Speed Up Your Database Client with Adaptable Multithreaded Prefetching

Nils Knafla
Dept. of Computer Science, University of Edinburgh
JCMB, King's Buildings
Edinburgh EH9 3JZ, United Kingdom
nk@dcs.ed.ac.uk

Abstract

In many client/server object database applications, performance is limited by the delay in transferring pages from the server to the client. We present a prefetching technique that can avoid this delay, especially where there are several database servers. Part of the novelty of this approach lies in the way that multithreading on the client workstation is exploited, in particular for activities such as prefetching and flushing dirty pages to the server. Using our own complex object benchmark we analyze the performance of the prefetching technique with multiple clients, multiple servers and different buffer pool sizes.

1 Introduction

Much of the research and development effort in high-performance database systems has focused on exploiting parallel computing on the database server platform. However, the falling prices of shared-memory multiprocessors makes such machines feasible as client hosts. This raises the question of how a multiprocessor client machine can be exploited to improve database response time. In one approach DeWitt et al. [8] used so-called *ParSets* in the OO7 benchmark traversals [4] to invoke a method on every object in a set in parallel. This approach can boost performance where the application is CPU-bound. However, for some applications, the performance bottleneck is dominated by the delay that results from the client waiting for the transfer of pages from the server and writing pages to the server. We address this problem by using multithreading to parallelise the I/O for prefetching and flushing. Various techniques of buffer management [1], [2] have been used to reduce traffic between the client and the server. However, in this paper, we concentrate on techniques that attempt to effect page transfers in a more timely fashion.

Prefetching has been studied before in many areas of computing such as operating systems, microprocessor design, compiler construction, the world wide web and databases. Early work on prefetching in databases [18] was able to exploit sequential access patterns. In object-oriented databases, the object relationships are often used to predict future accesses. The object structure and the pointer relationships provide important information for object access patterns.

Chang and Katz [5] predict future accesses from the data semantics in terms of inheritance and structural relationships. Objects are stored in pages and they prefetch the immediate object reference only. Due to the high cost of a page fetch, this technique has only a limited effect on reducing elapsed time. Cheng and Hurson [6] extended this work by adding multiple hints, a prefetching depth and physical storage considerations. Instead of using a single hint, a series of hints are given for all types of relationships. Prefetch depths were added to the prefetching hints according to the semantics of each relationship. For example, an application may require the access to follow configuration links recursively or to follow version links at a maximum of three levels away. Physical storage considerations are used to impose a limit on high cost I/O.

A complex assembly operator to load component objects recursively in advance was introduced by Keller et al. [13]. The application traversal was performed by three different scheduling algorithms, *depth-first*, *breadth-first* and an *elevator* algorithm (which schedules disk access to objects based on their physical location). Objects were clustered (according to their type or to the composite object structure) or unclustered. This technique is suitable for small object nets; in larger object nets, recursively prefetched objects might already have been replaced by the buffer replacement strategy.

In the Fido system [16], the prefetching technique employs an associative memory to recognize access patterns within a context over time. In training mode, object access information is gathered and stored with a *nearest-neighbor* associative memory. In prediction mode, this information is used to recognize previously encountered situations. Unfortunately they did not mention the costs of maintaining the associative memory.

Gerlhof developed an architecture for prefetching [10] and a so-called *Prefetch Support Relation* (PSR) [11]. The PSR stores the precomputed page answer of an operation, i.e. the identifiers of all pages that were accessed during the execution of an operation. Karpovich and Grimshaw [12] developed an extensible file system, ELFS, in which they used user hints to predict the file access pattern.

In Thor [7] each fetch request from the client causes the server to select a prefetch group containing the object requested and possibly some other objects. A fetch request is processed to completion, determining all members of the prefetch group, before any objects are sent to the client. The disadvantage of this approach is that the server is already a bottleneck and additional prefetch requests further increase the server workload.

