at time ω. The probability starting in object i that $(X_n)_{n\geq 0}$ ever hits α is then

$$h_i^\alpha = \mathbb{P}_i(H^\alpha < \infty). \tag{5.5}$$

The mean time taken for $(X_n)_{n\geq 0}$ (the navigation process) to reach α (a non-resident page) is either n steps or ∞ steps and given by

$$k_i^\alpha = E_i(H^\alpha) = \sum_{n<\infty} n\,\mathbb{P}(H^\alpha = n) + \infty\,\mathbb{P}(H^\alpha = \infty) \tag{5.6}$$

The mean hitting time and the hitting probability can be calculated by linear equations. With Theorem 1 we are able to establish the equations for the hitting probability.

Theorem 1 *The vector of hitting probabilities $h^\alpha = (h_i^\alpha : i \in O)$ is the minimal non-negative solution to the system of linear equations*

$$\left\{ \begin{array}{lcll} h_i^\alpha & = & 1 & \mathit{for}\; i \in \alpha \\ h_i^\alpha & = & \sum_{j\in O} p_{ij} h_j^\alpha & \mathit{for}\; i \notin \alpha \end{array} \right. \tag{5.7}$$

[3] Time in the context of a DTMC means simply a number of steps.

The mean hitting time can also be calculated by linear equations:

Theorem 2 *The vector of mean hitting times $k^\alpha = (k^\alpha : i \in O)$ is the minimal non-negative solution to the system of linear equations*

$$\begin{cases} k_i^\alpha &= 0 & \textit{for } i \in \alpha \\ k_i^\alpha &= 1 + \sum_{j \notin \alpha} p_{ij} k_j^\alpha & \textit{for } i \notin \alpha \end{cases} \tag{5.8}$$

The proof for both theorems can be found in [Norris, 1997]. We solve these equations online by an iterative method called *conjugate gradient* and off-line by the *LU decomposition algorithm* (for more implementation details see Section 5.4.1). In addition, we define the two following rules describing the adaption to our environment:

Rule 1: Let λ be the set of states corresponding to the objects in O that have a path to an object in a page α. For the setting of the equations to calculate the hitting probability (according to Theorem 1) and the mean hitting time (according to Theorem 2) we only consider states that are elements of λ ($o_i \in \lambda$).

Rule 2: To calculate the mean time that we hit a page α we have to consider only transitions from states in λ to states in λ. If the condition $\forall o_i \in O, \sum\{x : \exists o_j \in O : (o_i, x, o_j) \in R\} = 1$ is not fulfilled anymore because a state is not in λ then we have to recalculate the probability transitions. The new probability values for x_i' are computed by a method called **re-normalisation**:

$$x_i' = \frac{x_i}{\sum_{j=1}^{m} x_j} \tag{5.9}$$

where we only consider transition probabilities x_j to the objects corresponding to the states in λ. The new value for x_i' is computed by dividing its old value by sum of object transitions that have a path to states in α.

Example re-normalisation: Suppose we have $o_t, o_u, o_v, o_w \in O$ and $x_1, x_2, x_3 \in [0,1]$ with the transitions $\mathrm{o}_t \xrightarrow{x_1} o_u$, $\mathrm{o}_t \xrightarrow{x_2} o_v$ and $\mathrm{o}_t \xrightarrow{x_3} o_w$. Let $o_t, o_u, o_v \in \lambda$ and $o_w \notin \lambda$. Then the values for x_1' and x_2' are computed by $x_1' = x_1/(x_1 + x_2)$ and for $x_2' = x_2/(x_1 + x_2)$.

Example hitting equations: Figure 5.2 depicts a simple example of objects that are resident in a page with references to other objects. The probability to hit page 2 starting in object o_1 is computed as follows:

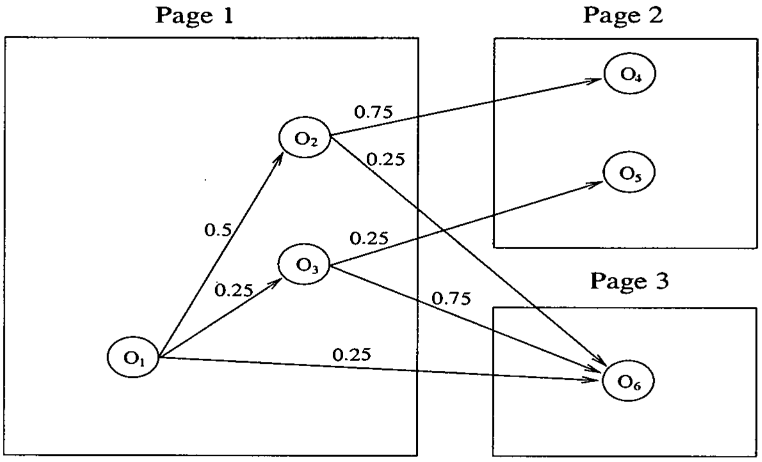


Figure 5.2: Probability graph of object accesses.

$$
\begin{aligned}
h_5 &= 1\\
h_4 &= 1\\
h_3 &= 0.25h_5\\
h_2 &= 0.75h_4\\
h_1 &= 0.5h_2 + 0.25h_3
\end{aligned}
$$

These linear equations can be solved by the method of substitution or other methods. After solving the equations each variable, corresponding to objects, has then a value indicating the probability of traversing to page 2. As a result, the access probability of page 2 is 0.4375 and doing the same for page 3 would result in 0.5625. The mean time to access page 2, starting in o_1, is then computed by the following equations:

$$
\begin{aligned}
k_5 &= 0\\
k_4 &= 0\\
k_3 &= 1 + 1k_5\\
k_2 &= 1 + 1k_4\\
k_1 &= 1 + \frac{2}{3}k_2 + \frac{1}{3}k_3
\end{aligned}
$$

Starting from o_1 the mean time to access page 2 is 2 and to page 3 is 1.75. The transition values in the formulae to compute k_1 to k_3 are obtained according to Rule 2.

5.3 Cost-Benefit Model

This section gives details of the definition of an incorrect prefetch in Section 5.3.1 and the definition of a correct prefetch in Section 5.3.2. At the end we also discuss the advantage of a multiple page request.

5.3.1 Cost of an Incorrect Prefetch Request

This cost describes the additional elapsed time for the application due to an incorrect prefetch. Every incorrect prefetch imposes a higher synchronisation cost for the DemandThread to access global data. Table 5.1 shows the cost parameters which influence the cost of an incorrect prefetch. The escalating use of synchronisation variables has a negative influence on the DemandThread which can be formally described by:

$$CIP = C_{CS} + C_M + C_R \tag{5.10}$$

The replacement cost C_R is a problematic parameter for a system implementation because the accurate prediction of a buffer replacement is difficult. If we would use always the high value of C_R, i.e. the cost of a page fetch, then the value of CIP would be imprecise for most prefetch decisions. On the other side if we use the lower value, i.e. 0, we would neglect the re-access of the evicted page. We have decided to use the value of 0 for our simulation. In the future we will investigate applying page probabilities for the replacement decision. If we would have a value for the access probability of a page then we could multiply the probabilty with C_R which gives a more accurate value for CIP.

Parameter	Description
C_{CP}	Cost for client processing which includes auditing, buffer management (except C_R), IO, concurrency control, network processing and memory management.
C_{CS}	Increased cost of context switches due to prefetch threads. Let $C_{CS(1)}$ be the context switch cost for one prefetch thread and let $\sigma_{(p)}$ be the scale-factor dependent on the number of prefetch threads p. $$C_{CS} = C_{CS(1)} \cdot \sigma_{(p)}$$
C_M	Additional waiting time and processing cost for the DemandThread to acquire and release mutexes. Let $C_{CM(1)}$ be the mutex cost for one prefetch thread. $$C_{CM} = C_{CM(1)} \cdot \sigma_{(p)}$$
C_{PR}	Cost for using prefetch information (not for solving the hitting times equations).
C_{PW}	Cost for waiting for a page request from sending to receipt. Let $C_{PW(1)}$ be the waiting cost for a request to the server with 1 client and $\delta_{(c)}$ a scale-factor for the delay of a page fetch dependent on the number of clients c at the server. $$C_{PW} = C_{PW(1)} \cdot \delta_{(c)}$$
C_R	Cost for the replacement of a page with a prefetched page. The evicted page must be accessed again before the prefetched page. $$C_R = \begin{cases} C_{PW} + C_{CP} & \text{if page is accessed again} \\ 0 & \text{otherwise} \end{cases}$$
C_S	Cost for the DemandThread to wait on a semaphore (only when the DemandThread stalls for the prefetched page).
B_P	Benefit for prefetching one page. Let C_O be the cost of processing one object; recall that d is the prefetch distance parameter. $$B_P = \begin{cases} C_{PW} + C_{CP} & \text{if prefetched page is resident on access} \\ C_O \cdot d & \text{otherwise} \end{cases}$$

Table 5.1: Cost/benefit parameters.

5.3.2 Benefit of a Correct Prefetch Request

The maximum saving for a prefetch is only achieved when the prefetched page arrives at the client before application access. The benefit BCP depends on the amount of savings minus the prefetch costs:

$$BCP = B_P - C_{CS} - C_M - C_{PR} - C_R - C_S \tag{5.11}$$

The appropriate setting of B_P is another influential parameter. Its value is mainly determined by the stall time for a server request, C_{PW}. We try to estimate C_{PW} according to the workload of the server. This estimation process is naturally very complex in real OODBMS but also crucial to start the prefetch at the right time.

5.3.3 Cost and Benefit of a Multiple-Page-Request

If we predict multiple pages to prefetch according to constraint (5.1) we could demand them by a single request from the server. The server would read the pages from disk and send them back to the client either (a) separately when time constraints are tight or (b) in a batch if time is not a problem.

A multiple-page-request has the advantage that the processing cost on the client and the server is lower than a sequence of single requests (which reduces C_{CP} and C_{PW}) because some functions have to be executed only once. It also reduces the network costs (which affects C_{PW}). The costs of thread management (C_M and C_{CS}) are also lower because multiple pages are requested by just one thread.

5.4 Performance Analysis

The relevant performance parameters from the ESM implementation for the simulation are described in Section 5.4.1. Then we present the results from our simple benchmark test in Section 5.4.2 and from the OO1 test in Section 5.4.3.

5.4.1 Implementation Results

5.4.1.1 Simulation Parameters

In this section we present the results that we obtained from timing ESM with one client connected to the server. We used the same machines and disks as

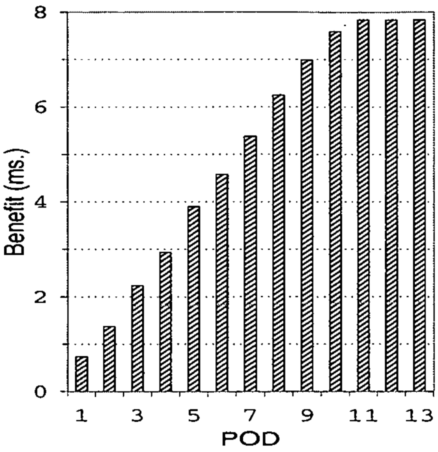


Figure 5.3: Savings of one prefetch dependent on the POD.

described in Section 4.4.1 with the only difference being that Dual-II is now used as the server. The *Sun Fast Ethernet* network runs at 100 Mb/sec.

For a start we have to compute POD_{pf}. Dividing the cost of a page fetch from disk (7943 μs[4]) by the cost of processing one object (799 μs) results in a value of 10 for POD_{pf}.

In Figure 5.3 we show how the amount of savings that can be achieved by one prefetch request depends on the prefetch distance. A prefetch is already successful with a distance of 1 (POD_{min}) and the maximum improvement is achieved at a distance of 11 objects (POD_{max}). From these empirical results we developed a benefit formula (5.12) which computes the amount of savings of a prefetch given the distance. We used the least squares method to find the line of best fit relating distance and benefit which results in:

$$benefit(d) = -782\mu s \cdot d + 109\mu s \qquad (5.12)$$

The variable benefit in formula 5.12 is almost as high as the cost of object processing. The benefit values are highly influenced by the settings of the object processing cost and the page fetch cost. We also measured the cost of an incorrect prefetch which is about 1573 μs higher than a demand fetch.

For an efficient implementation to solve the hitting times equations we compared 5 algorithms. We used two direct methods - Gauss-Jordan (GJ) and

[4] This is an optimistic value. We assume disk pages to be stored in clusters which reduces the average seek time.

LU-decomposition (LU), which compute an exact solution; and three iterative methods - Successive Over-Relaxation (SOR), Gauss-Seidel (GS) and Conjugate Gradient (CG). A detailed description of these algorithms can be found in [Stewart, 1994].

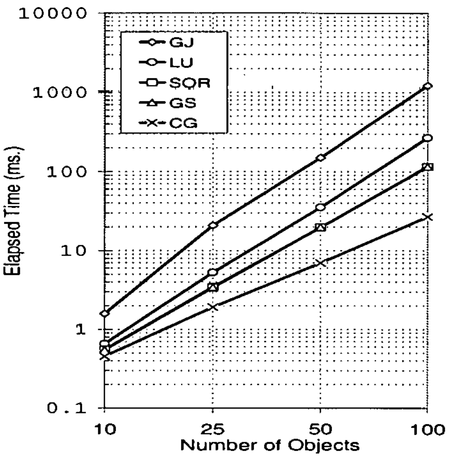


Figure 5.4: Computation time to solve linear equations.

Parameter		Setting (μs)
Object processing		799
1 Page Prefetch :	Page fetch	7943
	Incorrect prefetch	1573
	Variable benefit	-782
	Fixed cost benefit	109
	POD_{min} / POD_{max}	1/11
2 Page Prefetch :	Page fetch	9804
	Incorrect prefetch	2360
	Variable benefit	-782
	Fixed cost benefit	163
	POD_{min} / POD_{max}	1/13

Table 5.2: Simulation parameters.

5.4.1.2 Prediction Costs

Figure 5.4 shows the elapsed times of these algorithms to compute a matrix with n objects. LU is the faster direct method and CG is the fastest iterative method. Therefore we use the CG method in a separate thread on-line and the LU method off-line. In our application the number of objects in the matrix is determined by the number of objects in the **equivalence class**, i.e. all the objects that have a path from the current page to the adjacent pages that have to be computed. If the thread for the computation gets too busy we have to continue the computation after the transaction. Please note that all the computation is done automatically and the database administrator has only a few adjustments to make, e.g. determining the user object processing cost.

The amount of computation that we can do on-line is dependent on the amount of client processing time per object. For example if we process 10 objects with a processing time of 1ms each then we could compute in parallel an equivalence with 64 objects using the CG algorithm. Higher object process times would even provide more overlapping time for prediction. The amount of overlapping time therefore restricts how far we can predict future object access. At the moment we predict only the directly adjacent pages but according to rapid developments in CPU technology (Figure 1.4) it is soon possible to compute multiple depths of adjacent pages.[5]

5.4.2 Simulation Results from a Simple Benchmark

In the simulation we want to present the benefits of prefetching which are very dependent on the object relationship structures and the probability transitions. For example a linked list is a very easy candidate for prediction whereas objects with a fan-out of 10 referenced objects are very difficult to predict.

We used the discrete process based simulation package C++Sim [Little and McCue, 1993]. Every component of our client/server architecture (client, server, disk, prefetch engine and network) is simulated as a process so that each component can run in parallel. Each process has an associated queue for incoming requests and on completion the request is passed on to the next component. For the hold times of a process we used the timing results from the ESM implementation, e.g. time of a buffer replacement. The network cost is computed by a fixed transmission cost and a variable cost depending on the number of bytes to be transferred. The success of prefetching is measured by the waiting time of the client. If a prefetch is performed and the client requires the page then it will wait until the receipt of the page and the total elapsed time of the application will only consider the client waiting time. In Table 5.2 we present the constant cost factors of our simulation. The values for the incorrect prefetch indicate the

[5] Prediction computation then still dependent on client processing but can also be overlapped with user waiting time.

extra elapsed times due to an incorrect prefetch. We assume that the buffer space is infinite.

Algorithm 5.1 *Prefetch Algorithm P0 at analysing time*

```
Compute hitting probabilities according to Theorem 1
Compute mean hitting times according to Theorem 2
```

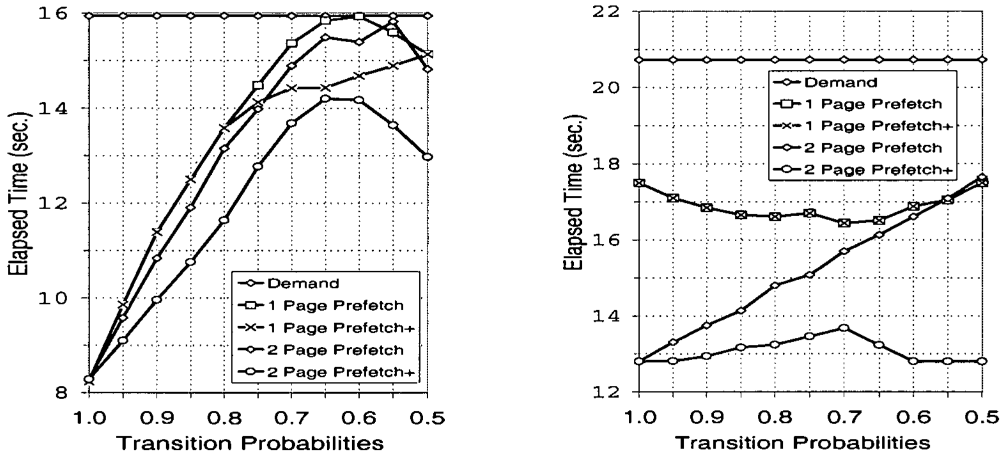
Algorithm 5.2 *Prefetch Algorithm P0 at run time*

```
/* get highest probability page from probability data structure */
get highest probability page and assign to $pp$
if $pp$ is not in buffer and not prefetched then
  /* get object distance from current object to $pp$ */
  get $d$ from hitting times data structure for $pp$
  /* if $d$ is in prefetch range */
  if $d$ >= $POD_{min}$ and $d$ <= $POD_{max}$ then
    if probability($pp$) > $CIP$ / $BCP(d)$ + $CIP$ then
      /* get highest heat of all objects within depth $dp$ */
      get heat($o_h$, $pp$)
      /* if heat from current object > $o_h$*/
      if heat($o_i$, $pp$) > heat($o_h$, $pp$) then
        prefetch $pp$
      end if
    end if
  end if
end if
```

The prefetch algorithm for this set of experiments consists of two parts. The first part computes the hitting times and hitting probabilities (Algorithm 5.1). This computation can take place on-line or off-line but has to be finished before starting part two. The second part is executed at application run time and makes a prefetch decision (Algorithm 5.2).

To test our prefetch algorithm we created two simple benchmarks. In both benchmarks every branch object has an out-degree of 2. In the first benchmark the distance from the entry object in the page to an object in another page is 10. In this distance there are 4 branch objects and 6 non-branch objects which makes a total of 62 objects in a page and 16 references to different pages. Every page has the same structure and we access 1000 pages. In Figure 5.5(a) we show the result of this test with 3 applications: *Demand*, a *1 Page Prefetch* (*1PP*) and a *2 Page Prefetch* (*2PP*) technique. *1PP* and *2PP* are variations of Algorithm 5.2 which fetch either 1 or 2 pages at the same time respectively.

The applications with a + sign consider the *heat* parameter to start the prefetch at the best possible object.



(a) (b)

Figure 5.5: Result of the simple simulation test.

