A Prefetching Technique for Object-Oriented Databases

Technical Report ECS-CSG-28-97

Nils Knafla
Dept. of Computer Science, University of Edinburgh *

January 10, 1997

Abstract

We present a new prefetching technique for object-oriented databases which exploits the availability of multiprocessor client workstations. The prefetching information is obtained from the object relationships on the database pages and is stored in a *Prefetch Object Table*. This prefetching algorithm is implemented using multithreading. In the results we show the theoretical and empirical benefits of prefetching. The benchmark tests show that multithreaded prefetching can improve performance significantly for applications where the object access is reasonably predictable.

Keywords: prefetching, distribution, object-oriented databases, performance analysis, multithreading, application access pattern, storage management

1 Introduction

Two industry trends in the performance/price ratio of hardware systems have implications for the efficient implementation of object-oriented database management systems (OODBMSs) in a client/server computing environment. Firstly, the continuing fall in price of multiprocessor workstations means that such machines are cost effective as client hosts in OODBMSs. Secondly, although the performance/price ratios of both processors and disks are improving, the rate of improvement is greater for processors. Hence, the disk subsystem is emerging as a bottleneck factor in some applications. Recent advances in high bandwidth devices (e.g. RAID, ATM networks) have had a large impact on file system throughput. Unfortunately, access latency still remains a problem due to the physical limitations of storage devices and network transfer latencies.

In order to reduce access latency database systems cache pages in the buffer pools of both the client and server. Prefetching is an optimisation technique which reads pages into the database buffer before the application requests them. A successful prefetching technique is dependent on the accuracy of predicting the future access. If accuracy is high, performance can be improved due to the high cost of a page fetch. If accuracy is poor, the performance can actually decrease due to cache pollution, channel congestion and additional workload for the server.

The fate of OODBMSs will largely depend on their performance in comparison to relational databases. The simple tabular structures of relational databases and the set-at-a-time semantics of retrieval languages such as SQL make it easy to parallelise relational database servers. However, in an OODBMS the structures are complex and typically the retrievals chase pointers. Furthermore, in most OODBMSs the bulk of the processing occurs on the client. To our knowledge the only commercial database that uses prefetching to load a whole object tree in advance is GemStone. Using information on the object structure for prefetching could significantly improve the performance of OODBMSs.

In this report, we present a new prefetching technique for page server systems. The prediction information is obtained from the object structure on the database pages and is stored in a *Prefetch Object Table* (POT) which is used at run time to start prefetch requests. We implemented this technique in the EXODUS storage manager (ESM) [1]. We also incorporated Solaris threads into ESM to have the application thread and the prefetching thread running on different processors in the client multiprocessor.

In section 2 we give a classification of related work in the area of prefetching. Section 3 describes the design of the POT and a buffer replacement strategy.

*JCMB, King's Buildings, Edinburgh EH9 3JZ, UK, Tel.: +44 131 650 5962, Fax: +44 131 667 7209, Email: nk@dcs.ed.ac.uk

The system architecture of ESM and prefetching is described in section 4. In section 5 we present the results of the performance tests. Finally, in section 6 we conclude the work and give ideas for future work.

2 Related Work

The concept of prefetching has been used in a variety of environments including microprocessor design, virtual memory paging, compiler construction, file systems, WWW and databases. Prefetching techniques can be classified by many dimensions: the design of the predictor, the unit of I/O transfer in prefetching, the start time for prefetching, or the data structures for storing prediction information.

The design of the predictor is very important to allow prefetching to obtain high accuracy whilst minimising overhead for the prediction decision. Predictors can be further classified as *strategy-based*, *training-based* or *structure-based*.

Strategy-based prefetching is used internally (One Block Lookahead) (OBL) [9] or explicitly by a programmer's hint [14]. In the Thor [12] database, an object belongs to a *prefetch group*. When an object of this group is requested by the client, the whole object group is sent to the client.