Our prefetching technique observes the client processing on the object 'net'. If the application moves towards a non-resident object, the page of the object is a candidate for prefetching. We try to time the prefetch request so that the request is not started too early but the page arrives before application access. We implemented this technique by extending the EXODUS storage manager (ESM) [3].

In section 2 we describe the design of our prefetching technique. An overview of ESM and the prefetching architecture is given in section 3. In section 4 we present the performance results of our benchmark and in section 5 we conclude our work.

2 The Prefetching Design

2.1 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 [4]. 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 4.2).

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 (Object Identifier) 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] Additionally we could use the expected amount of processing from the application.

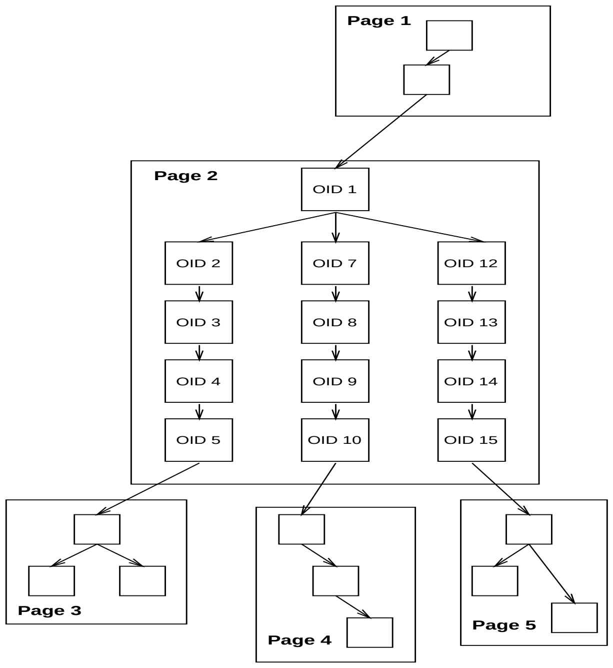


Figure 1. Object relationship

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[2].

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 analyzing 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 and OID 15 and identify OID 1 as the PSO for both. After analyzing 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 [13], 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 [5], we look further ahead for objects to prefetch than the immediate object.

Page [PageID]	PSO [SlotIndex]	ORO [SlotIndex]	RefPage [PageID]	RefOID [SlotIndex]

Figure 2. Entry in POT

Each page of the database is analyzed off-line. The Analyzer stores this information in the POT for every database root[3]. Fig. 2 depicts the layout of one entry in the POT. 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. 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 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 fre-

[2] This step is executed after we have defined all PSOs from the OROs in a page.

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

quency 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 use only special data members of the object for prefetching.

2.2 Replacement Policy

In the ESM client it is possible to open buffer groups with different replacement policies (LRU and MRU). Freedman and DeWitt [9] 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.

3 System Architecture

3.1 The EXODUS Storage Manager

We implemented the prefetching technique in ESM. The EXODUS client/server database system [3] 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 optimizer generator, and tools for constructing query-language optimizers.

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[4].

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.

[4] Object identifiers containing a physical and logical component (in ESM page number and slot number).