We varied the transition probabilities (**tp**) of the branch objects from 1.0 down to 0.5. Every branch object has two pointers to other objects which have to sum up to 1.0. On the x-axis of all figures we state the tp of the first pointer of an object. High probability transitions result in the computation of high probability pages, for example a tp of 1.0 always produces pages with an access probability of 1.0. The navigation through the object graph was controlled by a **draw-operator**[6]. The *Demand* application has constant values and is independent of the transition probabilities. *1PP* and *1PP+* perform equally well until the tp of 0.8, after which *1PP+* is better because of a later but more accurate prefetch. *2PP+* is always better than *2PP* and the 1 page prefetching techniques because it has a higher hit ratio. At the tps of 0.65 and 0.6 all prefetching applications suffer from a bad hit ratio imposed by difficult page predictions. Figure 5.5(a) shows the general advantage of our technique: If the tps allow prefetching it can reduce elapsed time drastically but if not it will not decrease performance.[7]

In the second benchmark the distance from an entry object to an object in another page is now 16, with 3 branch objects in between which results in 8 references to other pages. The major difference to benchmark 1 is that the last branch object in the high probability path has a 0.5 probability to both objects.

[6] Given a probability value it decides to continue navigation with reference 1 or 2.

[7] This assume that the prefetched pages do not evict pages that will be accessed earlier and incorrect prefetches do not replace pages that will be accessed again.

This results in 2 referenced pages with high probabilities. Figure 5.5(b) shows the result of this test. In general the prefetching application shows a better performance than in the previous benchmark because there are only 8 adjacent pages and these pages are easier to predict. *1PP* and *1PP+* show the same elapsed time for all tps. Neither application performs well at the probability of 1.0 because the last branch object with 0.5 probability imposes a high incorrect prefetching time for both. *2PP+* shows its superiority again especially with lower probabilities because of a more accurate prefetch. Figure 5.5(b) also shows that it is beneficial to prefetch given all possible probabilities.

5.4.3 Simulation Results from the OO1 benchmark

In another set of experiments, we tested our prefetching technique with the OO1 benchmark structures [Cattell, 1992]. We opted for the OO1 benchmark because it is possible to specify the degree of clustering and furthermore object relationships are not too complex. To recall the benchmark structure, every object has a medium size and has three pointers to other objects. In our environment we restrict the number of out-going pointers to two because it makes the navigation of pointers easier to understand.

The random connections between objects are selected to produce some locality of reference. Specifically, 90 percent of the connections are randomly selected among the 1 percent of objects that are "closest" and the remaining connections are made to any randomly selected object. Closeness is defined using the object with the numerically closest object ID's. In our experiments we varied the closeness between 80, 90 and 100 percent and call it to **cluster factor**. Objects are clustered according to their OID.[8]

The simulation cost parameters for one page fetch are presented in Table 5.3. The value for the average disk access is obtained from the performance specification of the Seagate Cheetah 18. The values for the client and server processing costs are the timing results from our ESM implementation. The tests were performed on a machine with 50 MHz but state-of-the-art machines have a cycle speed of up to 600 MHz. We therefore divided the timing values for sole processing by 12 and improved the values for the memory access by about 0.35%. The amount of object processing is an important parameter for the overlapping time with the prefetch and a variation of this value can be found in Figure 5.19.

[8] For example, if there are 50 objects in page then objects with OID 1-50 are placed in page 1 and objects with OID 51-100 are placed in page 2 and so on.

Parameter	Setting (μs)
Client Object Processing Cost	1000
Average Disk Access Cost	8527
Total Network Transfer Cost	231
Total Client Processing Cost	557
Total Server Processing Cost (page resident)	336
Total Server Processing Cost (page not resident)	493

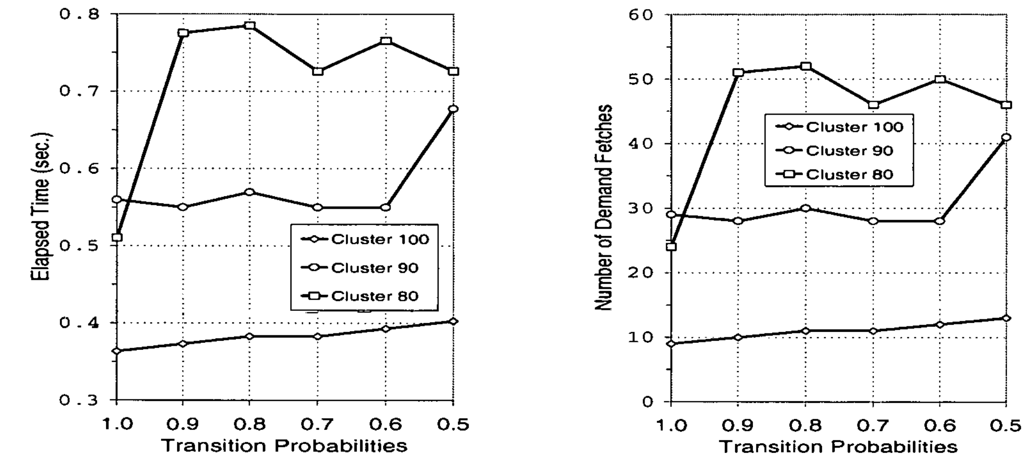
Table 5.3: Simulation parameters for one page fetch operation.

In the first set of experiments we evaluate the benefits of prefetching under different levels of clustering. Therefore we set the buffer pool size to infinite and access only a part of all database pages in order to prefetch correct and incorrect pages from the server. In the second part of the tests we examine the effect of limited buffer pool space on prefetching and the choice of a buffer replacement strategy. In these tests we access every database page several times. Please note that the database size and the number of accessed pages is quite small but the results with a higher number of pages would be very similar.

In our experiments we navigate through the object graph by traversing pointers. Every object has two pointers to other objects. In the experiments the tp of the first reference is varied from 1.0 down to 0.5 in 0.1 steps and the tp of the second reference is the remaining amount to sum both reference up to 1.0. At run time the navigation through the object graph is determined by a draw-operator which decides to follow either object reference 1 or 2 depending on the tp. The simulation is terminated when we have processed a fixed number of objects.

5.4.3.1 Result of the Demand Applications

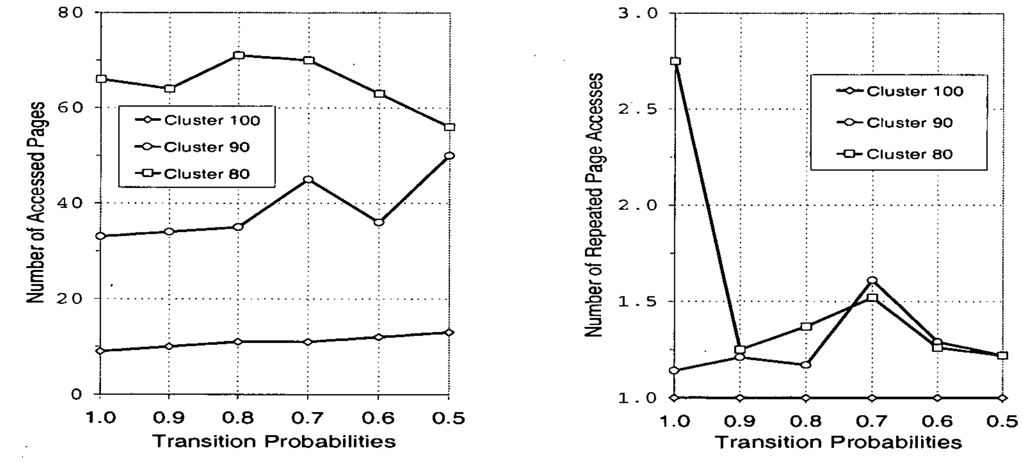
In Figure 5.6(a) we see the elapsed times of all demand applications under different cluster factors. Due to the fact that the client processing time of all applications is the same, the elapsed times is solely dependent on the page fetch times. Figure 5.6(b) shows the number of demand fetches of all applications under different cluster factors. The shapes of the graphs are very similar to the graphs in Figure 5.6(a). In Figure 5.6(a) we can see that a lower clustering factor induces a higher application elapsed time. The values of the tps have only a limited effect on the demand applications. For example, at tp 1.0 the number of demand fetches is much lower for *Cluster 80* because a higher number of



(a) (b)

Figure 5.6: Characteristics of the Demand applications under different cluster factors. Figure (a) shows the elapsed times of the applications and Figure (b) shows the number of demand page fetches.

cycles in the object graph causes less pages to be fetched. The *Cluster 100* has a modest increase in the number of demand fetches since the number of accessed objects per page is lower.



(a) (b)

Figure 5.7: Characteristics of the Demand applications under different cluster factors. The number of accessed pages is illustrated in Figure (a) and number of repeated page accesses in Figure (b).

The total number of accessed pages is shown in Figure 5.7(a). The basic finding of this test is that a smaller cluster factor increases the number of accessed pages. *Cluster 90* and *Cluster 100* have a slight increase in the number of page accesses with lower tps whereas *Cluster 80* decreases after tp 0.8. For example, *Cluster 100* traverses at lower tps more often via the second object reference to the next object. The second referenced object has a higher OID than the first object, therefore the number of accessed objects in the page gets lower and consequently the total number of accessed pages gets higher. The number of repeated accesses to a page is shown in Figure 5.7(b). *Cluster 80* has a high repetition value at tp 1.0 which reflects the result of the low number of demand fetches in Figure 5.6(b). In general most pages are accessed only once or twice and the 0.7 tp application has a slightly higher page access factor.

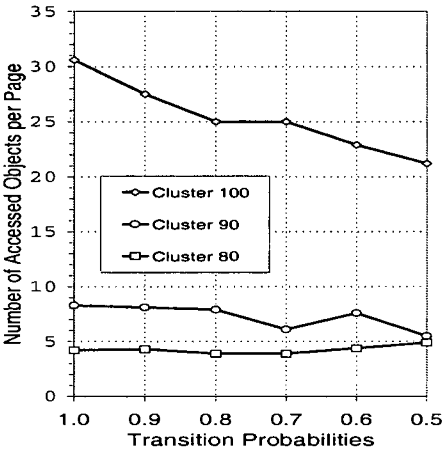


Figure 5.8: Number of accessed objects per page.

The number of accessed objects per page is important for having some overlapping time for prefetching. As Figure 5.8 shows, a lower cluster factor reduces the number of accessed objects in a page. *Cluster 80* and *Cluster 90* display an almost independent behaviour of the tps whereas *Cluster 100* reduces the number of accessed objects with lower tps. As previously explained, the reason for this behaviour is that with lower tps the number of second pointer traversals increases.

5.4.3.2 Result of the Prefetch Applications

Algorithm 5.3 *Prefetch Algorithm P1*

```
/* pp is one page or a set of pages with the same probability */
get highest probability page and assign to $pp$
for each $pp_i$
  if $pp_i$ is not in buffer and not prefetched then
    /* if page probability is higher than threshold $t$ */
    if probability($pp_i$) > $t$ then
      prefetch $pp_i$
    end if
  end if
end for
```

Algorithm 5.4 *Prefetch Algorithm P2*

```
/* get all the pages with a probability over threshold $t$ */
get all pages with a probability > $t$ and assign to $pp$
for each $pp_i$
  if $pp_i$ is not in buffer and not prefetched then
    prefetch $pp_i$
  end if
end for
```

The prefetch algorithms for these experiments are slightly different from the techniques we used for the simple benchmark. For this set of experiments we only consider the probability of a page and not the mean time to access a page. We designed two techniques: *P1* (Algorithm 5.3) and *P2* (Algorithm 5.4) which prefetch pages with probability higher than a specific threshold. We varied the threshold from 0.0 to 1.0 and show only the result of best performing prefetch application.

Algorithm 5.5 *Disk Probability Algorithm DP*

```
/* look for demand request in queue q */
get demand requests dr from q
if dp exists then
  serve pd
else
  /* get information from client via network */
  get probabilities for disk requests and other infos
  get prefetch request pr from q that stalls client
  if pr exists then
    serve pr
  else
    get page pp with highest probability from q
    serve pp
  end if
end if
```

Algorithm 5.6 *Disk Probability Algorithm DP2*

```
/* look for demand request in queue q */
get demand requests dr from q
if dp exists then
  serve pd
else
  /* get information from client via network */
  get probabilities for disk requests and other infos
  get prefetch request pr from q that stalls client
  if pr exists then
    serve pr
  else
    /* difference to DP: probability must be higher than zero */
    get page pp with highest probability ( > 0) from q
    serve pp
  end if
end if
```

The probabilities that we used for the prefetch decision could also be used for the disk queue. The transfer of the page probabilities increases the network workload but network processing is cheaper than disk processing. We developed two techniques: *DP* (Algorithm 5.5) and *DP2* (Algorithm 5.6) which are very similar but differ in the threshold parameter. *DP* serves requests with a current page probability of 0 and *DP2* does not. We combined both prefetch algorithms with both disk queue algorithms and show also the results for the prefetch algorithms which do not consider the probabilities at the disk.

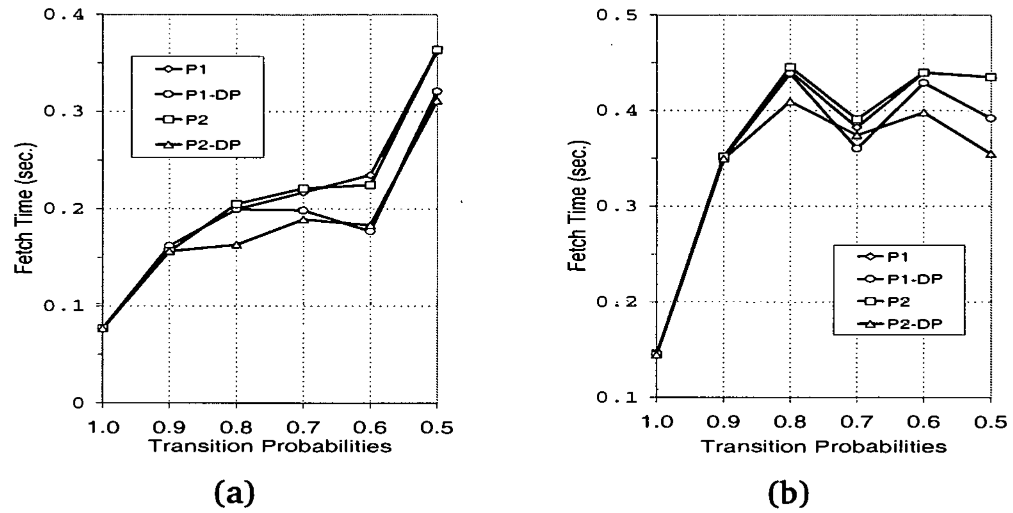


Figure 5.9: Result of the prefetch applications: P1, P1-DP, P2 and P2-DP. Figure (a) shows the result for the cluster factor 90 and Figure (b) for the cluster factor 80.

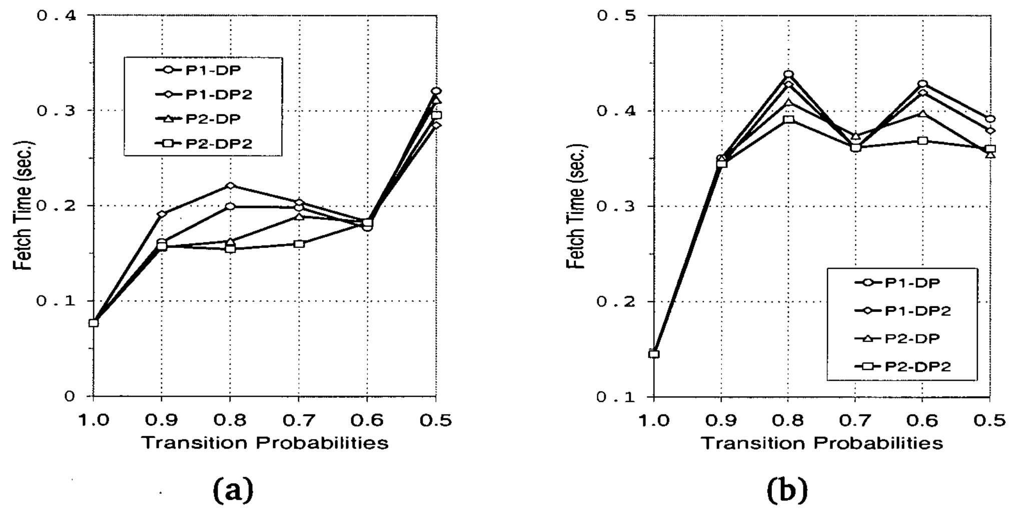
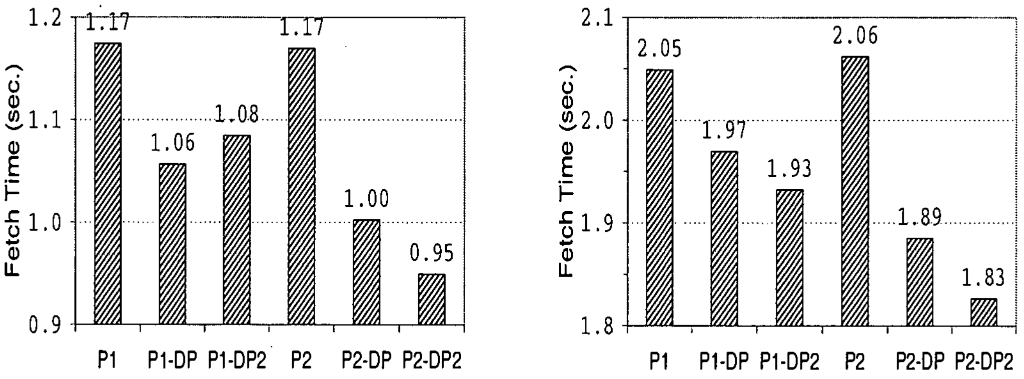


Figure 5.10: Result of the prefetch applications: P1-DP, P1-DP2, P2-DP and P2-DP2. Figure (a) shows the result for the cluster factor 90 and Figure (b) for the cluster factor 80.

In Figure 5.9 and Figure 5.10 we show the results of all prefetch applications. The total fetch time of all transition probabilities is presented in Figure 5.11. Prefetch application *P2-DP2* clearly performs best and *P2-DP* second best. This result shows that prefetching multiple pages over a specific threshold at the same time is more effective than only prefetching the highest page. Behind applications *P2-DP* and *P2DP2* the *P1-DP* and *P1-DP2* show an average result. The worst prefetch applications are *P1* and *P2* which perform similarly for both cluster factors. The conclusion of this test is that applications that use



(a) **(b)**

Figure 5.11: Total fetch time of all prefetch applications for transition probabilities from 1.0 to 0.5. Figure (a) shows the result for the cluster factor 90 and Figure (b) for the cluster factor 80.

page probabilities for the disk queue perform much better.

