A Prefetching Technique for Object-Oriented Databases

Nils Knafla

Dept. of Computer Science
University of Edinburgh
United Kingdom
Email: nk@dcs.ed.ac.uk

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, object-oriented databases, distribution, 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 since the penalty of waiting for the completion of a page fetch is so high. 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 retrieval chases pointers. Furthermore, in most OODBMSs the bulk of the processing occurs on the client: the server merely serves pages.

In this paper, 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. Our technique is different from existing techniques in the fact that we use an adaptive mechanism that prefetches pages dependent on the navigation through the object net. 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 an overview of the related work in the area of prefetching. How we predict pages to prefetch and store this information is described in section 3. The prefetching architecture is explained in section 4. In section 5 we present the theoretical results and the performance measurements. Finally, in section 6 we conclude our work and give an idea of 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. According to [7] predictors can be further classified as *strategy-based*, *training-based* or *structure-based*.

Strategy-based prefetching has an explicit programmed strategy which is used internally (One Block Lookahead [9]) or 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. For example, Fido [13] 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. If a data compressor successfully compresses the data, then its probability distribution on the data must be realistic and can be used for effective prediction.

Structure-based predictors obtain information from the object structure. Chang and Katz's technique [3] predicts the future access from the data se-

mantics in terms of structural relationships, e.g. inheritance, configuration and version history. They prefetch the immediate component object or immediate ancestor/descendent in a version history. An assembly operator for complex objects to load sub-objects recursively in advance was introduced by Keller [10]. The traversal was performed by different scheduling algorithms (*depth-first* and *breadth-first*). Our prefetching technique [11] also belongs to the *structure-based* approach.

In object-oriented databases the unit of I/O is an object (*Object Server*) or a page (*Page Server*) or a larger conglomeration, e.g. a segment. An object server prefetches an object or a group of objects [5] and a page server prefetches one or more pages ([4], [8]). 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*).

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 CAD, CAM, CASE and Office automation. These applications consist 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.

In a page server, like ESM, objects are clustered into pages. Good clustering is achieved when references to objects in the same page are maximized and references to objects on other pages are minimized. In our benchmark we use a *composite object clustering* technique.

The general idea of our technique is to prefetch references to other pages in a complex object structure net (e.g. OO7). We obtain the prefetch 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 because of time and resource limitations. Instead, we observe the client navigation through the object net. 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 optimal distance of n objects from the PSO to the ORO object which is necessary to provide enough processing to overlap with prefetching. 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, some saving can still be achieved (see section 5.2).

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 fig. 1 we have a complex object hierarchy. The object with the OID[2] 1 would be defined as a *Branch Object* because it contains a branch in the tree of objects. Objects with OID 2, OID 7 and OID 12 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 algorithm:

1. Retrieve the OID of the ORO and of the object in the next page referenced by the ORO.
2. Compute the POD to define the distance of n objects from PSO to ORO.
3. Determine the PSO by following the object reference n objects backwards from the ORO. If there are not enough objects in the reference chain before the ORO, then we will identify the first object in the page from the reference chain to achieve at least some saving.
4. If the object is already identified as a PSO and the previously identified PSO has a different *Post-Branch Object* then we would identify the *Post-Branch Objects* of the object as PSOs[3].

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 fig. 1 we would identify OID 5, OID 10 and OID 15 as OROs. In this example we assume a POD of 4 objects. On analysing page 2 we would identify OID 5 as an ORO. From OID 5 we would go through the chain backwards by 4 objects and identify OID 1 as a PSO. Then we would do the same for the OROs OID 10

[1] Additionally we could use the expected amount of processing from the application.
[2] OID = Object Identifier
[3] This step is executed after we have defined all PSOs from the OROs in a page.

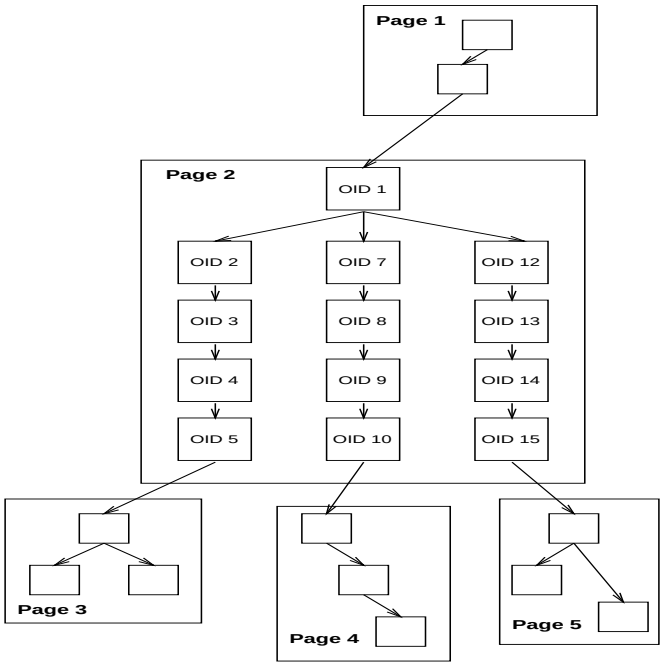

Fig. 1. Object relationship

and OID 15 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*. In this case we would identify the *Post-Branch Objects* of OID 1 (OID 2, OID 7 and OID 12) as PSOs instead of OID 1.

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 [10], we do not prefetch all references recursively; instead we select the pages to prefetch, dependent on the client processing. Recursive object prefetching has also the problem that prefetched pages can be replaced again before access. Adaptive object prefetching limits the number of prefetch pages to the adjacent pages. In contrast to [3], we look further ahead for objects to prefetch than the immediate object.

Each page of the database is analysed off-line. The Analyzer stores this information in the POT for every database root[4]. 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. 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.

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

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 OODBMSs. Applications traverse an object collection with a cursor. With PSO and ORO 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.

In future work we want to investigate performance and behaviour when the POT predicts a large number of pages. For this case we could use a multiple page request. To further reduce the number of pages, we could maintain information about a frequency count on how often the referenced page is accessed from this ORO. The total frequency count for a page would be computed by adding up all frequency count values of the OROs having the same referenced page. This total frequency count combined with a threshold makes the prefetch decision. Another possibility is to declare special data members of the object which are as important for prefetching.

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 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 System Architecture

4.1 The EXODUS Storage Manager

For the evaluation of the prefetching technique we chose the EXODUS storage manager to implement this idea. The EXODUS Client/Server database system was developed at the University of Wisconsin. The basic representation for data in the storage manager is a variable-length byte sequence of arbitrary size. Objects are referenced using structured OIDs[5]. On these basic storage objects, the storage manager performs buffer management, concurrency control, recovery, transactions and a versioning mechanism.

4.2 The Prefetching Architecture

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. As depicted in fig. 2, the database client is multithreaded.

[5] Object identifier containing a physical and logical component (in ESM page number and slot number)

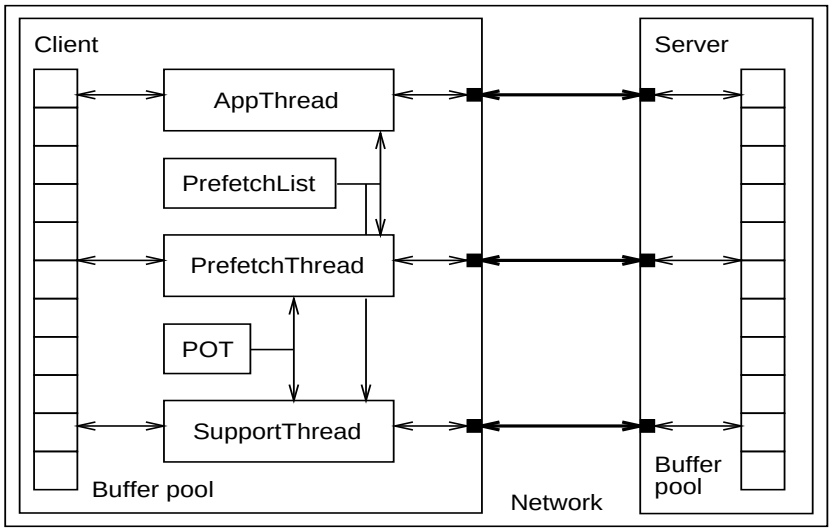

Fig. 2. Architectural Overview

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. The ESM server is not multithreaded[6].

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*. The number of *SupportThread*s is determined by the number of simultaneous prefetch requests. Each *SupportThread* runs on its own LWP[7] and while one *SupportThread* blocks on I/O, another *SupportThread* insert its page into the buffer pool.

[6] But ESM runs many tasks, as concurrent processes, on one processor

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

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.

5 Performance Evaluation

5.1 System Environment

For the ESM server we need a machine (called Dual-S[8]) 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 1 presents the performance parameters of the machines. Dual-C and Uni are also used as database clients. The network is Ethernet running at 10Mb/sec. The disk controller is a Seagate ST15150W (performance parameters in table 2).

Parameter	Dual-S	Quad	Dual-C	Uni
SPARCstation	20 Model 612	10 Model 514	20 Model 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 1: Computer performance specification

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

Table 2: Disk controller performance

5.2 Theoretical Results

The success of prefetching is dependent on the completion of the prefetch request before access. We define the cost of object processing to be C_o. Let C_{op} denote

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

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:

$$C_o = C_{op} + C_{oa} \tag{1}$$

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):

$$S_{or} = \begin{cases} C_p & \text{if } (C_o \cdot N_o \geq C_p) \\ C_o \cdot N_o & \text{otherwise} \end{cases} \tag{2}$$

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:

$$S_p = \sum_{i=1}^{n} S_{or}(i) \tag{3}$$

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):

$$S_r = (\sum_{j=1}^{q} S_p(j)) - C_t - C_s \tag{4}$$

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

$$savings = \frac{RT_d - RT_p}{RT_d} \cdot 100 \tag{5}$$

But the percentage of savings is always dependent on the amount of processing required on the page. For example in table 3 a page fetch costs 2 time units. With 10 CPU time units the saving is only 16 % but with 2 CPU time units the saving is 50 %. Therefore we plan to use a more accurate formula to

CPU	Page Fetch	Savings in percent
10	2	16 %
2	2	50 %

Table 3: Savings in percent

compute savings in percent. T_{sp} is the saved time with prefetching and T_p is the total time of all page fetches:

$$savings = \frac{T_{sp}}{T_p} \cdot 100 \tag{6}$$

We did not use this formula because 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.

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 (application with no prefetching) elapsed times.

In fig. 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.

In fig. 4a and 4b 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 longer network connection and slower access to the socket. Quad has the same cycle speed as Dual-C but a higher saving. Dual-C 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 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 elapsed-time savings under varying object access rates from the application (from 10 objects to 100 objects accessed). Fig. 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. In fig. 6 Prefetch1 means a prefetching version with just one *PrefetchThread* and Prefetch2 means a version with one *PrefetchThread* and one *SupportThread*. 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. 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 fig. 7 show that with more processing the savings in percent get smaller. This is because the application is more and more dominated by CPU processing (as explained in section 5.2).

In this test we studied the impact of wrong prefetches. We fetched 100 wrong pages from 200 page fetches. The other important parameter is the prefetch object distance. We used the distances of 1, 20 and 100. Recall that we always fetch 2 new pages from one page (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. Fig. 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 albeit of only 4 percent.

The last test measures the effect of additional clients on the Demand and the prefetching versions. In general each additional client increases the workload of the server and the network. If the prefetch request is completed before access, prefetching should improve performance even more with additional clients. If the server becomes a bottleneck and prefetch requests have to queue up at the server, prefetching can actually decrease performance. Fig. 9 shows that Demand

decreases performance significantly with 4 clients and the prefetching versions decrease performance with 7 clients.

6 Conclusions and Future Directions

In this paper 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 Quad machine, nearly 9% faster on Dual-C and 18% faster on Uni. 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%.

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.

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*, pages 474–499. Morgan Kaufmann, 1990.
2. M.J. Carey, D.J. DeWitt, and J.F. Naughton. The OO7 Benchmark. In SIGMOD [15], pages 12–21.
3. E.E. Chang and R.H. Katz. Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS. In *Proc. of the ACM SIGMOD Conference on the Management of Data*, pages 348–357, Portland, Oregon, June 1989.
4. K.M. Curewitz, P. Krishnan, and J.S. Vitter. Practical Prefetching via Data Compression. In SIGMOD [15], pages 257–266.
5. M.S. Day. *Client Cache Management in a Distributed Object Database*. PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, 1995.
6. C.S. Freedman and D.J. DeWitt. 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, May 1995.

7. C.A. Gerlhof and A. Kemper. A Multi-Threaded Architecture for Prefetching in Object Bases. In *Proc. of the Int. Conf. on Extending Database Technology*, pages 351–364, Cambridge, UK, March 1994.
8. C.A. Gerlhof and A. Kemper. Prefetch Support Relations in Object Bases. In *Proc. of the Sixth Int. Workshop on Persistent Object Systems*, pages 115–126, Tarascon, Provence, France, September 1994.
9. M. Joseph. An analysis of paging and program behaviour. *The Computer Journal*, 13(1):48–54, February 1970.
10. T. Keller, G. Graefe, and D. Maier. Efficient Assembly of Complex Objects. In *Proc. of theACM SIGMOD Int. Conf. on Management of Data*, pages 148–157, Denver, USA, May 1991.
11. N. Knafla. A Prefetching Technique for Object-Oriented Databases. Technical Report ECS-CSG-28-97, Department of Computer Science, University of Edinburgh, January 1997.
12. B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A.C. Myers, and L. Shira. 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, June 1996.
13. M. Palmer and S.B. Zdonik. Fido: A Cache That Learns to Fetch. In *Proc. of the 17th Int. Conf. on Very Large Data Bases*, pages 255–264, Barcelona, Spain, September 1991.
14. R.H. Patterson and G.A. Gibson. Exposing I/O Concurrency with Informed Prefetching. In *3rd Int. Conf. on Parallel and Distributed Information Systems*, pages 7–16, Austin, Texas, September 1994.
15. *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, Washington, USA, May 1993.
16. A.J. Smith. Sequentiality and Prefetching in Database Systems. *ACM Transactions on Database Systems*, 3(3):223–247, September 1978.
17. J.S. Vitter and P. Krishnan. Optimal Prefetching via Data Compression. In *Proc. 32nd Annual Symposium on Foundations of Computer Science*, pages 121–130, San Juan, Puerto Rico, October 1991. IEEE Computer Society Press.

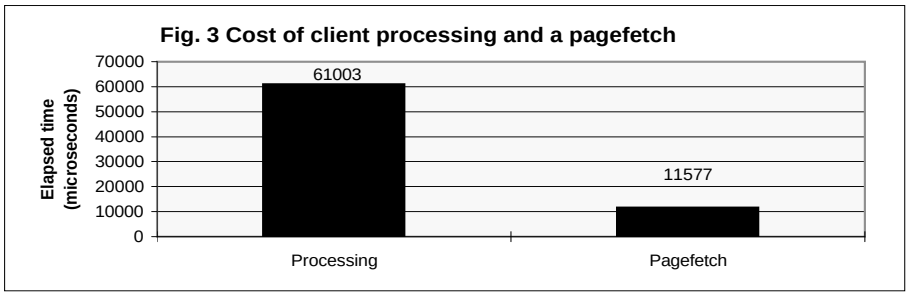
Fig. 3 Cost of client processing and a pagefetch
Elapsed time (microseconds)
70000
60000
50000
40000
30000
20000
10000
0
61003
11577
Processing
Pagefetch

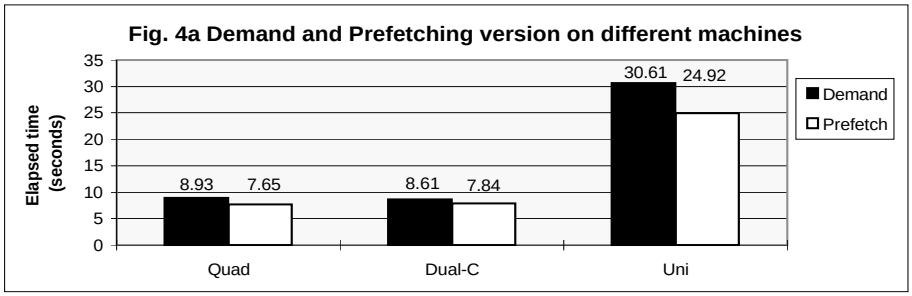
Fig. 4a Demand and Prefetching version on different machines
Elapsed time (seconds)
35
30
25
20
15
10
5
0
8.93
7.65
8.61
7.84
30.61
24.92
Demand
Prefetch
Quad
Dual-C
Uni

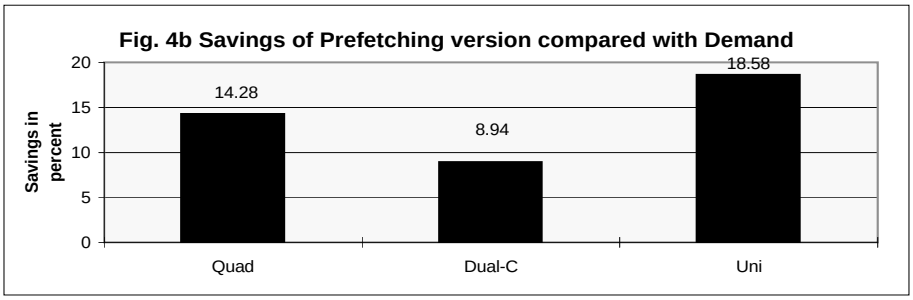
Fig. 4b Savings of Prefetching version compared with Demand
Savings in percent
20
15
10
5
0
14.28
8.94
18.58
Quad
Dual-C
Uni

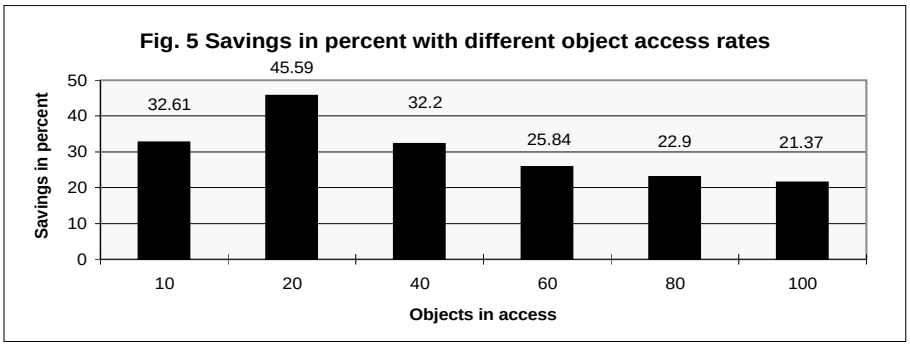
Fig. 5 Savings in percent with different object access rates
Savings in percent
50
40
30
20
10
0
32.61
45.59
32.2
25.84
22.9
21.37
10
20
40
60
80
100
Objects in access

Fig. 6 Multiple PrefetchThreads under different PODs

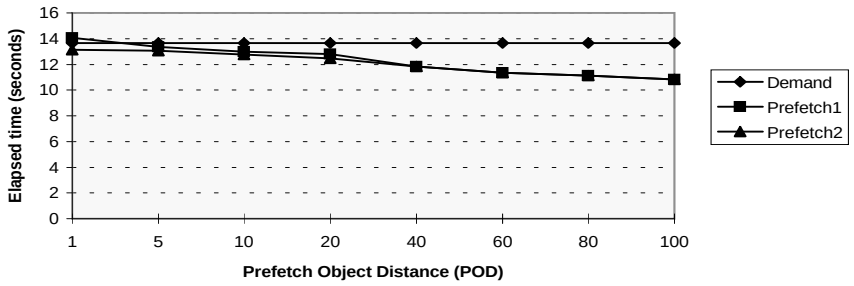

Fig. 7 Savings under different IRTs

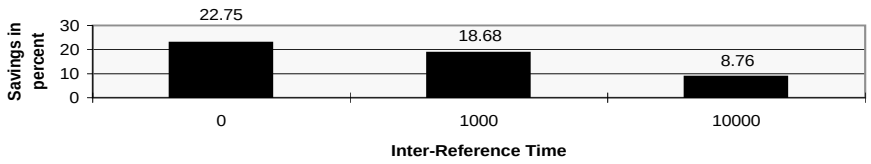

Fig. 8 Prefetching 100 incorrect and 100 correct pages

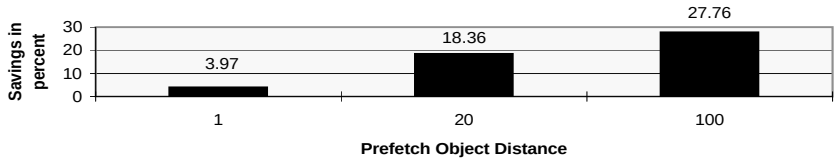

Fig. 9 Demand and Prefetching versions with multiple clients