Training-based predictors use repeated runs to analyse access patterns. Fido [13] is an example that prefetches by employing an associative memory to recognise access patterns within a context over time. Data compression techniques for prefetching were first advocated by Vitter and Krishnan [17]. The intuition is that data compressors typically operate by postulating a dynamic probability distribution on the data to be compressed. This distribution is used for prediction. In the area of microprocessors, Lee and Smith [11] designed a branch-prediction table which is a small associative memory that retains the addresses of recently executed branches and their targets.

Structure-based predictors (mostly used in OODBMSs) obtain information from the object structure. Chang and Katz's technique [3] predicts the future access from the data semantics in terms of inheritance and structural relationships, e.g. configuration and version history. An Assembly operator for complex objects to load sub-objects in advance was introduced by Keller [10]. The traversal was performed by different scheduling algorithms (*depth-first* and *breadth-first*).

In object-oriented databases the unit of I/O is an object (*Object Server*) or a page (*Page Server*). An object server prefetches an object or a group of objects ([3], [4]) and a page server prefetches one or more pages ([5], [7]). Another possible classification of prefetching is the time factor. Smith [16] proposed two policies: (a) prefetch only when a buffer fault occurred (*demand prefetch*), (b) prefetch at any time (*prefetch always*). Prediction information must be efficiently stored on disk and accessed in memory. This information can be stored in tables ([7], [8], [11]) or *Markov-chains* ([5], [13]).

3 The Prefetching Design

3.1 Prefetch Object Table

OODBMSs can store and retrieve large, complex data structures which are nested and heavily interrelated. Examples of OODBMS applications are CASE, CAD, CAM and Office automation. These applications consists of objects and relationships between objects containing a large amount of data. A typical scenario is laid out by the OO7 benchmark [2]. It comprises a very complex assembly object hierarchy and is designed to compare the performance of object-oriented databases. A key component of the benchmark is a set of composite parts. Each composite part has an associated graph of between 20 to 200 atomic parts. The higher-level design is an assembly hierarchy made up of complex assemblies at the top (which point to other assemblies) and base assemblies at the bottom (which point to composite parts). There are seven levels in the assembly hierarchy. The whole hierarchy requires 11 MB in the small database and 103 MB in the medium database on disk.

In a page server, like ESM, objects are clustered into pages. Good clustering is achieved when references to objects in the same page are maximised and references to objects on other pages are minimised.

The general idea of our technique is to prefetch references to other pages in a complex object structure net (e.g. OO7). 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 because of time and resource limitations. Instead, we follow the application processing of the object structure. We know which objects have *Out-Refs* and when we identify that the application is processing towards such an *Out-Ref-Object* (ORO) 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 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 ORO object which is necessary to prefetch a page early enough to finish the request before access. Let C_{pf} denote the cost of a page fetch and let C_{op} denote the cost of object preparation. The cost of object preparation is the ESM client processing time before the application can work on the object[1]. Then POD is computed as follows:

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

If the prefetch starts before the POD, a maximum saving is guaranteed, however, if it starts after the POD, but before access, there is a lower saving (see section 5.2).

2. Branch Objects

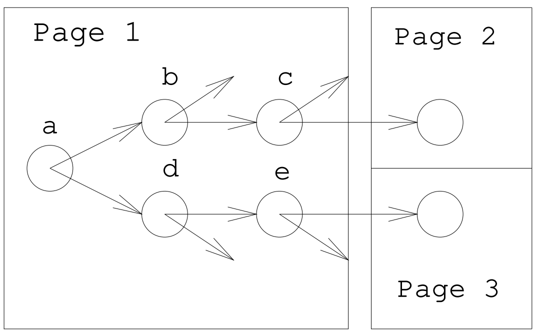

Figure 1: Object relationship

Fig. 1 shows an example of a complex object with 2 references. The application might access the root object a and then objects b and c. Then it appears that the application will follow this chain to page 2. In this case we would define object b and d as the PSOs because they are the first objects after the branch. When object b is accessed then page 2 is a candidate for prefetching, otherwise when object d is accessed page 3 is the candidate. This object branch technique reduces the number of adjacent pages to prefetch.