3.2 The Prefetching Architecture

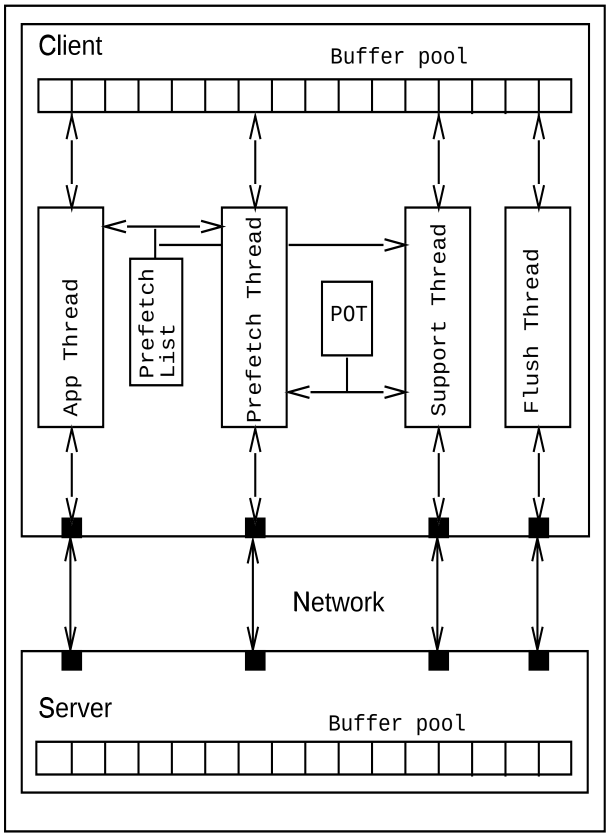


Figure 3. Prefetching architecture

In this section we describe how prefetching is incorporated into ESM. For the concurrent execution of the application and the prefetch system we used 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 fig. 3, 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. The *FlushThread* is responsible for flushing dirty pages to the server.

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 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 into 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 *SupportThread*s is determined by the number of simultaneous prefetch requests. Each *SupportThread* runs on its own LWP[5] 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[6] 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 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 systems performance and is part of future work.

For the parallel execution of threads on the client, synchronization 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 semaphore informs the *SupportThread* that there is a page to prefetch.

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. The Solaris thread interface provides a function to give the threads priorities. 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.[7] The *PrefetchThread* has a 50 percent priority[8] and the *SupportThreads* have low priorities.

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

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

Parameter	Server	Client
SPARCstation	20 Model 612	10 Model 514
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	Disk controller
External Transfer Rate	9 Mbytes/sec
Average Seek (Read/Write)	8 msec
Average Latency	4.17 msec

Table 2: Disk controller performance

4 Performance Evaluation

4.1 System Environment

In table 1 we give a specification of the computers used in our experiments. The client machine has 4 processors. The Ethernet network is running at 10 Mb/sec. The performance of the disk controller (Seagate ST15150W) is presented in table 2.

4.2 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:

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

[7] On a uniprocessor, a subtler approach to allocating priorities would be needed in order to strike a balance between application processing and prefetching.

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

The cost of a page fetch, C_p, is dependent on client and server processing and the network. Cl_b denotes the cost of client processing on the buffer pool; N_t denotes the cost of network transfer; S_b is the cost of server processing on the buffer pool; S_q is the server queueing cost and S_d is cost for the disk access. C_p is calculated by:

$$C_p = Cl_b + N_t + S_b + S_q + S_d + N_t + Cl_b \qquad (2)$$

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 C_p:

$$S_{or} = \begin{cases} C_p & if(C_o \cdot N_o \geq C_p) \\ C_o \cdot N_o & otherwise \end{cases} \qquad (3)$$

If there is enough processing (i.e. $C_o \cdot N_o$) to overlap then the saving is the cost of a page fetch. If not, there is also, albeit 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:

$$S_p = \sum_{i=1}^{n} S_{or}(i) \qquad (4)$$

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):

$$S_r = (\sum_{j=1}^{q} S_p(j)) - C_t - C_s \qquad (5)$$

4.3 Benchmark description

For the evaluation of the prefetching technique we created a benchmark with complex objects. The requirements for the 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.