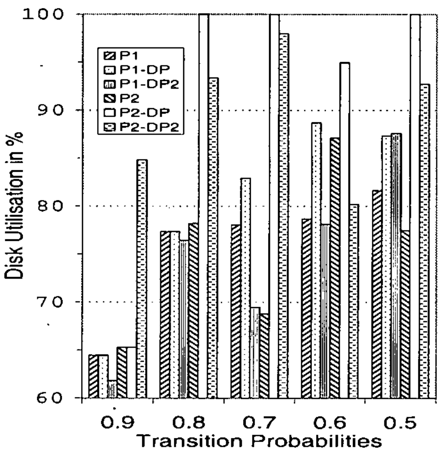
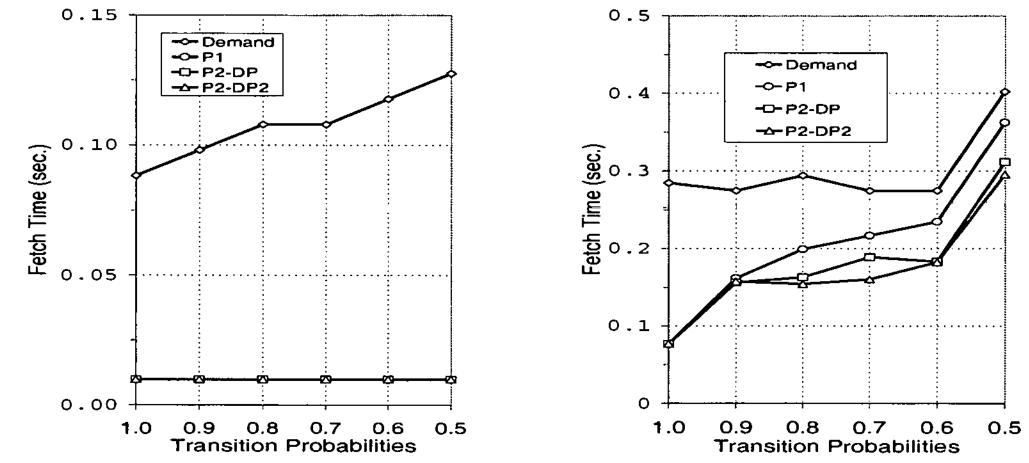


Figure 5.12: Disk utilisation for cluster 90 applications.

For every prefetching technique it is important to achieve a high utilisation of the disk. In Figure 5.12 the disk processing time of the prefetch applications is presented as a percentage of the total simulation time. *P2-DP* clearly has the highest utilisation, and with *P2-DP2* having the second highest. This result is no surprise because *P2-DP* retrieves many pages with low priorities. *P2-DP2* uses the disk more effectively as low priority requests do not block high priority requests. The lowest utilisation has *P1-DP2* followed by *P2* and *P1*. All *DP2* applications have a lower utilisation than *DP* applications because they do not serve low priority disk pages.

5.4.3.3 Comparison of the Demand and Prefetch Applications



(a) (b)

Figure 5.13: Performance of the Demand application and the three prefetch applications: P1, P2-DP, P2-DP2 with cluster factor 100 in Figure (a) and cluster factor 90 in Figure (b).

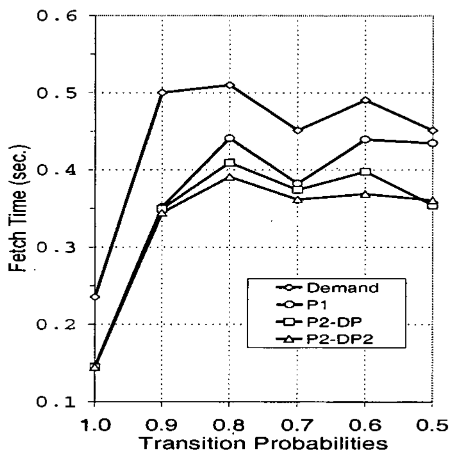


Figure 5.14: Performance of the Demand application and the three prefetch applications: P1, P2-DP, P2-DP2 with cluster factor 80.

The result of the *Demand* and three prefetch applications *P1*, *P2-DP* and *P2-DP2* under the cluster factors of 100, 90 and 80 can be found in Figure 5.13 and Figure 5.14. The corresponding improvements in % of the prefetch applications *P1* and *P2-DP2* for the cluster degree of 90 and 80 are presented in Figure 5.15 and Figure 5.16 respectively. The improvements of prefetch applications with a cluster factor of 100 are always 100%. The results show two basic facts: Firstly, lower tps reduce the savings potential of prefetching. Secondly, lower cluster

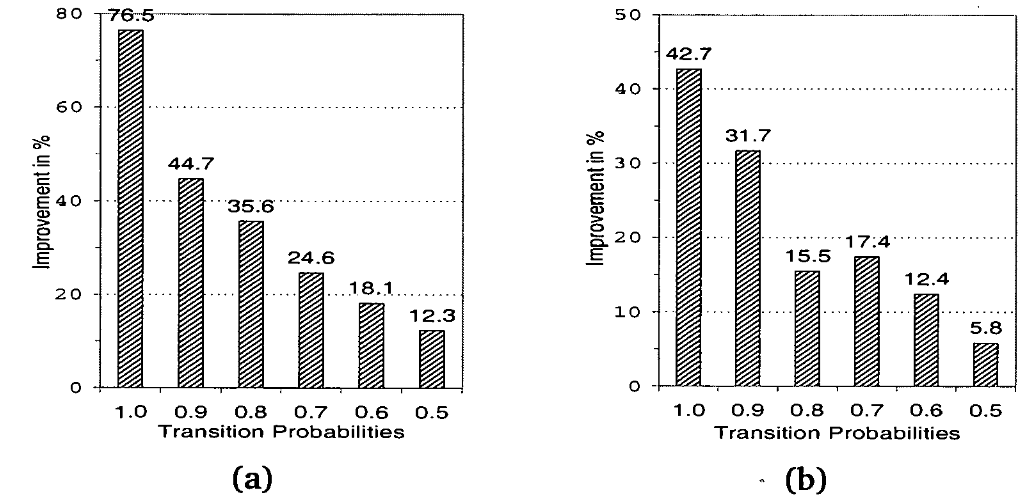


Figure 5.15: Improvements of the prefetch application P1 in % under the cluster factors of 90 (a) and 80 (b).

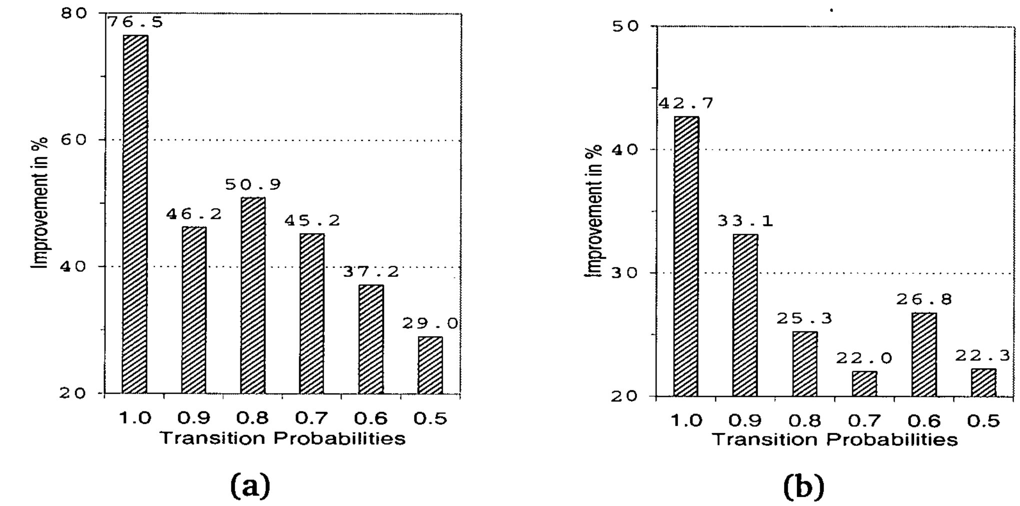
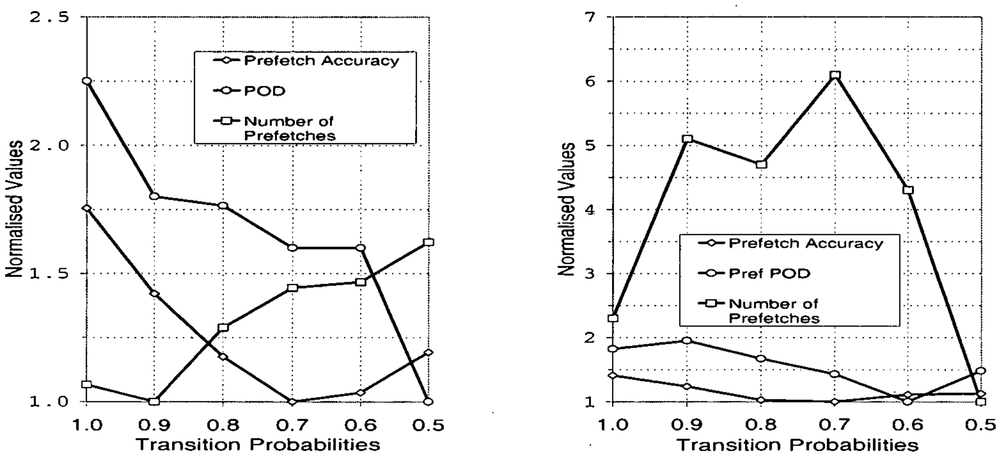


Figure 5.16: Improvements of the prefetch application P2-DP2 in % under the cluster factors of 90 (a) and 80 (b).

factors reduce the benefits of prefetching. This result is mainly due to the fact that we prefetch only adjacent pages in these experiments and not multiple pages in depth. If we would prefetch multiple depths then the reductions in page fetch time would be higher, but the prediction costs would be much higher as well. Prefetching multiple depths will be more feasible in the future with higher performance processors. At the end of the section we demonstrate the higher potential of the *Cluster 80* version in fetch time reduction.

To explain the shrinking improvements of the *P1* applications with lower tps we compared three decisive parameters for prefetching: the prefetch accuracy, the prefetch object distance and the number of prefetches in Figure 5.17. For the *Cluster 90* applications we observe that the POD has probably the highest effect on the prefetch improvements. It decreases from tp 1.0 to tp 0.5 like the prefetch improvements. The prefetch accuracy decreases to tp 0.7, inline with the prefetch improvements, but then increases again. The number of prefetches increases with lower tps, resulting in only a small effect on the fetch times. The applications with a cluster factor of 80 show a similar result. The prefetch distance and prefetch accuracy decreases with lower tps. The number of prefetches is high for medium tps. The improvements at tp 0.8 is lower because accuracy and the number of prefetches are reduced. Tp 0.7 achieves a higher improvement due to a larger number of prefetches. Tp 0.5 has a higher POD and a higher accuracy but a much lower number of prefetches results only in a small saving.



(a) (b)

Figure 5.17: Normalised values for P1 considering prefetch accuracy, prefetch object distance and the number of prefetches for the applications with a cluster factor of 90 (a) and 80 (b).

The improvements for the *P2-DP2* (Figure 5.16) are higher than for *P1*, especially at lower tps. The improvements for the cluster 90 applications fall from tp 1.0 down to tp 0.5, with the exception of the tp 0.9 application. The reason for this exception is that the POD and the number of prefetches is quite low (Figure 5.18). At the tps of 0.8 and 0.7 there is a high number of prefetches combined with a high POD which results in a low prefetch accuracy. The applications with a cluster factor of 80 have low improvements at middle tps. For example at tp 0.7, the *P2-DP2* has a similar fetch time to tp 0.6 but the explanation for the low improvement is the good result of the *Demand* application

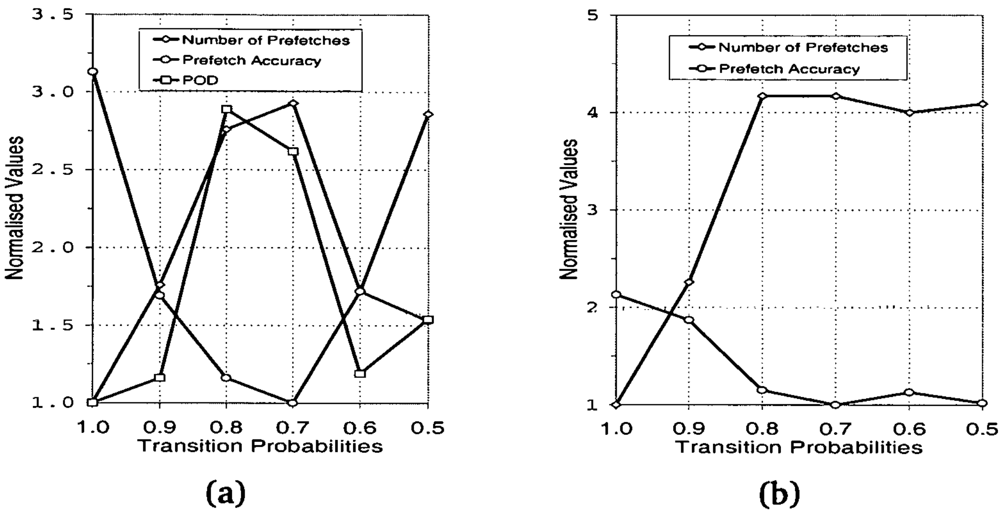


Figure 5.18: Normalised values for P2-DP2 considering prefetch accuracy, prefetch object distance and the number of prefetches for the applications with a cluster factor of 90 (a) and 80 (b). We do not show the POD in Figure (b) because of difficulties in the measurements.

which cannot be compensated by *P2-DP2*.

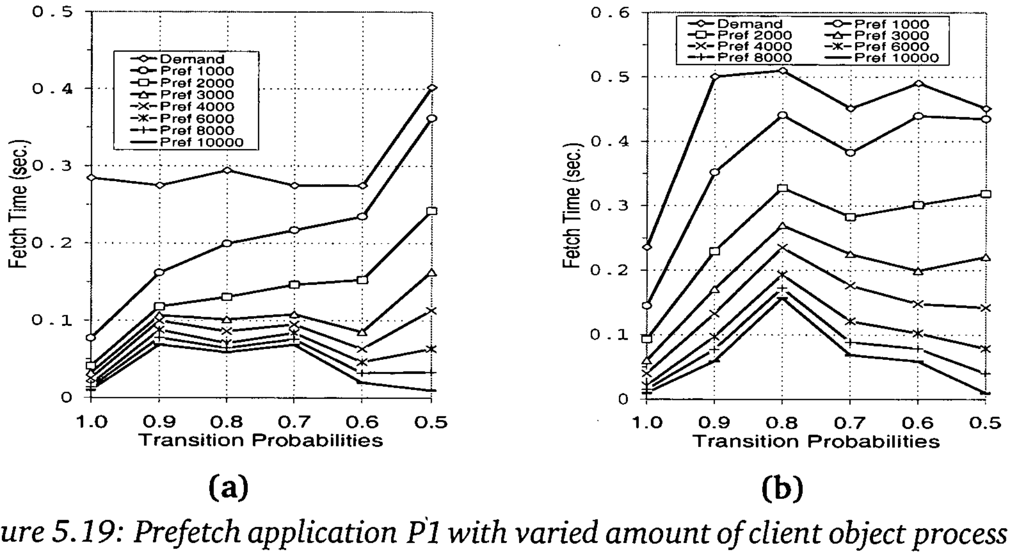
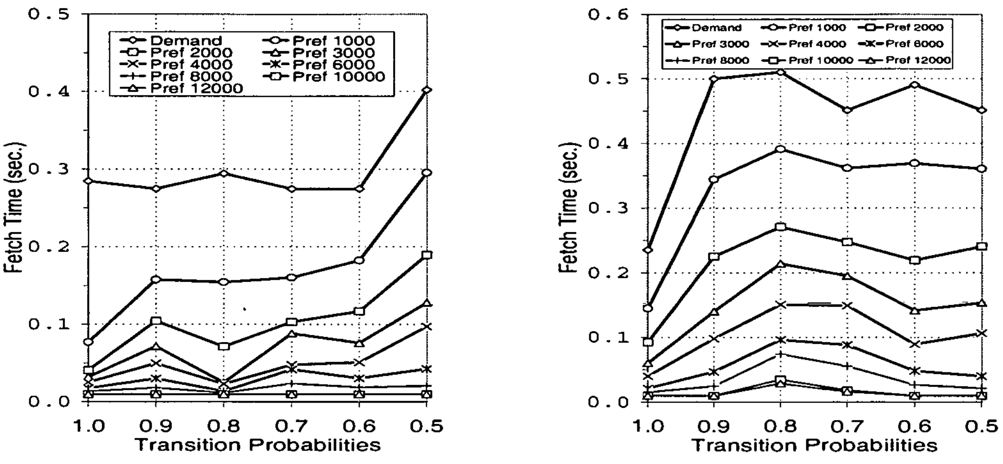


Figure 5.19: Prefetch application P1 with varied amount of client object processing. Figure (a) show the results for the cluster factor 90 and Figure (b) for the cluster factor 80.

In another experiment we want to show the full potential of prefetching versions with lower cluster factors. Therefore, we started the prefetch applications with different amounts of client processing. Higher amounts of client processing ensures more overlapping time for prefetching. We varied the client processing time from 1000 to 12000 μs per object. The results in Figure 5.19



(a) **(b)**

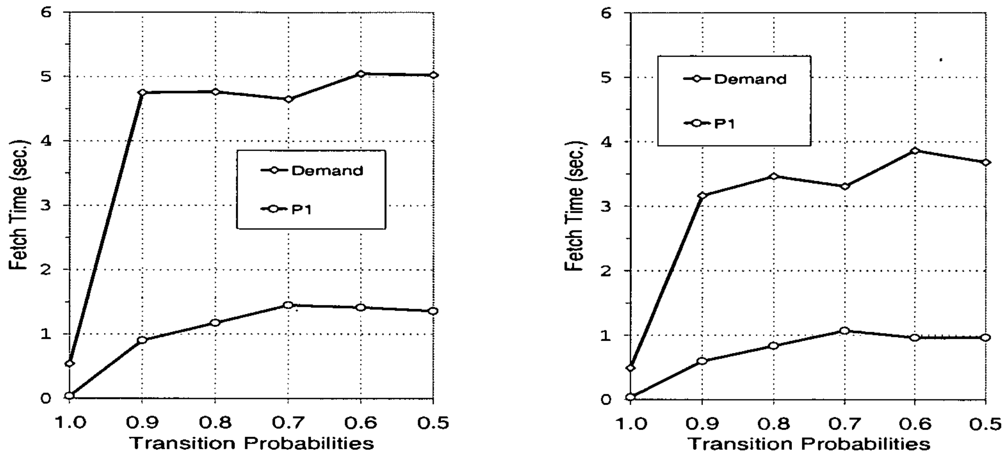
Figure 5.20: Prefetch application P2-DP2 with varied amount of client object processing. Figure (a) show the results for the cluster factor 90 and Figure (b) for the cluster factor 80.

and Figure 5.20 show that higher client processing times ensure higher savings. The prefetch applications with a cluster factor of 80 reveal higher reductions in fetch time than the *cluster 90* prefetch applications. The conclusion of this test is that lower cluster factors result in more page fetch time that can, potentially, be reduced by prefetching.

The result of Figure 5.20 also shows that *P2-DP2* can achieve higher processing times to reduce fetch time to a minimum. P1 cannot reduce fetch times any further with higher processing times because it would refuse to prefetch a page with high probability if there is another page with a higher probability.