From the definition of PSOs, the branch objects are responsible for what to prefetch (page ids) and the POD is responsible for when to prefetch. This decision is always made on the individual case of the object structure. Sometimes a compromise between the two is needed, e.g. either prefetch right on time with lower accuracy (because an important branch object was not passed yet) or prefetch after the object distance with higher accuracy (a branch object was passed but there is not enough time to finish before access). The order of the prefetch requests is determined by the shortest path to OROs: urgent pages are prefetched earlier.

Each page of the database is analysed off-line, firstly to find the ORO and then to determine the PSO. The Analyser stores this information in the POT. Entries for a page are stored together and accessed by a hash table in memory. The overhead for this table is quite low because it only contains only a few objects from the page.

The POT is not only useful for complex objects, it can also be used for collection classes (linked list, bag, set or array) in OODBMSs. Normally the application works with a cursor on collections. With PSO and ORO it would be possible to prefetch the next page at the right time. It is important in our technique that the object size is smaller than the page size. If not, prefetching can be used to bring the whole object into memory.

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 [6] proposed a LRU replacement strategy with one chain for demand reads and one chain for prefetching. We also 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 System Architecture

4.1 The EXODUS Storage Manager

For the evaluation of the prefetching technique we chose ESM to implement this idea. The EXODUS Client/Server database system [1] 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 optimisers.

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

[1] Additionally we could use the expected amount of processing from the application

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.

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. Transactions are implemented using a shadowing and logging technique. Client and Server communicate via the socket interface. The client specifies the requested data in a message structure and sends it to the server. The server updates this structure and responds with the attached 8K page.

We did not make any changes to the software in order to get a fair comparison of the demand version with the new prefetching version.

4.2 The Prefetching Architecture

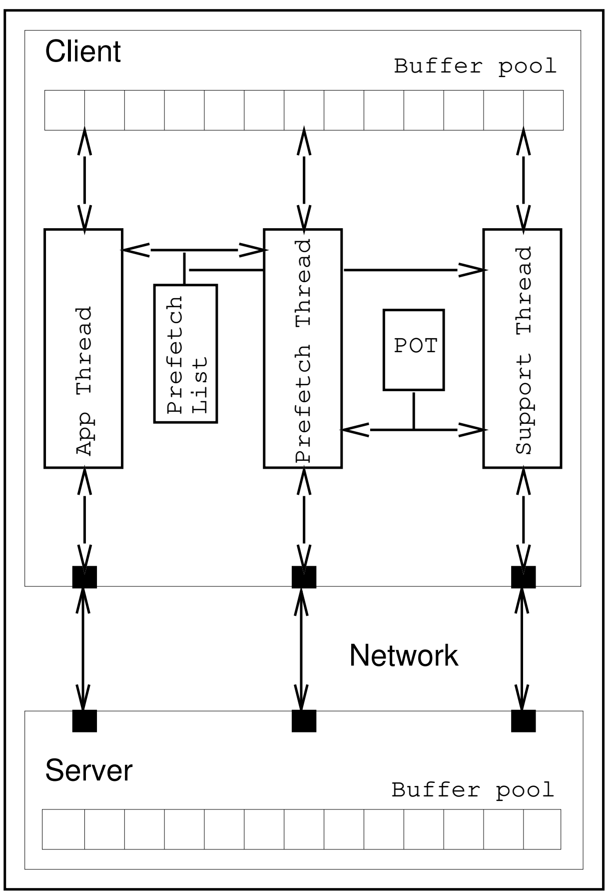

Figure 2: Architectural Overview

In this section we describe how prefetching is incorporated into ESM. For the concurrent execution of the application and the prefetch system we use the Solaris thread interface. Multithreading combined with prefetching has the benefits of:

1. Increased application throughput and responsiveness;
2. Performance gains from multiprocessing hardware (parallelism);
3. Efficient use of system resources.