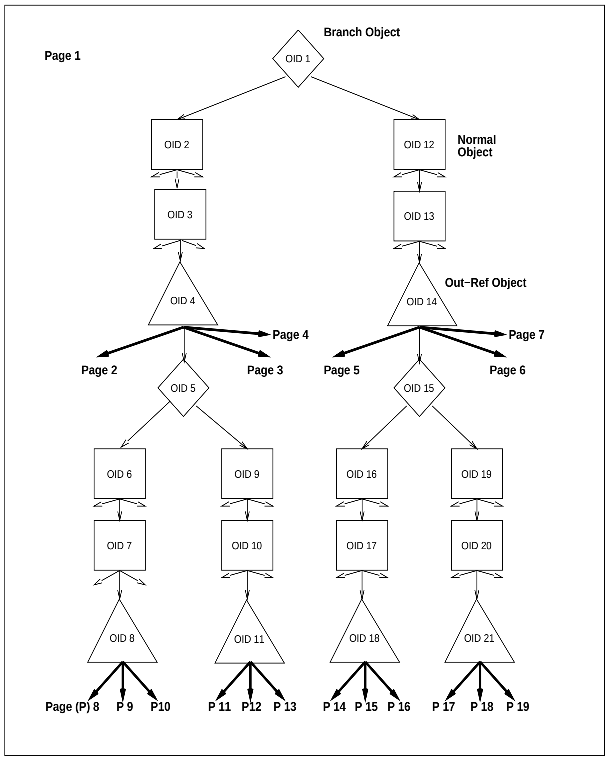


Figure 4. Benchmark structure of one page

In fig. 4 we depict the design of one page[9]. There are three types of objects: *Branch Objects*, *Out-Ref-Objects* (ORO) 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 different pages which are 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 dereferenced and then the application continues in page 1. At the lower levels (e.g. Pages 8 to 19) two pages are dereferenced 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 will be high. In [14], [15] we presented a benchmark which was less I/O intensive.

[9] Every page has the same structure.

Fig. 4 needs some explanation concerning the number of *Normal Objects*. The number of *Normal Objects* before an *Out-Ref-Object* 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.

4.4 Performance Measurements

All the tests were made in a multi-user environment. Because we were unable to get exclusive access to the machines and network it is not possible to compare the absolute times of different figures. Nevertheless, we made the tests at a time when the workload was low. Savings in percent mean the percent saving of a prefetching version compared with a version without prefetching and multithreading (Demand).

In fig. 5a we present the results of our benchmark. Prefetch1 means one PrefetchThread supports the application; Prefetch2 means there is one PrefetchThread and one SupportThread and Prefetch3 has one PrefetchThread and two SupportThreads. Fig. 5a 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 Prefetch3 has the best performance because it achieves the optimal number of prefetch threads for page requests. Fig. 5b shows the savings of the prefetching versions in percent. Prefetch1 only provides a 5% improvement, compared with Prefetch3 which achieves a saving of 23%.

The effect of additional clients (Demand versions running on other machines) is shown fig. 6. Prefetch2 always performs better than Demand. At the level of 3 clients, Prefetch1 performs worse than Demand. This is because the cost of the thread management is higher than the benefit of prefetching. In general each client increases the network workload, the server processing and the work for the disk manager. If the prefetch request is completed before client access the savings in percent would increase in a multi-client environment. If the prefetch request would be served at the same time at the server or even later, prefetching would decrease performance due to the additional costs of the thread management.

In fig. 7 we present the results 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 conditions for the hardware. Fig. 7 shows that all versions improve performance in the distributed environment.

The size of the buffer pool has an important impact on the performance of the prefetch technique. We compared performance for 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 fig. 8, the Prefetch version can increase slightly 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 synchronization variables. The write versions behave very similarly. A higher 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 we stopped the time after the commit of the transaction. For the read versions the result are the same as in fig. 8, 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 another version, called *Prefetch write mt flush* with 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 their elapsed time increases after 50. 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 test proves that multithreading is not only useful for prefetching; flushing dirty pages to the server is also a suitable application for multithreading.

In fig. 10 we compared our presented prefetching technique (using a POT table) with the technique presented in [5] which prefetches only adjacent references from the current object. The prefetch adjacent version achieved quite a surprisingly good result because of the structure of the benchmark. All three page fetches are from one object. Two prefetches can be made at the same time as the first page is prefetched. Therefore only the first page fetch has a sufficiently large penalty to arrive late.