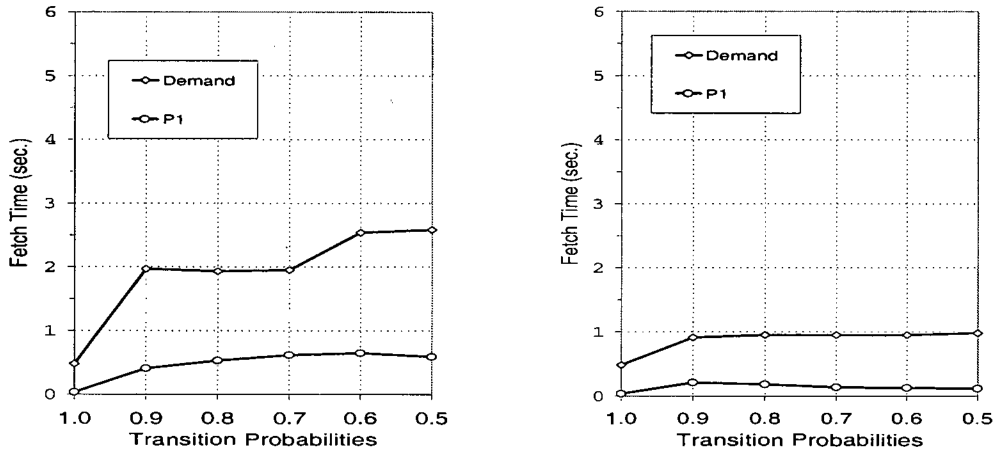
5.4.3.4 Effect of Buffer Pool Sizes

In this set of experiments we evaluated the effect of buffer pool sizes on prefetching technique *P1*. We varied the pool size between 10, 30, 50 and 100 buffers using LRU replacement. We increased the total number of objects to be accessed to have more repeated accesses to every page. Figure 5.21 and Figure 5.22 show that smaller buffer sizes are beneficial for prefetching because the hit rate is low and many pages have to be fetched from the server. In Figure 5.21 neither prefetch application performs well at tp 0.7. The reason for this behaviour is that although the prefetch distance and the number of prefetches is higher than the neighbour application of tp 0.8; it suffers from low accuracy with many incorrect prefetches. Another observation is that low tps increase



(a) **(b)**

Figure 5.21: Effect of the buffer pool sizes on the Demand and P1 application with LRU replacement and a cluster factor of 90. Figure (a) shows the result for 10 buffer frames and Figure (b) for 30 frames.



(a) **(b)**

Figure 5.22: Effect of the buffer pool sizes on the Demand and P1 application with LRU replacement and a cluster factor of 90. Figure (a) shows the result for 50 buffer frames and Figure (b) for 100 frames.

fetch time with only a few page buffers. The direct effect of decreasing buffer pool size on the tp 0.9 application is shown in Figure 5.23.

The current LRU-replacement strategy is not very efficient because it does not consider the computed page probabilities for replacement decisions. We designed a replacement policy, called LRU-Prob, which does not replace a page if it has a probability higher than some threshold. We varied the threshold for the replacement decision between 0.0 and 0.5. The best results are presented

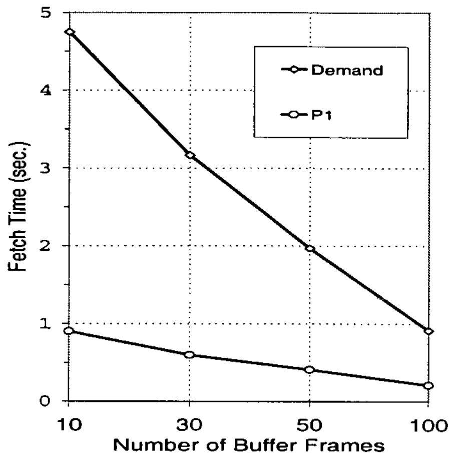


Figure 5.23: Effect of a decreasing number of buffer frames on the application with tp of 0.9 and a cluster factor of 90.

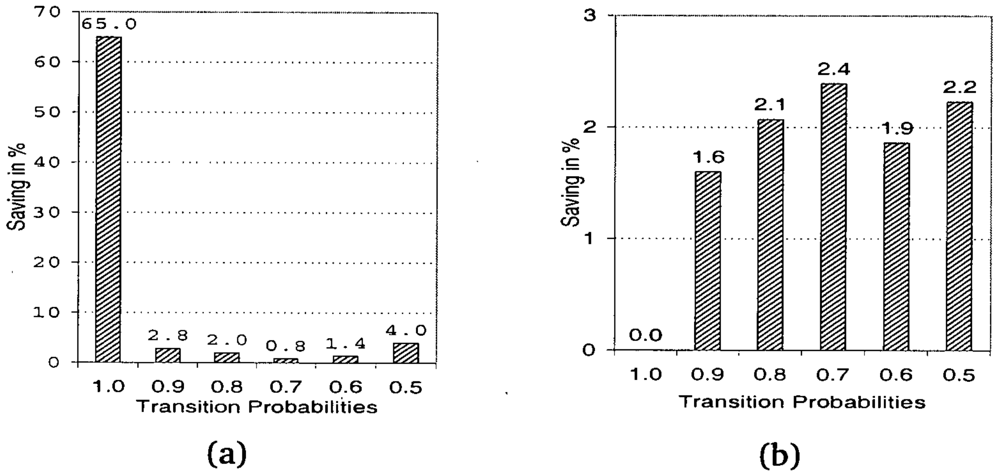


Figure 5.24: Improvement of the LRU-Prob replacement policy compared with a simple LRU policy under a cluster factor of 80. Figure (a) shows the result for 10 buffer frames and Figure (b) for 30 buffer frames.

in Figure 5.24 and Figure 5.25. Generally, the consideration of the page probability improves the total fetch time between 1 and 3%. A surprisingly high improvement of 65% is achieved at tp 1.0 with 10 buffer frames. The client hit rate of LRU is identical to LRU-Prob but LRU-Prob has a much higher hit rate at the server which reduces the page fetch time.

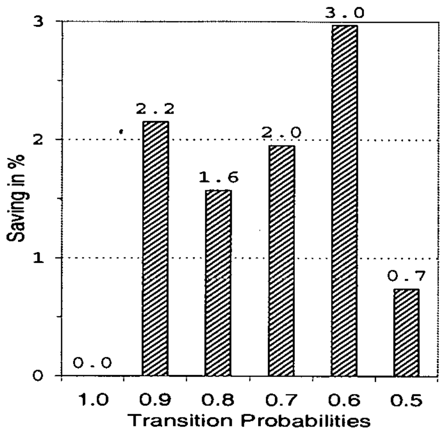
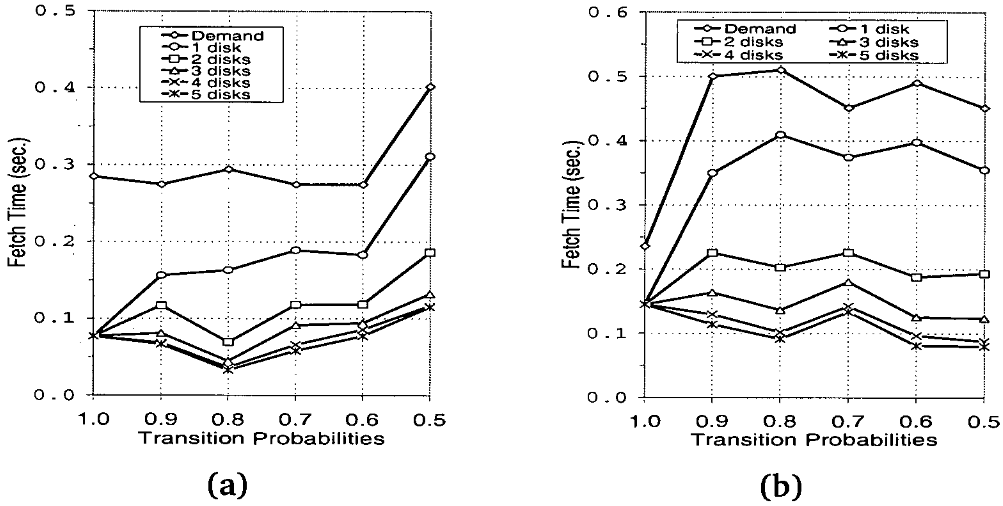


Figure 5.25: Improvement of the LRU-Prob replacement policy compared with a simple LRU policy for 50 buffer frames under a cluster factor of 80.



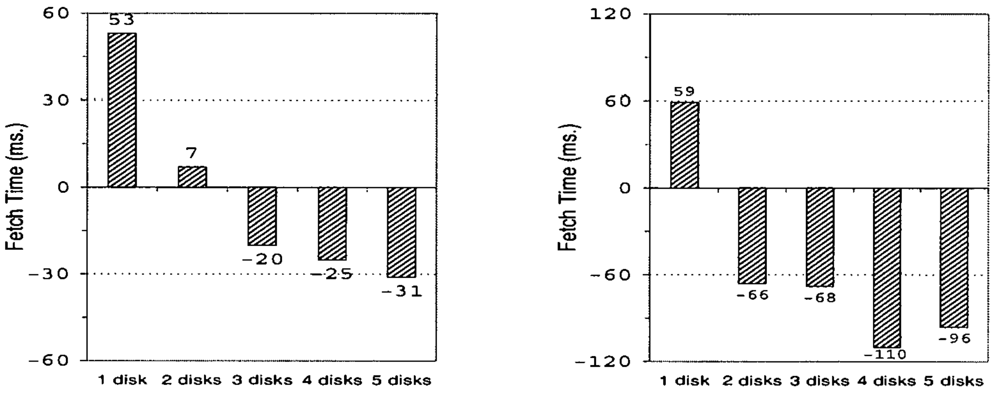
(a)

(b)

Figure 5.26: Effect of parallel disk accesses on the performance of the prefetch application P2-DP with n disks. Figure (a) shows the result for the applications with a cluster factor of 90 and Figure (b) for a cluster factor of 80.

5.4.3.5 Effect of Parallel Disk Accesses

The disk access has a high percentage of the total page fetch cost and this percentage will increase in the future. In this experiment we use a technique called disk striping with which we try to parallelise disk accesses by increasing the number of disks at the server for processing client requests. We assume that database pages are perfectly partitioned over multiple disks so that a free disk can perform any outstanding request. The result of this experiment is shown in Figure 5.26. The prefetch application P2-DP shows good improvements on up to 3 disks. The *4 disks* application improves even further for a cluster factor of 80. In general, a larger number of disks has a higher benefit for the applications with a cluster factor of 80.



(a) (b)

Figure 5.27: Reduction of fetch time of P2-DP over P2-DP2 (negative values show improvements of P2-DP). We compare the sum of the fetch times of all transition probabilities in Figure (a) for cluster factor 90 and in Figure (b) for cluster factor 80.

P2-DP did not achieve such a good result as *P2-DP2* with 1 disk, however the result could be different with multiple disks. If there is more disk processing power then prefetching pages that have a reduced probability, even as small as zero, could be beneficial under two different circumstances. Firstly, during the expensive seek operation the probability could rise and secondly the prefetched page could be accessed at a later time. In Figure 5.27 we demonstrate that *P2-DP* performs better with 3 disks for cluster factor 90 and with only 2 disks for cluster factor of 80.

5.5 Summary

In this chapter we presented a prefetch algorithm that uses page probabilities for the prefetch decision. We compute the page probability by evaluating all paths from the current object to objects in the adjacent pages. The object relationships are modelled using a Discrete-Time Markov Chain (DTMC) and a method called *hitting times* is used to compute the page access probability. If the probability of a page is higher than a threshold defined by cost/benefit parameters then the page is a candidate for prefetching. We developed a model that distinguishes between costs for an incorrect prefetch and benefits of a correct prefetch.

We evaluated the prefetching technique with our own, simple benchmark and the OO1 benchmark. The key findings of the simple benchmark tests are:

- High transition probabilities result in high page probabilities and consequently prefetching can reduce elapsed time drastically.
- Prefetch applications that consider the *heat* parameter for the start of the prefetch show better performance than applications which do not.
- Pages with fewer out-going references are easier to predict and ensure higher prefetch savings.
- The general advantage of our prefetching technique is that if the transition probabilities allow prefetching it can reduce elapsed time drastically but if not it will not decrease performance.

The key results of the OO1 benchmark implementation are:

- The prefetch application P2-DP2 which considers the page probability at the disk queue and prefetches multiple pages at the same time, performs best.
- All prefetch applications that consider page probabilities at the disk queue perform better than applications that do not.
- Lower cluster factors provide higher amounts of fetch time that can be reduced by prefetching.
- Prefetching only direct adjacent pages results in bigger improvements for prefetch applications with higher cluster factors.
- Prefetch accuracy and prefetch object distance are the most important parameters for fetch time reduction.
- A lower number of buffer frames increases the savings potential of prefetching applications.
- Our LRU-Prob replacement strategy outperforms simple LRU replacement.

- Multiple parallel disks increase the performance of prefetching applications.

- Prefetch application P2-DP performs better than P2-DP2 with multiple disks.

In this chapter we studied extensively the performance of the prefetch algorithm PMC and its variants P0, Pl and P2. Pl, which fetches only the highest probability page at a time could not achieve the good performance of P2, which fetches all pages above a threshold. Now we have studied prefetching combined with buffer management and clustering but the granularity of a prefetch is another important issue. In Chapter 6 we compare the unit of a page versus the unit of a group of objects to be transferred between client and server.

Chapter 6

Page versus Object Prefetching

6.1 Introduction

Most OODBMSs and prefetching techniques can be classified as either page or object server systems.[1] If the system is a page server the prefetching technique will prefetch one or multiple pages [Krishnan, 1995; Gerlhof, 1996; Knafla, 1997d; Knafla, 1998b] and in case of an object server one or a group of objects [Chang, 1989; Keller et al., 1991; Palmer and Zdonik, 1991; Day, 1995].

All the previous research was conducted in one of these two types but there is no research, to the best of our knowledge, which assesses a prefetching technique by comparing its performance in a page server and object server implementation. For both systems the important problem to solve is how to avoid the I/O bottleneck. Here we have to distinguish two cases at the server side: disk pages are resident at the server and disk pages are not resident at the server. In theory, if all pages are resident the object prefetching technique has the advantage that it can put together all the relevant objects for a prefetch request independent of how these objects are dispersed on pages. If pages are not likely to be resident, a page prefetching technique has the advantage that it requests only a few high priority disk pages.

Day [Day, 1995; Liskov et al., 1996] made an interesting study in the Thor database in which he compared the performance of a page server system (without prefetching) with an object server system that prefetches groups of objects. The motivation was that the performance of a single-object fetching system is unacceptable [DeWitt et al., 1990; Hosking and Moss, 1993]. Thor trans-

[1] Some OODBMSs products, e.g. GemStone, avoid the problem by executing the request on the server and only return the result to the client.

fers groups of objects from server to client. On receiving a fetch request, a Thor server selects objects to send in response. The group of objects selected is called a **prefetch group**. Thor's dynamic selection of the group contrasts with most distributed object databases which cluster objects statically into pages and transfer pages. The selection of objects considers various techniques that prefetch objects in the transitive closure of the current object. It can use depth-first or breadth-first search and considers whether the object is resident at client or not. The performance evaluation was made with the OO7 benchmark [Carey et al., 1993] using the dense and sparse traversals. The general result showed that the best technique was **bf-cutoff**: a simple breadth-first traversal that cuts off its exploration upon encountering an object already sent to the client. The page server system performed reasonably well in the dense traversal where the access pattern was similar to the clustering. The problem with this study is that it does not give a fair comparison between an object and page server prefetching system because:

1. It assumes that all pages are resident in the server buffer pool. This is advantageous to object prefetching as explained before.
2. It compares an object prefetching technique with a demand paging system.

A general problem with this type of group prefetching is that if the server is already the bottleneck of the system then the additional selection process of objects costs valuable server execution time. Another problem is the knowledge about the client cache: either it discards this information or it needs to be transferred to the server. The advantage is that the prefetching information is available locally.

The differences between a demand object, page and file server systems was studied by DeWitt et al. [DeWitt et al., 1990]. The object server transfers only one object between client and server at a time and both client and server can execute methods. The page server transfers 4-KB pages. The file server uses NFS to access files and is not relevant to our study. The general result of this research was that the object-server architecture is relatively insensitive to clustering. It is sensitive to a client's buffer size up to a certain point, after which the cost of fetching objects using the RPC mechanism dominates. They conclude that for the object server it is viable to send a group of objects. The page-server architecture is very sensitive to the size of the client's buffer pool and to clustering when traversing or updating complex objects. While the page-server architecture is far superior on sequential scan queries, the object server architecture demonstrates superior performance when the database is poorly clustered or the workstation's buffer pool is very small in relation to the size of the database.

Another study concentrated on the interaction of locking and the database architecture [Carey et al., 1994c]. They presented three page server variants that allow concurrent data sharing at the object level while retaining the performance advantages of shipping pages to the client. The results indicated that a page server is preferable to an object server. Moreover, the adaptive page server with object locking was shown to provide very good performance and outperformed the pure page and object server. Both Carey et al. [Carey et al., 1994c] and DeWitt et al. [DeWitt et al., 1990] study the difference in the object/page architectures without considering prefetching.

The performance evaluation of three commercial OODBMSs was subject to the study of [Hohenstein et al., 1997]. Two of the three systems were page servers and one object server. In contrast to previous benchmark studies this evaluation was performed with a concrete data warehouse application. One result is that OODBMSs differ substantially in their performance, often more than standard benchmarks had shown [Carey et al., 1994b]. Another outcome is that numerous tuning possibilities can improve the performance considerably. Results showed that the architecture of the system (page vs. object server) is often more important than the object-oriented paradigm itself. For the application, the object server has, besides the superior performance, an additional advantage due to the finer locking granularity.

In this chapter we compare the performance of a prefetching page server system with a prefetching object server system. The prefetching algorithms are presented in Section 6.2. Section 6.3 presents the environment for the simulation. All the results of the performance evaluation are given in Section 6.4. Finally, Section 6.5 summarises this evaluation.

6.2 Prefetch Algorithms

We compare some page prefetch techniques with several variants of object prefetch techniques. Every pointer has an assigned transition probability (tp). All techniques compute the access probability of objects according to the tps between objects which can be between 0 and 1. The page prefetch algorithms are described in Section 6.2.1 and the object algorithms in Section 6.2.2.

6.2.1 Page Prefetch Algorithms

The page prefetch technique loads the page with the highest access probability from the current object being processed. We described the computation process in the previous chapter. One way in which this technique differs is that we are not using a minimal threshold to start the prefetch; instead it always prefetches the page with the highest probability and only one page at a time. Moreover, it does not take into account any negative effects of prefetching. We compare four different techniques:

1. *PSDem*

   A demand application without any prefetching which loads every missing page on an object fault.

2. *PSPref*

   A prefetch application that fetches always the highest probability page. It can fetch only one page at the time.

3. *PSDemHit, PSPrefHit*

   Identical applications as *PSDem* and *PSPref* but all the requested pages are resident in the servers' buffer pool. No disk request is required.

6.2.2 Object Prefetch Algorithms

The object prefetch techniques always load a group of objects in advance. The relevant group of objects is computed by the *Chapman-Kolmogorov equations* [Ross, 1997]. Let i be the current object and j another object, then $\mathbb{P}_{ij}^{n+m}$ denotes the probability to access object j from object i in $n + m$ steps[2]. Let k be an intermediate object between i and j which could be one object or a set of objects. Then we compute $\mathbb{P}_{ij}^{n+m}$ firstly from object i to k in n steps and then secondly from object k to j in m steps:

[2] A step means the traversal from one object to another object.

$$\mathbb{P}_{ij}^{n+m} = \sum_{k=0}^{\infty} \mathbb{P}_{ik}^{n} \mathbb{P}_{kj}^{m} \text{ for all } n, m \geq 0, \text{all } i, j \tag{6.1}$$

These equations can be solved by matrix multiplications.[3] If the probability of an object is higher than a threshold then we insert this object into the request group. We use four different threshold parameters (0.0, 0.001, 0.01 and 0.1). Another important parameter for the prediction is the lookahead n. We vary the depth of this parameter from 1 to 19 objects. In our tests we will compare the following object prefetch and demand techniques:

1. *OSDem*

   On an object fault, this technique fetches the missing object and all the objects in the lookahead n. It fetches all the non-resident pages from disk and sends the whole group of objects to the client.