As depicted in figure 2, 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. A *SupportThread* has the same task as the *PrefetchThread* with the only difference being that it is scheduled by the *PrefetchThread*. Each thread has one associated socket. The POT informs the *PrefetchThread* which pages are candidates for prefetching from the current processing of the application. The *Prefetch List* 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 not already resident. If not, the page is inserted in the Prefetch List and the request is sent to the server. The server responds with the demanded page and the client inserts the page in its buffer pool. Eventually the page is removed from the Prefetch List and inserted into the hash table of the buffer pool.

If the POT predicts multiple pages, *SupportThreads* help the *PrefetchThread*; this is useful when the prefetch object distance is short. The number of *SupportThreads* is determined by the number of simultaneous prefetch requests. Each *SupportThread* runs on its own LWP[2] and while one *SupportThread* blocks on I/O another *SupportThread* can insert its page into 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 Prefetch List. If the page has been prefetched the *AppThread* waits on a semaphore until the page arrives, otherwise it sends a demand request to the server.

The ESM server is not multithreaded[3] and performs each request sequentially. But the server forks a new process for the disk management. The server and disk manager communicate via shared memory. The server puts a request for a new page in a disk queue

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

[3]But ESM runs many tasks sequentially at the same time on one processor.

Parameter	Havra	Papa
SPARCstation	20 Model 612	10
Main Memory	192 MB	224 MB
Virtual Memory	624 MB	515 MB
Number of CPUs	2	4
Cycle speed	60 MHz	50 MHz

Table 1: Computer performance specification

Parameter	Craro	Faither
SPARCstation	20 Model 502	ELC (4/25)
Main Memory	512 MB	24 MB
Virtual Memory	491 MB	60 MB
Number of CPUs	2	1
Cycle speed	50 MHz	33 MHz

Table 2: Computer performance specification

and the disk manager reads the page from disk and copies it into the buffer pool of the server. Incorporating threads into the server would further improve the whole system performance and is part of future work.

For the parallel execution of threads 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. When either the *AppThread* or *PrefetchThread* are idle they wait on a semaphore. A condition variable[4] informs the *SupportThread* that there is a page to prefetch.

5 Performance Evaluation

5.1 System Environment

For the ESM server we need a machine (called Havra) configured with a large quantity of shared memory and enough main memory to hold pages in the buffer pool. To take full advantage of multi-threading we chose a four processor machine (called Papa) for the client. Table 1 presents the performance parameters of the machines. For some other tests on the client we used the machines in Table 2. The network is Ethernet running at 10Mb/sec. The disk controller is a Seagate ST15150W (performance parameters in table 3).

[4] A synchronisation variable that allows the users to specify arbitrary conditions on which to block

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

Table 3: Disk controller performance

5.2 Theoretical Results

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_o. Let C_{op} denote the cost of preparing one object for application access and let C_{oa} denote the cost of processing on the object from the application plus waiting time. C_o is calculated by:

$$C_o = C_{op} + C_{oa}$$

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

$$\begin{aligned} &\text{if } (C_o \cdot N_o \geq C_p) \\ &\qquad S_{or} = C_p \\ &\text{else} \\ &\qquad S_{or} = C_o \cdot N_o \end{aligned}$$

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

$$S_p = \sum_{i=1}^{n} S_{or}(i) \tag{1}$$

Finally, the saving of the total run is defined by S_r 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):

$$S_r = (\sum_{j=1}^{q} S_p(j)) - C_t - C_s \tag{2}$$

In our performance test we measured the run times for the demand version (RT_d) and for the prefetching version (RT_p). The savings are computed as follows:

CPU	Page Fetch	Savings in percent
10	2	17 %
2	2	50 %

Table 4: Savings in percent

$$savings = \frac{RT_d - RT_p}{RT_d} \cdot 100 \qquad (3)$$

But the percentage of savings is always dependent on the amount of processing required on the page. For example in table 4 a page fetch costs 2 time units. With 10 CPU time units the saving is only 17 % but with 2 CPU time units the saving is 50 %. Therefore we should use a more accurate formula to compute savings in percent. T_{sp} is the saved time with prefetching and T_p is the total time of all page fetches:

$$savings = \frac{T_{sp}}{T_p} \cdot 100 \qquad (4)$$

We did not use this formula as it requires a more complicated measurement technique.