In the last test we showed the effect of incorrect prefetch requests. We used the same benchmark for this test with the only difference being that we navigate only to one page from the ORO. The other two pages can be used for incorrect prefetches. In fig. 11 *Two incorrect* means prefetching two pages incorrectly from an ORO; *One incorrect* means prefetching one incorrectly and Correct means optimal prefetching. The Inter-Reference Time (IRT) simulates overhead for client processing. The values of the IRTs are the number of loop iterations after every object access. The elapsed time of 3560 iterations is equal to the elapsed time of one object preparation for client access. 1675 iterations are equal to half of one object preparation and 850 iterations are equal to a quarter. Additional overhead for client processing could be produced by a pointer swizzling technique, application client processing or waiting time. *One incorrect* and *Correct* always perform better than Demand. After an IRT value of 850, *Two incorrect* can also improve performance.

5 Conclusions and Future Work

We presented a prefetching technique for object-oriented databases using multithreading. At run time we observe the application client processing and if the application moves towards a page which is not resident we will prefetch this page using the information of the POT. All information of the POT is collected off-line by an analyzer. This information is used at run time by the prefetch threads. We also use multithreading for flushing dirty pages to the server. This is especially useful at the end of a transaction when many pages have to be flushed to the server. The results of our benchmark showed that prefetching can improve performance significantly if object access is reasonably predictable. The fastest prefetching version achieved a saving of 23%. Even will a prefetch accuracy of only 33%, performance can still be improved by prefetching. Prefetching is also successful with additional database clients as long as the server does not become a total bottleneck. We showed that the organisation of distributed databases is attractive for prefetching. Increasing the buffer size improved significantly the response time in a write transaction. To speedup the flush of dirty pages to the server we also used threads. Above a buffer size of 100 frames the multithreaded version has a 15% percent advantage over the sequential flush version.

In the future we will compare our technique with other proposed prefetching techniques. We will implement the OO7 benchmark to see how our technique will perform in this environment. We also look at real application structures for prefetching. Another possibility is to make the ESM server multithreaded. The problem is the synchronization 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. If the client demands multiple pages without strict time restrictions, our system will test the unit of I/O with a set of pages.

References

[1] J.-H. Ahn and H.-J. Kim. SEOF: An Adaptable Object Prefetch Policy For Object-Oriented Database Systems. In *Proc. of the 13th Int. Conf. on Data Engineering*, Birmingham, UK, Apr. 1997.

[2] P. Cao, E. Felten, A. Karlin, and L. Li. Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling. *ACM Trans. Comput. Syst.*, 14(4), Nov. 1996.

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

[4] M. Carey, D. DeWitt, and J. Naughton. The OO7 Benchmark. In SIGMOD [17], pages 12–21.

[5] E. Chang and R. 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.

[6] J. Cheng and A. Hurson. On the Performance Issues of Object-Based Buffering. In *Proc. First Int. Conf. on Parallel and Distributed Information System*, pages 30–37, Miami Beach, Florida, Dec. 1991.

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

[8] D. DeWitt, J. Naughton, J. Shafer, and S. Venkataraman. Parallelizing OODBMS traversals: a performance evaluation. *VLDB Journal*, 5(1):3–18, Jan. 1996.

[9] C. Freedman and D. 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.

[10] C. 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, Mar. 1994.

[11] C. 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, Sept. 1994.

[12] J.F. Karpovich, A.S. Grimshaw, J.C. French. Extensible File Systems (ELFS) An Object-Oriented Approach to High Performance File I/O. In *OOPSLA*, Portland, Oregon, Oct. 1994.

[13] T. Keller, G. Graefe, and D. Maier. Efficient Assembly of Complex Objects. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 148–157, Denver, USA, May 1991.