2. *OSPref*

   It prefetches all the objects in the lookahead n. In the case of a miss it sends a demand request to the server.

3. *OSDemHit, OSPrefHit*

   Identical applications as *OSDem* and *OSPref* but all disk pages are resident in the servers' buffer.

4. *OSServPref*

   The server prefetches the page with the highest priority into its buffer pool. The prefetch is executed so far in advance that when a client request arrives for a page it will be already resident in the buffer pool.

5. *OSPrefLimDem*

   The prefetch operations are executed as in *OSPref* but the demand operation fetches only objects from the page to which the faulted object belongs. The idea behind this optimisation is that stalling client time is reduced to the lowest level.

[3] These computations are very expensive and are used to compare equal probability values with a page server. We do not compare these computation costs in this study.

6. *OSAbortPref, OSAbortPrefNotDem*

   On the arrival of a new prefetch request, *OSAbortPref* will destroy all previous requests at the server from that client and will execute only the new request. It will also destroy all disk requests from the old client requests. All the objects that are resident are sent to the client from the aborted request. If the server would not destroy previous requests then new prefetch request, determined by the new context, would have to wait for old prefetch request to be served. The old requests could already be out-of-context and the savings potential of the new request is reduced. *OSAbortPrefNotDem* will only abort previous prefetch request but no demand requests.

7. *OSServSendDirect*

   The server sends the objects of the arriving disk pages directly to the client. This means one request could result in many initiated transfers from the server. Considering the fact that the disk is the slowest system part and that the network is much faster, nowadays, it could be advantageous to send every set of incoming objects separately to the client. Especially when the client is only interested in a small set of objects it does not have to wait until all objects are at the server side.

6.3 System Environment

For the performance evaluation we used the simulation language C++Sim [Little and McCue, 1993]. We used the same simulation that we have described in Section 5.4.1. In addition, we created two processes for prefetching: one for sending requests to the server and one for receiving data from the server. Both processes run in parallel with the client process. We also created two object buffers, one at the client and one at the server.

Table 6.1 shows the shared performance settings of both object and page server, Table 6.2 shows the values of the page server and Table 6.3 the values from the object server.

The cost of the network transfer depends on the number of bytes to be transferred. These variable costs are split, according to a measured percentage, into

[4] This is an optimistic value. We assume disk pages to be stored in clusters which reduces the average seek time.

Parameter	μs
Object processing time	1200
Fixed network cost for one transfer	1557
Variable network time for one transfer (per KB)	132
Var. network time client/network/server in %	45%/18%/37%
Client message send	267
Server message receive	156
Client message receive	156
Average disk access (seek+transfer)[4]	5615
Client server-request processing	1530

Table 6.1: Shared performance parameter of object/page server.

Parameter	μs
Page fetch time	12000
Server processing (page resident)	1359
Server processing (page not resident)	1664

Table 6.2: Page server performance specification.

processing at the client side (for receiving data), at the network (for transferring data) and at the server (for sending data). Every network data transfer has also an associated message block at the beginning which is read before the data. The total network cost is then ascertained by the variable cost plus the fixed network cost and message cost.

The page server has a page fetch time of 12 ms. If the object server fetches the same number of objects from the same page as the page server then its fetch time will be 12.1 ms (slightly higher because of the object overhead involved). At the server we distinguish between costs for processing resident or non-resident pages. If the page is not resident there is an overhead involved to communicate with the disk thread. A prefetch in a page server system must be started 10 processing objects ahead to achieve the total amount of savings. This is also true for the object server system if it fetches only one page from disk.

For the object server we have a low processing cost for the initial processing of a receipt. Moreover, we hold the server for processing, e.g. buffer management, before the disk request and after the disk request, e.g. auditing the page.

The benchmark structure we used for this test is a simple tree structure. Branch objects have two pointers to other objects. At each level in the depth of

Parameter	μs
Initial processing	18
Call disk	105
Processing non-resident page	790
Object lookup	5
Processing after disk arrival	978

Table 6.3: Object server performance specification.

the tree structure one branch object alternates with an object which has only one pointer to another object. Every branch object has an associated probability which is varied from 0.5 to 1.0. The navigation through the object graph was controlled by a *draw-operator*[5] There are 62 objects on a disk page (8 KB) and the object size is 132 bytes. A depth-first traversal would access 10 objects from the same page and the 11th would be an object from the next page. We assume perfect clustering for the objects on disk pages. Prefetching is only used to prefetch the objects/pages at the page border. We also assume the buffer space to be infinite and we do not consider any locking in our tests.

6.4 Performance Evaluation

6.4.1 Page Server Result

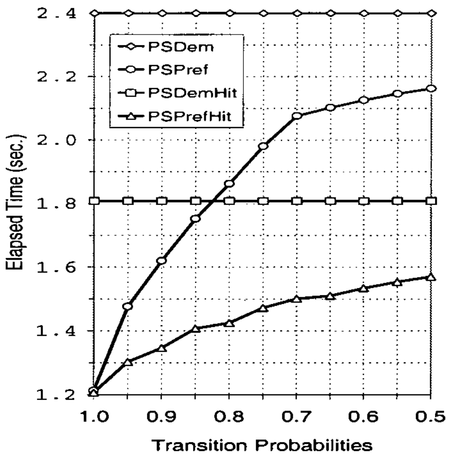


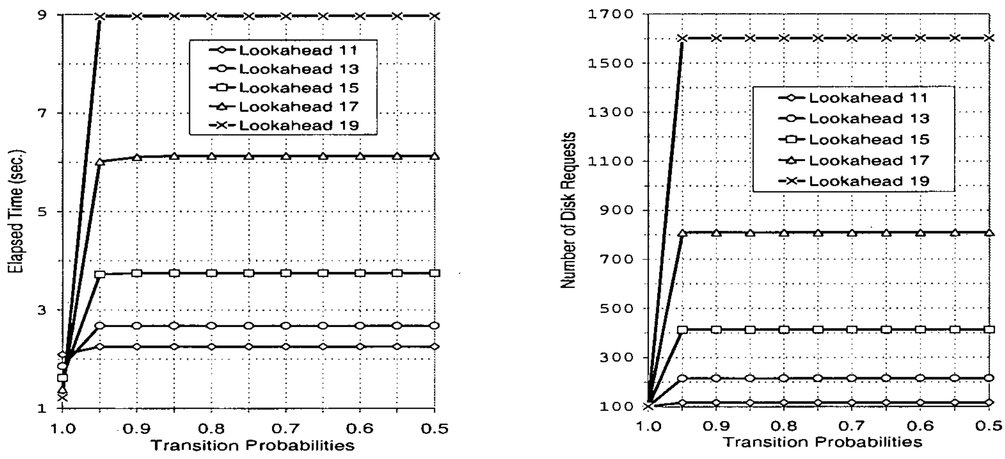
Figure 6.1: Page server result.

[5] Given a probability value it decides to continue navigation with reference 1 or 2.

Figure 6.1 shows the result of the page server applications. The performance of the demand applications is independent of the tps. At the probability of 1.0 *PSPref* is almost as good as *PSPrefHit*. With lower tps the prediction is less accurate which causes more incorrect prefetches and the elapsed time increases.

6.4.2 Object Server Result

The result of the object server with a threshold of 0.0 is presented in Figure 6.2. The number of the applications with the lookahead parameter indicates the depth of the object graph to be prefetched. At the tp of 1.0 *Lookahead 19* performs best. A lower lookahead reduces the amount of savings. At the other tps (0.95 down to 0.5) the elapsed times for all versions are immutable. The high lookahead applications perform very badly since they fetch a lot of pages from disk and consume a lot of server time.



(a) (b)

Figure 6.2: Object server result with threshold 0.0. In Figure (a) the result is measured in elapsed times and in Figure (b) in the number of disk requests.

The explanation for these times can be found in Figure 6.2(b). This chart shows the number of disk requests of the different applications and its shape is similar to the graph of Figure 6.2(a). The number of disk requests determines the elapsed time of the applications. All the main differences of single components can be found in Table 6.4. The *Lookahead 19* application always prefetches 16 pages from the current location of client processing and accesses only one of these pages. The number of server and disk waits means the number of times a request has to wait in a queue until it can be served.

Parameter	Lookahead 11	Lookahead 15	Lookahead 19
Server processing time	0.3	1.07	3.74
Disk processing time	0.65	2.32	8.99
Number of object requests	6292	23414	73346
Number of server waits	0	0	493
Number of disk waits	0	297	1583

Table 6.4: Object server result with threshold 0.0.

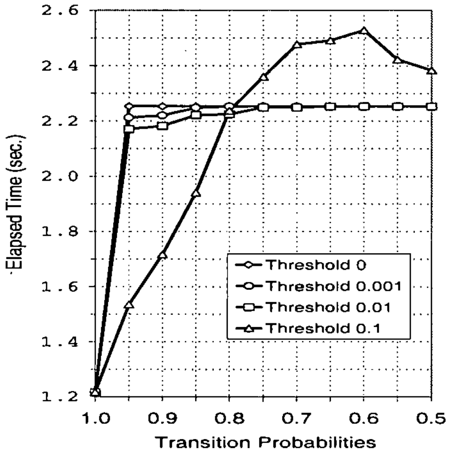


Figure 6.3: Object server with all four thresholds.

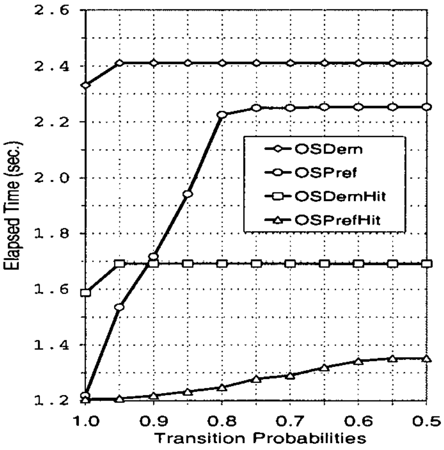


Figure 6.4: Final object server result.

Figure 6.3 shows the results of all four threshold applications. All applications with a threshold smaller than or equal to 0.01 have the same elapsed time between the tps of 0.75 down to 0.5 and at 1.0. At the other probabilities these applications vary slightly because of the different number of disk and object requests. A higher threshold reduces the number of object fetches but if it is too high it also aggrandises the number of demand fetches (e.g. threshold 0.1 has 4 additional demand fetches). In Figure 6.3, we see that *Threshold 0.1* performs well for tps lower than 0.8 because of the high probability relationships but then deteriorates due to a higher number of demand fetches. The *Threshold 0.01* application shows a high improvement between 0.95 and 0.85. The main difference of the application with the 0.95 tp can be seen in Table 6.5. The *Threshold 0.01* application starts the prefetch just two objects before access whereas *Threshold 0.1* has the best performance with a prefetch distance of 10 objects. The *Threshold 0.01* application has a short demand fetch time but a long prefetch wait time as it starts the prefetch too late. It fetches many objects with a few disk requests. The *Threshold 0.1* application prefetches more incorrect pages because of the prefetch distance but prefetches always arrive before access.

Parameter	Threshold 0.01	Threshold 0.1
Prefetch distance	2	10
Demand fetch time	0.109	0.335
Prefetch wait time	0.862	0.000
Number of disk requests	109	127
Number of object requests	3094	1386
Number of obj. demand group req.	5	27

Table 6.5: Object server result with transition probability 0.95 testing the 0.01 and 0.1 threshold applications.

In Figure 6.4 we see the final result of the object server performance. The *OSDem* has a constant performance; only at the transition probability of 1.0 is it slightly better because of fewer object fetches. *OSPref* has a sharp increase in performance until the probability of 0.75 and then stays constant. The explanation for this is the same as the difference between the *Threshold 0.01* and *Threshold 0.1* application in Figure 6.3. Before the 0.75 tp the prefetch distance is 10 and savings are high; after 0.8 the distance is 2. *OSDemHit* shows similar shape as *OSDem*. *OSPrefHit* performs best at all tps. Its elapsed time increases slightly with lower tps because of lower object destination probabilities and a later start of the prefetch.

6.4.3 Object and Page Server Comparison

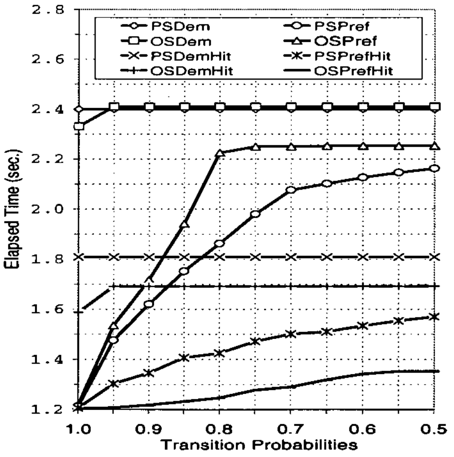


Figure 6.5: Object/page server comparison.

In Figure 6.5 all previously presented results are compiled into one graphic. *OSPref* is worse at every tp than *PSPref.* The highest difference is at 0.8 when *OSPref* uses a prefetch distance of 2 and *PSPref* still achieves the best result at a distance of 8. The result is different for the applications where the pages are resident at the server. *OSPrefHit* is able to select all the important objects for an object group without doing any expensive page fetches whereas *PSPrefHit* is restricted to objects of one page. *PSDem* and *OSDem* are almost identical: at 1.0 *OSDem* is slightly better because of less object fetches and after 1.0 *PSDem* improves as it has less overhead in object management. *OSDemHit* clearly performs better than *PSDemHit* in all probabilities.

6.4.4 Object Server Performance Optimisations

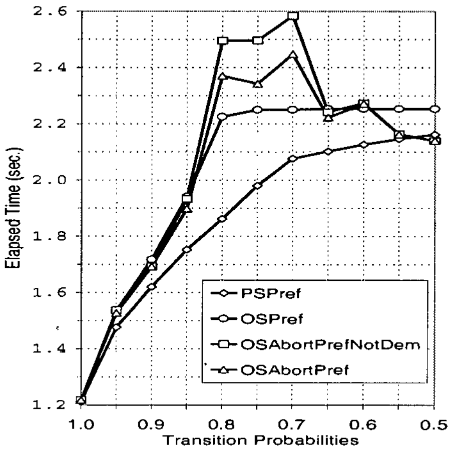


Figure 6.6: Server prefetch abort.

In the previous object prefetch technique the client sends many prefetch requests to the server. The server can only respond to such requests once all of the required objects are resident in its address space. Consequently, there are performance advantages if the server handles multiple requests concurrently from this point the server can do useful work whilst awaiting the arrival of pages from the disks. Some of these requests could be out of date, i.e. the applications' navigation is ahead of the prefetch request or the application changed its navigation totally.

We developed the two algorithms *OSAbortPref* and *OSAbortPrefNotDem,* shown in Figure 6.6, that abort all previous requests from the client at the server on the receipt of a new request. Every new client request contains all the relevant objects for the client processing; some of the objects in the new request could be part of older requests. Therefore we are able to abort all older requests. The server will abort all the disk requests from a client request and the client request itself. *OSAbortPref* also aborts previous demand requests from the client whereas *OSAbortPrefNotDem* only aborts previous prefetch requests. The performance of both abort versions are close to the *OSPref* version. Between the tps of 0.7 and 0.8 both abort versions perform badly. The reason for these performance characteristics can be gleaned from Table 6.6.

Parameter	No Abort	Abort
Demand fetch time	0.109	0.097
Prefetch wait time	0.941	1.151
Total server time	0.354	1.850
Total disk time	0.646	2.184
Number of disk requests	115	389
Number of object requests	6090	15101
Number of prefetch group requests	99	199

Table 6.6: Object prefetching with and without abort. Transition probability: 0.7, prefetch distance: 2 and threshold: 0.01.

The abort prefetch application does not check which objects are currently prefetched. It computes the relevant objects from the current context and sends this request away. This approach obviously increases the number of object requests. The server and disk processing time is also higher because of the processing overhead at the server. More object requests involve more disk requests and the disk system is busy with requests that are later not needed because of an abort. Every time a served disk request arrives at the server, the server will check if the request is aborted. If yes, it sends all the objects that are resident to the client and reduces the open request list from the client. The overhead for the server processing and the higher number of the network transfers increases the server time. *OSAbortPrefNotDem* and *OSAbortPref* do better than *PSPref* and *OSPref* at the 0.5 probability but cannot improve performance at the other probabilities.

The object server showed its superiority when all pages are resident at the server. For times when the server workload is low the server could prefetch pages from disk according to the prefetch information from a client. *OSServPref* starts a disk request after the service of a prefetch request. The server prefetches the highest probability page of an additional lookahead that is not in the current

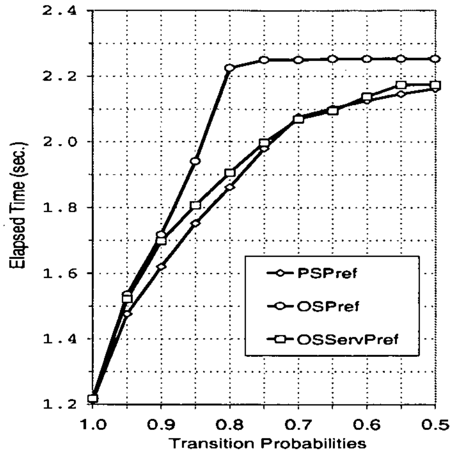


Figure 6.7: Server prefetching.

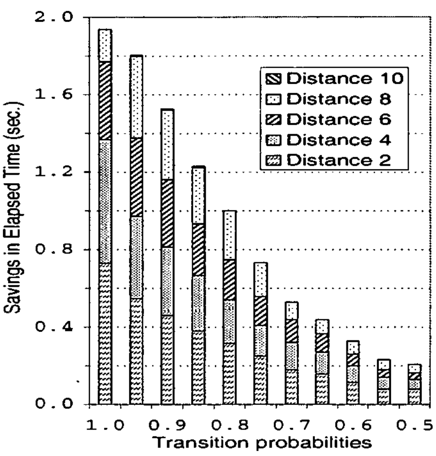


Figure 6.8: Server prefetch improvements for threshold 0.0.

request. Server prefetches have an extra low priority queue at the disk. If the queue still has requests from the same client it will delete all old requests. Figure 6.7 demonstrates that *OSServPref* performs much better than *OSPref* and very close to *PSPref* at higher tps. Of course, *PSPref* could also do server prefetching. In the current implementation *OSServPref* fetches only one page at a time from the disk. Fetching multiple pages could provide an even better result. Figure 6.8 shows that most of the savings can be achieved at higher tps because these are associated with the best prediction possibilities. Also the applications with the shorter distances achieve the best savings. The distance 10 application cannot achieve any improvement.

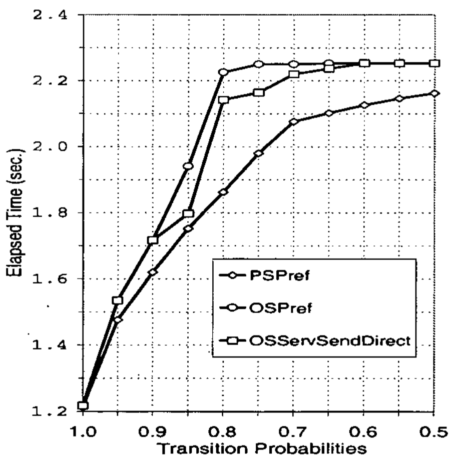


Figure 6.9: Direct sending of pages.

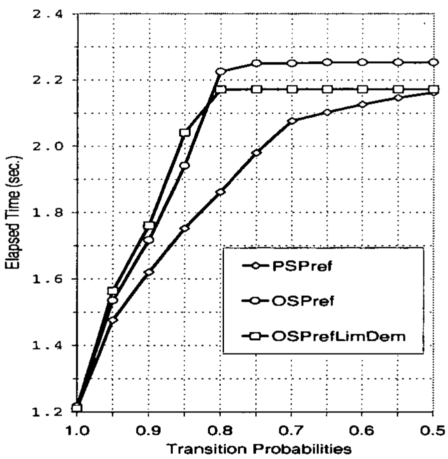


Figure 6.10: Prefetching with a limited demand fetch.

Another performance optimisation for the object server is the separate sending of pages to the client after receiving them from disk. This disk access is the most expensive part of the object fetch. The client could be waiting already for objects of that page and therefore it would make sense to start a network transfer after the disk receipt. Figure 6.9 presents the result of this test. *OSServSendDirect* shows a good improvement between the tps of 0.65 and 0.85 but cannot achieve the result of *PSPref.* The best improvement is at 0.85. Table 6.7 shows the differences of a 0.85 transition probability version. The prefetch wait time of the send early version is much lower because this version does not wait for the complete service at the server. The disadvantage of the send early application is that it increases the network service time with a fixed cost for every message. The server time is also increased because of the encoding of the message.

Parameter	Normal (sec.)	Send Direct (sec.)
Prefetch wait time	0.154	0.009
Total network time	0.049	0.068
Total server time	0.424	0.462

Table 6.7: Direct sending of pages at transition probability 0.85. Prefetch distance: 10 and threshold: 0.1.

In general every demand read requests all the objects according to a depth value from the server without considering the location of the objects in pages. *OSPrefLimDem* limits the object transfer to objects that are located on the same page as the faulted object. Figure 6.10 shows the result of *OSPrefLimDem* application. It performs worse than *OSPref* at tps of less than 0.85 but then performs better than *OSPref* at higher tps. At tp below 0.8 *PSPref* fetches many objects in the demand fetch unnecessary which can be avoided with *OSPrefLimDem.* However, *OSPrefLimDem* cannot achieve the performance of *PSPref.* At tps greater than 0.8, the better prediction possibilities combined with higher PODs give *OSPref* better conditions for prefetching.

6.5 Summary

We compared the performance of a prefetching page server with that of a prefetching object server. We simulated the object access pattern by assigning tps to the object relationships. According to the tps we compute the access probability of pages and objects. The key findings of this simulation study are:

- The prefetching page server outperforms the prefetching object server due to the higher number of disk requests of the object server.
- If all pages are resident at the server the prefetching object server performs better because it can select the group of objects to be prefetched.
- Applications that abort previous requests from the same client offer no performance advantage. Only at the tps around 0.5 did the prefetching abort applications show encouraging results.
- Combined client-initiated and server-initiated prefetching is attractive for object servers.
- An object server should limit the number of objects in a demand fetch if server pages are not resident and extend the number of object fetches with prefetching.
- If an object group request involves many pages to be fetched from disk, the direct sending of the page to the client is most efficient.

This chapter completes the research work. In the next chapter we will summarise all chapters of this thesis and give an outlook to future work.

Chapter 7

Conclusions

This final chapter will consolidate all of the material presented thus far and focus the aims and achievements of this thesis. This will involve a summary of all of the earlier chapters in Section 7.1. In Section 7.2 we explicitly highlight the original contribution to knowledge which has been made. We discuss the technology trends on PMC in Section 7.3. Finally, in Section 7.4 we will offer a number of topics for further research. This will conclude the main body of our thesis.

7.1 Summary

In the thesis we discussed several aspects of prefetching algorithms for object-oriented databases. We now give a short summary of the main issues that have been studied.

In Chapter 2 we gave an insight into the basic components of OODBMSs which have to be closely co-ordinated with a prefetching technique. The object identifier is used at prediction and prefetch time, and consequently has a big impact on the computation speed and occupied disk space of prediction information. The choice between an object or page server architecture has also an influence on the savings potential of prefetching which has been studied in Chapter 6. Another strong dependency exists between clustering and prefetching which is the subject of Chapter 5.

Prefetching has been studied extensively over the last 20 years. Chapter 3 gave an overview of related work. We concentrated on subjects that are relevant to OODBMSs. There are several aspects of optimisation to reduce response time in a client/server environment:

1. Reduction of seek times.

2. Reduction of idle times.

3. Global resource optimisation.

4. Group requests for multiple pages.

5. Overlapping of CPU and I/O.

6. Parallel disk access.

Previous prefetching techniques in the area of OODBMS concentrated on loading a set of objects in the forward direction from the current object. The physical storage consideration of object relationships and the timing of a prefetch were neglected. In addition, no study took the probability of traversing object relationships into account.

We implemented a prefetch environment into the EXODUS storage manager in Chapter 4. We extended the database client with threads for predicting and prefetching data from the server. We also designed a structure-based prefetching technique called OSP. It looks for non-resident objects in the forward direction up to a depth determined by the time a page fetch and the expected time of object processing. It also assigns weights to pages by counting the number of objects that have references to that page. OSP has a low overhead and works effectively when the number of adjacent pages is small.

From ESM implementation we learned that the total reduction of elapsed time is dependent on the CPU-I/O ratio. We achieved a reduction of up to 23%. Using multiple prefetch threads can improve performance as a result of intensified parallelism at the client. The maximum number of all threads for prefetching and processing should not be higher than the number of processors available. Another finding was that a multiple-server architecture is more attractive for prefetching than a single server architecture, even where the single server has a power that is comparable to the combined power of the multiple servers. Finally, the percentage of incorrect prefetches is vital for the success of prefetching. The complex benchmark result substantiated that one incorrect prefetch was acceptable but two incorrect prefetches without additional client processing were unacceptable.

In Chapter 5 we developed a prefetch algorithm to compute the access probability of pages by analysing object relationships. We compute the probability of accessing a page and the mean time to access. The object relationships are modelled using a Discrete-Time Markov Chain and a method called *hitting times* is used to compute the page access probability. The simulation results show that high transition probabilities result in high page probabilities and consequently prefetching can reduce elapsed time drastically. In addition, pages with fewer out-going references to adjacent pages are easier to predict and ensure higher prefetch savings. Other key findings from the OO1 simulation are:

- All prefetch applications that consider page probabilities at the disk queue perform better than applications that do not.
- Lower cluster factors provide higher amounts of fetch time that can be reduced by prefetching.
- Prefetching only directly adjacent pages results in bigger improvements for prefetch applications with higher cluster factors.
- Prefetch accuracy and prefetch object distance are the most important parameters for fetch time reduction.
- A smaller number of buffer frames increases the savings potential of prefetching applications.
- Multiple parallel disks increase the performance of prefetching applications.

Finally, in Chapter 6 we compared the performance of a prefetching page server with that of a prefetching object server. The general result of the test was that the prefetching page server outperforms the prefetching object server due to fewer disk requests at the server. In a test where all the pages are resident at the server the prefetching object server performs better because it can select the group of objects to be prefetched.

7.2 Contribution

Now that we have summarised the conclusions of this thesis, we will explicitly highlight how our work makes an original contribution to knowledge. Accordingly, the following points are offered as the principal scholarly contributions that have been made by this thesis:

1. A classification of prefetching techniques according to their applied prediction method and other aspects.

2. An OODBMS implementation of a client/server prefetching architecture on multiprocessor machines. We evaluated the behavior and performance of multiple threads on multiprocessors.

3. We designed a new object structure-based prefetching technique, called OSP, which considers the weight of pages and the timing of a prefetch.

4. A theoretical study about the speedup of prefetch applications.

5. A prefetching algorithm PMC, which analyses the probability of object relationships, computes the page probability and mean time to access. From this basic idea we developed multiple deviated prefetch algorithms. We also developed a cost model for the benefit and extra costs of prefetching.

6. We evaluated the performance of a prefetching object server versus a prefetching page server.

7. We conducted the first experimental study that considers the probability of object traversal for prefetching.

7.3 Discussion

We have seen the good performance of prefetch algorithm PMC in Chapter 5, but in the future PMC will be even more valuable. CPU performance is improving quickly and the parallel use of CPUs also provides more processing power. The performance of disk is expected to rise only at a slow rate. This means that disk retrieval time will increase its percentage in page fetch time. This trend suggests that the scheduling of disk requests will be even more important.

Faster CPUs reduce the duration of the computation-intensive parts of a page fetch, and they also reduce the client processing time to be overlapped with the prefetch. Either the prefetch has to be started earlier or user waiting time is used for overlapping. On the other side applications also grow in their complexity which increases the overlapping time again.

Comparing the future performance development of disks and CPUs, indicates that powerful CPUs should be used to predict application access and thereby avoid slow disk processing. CPUs of the future generation tend be idle most of time: this idle time is perfect for off-line prediction. A higher amount of disk requests gives the disk optimiser also the chance to order requests and take advantage of the high disk transfer rates.

Another future trend is that network transfer times also improve at a high speed. This makes the transfer of prediction information cheaper. Besides the actual page transfer, the server sends updates on the object structure to the client and the client sends updated values of the page probabilities to the server. Therefore, information can often be piggy-backed to the data transfer.

7.4 Future Work

To complete the thesis we will present a few suggestions for further research. We have covered a lot aspects of prefetching, e.g. the interaction with clustering and buffer management; overlapping times for prefetching; prediction costs and the granularity of a prefetch. There is therefore much for further investigation. However, to contain the discussion, we have selected the following topics which we feel offer the most potential for expansion.

7.4.1 Integrated Multi-User Prefetching

The size of future applications and the amount of data every client will request from the server will rise. All the prefetch requests from multiple clients have to be integrated by a multiple user prefetching technique. The integration process is important at the server to decide about the relevance of all incoming prefetch requests. The client could provide information such as the mean time to access and the access probability to the server. The server then decides which client has to be served first. This is especially important for the order of the disk queue as we have seen in Section 5.4.3. Of similar importance is the replacement decision of pages in the buffer pool. The server can use the page probabilities for ordering disk requests but keeping them up-to-date for a replacement decision is difficult. Therefore the replacement strategy depends on the trade-off between a high-accuracy solution with page probabilities and the communication cost for providing this information.

7.4.2 Multithreaded ESM Server

The ESM server is not multithreaded at the moment but it can run many tasks as concurrent processes on one processor. If one task stalls for I/O another task is scheduled on the processor. The server also forks a new process, the disk manager, for every disk volume. Communication between server and disk manager is achieved by shared memory.

This current architecture does not exploit parallelism as much as it could do. It only allows to run one server process on a multiprocessor and the communication via shared memory is expensive. Using threads at the server would avoid both problems and would speed up our client/server prefetching architecture. Every client request would be performed by a separate thread. When the thread is waiting for I/O, another thread can be executed on the processor. This architecture would be especially valuable for the idea presented in Section 7.4.1.

7.4.3 Noise Influence on PMC

In a multi-user environment many clients make read and write requests to the server. Many requests increase the average page fetch time and influence the time when a prefetch has to be started. In our model it affects the right setting of POD_{max}. The server could give the client information about the current workload and the client considers it in the prefetch decision. The problem is the object relationship patterns are often complex and if we start the prefetch too early it may turn out to be incorrect. Therefore if we set POD_{max} too high it may cause more harm than good. Another problem may be a high fluctuation in the server workload and until the client is notified many requests are wrongly timed.

Another noise parameter is frequency of updates. Updates on data values are no problem but updates on pointers are problematic. If pointers are updated the current prediction information of the equivalence class have to be invalidated and to be rebuilt. As long as the information is invalidated no prefetch can be issued. The server has to send information of the new object relationship structure to all affected clients and the clients have to re-compute the prediction information. Therefore a large number of updates on pointers increases the prediction computation time at the client, the network workload and the server processing time. In general, we want to investigate all aspects of higher workloads induced by a multi-user environment.

7.4.4 Buffer Replacement based on Probabilities

In Chapter 5 we learned that the probability computation of direct adjacent pages is feasible with up-to-date computers. The computation for multiple depths of adjacent pages may be difficult at the moment. In the future the processing power of CPUs will increase drastically which will enable us to compute future access much further ahead. The enhanced information could be used for prefetching and buffer replacement. Every page has an associated probability depending on the current position in the navigation. If a candidate page has a higher probability than the lowest probability of a page in the buffer pool then we would start the prefetch and replace the low probability page. The probability information is important for a replacement decision at the client and the server. At the server we have to integrate the navigation process of many clients.

Bibliography

Acharya, S. (1998). *Broadcast Disk: Disseminiation-based Data Management for Asymmetric Communication Environments.* PhD thesis, Department of Computer Science, Brown University.

Acharya, S., Franklin, M., and Zdonik, S. (1996a). Disseminating Updates on Broadcast Disks. In *Proc. of the 22th Int. Conf. on Very Large Data Bases,* pages 354–365, Bombay, India. Morgan Kaufmann Publishers.

Acharya, S., Franklin, M., and Zdonik, S. (1996b). Prefetching from a Broadcast Disk. In *Proc. of the 11th Int. Conf. on Data Engineering,* pages 276–287, New Orleans, LA, USA.

Acharya, S., Franklin, M., and Zdonik, S. (1997). Balancing Push and Pull for Data Broadcast. In *Proc. of the 1997 ACM Sigmod Conf.*, pages 183–194, Tucson, Arizona.

Agarwal, B. (1995). Configuring and Tuning the Data Cache in SQL Server 11.0. Sybase Engineering TechNotes.

Agrawal, R., Dar, S., and Gehani, N. (1993). The O++ Database Programming Language: Implementation and Experience. In [ICDE, 1993], pages 61–70.

Agrawal, R. and Gehani, N. (1989). ODE (Object Database and Environment): The Language and the Data Model. In [SIGMOD, 1989], pages 36–45.

Ahn, J. and Kim, H. (1997). SEOF: An Adaptable Object Prefetch Policy For Object-Oriented Database Systems. In *Proc. of the 13th Int. Conf. on Data Engineering,* pages 4–13, Birmingham, UK.

Albers, S., Garg, N., and Leonardi, S. (1996). Minimizing Stall Time in Single and Parallel Disk Systems. Technical Report MPI-I-97-1-024, Max-Planck-Institut für Informatik, Saarbrücken, Germany.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Proc. of the SJCC*, 31:483–485.

Andrews, T. and Harris, C. (1987). Combining Language and Database Advances in an Object-Oriented Development Environment. In *Proc. of the Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 430–440, Orlando, FL. ACM.

Arunachalam, M. and Choudhary, A. (1995). A Prefetching Prototype for the Parallel File System on the Paragon. In [SIGMETRICS, 1995], pages 321–323.

ASPLOS (1992). *Fifth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Boston, MA.

Atkinson, M., Bailey, P., Chisholm, K., Cockshott, W., and Morrison, R. (1983). An Approach to Persistent Programming. *The Computer Journal*, 26(4):360–365.

Baer, J.-L. and Chen, T.-F. (1991). An effective on-chip preloading scheme to reduce data access penalty. In *Proc. of the Int. Conf. on Supercomputing*, pages 176–186, Cologne, Germany.

Baier, J. and Sager, G. (1976). Dynamic Improvement of Locality in Virtual Memory Systems. *IEEE Transactions on Software Engineering*, 2(1):54–62.

Banatre, M., Issarny, V., Leleu, F., and Charpiot, B. (1997). Providing quality of service over the Web: a newspaper-based approach. *Computer Networks and ISDN Systems*, 29(8):1457–1465.

Bancilhon, F., Delobel, C., and Kanellakis, P. (1992). *Building an Object-Oriented Database System - The Story of O_2*. Morgan Kaufmann Publishers.

Barve, R., Kallahalla, M., Varman, P., and Vitter, J. (1997). Competitive Parallel Disk Prefetching and Buffer Management. In [IOPADS, 1997], pages 47–56.

Belady, L. (1966). A study of replacement algorithms for virtual storage. *IBM Systems Journal*, 5:78–101.

Bell, T., Cleary, J., and Witten, I. (1990). *Text Compression*. Prentice-Hall.

Bertino, E., Saad, A., and Ismail, M. (1994). Clustering techniques in object bases: a survey. *Data & Knowledge Engineering*, 12(3):255–275.

Bestavros, A. (1995). Using Speculation to Reduce Server Load and Service Time on the WWW. In *Proc. of the 1995 ACM CIKM Int. Conf. on Information and Knowledge Management.*, pages 403–410. ACM.

Bestavros, A. (1996). Speculative Data Dissemination and Service to Reduce Server Load, Network Traffic and Service Time. In *Proc. of the 1996 Int. Conf. on Data Engineering*, pages 180–189, New Orleans, Louisiana.

Bianchini, R. and LeBlanc, T. (1994). A preliminary evaluation of cache-missed-initiated prefetching techniques in scalable multiprocessors. Technical Report 515, University of Rochester, Computer Science Department, NY, 14627.

Biliris, A. and Panagos, E. (1995). A High Performance Configurable Storage Manager. In [ICDE, 1995], pages 35–43.

Burdorf, C. and Cammarata, S. (1990). Prefetching Simulation Objects in a Persistent Simulation Environment. In Antonio Guasch, editor, *Proc. of the SCS Multiconference on Object Oriented Simulation*, pages 68–74, San Diego, CA. Society for Computer Simulation.

Butler, H. (1987). *Persistent Lisp: Storing Interobject References in a Database*. PhD thesis, Computer Science Division, EECS Department, University of California at Berkeley.

Calder, B. and Grunwald, D. (1994). Fast and accurate instruction fetch and branch prediction. In *Proc. of the Int. Symp. on Computer Architecture*, pages 22–32, Chicago, IL.

Calder, B., Grunwald, D., Jones, M., Lindsay, D., Martin, J., Mozer, M., and Zorn, B. (1997). Evidence-Based Static Branch Prediction Using Machine Learning. *ACM Transactions on Programming Languages and Systems*, 19(1):188–222.

Cao, P. (1996). *Application-controlled File Caching and Prefetching*. PhD thesis, Department of Computer Science, Princeton University.

Cao, P., Felten, E., Karlin, A., and Li, K. (1995a). A Study of Integrated Prefetching and Caching Strategies. In [SIGMETRICS, 1995], pages 188–197.

Cao, P., Felten, E., Karlin, A., and Li, K. (1995b). Implementation and Performance of Integrated Application-Controlled Caching, Prefetching and Disk Scheduling. Technical Report CS-TR-493-95, Department of Computer Science, Princeton University.

Cao, P., Felten, E., Karlin, A., and Li, L. (1996). Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343.

Carey, M., DeWitt, D., Frank, D., Graefe, G.; Richardson, J., Shekita, E., and Muralikrishna, M. (1986a). The Architecture of the EXODUS Extensible DBMS. In Dittrich, K. and Dayal, U., editors, *Proc. of the ACM/IEEE Int. Workshop on Object-Oriented Database Systems*, pages 233–255, Pacific Grove, CA. IEEE Computer Society Press.

Carey, M., DeWitt, D., Franklin, M., Hall, N., McAuliffe, M., Naughton, J., Schuh, D., Solomon, M., Tan, C., Tsatalos, O., White, S., and Zwilling, M. (1994a). Shoring Up Persistent Applications. In [SIGMOD, 1994], pages 383–394.