5.3 Performance Measurements

For the evaluation of the prefetching technique we created a benchmark with complex objects. The structure of the benchmark should be complex with many relationships between objects, but not too complex for comprehension. 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 two different pages. 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 (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.

All time measurements are in real time. We also measured CPU time but this does not include the user or system waiting time. Although the tests were made in a multi-user environment the workload of the machines and the 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 different tests. In figures 4b to 8 the savings in percent are the savings of the prefetching version compared with the demand version times.

In figure 3 we compared the cost of one prefetch request to processing 101 objects in a page. The processing time of 61 milliseconds is about 5 times higher than the time to prefetch one page which took 11 milliseconds. Most of the processing is due to an audit function that calculates the slot space of the page. Expensive components of the prefetch request are the waiting time until the page arrives from the server and the subsequent reading of the page from the socket into memory. The waiting time is dependent on the network speed, the disk access from the server and the workload of the server.

In figure 4a and 4b we present the results of our benchmark. The tests on the client side were made on the four processor machine Papa, the two processor machine Craro and on the uniprocessor Faither. The server runs on the two processor machine Havra. The prefetching version is always faster than the demand version. The best result was made on the slow Faither machine because of its longer network connection and slower access to the socket. Papa has the same cycle speed as Craro but a higher saving. Craro and Papa have in opposition to Faither 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 see the maximum speedup of prefetching.

As mentioned in section 5.2 the saving of prefetching is dependent on the percentage amount of processing of the application. Having 101 objects on one page, we compared the run-time savings under varying object access rates from the application (from 10 objects to 100 objects accessed). Figure 5 shows 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. Increasing the number of objects gradually decreases the savings.

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 *SupportThreads* to prefetch simultaneously. We compared different prefetch object distance parameters to see under which conditions more *SupportThreads* are useful. The number of *SupportThreads* is dependent on the number of prefetch requests (in our test there is only one *SupportThread*). In figure 6 we can see that the savings get higher as the prefetch object distance gets smaller. The maximum speedup of 7.85 is with a distance of 1 because both pages are urgently needed and prefetched in parallel. With a distance of 10 the saving is only 6.9. Under no time restrictions, additional prefetch threads produce more overhead and are not useful.

The application fetches all objects by OID into memory without any processing on the objects or any waiting time. Also 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 and called it Inter-Reference Time (IRT). The results in figure 7 show that with more processing the savings in percent gets smaller. This is because the application is more and more dominated by CPU processing (as explained in section 5.2).

In the last test we studied the impact of wrong prefetches. Normally a wrong prefetch is not a serious problem when there is enough time and enough resources to make another prefetch in parallel computing. At the client both prefetch threads can run concurrently. The only problem is the increased workload for the server and the network. In this run we fetched 100 wrong pages from 200 page fetches. The other important parameter is the prefetch object distance. We used the distance of 1, 20 and 100. Recall that we always fetch 2 new pages from one page (now one correct and one incorrect page). The distance of 100 is sufficient to do a wrong prefetch, the distance of 20 is critical to do one prefetch right on time and with the distance of 1, the prefetch is always late. Figure 8 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 of almost 4 percent.

6 Conclusions and Future Directions

In this report we presented a prefetching technique for complex object relationships in a page server. The object structure of the database is analysed and stored in a *Prefetch Object Table*. During the run time of the application, this table is consulted to make the right prefetches on time. We used the object pointers to make predictions for future access. If the application follows such an object reference chain, we know the object that points to an object in the next page therefore making this page a candidate for prefetching. We also use the branch information of the complex relationships to predict the next pages as accurately as possible. If there are more prefetches to do at the same time we use more threads to get all prefetches before the application requires access.