[14] N. Knafla. A Prefetching Technique for Object-Oriented Databases. Technical Report ECS-CSG-28-97, Department of Computer Science, University of Edinburgh, Jan. 1997.

[15] N. Knafla. A Prefetching Technique for Object-Oriented Databases. In *Advances in Databases, 15th British National Conference on Databases, BNCOD 15*, London, United Kingdom, July 1997.

[16] M. Palmer and S. 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, Sept. 1991.

[17] *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, Washington, USA, May 1993.

[18] A. Smith. Sequentiality and Prefetching in Database Systems. *ACM Trans. Database Syst.*, 3(3):223–247, Sept. 1978.

Fig. 5a Demand and prefetching versions

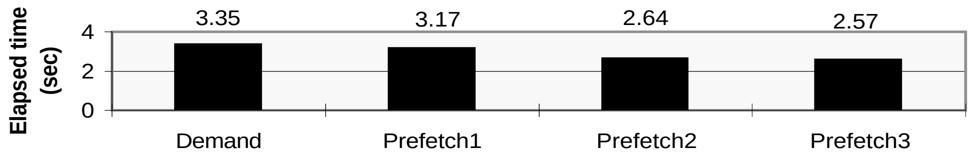


Fig. 5b Savings of the prefetching versions

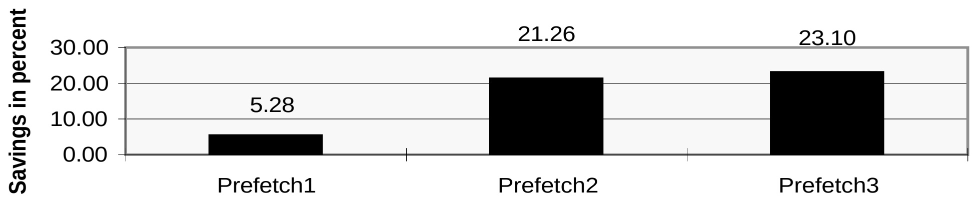


Fig. 6 Demand and prefetching versions with multiple clients

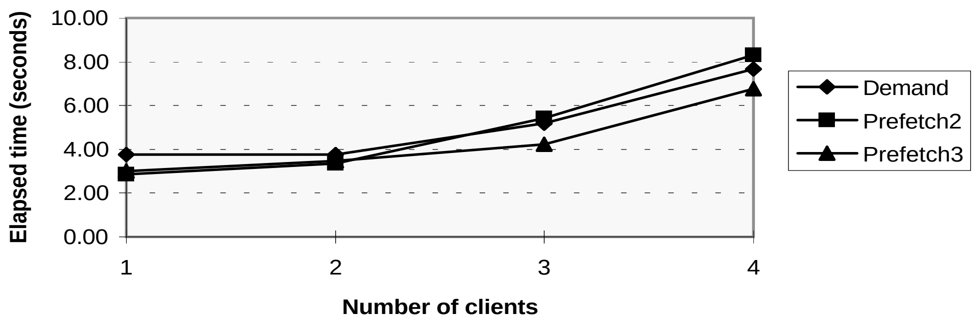


Fig. 7 Data on one server or two servers

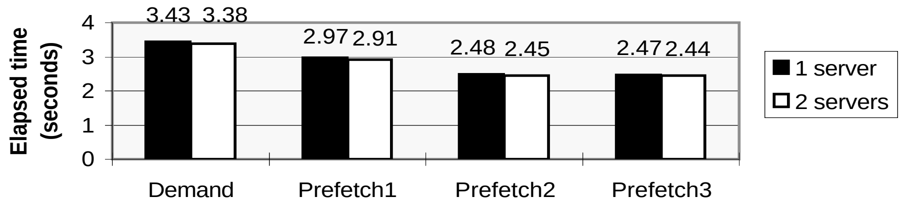


Fig. 8 Impact of different buffer pool sizes (stopped before commit)