Carey, M., DeWitt, D., Graefe, G., Haight, D., Richardson, J., Schuh, D., Shekita, E., and Vandenberg, S. (1990). The EXODUS Extensible DBMS Project: An Overview. In Zdonik, S. and Maier, D., editors, *Readings in Object-Oriented Database Systems*, pages 474–499. Morgan Kaufmann Publishers.

Carey, M., DeWitt, D., and Naughton, J. (1994b). The OO7 Benchmark. Technical Report Technical Report 1140, Computer Science Department, University of Wisconsin- Madison.

Carey, M., DeWitt, D., Richardson, J., and Shekita, E. (1986b). Object and File Management in the EXODUS Extensible Database System. In *Proc. of the Twelfth Int. Conf. on Very Large Data Bases*, pages 91–100, Kyoto, Japan.

Carey, M., DeWitt, J., and Naughton, J. (1993). The OO7 Benchmark. In [SIGMOD, 1993], pages 12–21.

Carey, M., Franklin, M., and Zaharioudakis, M. (1994c). Fine-Grained Sharing in a Page Server OODBMS. *SIGMOD Records*, 5:359–370.

Cattell, R. (1992). Object Operations Benchmark. *ACM Transactions on Database Systems*, 17(1):1–31.

Cattell, R., editor (1993). *The Object Database Standard ODMG-93.* Morgan Kaufmann Publishers.

Cattell, R. (1994). *Object Data Management.* Addison-Wesley.

Chan, C., Ooi, B., and Lu, H. (1992). Extensible buffer management of indexes. In *Proc. of the Eighteenth Int. Conf. on Very Large Data Bases*, pages 444–454, Vancouver, Canada.

Chang, E. and Katz, R. (1989). Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS. In [SIGMOD, 1989], pages 348–357.

Chang, E.-L. (1989). *Effective Clustering and Buffering in an Object-Oriented DBMS.* PhD thesis, Computer Science Division, EECS Department, University of California at Berkeley.

Chaudhuri, S., Ghandeharizadeh, S., and Shahabi, C. (1995). Avoiding Retrieval Contention for Composite Multimedia Objects. In *Proc. of the 21st Int. Conf. on Very Large Data Bases*, pages 287–298, Zurich, Switzerland.

Chee, C., Lu, H., Tang, H., and Ramamoorthy, C. (1997). Improving I/O Response Times Via Prefetching and Storage System Reorganization. In *The 21st Ann. Int. Computer Software and Applications Conf.*, pages 143–148, Washington, D.C.

Chen, P., Lee, E., Gibson, G., Katz, R., and Patterson, D. (1994). RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185.

Chen, T.-F. (1993). *Data prefetching for high-performance processors.* PhD thesis, Department of Computer Science and Engineering, University of Washington.

Chen, T.-F. and Baer, J. (1994). A Performance Study of Software and Hardware Data Prefetching Schemes. In *Proc. of the 21st Ann. Int. Symp. on Computer Architecture*, pages 223–232, Chicago, IL. IEEE Computer Society Press.

Chen, T.-F. and Baer, J.-L. (1992). Reducing Memory Latency via Non-blocking and Prefetching Caches. In [ASPLOS, 1992], pages 51–61.

Chen, T.-F. and Baer, J.-L. (1995). Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Transactions on Computers*, 44(5):609–623.

Cheng, J. and Hurson, A. (1991a). Effective Clustering of Complex Objects in Object-Oriented Databases. In [SIGMOD, 1991], pages 22–31.

Cheng, J. and Hurson, A. (1991b). On the Performance Issues of Object-Based Buffering. In [PDIS, 1991], pages 30–37.

Cho, S. and Cho, Y. (1996). Page Fault Behavior and Two Prepaging Schemes. In *Proc. of the 1996 IEEE 15th Ann. Int. Phoenix Conf. on Computers and Communications*, pages 15–24, New York.

Chou, H.-T., DeWitt, D., Katz, R., and Klug, A. (1985). Design and Implementation of the Wisconsin Storage System. *Software - Practice and Experience*, 15(10):943–962.

Cleal, D. (1996). Optimising Relational Database Access. *Object Expert*, 1(3):32–38.

Copeland, G. and Maier, D. (1984). Making Smalltalk a Database System. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 316–325, Boston, MA.

Cortes, T., Girona, S., and Labarta, J. (1997). Avoiding the Cache-Coherence Problem in a Parallel/Distributed File System. Technical Report UPC-CEPBRA-1996-13, Departament d'Arquitectura de Computadors, Universitat Politecnica de Catalunya, Barcelona.

Crovella, M. and Barford, P. (1997). The Network Effects of Prefetching. Technical Report 97–002, Computer Science Department, Boston University.

Cunha, C. (1997). *Trace Analysis and its Applications to Performance Enhancements of Distributed Informations Systems*. PhD thesis, Computer Science Department, Boston University.

Cunha, C. and Jaccoud, C. (1997). Determining WWW User's Next Access and Its Application to Pre-fetching. Technical Report TR-95-011, Computer Science Department, Boston University.

Curewitz, K., Krishnan, P., and Vitter, J. (1993). Practical Prefetching via Data Compression. In [SIGMOD, 1993], pages 257–266.

Dahlgren, F. and Stenström, P. (1995). Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiporocessors. In *Proc. IEEE Symp. on High-Performance Computer Archictecture*, pages 68–77, Raleigh, North Carolina.

Datta, A., Mukherjee, S., Viguier, I., and Veloo, R. (1995). PAPER (Prefetching Anticipatorily and Priority basEd Replacement): A High Performance, Low Overhead Buffer Management Scheme for Real-Time, Active Database Systems. Technical Report RTRG-TR-95-02, Department of Management Information Systems, University of Arizona.

Day, M. (1993). Object Groups May Be Better Than Pages. In Penny Storms, editor, *Fourth Workshop on Workstation Operating Systems*, pages 119–122, Napa, California. IEEE Computer Society Press.

Day, M. (1995). *Client Cache Management in a Distributed Object Database.* PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science.

Delobel, C., Lecluse, C., and Richard, P. (1995). *Databases: From Relational to Object-Oriented Systems*. International Thomson Publishing.

DeWitt, D., Maier, D., Futtersack, P., and Velez, F. (1990). A Study of Three Alternative Workstation-Server Architecture for Object-Oriented Database Systems. In [VLDB, 1990], pages 107–121.

DeWitt, David; Gray, Jim (1992). Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85–98.

Effelsberg, W. and Härder, T. (1984). Principles of Database Buffer Management. *ACM Transactions on Database Systems*, 9(4):560–595.

Farkas, K., Jouppi, N., and Chow, P. (1995). How useful are non-blocking loads, stream buffers and speculative executions in multiple issue processors? In [HPCA, 1995], pages 78–89.

Fischer, J. and Freudenberger, S. (1992). Predicting conditional branch directions from previous runs of a program. In [ASPLOS, 1992], pages 85–95.

Fleming, T., Midkiff, S., and Davies, N. (1997). Improving the Performance of the World Wide Web over Wireless Networks. In *GLOBECOM 97. IEEE Global Telecommunications Conf.*, pages 1937–42, Phoenix, AZ, USA.

Franklin, M., Carey, M., and Livry, M. (1993). Local Disk Caching for Client-Server Database Systems. In *Proc. of the Nineteenth Int. Conf. on Very Large Data Bases*, pages 641–654, Dublin, Ireland. Morgan Kaufmann Publishers.

Freedman, C. and DeWitt, D. (1995). The SPIFFI Scalable Video-on-Demand System. In *Proc. of the ACM SIGMOD/PODS95 Joint Conf. on Management of Data*, pages 352–363, San Jose, CA.

Fu, J. and Patel, J. (1991). Data prefetching in multiprocessor vector cache memoies. In *Proc. of the Intl. Symp. on Computer Architecture*, pages 54–63.

Garcia-Molina, H. and Salem, K. (1992). Main Memory Database Systems: An Overview. *IEEE Knowledge and Data Engineering*, 4(6):509–516.

Gassner, P., Lohman, G., Schiefer, K., and Wang, Y. (1994). Query optimization in the IBM DB2 family. Technical Report RJ9734, IBM Almaden Research Center.

GemStone (1991). Product Overview.

Gerlhof, C. (1996). *Optimierung von Speicherzugriffskosten in Objektbanken: Clustering und Prefetching*. PhD thesis, Faculty for Mathematics and Computer Science, University of Passau.

Gerlhof, C. and Kemper, A. (1994a). A Multi-Threaded Architecture for Prefetching in Object Bases. In *Proc. of the Int. Conf. on Extending Database Technology*, pages 351–364, Cambridge, UK.

Gerlhof, C. and Kemper, A. (1994b). Prefetch Support Relations in Object Bases. In *Proc. of the Sixth Int. Workshop on Persistent Object Systems*, pages 115–126, Tarascon, France.

Gerlhof, C., Kemper, A., Kilger, C., and Moerkotte, G. (1993). Partition-Based Clustering in Object Bases: From Theory to Practice. In *Foundations of Data Organization and Algorithms. Proc. of the Seventh Int. Conf.*, number 730 in Lecture Notes in Computer Science, pages 301–316, Chicago, IL, USA. Springer-Verlag.

Ghandeharizadeh, S., Ramos, L., Asad, Z., and Qureshi, W. (1991). Object Placement in Parallel Hypermedia Systems. In [VLDB, 1991], pages 242–254.

Gibson, G., Patterson, R., and Satyanarayanan, M. (1992). Disk Reads with DRAM Latency. In Penny Storms, editor, *Proc. Third Workshop on Workstation Operating Systems*, pages 126–131, Key Biscayne, FL. IEEE Computer Society Press.

Gokhale, V. (1997). Design of the 64-bit Option for the Oracle7 Relational Database Management System. *Digital Technical Journal*.

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley.

Gollapudi, S. and Zhang, A. (1998). Buffer model and management in distributed multimedia presentation systems. *Multimedia Systems*, 6(3):206–18.

Gornish, E., Granston, E., and Veidenbaum, A. (1990). Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proc. of the Int. Conf. on Supercomputing*, pages 354–368, Amsterdam, Netherlands.

Griffioen, J. and Appleton, R. (1993). Automatic Prefetching in a WAN. In *IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 8–13, Princeton, NJ.

Griffioen, J. and Appleton, R. (1994). Reducing File System Latency using a Predictive Approach. In *Proc. of the 1994 USENIX Conf.*, pages 197–207, Boston, MA.

Griffioen, J. and Appleton, R. (1995a). Improving File System Performance via Predictive Caching. In [PDCS95, 1995], pages 165–170.

Griffioen, J. and Appleton, R. (1995b). Performance Measurements of Automatic Prefetching. In [PDCS95, 1995].

Grimshaw, A. and Loyot, E. (1991). ELFS : Object-Oriented Extensible File Systems. Technical Report TR-91-14, Department of Computer Science, University of Virginia.

Grimsrud, K., Archibald, J., and Nelson, B. (1993). Multiple Prefetch Adaptive Disk Caching. *IEEE Knowledge and Data Engineering*, 5(1):88–103.

Hennessy, J. and Patterson, D. (1996). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers.

Hohenstein, U., Pleßner, V., and Heller, R. (1997). Evaluating the Performance of Object-Oriented Database Systems by Means of a Concrete Application. In *Proc. of the 8th Int. Workshop on Database and Expert Applications*, Toulouse, France.

Hornick, M. and Zdonik, S. (1987). A shared, segmented memory system for an object-oriented database. *ACM Transactions on Office Information Systems*, 5(1):70–95.

Horspool, R. and Huberman, R. (1987). Analysis and Development of Demand Prepaging Policies. *Journal of Systems and Software*, 7:183–194.

Hosking, A. and Moss, J. (1993). Object Fault Handling for Persistent Programming Languages: A Performance Evaluation. In *Proc. of the Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 288–303, Washington, DC.

HPCA (1995). *Proc. First IEEE Symp. on High-Performance Computer Architecture*. IEEE Computer Society Press.

IBM (1994). *DB2/6000 Version 2 and DB2/2 Version 2. Annoucement number 294–320.*

IBM (1997). *DB2 Administration Guide.*

ICDE (1993). *Proc. of the 9th Int. Conf. on Data Engineering*, Vienna, Austria.

ICDE (1995). *Proc. of the 11th Int. Conf. on Data Engineering*, Taipei, Taiwan.

IOPADS (1997). *Fifth Workshop on I/O in Parallel and Distributed Systems*, San Jose, CA.

Itasca Systems, I. (1991). ITASCA Technical Summary, Release 2.0.

Jacobsen, Q. and Cao, P. (1998). Potential and Limits of Web Prefetching Between Low-Bandwidth Clients and Proxies. Technical Report 1372, Computer Science Department, University of Wisconsin- Madison.

Jagadish, H., Lieuwen, D., Rastogi, R., Silberschatz, A., and Sudarshan, S. (1994). Dali: A High Performance Main Memory Storage Manager. In [VLDB, 1994], pages 48–59.

Jauhari, R., Carey, M., and Livny, M. (1990). Priority-Hints: An Algorithm for Priority-Based Buffer Management. In [VLDB, 1990], pages 708–721.

Jeon, H. and Noh, S. (1997). Improving Buffer Cache Performance with Prefetching: A Minimal Overhead Solution. Technical Report D&PS_TR_-E97702, Department of Computer Engineering, Hong-Ik University.

Jeon, H. and Noh, S. (1998). A Database Disk Buffer Management Algorithm based on Prefetching. In *Seventh Int. Conf. on Information and Knowledge Management*, Washington, DC.

Jeong, T., Ham, J., and Kim, S. (1997). A Pre-scheduling Mechanism for Multimedia Presentation Synchonization. In *Proc. IEEE Int. Conf. on Multimedia Computing and Systems*, pages 379–386, Ottawa, Canada.

Jiang, Z. and Kleinrock, L. (1997). Prefetching Links on the WWW. In *ICC'97*.

Jiang, Z. and Kleinrock, L. (1998). An Adaptive Network Prefetch Scheme. *Journal of Selected Areas in Communications*, 0(0).

Johnson, T. and Shasha, D. (1994). 2Q: a low overhead high performance buffer management replacement algorithm. In [VLDB, 1994], pages 439–450.

Joseph, D. and Grunwald, D. (1997). Prefetching using Markov Predictors. *Computer Architecture News*, 25(2):252–263.

Joseph, M. (1970). An analysis of paging and program behaviour. *The Computer Journal*, 13(1):48–54.

Kaehler, T. and Krasner, G. (1983). LOOM - Large Object-Oriented Memory for Smalltalk-80 Systems. In *Smalltalk-80: Bits of History, Words of Advice*, pages 251–270. Addison-Wesley.

Kaeli, D. and Emma, P. (1991). Branch history table prediction of moving target branches due to subroutine returns. In *Proc. of the Int. Symp. on Computer Architecture*, pages 34–42.

Kallahalla, M. and Varman, P. (1998). Improving Competitiveness of Parallel-Disk Buffer Management using Randomization. In *Proc. of 1998 Int. Conf. on Parallel Processing*, Minneapolis, Minnesota.

Karpovich, J., Grimshaw, A., and French, J. (1994). Extensible File Systems (ELFS) An Object-Oriented Approach to High Performance File I/O. In *Ninth Ann. Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 191–204, Portland, Oregon.

Keeton, K., Patterson, D., and Hellerstein, J. (1998). A Case for Intelligent Disks (IDISKs). *SIGMOD Records*, 27(3):42–52.

Keller, T., Graefe, G., and Maier, D. (1991). Efficient Assembly of Complex Objects. In [SIGMOD, 1991], pages 148–157.

Kemper, A. and Kossmann, D. (1993). Adaptable Pointer Swizzling Strategies in Object Bases. In [ICDE, 1993], pages 155–162.

Kemper, A. and Kossmann, D. (1994). Dual-Buffering Strategies in Object Bases. In [VLDB, 1994], pages 427–438.

Khoshafian, S. and Copeland, G. (1986). Object identity. In *OOPSLA*, pages 406–416, Portland, Oregon.

Kim, W., Garza, J., Ballou, N., and Woelk, D. (1994). Architecture of the ORION Next-Generation Database System. In Stonebraker, M., editor, *Readings in Database Systems*, pages 857–872. Morgan Kaufmann Publishers, second edition.

Kimbrel, T. (1997). *Parallel Prefetching and Caching*. PhD thesis, Department of Computer Science and Engineering, University of Washington.

Kimbrel, T., Cao, P., Felten, E., Karlin, A., and Li, K. (1996a). Integral Parallel Prefetching and Caching. In [SIGMETRICS, 1996].

Kimbrel, T. and Karlin, A. (1996). Near-optimal Parallel Prefetching and Caching. In *Proc. of the 37th Ann. Symp. Foundations of Computer Science*, pages 540–549, Burlington, Vermont.

Kimbrel, T., Tomkins, A., Patterson, H., B., B., Cao, P., Felten, E., Gibson, G., Karlin, A., and Li, K. (1996b). A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. Technical Report 96-09-01, Department of Computer Science and Engineering, University of Washington.

Klaiber, A. and Levy, H. (1991). An Architecture for Software-Controlled Data Prefetching. *Computer Architecture News*, 19(3):43–52.

Knafla, N. (1997a). A Prefetching Technique for Object-Oriented Databases. Technical Report ECS-CSG-28-97, Department of Computer Science, University of Edinburgh.

Knafla, N. (1997b). A Prefetching Technique for Object-Oriented Databases. In *Advances in Databases, 15th British National Conf. on Databases*, Lecture Notes in Computer Science, pages 154–168, London, United Kingdom. Springer-Verlag.

Knafla, N. (1997c). Predicting Future Page Access by Analysing Object Relationships. Technical Report ECS-CSG-35-97, Department of Computer Science, University of Edinburgh.

Knafla, N. (1997d). Speed Up Your Database Client with Adaptable Multithreaded Prefetching. In *Proc. of the Sixth IEEE Int. Symp. on High Performance Distributed Computing*, pages 102–111, Portland, Oregon. IEEE Computer Society Press.

Knafla, N. (1998a). An Adaptable Multithreaded Prefetching Technique for Client-Server Object Bases. *Cluster Computing*, 1(1):27–37.

Knafla, N. (1998b). Analysing Object Relationships to Predict Page Access for Prefetching. In *Proc. of the Eighth Int. Workshop on Persistent Object Systems: Design, Implementation and Use (POS-8)*, pages 160–170, Tiburon, California. Morgan Kaufmann Publishers.

Knafla, N. (1998c). Page versus Object Prefetching: A Performance Evaluation. Technical Report ECS-CSG-43-98, Division of Informatics, University of Edinburgh.

Kotz, D. and Ellis, C. (1990). Prefetching in File Systems for MIMD Multiprocessors. *IEEE Knowledge and Data Engineering*, 1(2):218–230.

Kotz, D. and Ellis, C. (1991). Practical Prefetching Techniques for Parallel File Systems. In [PDIS, 1991], pages 182–189.

Kraiss, A. and Weikum, G. (1997). Vertical Data Migration in Large Near-Line Document Archives Based on Markov-Chain Predictions. In *Proc. of the 23rd Int. Conf. on Very Large Databases*, pages 246–255, Athens, Greece.

Kraiss, A. and Weikum, G. (1998). Integrated Document Caching and Prefetching in Storage Hierarchies Based on Markov-Chain Predictions. *VLDB Journal*, 7(3):141–162.

Kratzer, K., Wedekind, H., and Zörntlein, G. (1990). Prefetching – A Performance Analysis. *Information Systems*, 15(4):445–452.

Krishnan, P. (1995). *Online Prediction Algorithms for Databases and Operating Systems*. PhD thesis, Department of Computer Science, Brown University.

Kroeger, T. and Long, D. (1996). Predicting File System Actions from Prior Events. In *Proc. of the USENIX 1996 Ann. Technical Conf.*, pages 319–328, San Diego, CA.

Kroeger, T., Long, D., and Mogul, J. (1997). Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In [USENIX, 1997], pages 13–22.

Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. (1991). The Objectstore Database System. *Communications of the ACM*, 34(10):50–63.

Lee, J. and Smith, A. (1984). Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Computer*, 17(1):6–22.

Lee, J.-H., Lee, M.-Y., Choi, S.-U., and Park, M. (1994). Reducing cache conficts in data cache prefetching. *Computer Architecture News*, 22(4):71–77.

Lee, K., Kallahalla, M., Lee, B., and Varman, P. (1997). Performance Comparison of Sequential Prefetch and Forecasting using Parallel I/O. In *Proc. of the IASTED Int. Conf. on Parallel and Distributed Computing and Networks*, Singapore.

Lee, K. and Varman, P. (1995a). Improving Parallelism in I/O Systems. In *Proc. of IEEE Singapore Int. Conf. on Networks / Int. Conf. on Information Engineering*, pages 210–214, Singapore.

Lee, K.-K. and Varman, P. (1995b). Prefetching and I/O Parallelism in Multiple Disk Systems. In *Proc. of the 1995 Int. Conf. on Parallel Processing*, pages 160–163, Urbana, IL.

Lei, H. and Duchamp, D. (1997). An Analytical Approach to File Prefetching. In *Proc. of the 1997 USENIX Ann. Technical Conf. (Anaheim CA, January 1997)*, pages 6–10.

Lilja, D. (1988). Reducing the Branch Penalty in Pipelined Processors. *IEEE Computer Society Press*, 21(7):47–55.

Lim, B. and Bianchini, R. (1996). Limits on the Performance Benefits of Multithreading and Prefetching. In [SIGMETRICS, 1996], pages 37–46.

Liskov, B., Adya, A., Castro, M., Day, M., Ghemawat, S., Gruber, R., Maheshwari, U., Myers, A., and Shira, L. (1996). Safe and Efficient Sharing of Persistent Objects in Thor. In *Proc. of the ACM SIGMOD/PODS96 Joint Conf. on Management of Data*, pages 318–329, Montreal, Canada.

Little, M. and McCue, D. (1993). Construction and Use of a Simulation Package in C++. Technical Report 437, Department of Computing Science, University of Newcastle, UK.

Liu, G. (1994). Exploitation of Location-dependent Caching and Prefetching Techniques for Supporting Mobile Computing and Communications. In *The 6th Int. Conf. on Wireless Communications*, Calgary, Canada.

Loon, T. and Bharghavan, V. (1997). Alleviating the latency and bandwidth problems in WWW browsing. In [USENIX, 1997], pages 219–230.

Madhyastha, T. and Reed, D. (1997). Input/Output Access Pattern Classification Using Hidden Markov Models. In [IOPADS, 1997], pages 57–67.

Maier, D., Graefe, G., Shapiro, L., Daniels, S., Keller, T., and Vance, B. (1994). Issues in Distributed Complex Object Assembly. In Özsu, M., Dayal, U., and Valduriez, P., editors, *Proc. of the Int. Workshop on Distributed Object Management*, pages 165–181, Edmonton, Canada.

Maier, D., Stein, J., Otis, A., and Purdy, A. (1986). Development of an Object-Oriented DBMS. *ACM SIGPLAN Notices*, 21:472–482.

McAuliffe, M. and Solomon, M. (1995). A Trace-Based Simulation of Pointer Swizzling Techniques. In [ICDE, 1995], pages 52–61.

McFarlin, S. and Hennessy, J. (1986). Reducing the costs of branches. In *Proc. of the Int. Symp. on Computer Architecture*, pages 396–403.

McVoy, L. and Kleiman, S. (1991). Extent-like performance from a UNIX file system. In *Proc. of the Winter 1991 USENIX Conf.*, pages 33–43, Dallas, Texas.

Mohan, C., Pirahesh, H., Tang, G., and Wang, Y. (1993). Parallelism in relational DBMSs: Possible approaches and a DB2 implementation. Technical report, IBM Almaden Research Center.

Moreno, E., Kofuji, S., and Cintra, M. (1997). Prefetching and Multithreading Performance in a Bus-Based Multiprocessors with Petri Nets. In *Proc. of the 3rd Int. Euro-Par Conf.*, pages 1017–1024, Passau, Germany.

Moss, J. (1990). Design of the Mneme Persistent Object Store. *ACM Transactions on Information Systems*, 8(2):103–139.

Moss, J. (1992). Working With Objects: To Swizzle or Not to Swizzle? *IEEE Transactions on Software Engineering*, 18(8):657–673.

Mowry, T., Lam, M., and Gupta, A. (1992). Design and Evaluation of a Compiler Algorithm for Prefetching. In [ASPLOS, 1992], pages 62–73.

Ng, R. and Yang, J. (1994). Maximizing Buffer and Disk Utilizations for News On-Demand. In [VLDB, 1994], pages 451–462.

Norris, J. (1997). *Markov Chains*. Cambridge series on statistical and probabilistic mathematics. Cambridge Uni Press.

Objectivity (1994). Objectivity/DB Technical Overview.

Ohara, M. (1996). Producer-oriented versus consumer-oriented prefetching: A comparison and analysis of parallel application programs. Technical Report CSL-TR-96-695, Department of Electrical Engineering and Computer Systems, Stanford University.

OMG (1997). CORBAservices: Common Object Services Specification 2.1.

O'Neil, E., O'Neil, P., and Weikum, G. (1993). The LRU-K page replacement algorithm for database disk buffering. In [SIGMOD, 1993], pages 297–306.

ONTOS (1995). ONTOS Object Integration Server (ONTOS OIS) Integrating Objects with Relational Databases. Technical Overview.

Oracle (1997). Oracle's JDBC Drivers Accessing the Oracle RDBMS from Java. An Oracle Technical White Paper.

Padmanabhan, V. (1995). Improving World Wide Web Latency. Technical Report UCB/CSD-95-875, Computer Science Division, EECS Department, University of California at Berkeley.

Padmanabhan, V. and Mogul, J. (1996). Using Predictive Prefetching to Improve World Wide Web Latency. *ACM SIGCOMM Computer Communications Review*, 26(3).

Pai, V., Schaefer, A., and P.J., V. (1994). Markov analysis of multiple-disk prefetching strategies for external merging. *Theoretical Computer Science*, 128(1):211–239.

Pai, V. and Varman, P. (1992). Prefetching with Multiple Disks for External Mergesort: Simulation and Analysis. In *Proc. of the 8th Int. Conf. on Data Engineering*, pages 273–282, Tempe, Arizona.

Palacharla, S. and Kessler, P. (1994). Evaluting stream buffers as a secondary cache replacement. In *Proc. of the Int. Symp. on Computer Architecture*, pages 24–33, Chicago, IL.

Palmer, M. and Zdonik, S. (1990). Predictive caching. Technical Report CS-90-29, Department of Computer Science, Brown University.

Palmer, M. and Zdonik, S. (1991). Fido: A Cache That Learns to Fetch. In [VLDB, 1991], pages 255–264.

Patterson, R. (1997). *Informed Prefetching and Caching*. PhD thesis, Department of Computer Science, Carnegie Mellon University.

Patterson, R. and Gibson, G. (1994). Exposing I/O Concurrency with Informed Prefetching. In *3rd Int. Conf. on Parallel and Distributed Information Systems*, pages 7–16, Austin, Texas.

Patterson, R., Gibson, G., Ginting, E., Stodolsky, D., and Zelenka, J. (1995). Informed Prefetching and Caching. In *Proc. of the Fifteenth ACM Symp. on Operating Systems Principles*, pages 79–95, Copper Mountain Resort, Colorado. ACM.

Patterson, R., Gibson, G., and Satyanarayanan, M. (1993). A Status Report on Research in Transparent Informed Prefetching. Technical Report CMU-CS-93-133, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA.

PDCS95 (1995). *Parallel and Distributed Computing Systems*, Orlando, Florida.

PDIS (1991). *Proc. First Int. Conf. on Parallel and Distributed Information Systems*, Miami Beach, Florida.

Phalke, V. and Gopinath, B. (1995). A miss history-based architecture for cache prefetching. In *Proc. of the Int. Workshop on Memory Management (IWMM)*, pages 381–398, Kinross, UK.

Robinson, J. and Devarakonda, M. (1990). Data cache management using frequency-based replacement. *Performance Evaluation Review*, 18(1):134–142.

Rochberg, D. and Gibson, G. (1997). Prefetching Over a Network: Early Experience with CTIP. *Performance Evaluation Review*, 25(3):29–36.

Rogers, A. and Li, K. (1992). Software Support for Speculative Loads. In [ASPLOS, 1992], pages 38–50.

Ross, S. (1997). *Introduction to Probability Models*. Academic Press.

Rubine, D., Dannenberg, R., Anderson, D., and Neuendorffer, T. (1994). Low-Latency Interaction through Choice-Points, Buffering and Cuts in Tactus. In *Proceedins of the Int. Conf. on Multimedia Computing and Systems*, pages 224–233, Boston, MA.

Sacco, G. and Schkolnick, M. (1986). Buffer management in relational database systems. *ACM Transactions on Database Systems*, 11(4):473–498.

Salem, K. (1991). Adaptive Prefetching for Disk Buffers. Technical Report TR-91-46, Department of Computer Science, University of Maryland.

Salem, K. and Garcia-Molina, H. (1986). Disk Striping. In *Proc. of the 2nd Int. Conf. on Data Engineering*, pages 336–42, Los Angeles, CA.

Schek, H., Paul, H., Scholl, M., and Weikum, G. (1990). DASDBS project: Objectives, experiences and future prospects. *IEEE Knowledge and Data Engineering*, 2(1):25–43.

Seltzer, M., Chen, P., and Ousterhout, J. (1990). Disk Scheduling Revisited. In *USENIX Proc. C++ Conf.*, pages 313–324, Washington DC.

Semiconductor Industry Association (1997). The National Technology Roadmap for Semiconductors.

Shah, V. and Kumar, B. (1995). Cluster Alignment and Parallel Algorithm for Multiple Prefetch Adaptive Disk Cache. *CSI Communications*, 18(11):22–24.

Shekita, E. and Zwilling, M. (1990). Cricket: A Mapped, Persistent Object Store. In *Proc. of the Fourth Int. Workshop on Persistent Object Systems*, pages 89–102, Martha's Vineyard, USA.

SIGMETRICS (1995). *1995 ACM SIGMETRICS Joint Int. Conf. on Measurement and Modeling of Computer Systems. SIGMETRICS '95/ PERFORMANCE '95*, Ottawa, Canada. ACM Press.

SIGMETRICS (1996). *Conf. on Measurement and Modeling of Computer Systems*, Philadelphia, PA. ACM.

SIGMOD (1989). *Proc. of the ACM SIGMOD Int. Conf. on the Management of Data*, Portland, Oregon.

SIGMOD (1991). *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, Denver, USA.

SIGMOD (1993). *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, Washington, USA.

SIGMOD (1994). *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, Minneapolis, Minnesota.

Singhal, V., Kakkad, S., and Wilson, P. (1992). Texas: An Efficient, Portable Persistent Store. In *Proc. of the Fifth Int. Workshop on Persistent Object Systems*, pages 11–33, San Miniato, Italy.

Smith, A. (1978a). Sequential Program Prefetching in Memory Hierarchies. *IEEE Computer*, 11(12):7–21.

Smith, A. (1978b). Sequentiality and Prefetching in Database Systems. *ACM Transactions on Database Systems*, 3(3):223–247.

Smith, A. (1982). Cache Memories. *ACM Computing Surveys*, 14(3):473–530.

Smith, A. (1985). Disk Cache - Miss Ratio Analysis and Design Considerations. *ACM Transactions on Computer Systems*, 3(3):161–203.

Song, I. and Cho, Y. (1993). Page Prefetching Based on Fault History. In *Proc. of the USENIX Mach III Symp.*, pages 203–213, Santa Fe, USA.

SPARCworks (1995). *SPARCworks / iMPact: Tools for Multithreaded Programming*. SunSoft, Mountain View, CA, USA.

Staehli, R. and Walpole, J. (1993). Constrained-Latency Storage Access. *IEEE Computer*, 26(3):44–53.

Stewart, W. (1994). *Introduction to the Numerical Solution of Markov Chains.* Princeton.

Stonebraker, M., Rowe, L., and Hirohama, M. (1990). The Implementation of POSTGRES. *IEEE Knowledge and Data Engineering*, 2(1):125–142.

Tait, C. and Duchamp, D. (1990). Detection and Exploitation of File Working Sets. Technical Report CUCS-050-90, Computer Science Department, University of Columbia.

Talcott, A., Yamamoto, W., Serrano, M., Wood, R., and Nemirovsky, M. (1994). The impact of unresolved branches on branch prediction scheme performance. In *Prof. of the Int. Symp. on Computer Architecture*, pages 12–21, Chicago, IL.

Tan, M., Roussopoulos, N., and Kelley, S. (1995). The Tower of Pizzas. Technical Report UMIACS-TR-95-52, Department of Computer Science, University of Maryland.

Teng, J. and Gumaer, R. (1984). Managing IBM Database 2 buffers to maximize performance. *IBM Systems Journal*, 23(2):211–218.

Tomkins, A. (1997). *Practical and Theoretical Issues in Prefetching and Caching.* PhD thesis, Department of Computer Science, Carnegie Mellon University.

Tomkins, A., Patterson, R., and Gibson, G. (1997). Informed Multi-Process Prefetching and Caching. In *Proc. of the ACM Int. Conf. on Measurements and Modeling of Computer Systems*, pages 100–114, Seattle, Washington.

Touch, J. and Farber, D. (1994). An Experiment in Latency Reduction. In *Proc. of the IEEE INFOCOM'94*, pages 175–183, Toronto, Canada.

Treiber, R. and Menon, J. (1995). Simulation Study of Cached RAID5 Designs. In [HPCA, 1995], pages 186–197.

Trivedi, K. (1977). An Analysis of Prepaging. *Computing*, 22(3):191–210.

Tsangaris, M. and Naughton, J. (1992). On the Performance of Object Clustering Techniques. Technical Report 1090 - 1992, Computer Science Department, University of Wisconsin- Madison.

Tsangaris, M. and Naughton, J. F. (1991). A Stochastic Approach for Clustering in Object Bases. In [SIGMOD, 1991], pages 12–21.

Tullsen, D. and Eggers, S. (1993). Limitations of cache prefetching on a bus-based multiprocessor. In *Proc. of the Int. Symp. on Computer Architecture*, pages 278–288, San Diego, CA.

Tullsen, D. and Eggers, S. (1995). Effective cache prefetching on bus-based multiprocessors. *ACM Transactions on Database Systems*, 13(1):57–88.

USENIX (1997). *Proc. of the USENIX Symp. on Internet Technologies and Systems*, Monterey, CA, USA.

VanderWiel, S. and Lilja, D. (1997). When Caches Aren't Enough: Data Prefetching Techniques. *IEEE Computer*, 30(7):23–30.

Varman, P. and Verma, R. (1996). Tight Bounds for Prefetching and Buffer Management Algorithms for Parallel I/O Systems. In *Proc. 16th Symp. Foundations of Software Technology and Theoretical Computer Science*, pages 200–211, Hyderabad, India.

Versant (1992). How to Evaluate Object Database Management Systems.

Vigna, E. (1997). Optimizing POET - Application-Specific Benchmarks. POET White Paper.

Vitter, J. and Krishnan, P. (1991). Optimal Prefetching via Data Compression. In *Proc. 32nd Ann. Symp. on Foundations of Computer Science*, pages 121–130, San Juan, Puerto Rico. IEEE Computer Society Press.

VLDB (1990). *Proc. of the 16th Int. Conf. on Very Large Data Bases*, Brisbane, Australia.

VLDB (1991). *Proc. of the 17th Int. Conf. on Very Large Data Bases*, Barcelona, Spain.

VLDB (1994). *Proc. of the 20th Int. Conf. on Very Large Data Bases*, Santiago, Chile.

Voelker, G., Anderson, E., Kimbrel, T., Feeley, M., Chase, J., Karlin, A., and H.M., L. (1998). Implementing Cooperative Prefetching and Caching in a Globally-Managed Memory System. In *Proc. of the 1998 ACM SIGMETRICS Conf. on Performance Measurement, Modeling, and Evaluation*, pages 33–43, Madison, Wisconsin.

Wang, Z. and Crowcroft, J. (1996). Prefetching in World Wide Web. In *Proc. of the 1996 IEEE Global Telecommunications Conf.*, London, UK.

Wedekind, H. and Zoerntlein, G. (1986). Prefetching in Realtime Database Applications. In Zaniolo, C., editor, *Proc. of the ACM SIGMOD 1986 Conf. on the Management of Data,* pages 215–226, Washington, DC. ACM.

Weikum, G. (1989). Set-oriented disk access to large objects. In *Proc. of the IEEE Conf. on Data Engineering,* pages 426–433, Los Angeles, CA.

Weikum, G., B., N., and Paul, H.-B. (1987). Konzeption und Realisierung einer mengenorientierten Seitenschnittstelle zum effizienten Zugriff auf Komplexe Objekte. In *Proc. of the GI Conf. Database Systems For Office Engineering and Scientific Applications,* pages 212–230.

Weikum, G., Hasse, C., Mönkeberg, A., and Zabback, P. (1994). The COMFORT automatic tuning project. *Information Systems,* 19(5):381–432.

White, S. (1994). *Pointer Swizzling Techniques for Object-Oriented Database Systems.* PhD thesis, Computer Science Department, University of Wisconsin-Madison.

White, S. and DeWitt, D. (1994). QuickStore: A High Performance Mapped Object Store. In [SIGMOD, 1994].

Wilson, P. and Kakkad, S. (1992). Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware. In *Proc. of the 1992 Workshop on Object Orientation in Operating Systems,* pages 364–377, Dourdan, France.

Wilson, P. R., Mukherjee, S., and Kakkad, S. (1994). Anomalies and Adaptation in the Analysis and Development of Prepaging Policies. *Journal of Systems and Software,* 27(2):147–153.

Wu, K.-L., Yu, P., and Teng, J. (1994). Data Placement and Buffer Management for Concurrent Mergesort with Parallel Prefetching. In *Tenth Int. Conf. on Data Engineering,* pages 418–427, Houston, Texas.

Yamaguchi, S., Chinen, K., and Unoue, H. (1997). WWW Cache Management and Its International Deployment. In *Worldwide Computing and its Applications WWCA97,* pages 267–280, Tsukuba, Japan.

Zdonik, S., Franklin, M. Alonso, R., and Acharya, S. (1994). Are Disks in the Air Just Pie in the Sky? In *Proc.. Workshop on Mobile Computing Systems and Applications*, pages 12–19. IEEE Computer Society Press.

Ziv, J. and Lempel, A. (1978). Compression of individual sequences via variable rate coding. *IEEE Trans. on Information Theory*, 24:530–536.

Zivkov, B. and Smith, A. (1996). Disk Caching in Large Databases and Time-shared Systems. Technical Report CSD-96-913, Computer Science Division, EECS Department, University of California at Berkeley.

Index