In the implementation and performance tests we evaluated the prefetching technique. The prefetching version was 14% faster on the Papa machine, nearly 9% faster on Craro and 18% faster on Faither. Reducing the number of accessed objects in a page increases the savings. With an access of 20 objects in a page we achieved a saving of 45%.

If the objects are clustered on a page with only a few out-going pointers, prefetching can close this gap of clustering. If objects are not clustered together very complex relationships with many out-going pointers, prefetching could increase running time because of the bad prediction possibilities. Our technique is especially useful for complex objects with longer object chains and the collection classes of OODBMSs.

This work will be continued in several directions. Firstly, we will look at the object structure of real applications to see how our technique will perform. We will test different levels of complexity with varying numbers of *Out-Refs*. If the application makes many updates of pointer references we will evaluate how this effects the performance of POT. Also we will implement our buffer management algorithm to test repeated access to pages. Another possibility is to make the ESM server multithreaded. The problem is the synchronisation of the threads to access global data concurrently and safely. For example if many threads want to access the buffer pool waiting time will be increased. Concurrency control is another problem to be solved as we must avoid too many pages being locked at the client.

References

[1] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. L. Vandenberg. The EXODUS Extensible DBMS Project: An Overview. In S.B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems.* 1990.

[2] M. J. Carey, D. J. DeWitt, J. F. Naughton. The OO7 Benchmark. [15].

[3] E. E. Chang and R. H. Katz. Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS. In *Proceedings of the ACM SIGMOD Conference on the Management of Data (Portland, Oregon, May-June 1989).*

[4] J. R. Cheng and A. R. Hurson. On the Performance Issues of Object-Based Buffering. In *Proc. First International Conference on Parallel and Distributed Information System (Dec. 4-6, 1991, Miami Beach, Florida.* IEEE Computer Society Press, December.

[5] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical Prefetching via Data Compression. [15].

[6] C. S. Freedman and D. J. DeWitt. The SPIFFI Scalable Video-on-Demand System. In *Proceedings of the ACM SIGMOD/PODS95 Joint Conference on Management of Data (22-25 May 1995, San Jose, CA).*

[7] C. A. Gerlhof and A. Kemper. Prefetch Support Relations in Object Bases. In *Persistent Object Systems.* Proceedings of the Sixth International Workshop on Persistent Object Systems (Tarascon, Provence, France, 5th–9th September 1994).

[8] K. S. Grimsrud, J. K. Archibald, and B. E. Nelson. Multiple Prefetch Adaptive Disk Caching. *IEEE Knowledge and Data Engineering*, 5(1), February 1993.

[9] M. Joseph. An analysis of paging and program behaviour. *The Computer Journal*, 13(1), February 1970.

[10] T. Keller, G. Graefe, and D. Maier. Efficient Assembly of Complex Objects. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (Denver, USA, 1991).* ACM.

[11] J. K. F. Lee and A. J. Smith. Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Computer*, 17(1), January 1984.

[12] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, L. Shira. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proceedings of the ACM SIGMOD/PODS96 Joint Conference on Management of Data (3-6 June 1996, Montreal, Canada).*

[13] M. Palmer and S. B. Zdonik. Fido: A Cache That Learns to Fech. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases (Barcelona, Catalonia, Spain, 3rd–6th September 1991).*

[14] R. H. Patterson and G. A. Gibson. Exposing I/O Concurrency with Informed Prefetching. In *3rd International Conference on Parallel and Distributed Information Systems (Austin, Texas September 1994).* IEEE Computer Society.

[15] *Proceedings of the ACM SIGMOD International Conference on Management of Data (Washington, USA, 1993).*

[16] A. J. Smith. Sequentiality and Prefetching in Database Systems. *ACM Transactions on Database Systems*, 3(3), September 1978.

[17] J. S. Vitter and P. Krishnan. Optimal Prefetching via Data Compression. In *Proceedings 32nd Annual Symposium on Foundations of Computer Science (1-4 Oct. 1991, San Juan, Puerto Rico).* IEEE Computer Society Press.