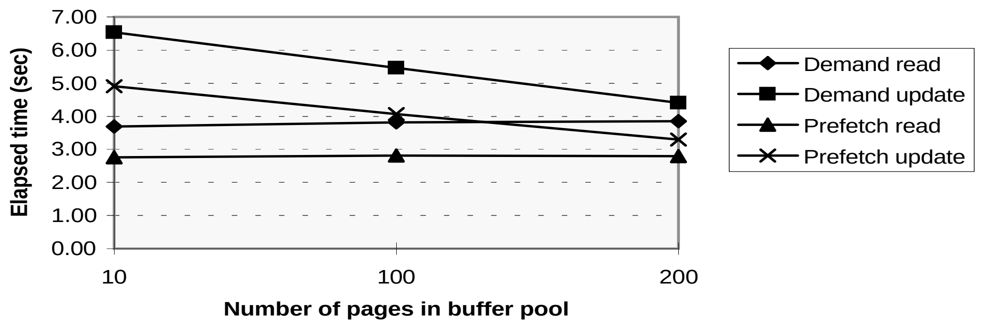


Fig. 9 Impact of different buffer pool sizes (stopped after commit)

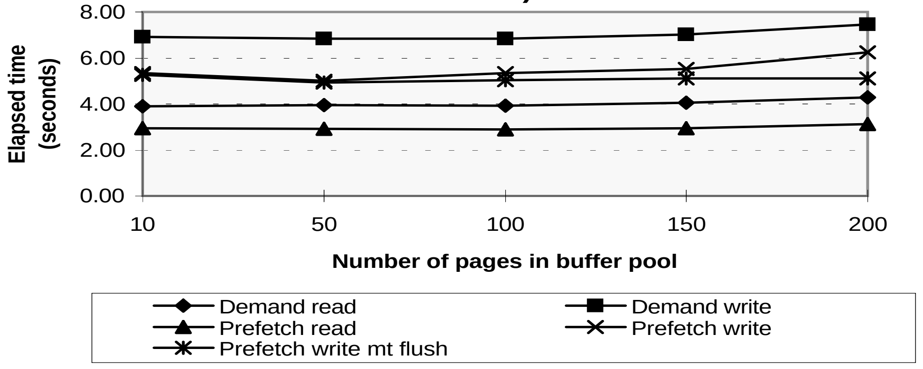


Fig. 10 Prefetch adjacent vs. Prefetch with POT

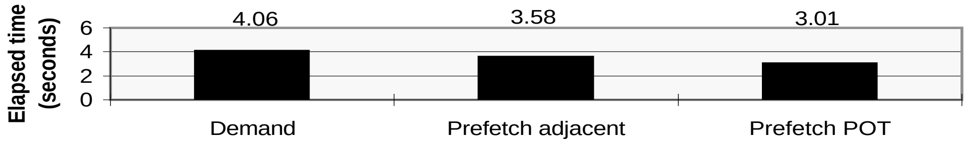
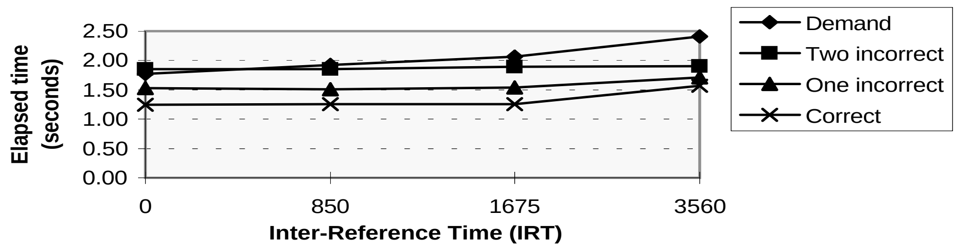


Fig. 11 Effect of incorrect prefetching


Elapsed time (seconds)
Inter-Reference Time (IRT)
Demand
Two incorrect
One incorrect
Correct