Fig. 3 Cost of Clientprocessing and Pagefetch

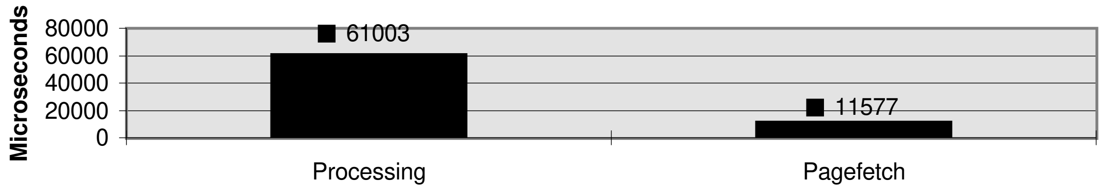

Fig. 4a Demand and Prefetch version

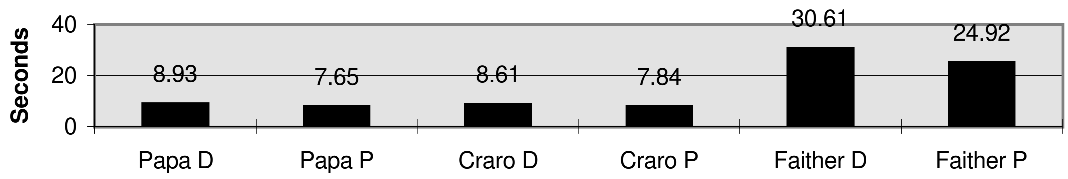

Fig. 4b Savings in Percent

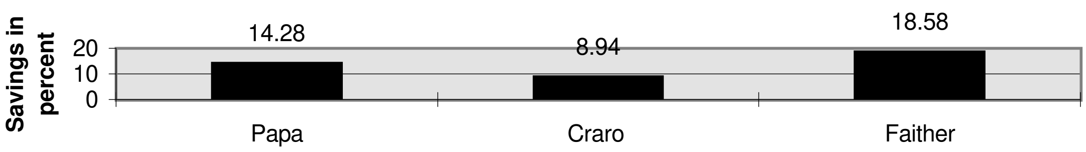

Fig. 5 Savings in percent with different object access rates

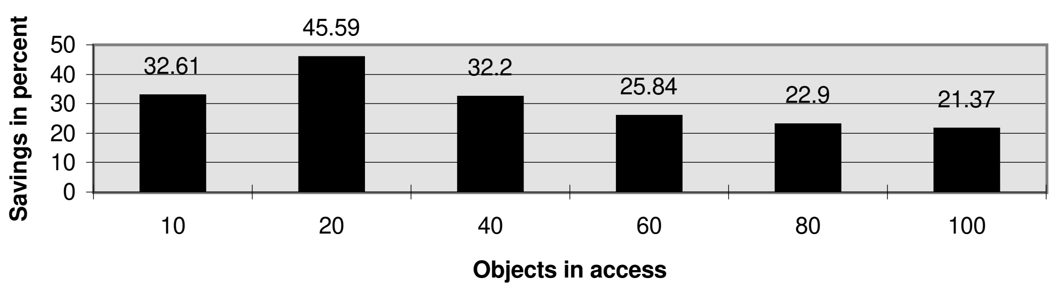

Fig. 6 Savings with 2 prefetch threads

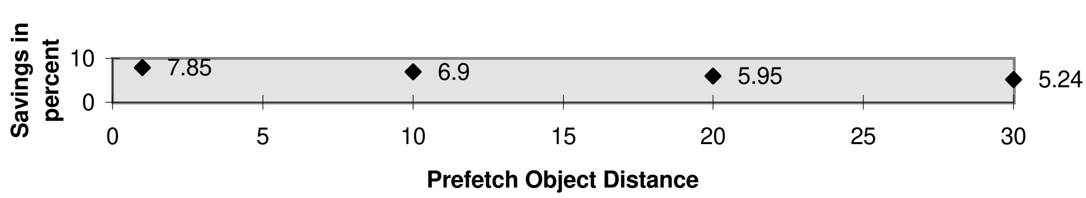

Figure 3:

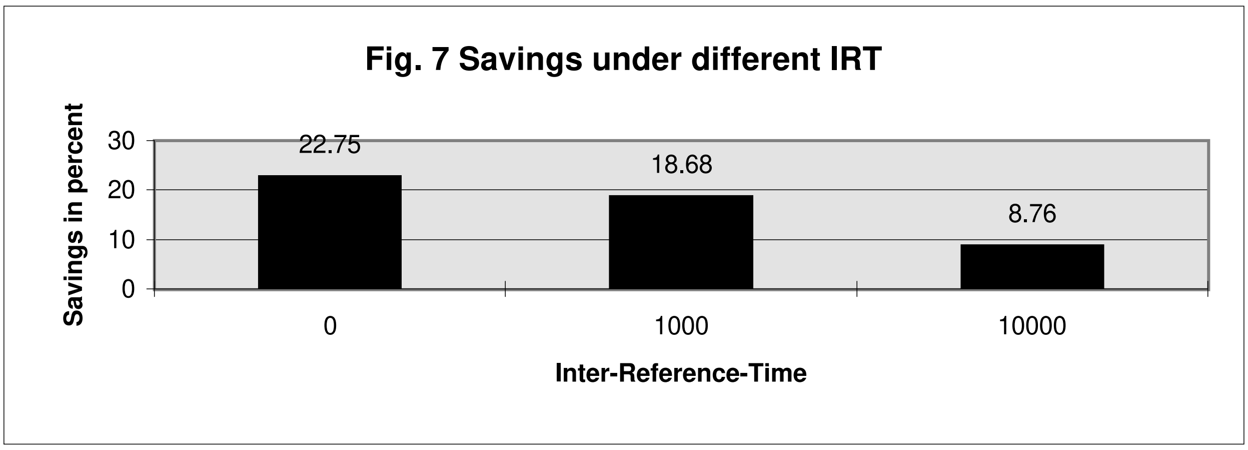
Fig. 7 Savings under different IRT
Savings in percent
30
20
10
0
22.75
18.68
8.76
0
1000
10000
Inter-Reference-Time

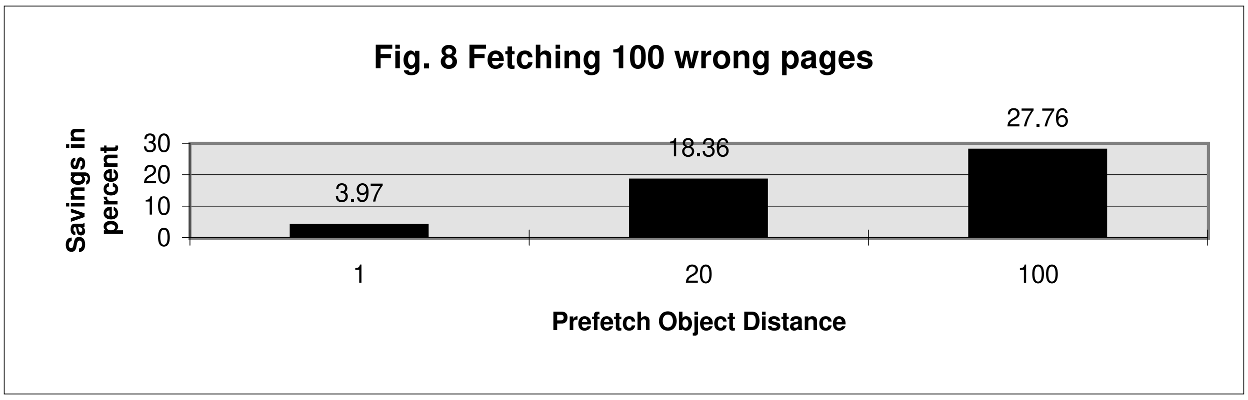
Fig. 8 Fetching 100 wrong pages
Savings in percent
30
20
10
0
3.97
18.36
27.76
1
20
100
Prefetch Object Distance

Figure 4: