SEOF: An Adaptable Object Prefetch Policy For Object-Oriented Database Systems*†

Jung-Ho Ahn and Hyoung-Joo Kim
OOPSLA Lab. Department of Computer Engineering
Seoul National University
Shilim-Dong Gwanak-Gu, Seoul 151-742, KOREA
{jhahn, hjk}@papaya.snu.ac.kr

Abstract

The performance of object access can be drastically improved by efficient object prefetch. In this paper we present a new object prefetch policy, Selective Eager Object Fetch(SEOF) which prefetches objects only from selected candidate pages without using any high level object semantics. Our policy considers both the correlations and the frequencies of fetching objects. Unlike existing prefetch policies, this policy utilizes the memory and the swap space of clients efficiently without resource exhaustion. Furthermore, the proposed policy has good adaptability to both the effectiveness of clustering and database size. We show the performance of the proposed policy through experiments over various multi-client system configurations.

1. Introduction

The advantages of object-oriented databases in terms of a rich data model for next generation database applications, such as CAD/CAM/CASE, AI expert shell, and multimedia office information system have become widely recognized[2, 5]. One of the key issues of such applications is the performance due to their computationally complex data management. To meet their stringent requirements, there have been numerous works concerning clustering and pointer swizzling which are major issues of high performance object management[5, 9, 12, 15, 22, 23].

The most common approach for building an object-oriented DBMS is the data shipping architecture in client-server environment: e.g., O2[2], ORION-1SX[14]. Therefore, in order to speed up object access it is necessary to reduce object misses at clients and to minimize client-server interactions. As such, it is well known that efficient object buffer management can significantly improve the overall performance.

However, object buffer management has a number of complex and difficult problems to be solved for efficient object buffering. The object buffer should handle fragmentations as well as heavy memory allocations and deallocations, since it should manipulate objects of various size. Moreover, a buffer consistency protocol may make the object buffer management harder[3].

It is also not easy for an object buffer to delimit the span of an object access. This is because the `FIX`/`UNFIX` protocol of page buffer management cannot be employed without any performance degradation of object access. So, efficient buffer replacement is very difficult, if not impossible.

Due to these difficulties related to object buffering, most previous works[3, 5, 19] to increase object buffer hit ratio have investigated efficient object prefetch policies rather than efficient object buffer replacement algorithms. An early work proposed the policy which exploits high level object semantics in terms of inheritance and structural relationship[5]. Alternative approaches based on profiling or learning of object access pattern have been studied in [3, 19].

The prefetch policy which uses high level object semantics is likely to help efficient retrieval of complex objects. However, this approach cannot predict general object access patterns which might result from invoking a method[7]. Moreover, it is not easy for the object manager fashioned in a *byte server* to understand all of object semantics because the byte server has no idea about class, relationship, or inheritance. In predictive schemes, it is expensive to keep profiling or learning of object access patterns on every access context and hard to realize these policies. And all of the previous works[3, 5, 9, 13, 22] took little thought of exhaustion

*This work was partially supported by Ministry of Education through Inter-University Semiconductor Research Center(ISRC-96-E-2026) in Seoul National University.

†This work was partially supported by Korea Ministry of Trade and Industry under grant NO. 943-20-4, “SOP Project”.

and competition of memory and swap resources, since most experiments were performed on small databases. And the performance issues in multi-client environments have been rarely studied.

Along this line, we set out to build a new object prefetch policy, *Selective Eager Object Fetch* that prefetches objects only from selected candidate pages without using any object semantics. We then studied the performance of our new policy over various multi-client system configurations.

The remainder of the paper is organized as follows. Section 2 discusses in further detail the techniques of object prefetch and their related works. In section 3, we introduce our Selective Eager Object Fetch algorithm and the intuition behind it. Section 4 describes the simulation model and presents the experiment results. Finally, conclusion of our study and some areas for future research are given in section 5.

2. Techniques of Object Prefetch

Most of object-oriented database applications have a strong tendency to cache a large number of objects in virtual memory and perform extensive computation on them. However, it is not desirable to keep objects in page frames, because pure page-based buffering leads to inefficient space utilization if databases are clustered poorly[3, 13, 14].

To solve this problem, many of object-oriented DBMSs are based on the dual-buffer architecture in which an object buffer functions on top of a page buffer[3, 14]: Examples are Itasca[10], Ontos[18], and Versant[21]. The partitioned buffering provides good space utilization by filtering out useless objects from the page buffer and allows efficient garbage collection.

Performance when accessing objects in the dual-buffer architecture primarily depends on the object buffer hit ratio. However, as mentioned above, object buffer management has many difficult issues to be resolved. Among them, delimiting the span of an object reference is the most significant problem, since unused objects cannot be displaced deliberately without the notion of it. So, adopting a replacement algorithm such as LRU to an object buffer might be a big burden, if not impossible. In addition, paging or buffer replacement in object-oriented database applications, whose working cycles are characterized as *load-work-save* model, are less important than in traditional ones[15]. Taking this considerations, many systems including O2, Objectivity/DB[16], Versant, Mneme[15], and EPVM[22] cache objects in virtual memory without object replacement and an underlying operating system takes all responsibility of memory management and swap I/O. That is, all fetched objects are kept in memory until a transaction commits or a reference is definitely finished.

Consequently, the performance of object access can be improved not by efficient buffer replacement but by efficient object prefetch. Moreover, prefetching objects is more profitable in object-oriented database applications, since fetched objects do not likely happen to be invalidated by other clients due to their aspects of high read/write ratio and weak data sharing[5, 15, 19].

We can classify object prefetch policies into three categories according to how to select candidate objects for prefetching. The first one is the aggressive eager prefetch scheme where all of objects are extracted from a page or a segment together upon the first of fetching an object from the page or the segment. ORION and ENCORE[8] use this scheme. This policy allows the good performance of object access by improving the object buffer hit ratio and it can be highly profitable in multi-client environments, since object hits save the work load on the server.

However, since a number of unneeded objects can be held in memory by eager prefetching, this policy may lead to many page faults and swappings as well as unnecessary copy overhead. Thus, this approach may induce a significant performance degradation although it can benefit from small and well clustered databases.

The second policy is equipped with advanced object semantics. That is, all of the objects linked with the requested one are fetched recursively on every object miss. Observing the access patterns of object-oriented database applications, Chang and Katz[5] proposed a run time clustering algorithm and a smart buffering policy which exploits the knowledge about inheritance and structural relationship. Their smart buffering uses access hints provided by a user and object relationships to obtain all to-be used objects in advance. It also gives high priority to the pages related with the accessed objects, if they are already cached.

This achieves extremely good performance for retrieving complex objects, if they are clustered well. When objects have multiple relationships, however some object access patterns cannot be incorporated with clustering[3, 19]. That is, this policy may actually cause more unnecessary object fetching than the first one if only a part of relationships are utilized. It also cannot predict general object access patterns and it is difficult to facilitate this scheme on a byte server

The third one is the predictive approach which predicts to-be used objects through profiling or learning of access patterns. In the profile-based policy[3], sophisticated buffering hints are stored in a profile to prefetch objects more precisely. As an alternative, Palmer and Zdonik[19] proposed a predictive cache that employs associative memory to recognize access patterns. In these policies, profiling or learning should be associated with each access context, since access patterns even on the same data may be different according to applications. However, it is difficult to profile or learn access patterns on every context with some accu-

racy.

Kemper and Kossmann[13] made another effort to improve the performance of object access, which tried to adapt to more complex and various object patterns by mixing page-based buffering and object-based buffering simultaneously.

Object prefetch has been also studied in many previous researches on swizzling, since these two topics are closely related[9, 12, 15, 22, 23]. Although these studies made observations mainly about various swizzling techniques, their results exhibit the evaluation of object prefetch schemes indirectly.

3. Selective Eager Object Fetch Policy

In general, the cost of object access in the dual-buffer architecture can be computed as:

$$Access\ Cost = O_{hit} \times C_{O_{hit}} + O_{miss} \times (P_{hit} \times C_{P_{hit}} + P_{miss} \times C_{P_{miss}} + C_{O_{miss}})$$

Where $O_{hit}(O_{miss})$ is the hit(miss) ratio of the object buffer, $P_{hit}(P_{miss})$ is the hit(miss) ratio of the page buffer, $C_{O_{hit}}(C_{O_{miss}})$ is the access cost for a hit(missed) object, and $C_{P_{hit}}(C_{P_{miss}})$ is the access cost for a hit(missed) page. This is explained by the steps in accessing an object: If the object is missed, the page which has the object is fixed and then the object is copied from it. In the same way, if the page is missed, the page should be fetched from a server or a disk device. Here, $C_{P_{miss}}$ is much greater than any other costs since it includes the cost for client-server interactions. Therefore, decreasing P_{miss} and O_{miss} is the key for improving the performance of object access[1]. First, an efficient page buffer replacement policy can raise the page buffer hit ratio. Several alternatives are possible including LRU, LFU, LRU-K[17], and 2Q[11]. In object-oriented DBMSs, these replacement polices may be as good as in traditional DBMSs[2].

As noted in section 2, however, an efficient object prefetch policy is needed to reduce object misses rather than an efficient object buffer replacement algorithm. Existing prefetch policies are already reviewed in section 2.

Now we drive our new object prefetch policy, Selective Eager Object Fetch. We decided to choose the eager object prefetch policy(the first policy classified in section 2) as the basis of our new algorithm, since this policy does not require any object semantics. Furthermore, this scheme induces little overhead for incorrect prefetches, because this approach does not issue any additional page requests for prefetching.

The OO1 object operation benchmark[4], which was developed to evaluate scientific and engineering applications, exhibits the aspect of navigational object access in general object-oriented database applications. The OO1 benchmark database consists of `Part` and `Connection` objects, where every `Part` is connected to three other `Parts` via `Connections`. The connections between `Parts` are selected randomly to produce 90%-1% clustering factor: 90% of the connections are to the closest 1% of `Part` objects. The `traversal` operation of the OO1 benchmark accesses all `Parts` connected to a randomly selected `Part` object recursively, up to 7 hops.

Figure 1 shows the access pattern of the `traversal` operation on the small OO1 benchmark database which contains 20,000 `Part` objects. The X and Y axes represent the page id where accessed objects reside in and the sequence of object accesses respectively. Figure 1(a), which shows the access pattern when running `traversal` 10 times, indicates the two distinct localities of page accesses on each traversal: The one is for accessing `Parts` and the other is for `Connections`. Figure 1(b) scales up a portion of figure 1(a).

Given this access pattern, we found that the page which contains many objects accessed by `traversal` is continuously referenced at regular intervals. Figure 1(b) shows some examples: page 681, 682, ..., 685. On the other side, the page which is accessed intermittently is considered to have only a few objects to be fetched. This observation allows us to select the page which has been accessed successively as a candidate from which objects are eagerly prefetched.

However, there may exist a group of objects whose objects are created and stored sequentially and also retrieved together on every access. So, choosing candidate pages only by access frequencies can mislead prefetching since only a few objects may be actually used. Page 705, 706, and 708 in figure 1(b) are not accessed again after fetching two or three objects in a row.

To solve the problem described above, the correlation of fetching objects should be factored out. Until now, the correlation problem has been considered only by several page buffer management schemes. Frequency-based replacement algorithm excludes out the surge of references to a page by not incrementing the reference count if the page has been referenced repeatedly in a short interval[20]. From this, the policy does not give high priority to the page which is accessed repeatedly in a short interval and yet relatively infrequently referenced overall. The problem of correlated references is also mentioned in LRU-K[17] and 2Q[11] in a similar manner.

This factor can be incorporated into the object prefetch

[1] We are primarily concerned with the page server architecture, since it is most popular. But our work can be applied to the object-server architecture easily.

[2] Unfortunately, so far as we know, there does not exist any replacement policy tuned for object-oriented database systems.

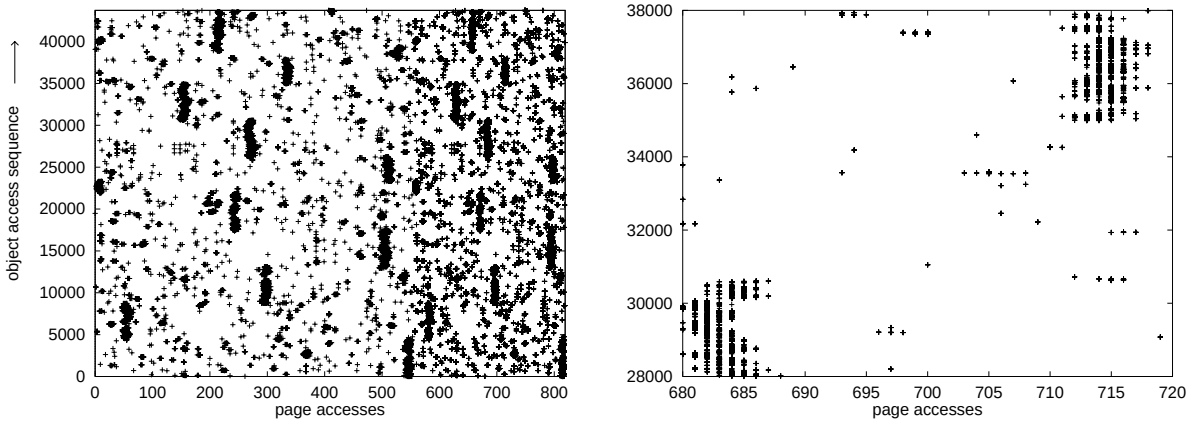


Figure 1. Access Pattern of OO1 `Traversal` Operation

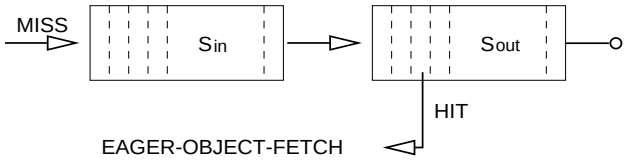


Figure 2. S_{in} and S_{out} in SEOF

policy, where a page is not selected as a candidate if the page is referenced successively in a short interval. Since the page with correlated references seems to contain only a few objects to be used, fetching only a requested object would save the buffer space. Furthermore, the page is very likely to be hit since the time interval of fetching objects from the page is sufficiently short.

As such, our Selective Eager Object Fetch(SEOF) algorithm is based on the two ideas addressed below:

- The page which has been referenced repeatedly in a short interval seems to have only a few objects to be fetched. This is considered as the correlation of object fetches.
- If there are frequent non-correlated references to a page, the page is likely to have many objects to be used.

The conceptual outline of SEOF algorithm is as follows. SEOF maintains two FIFO queues[3], S_{in} and S_{out} as shown in figure 2. Two queues are $Thresh_{S_{in}}$ and $Thresh_{S_{out}}$ long respectively. On every fixing a page for a missed object, SEOF places the page in S_{in}, if the page does not belong to either of S_{in} and S_{out}. Here, when S_{in} becomes larger than $Thresh_{S_{in}}$, the first-come entry of S_{in} is moved to S_{out}. S_{out} keeps its length in the same way. If the fixed page is in S_{out} SEOF fetches all uncached objects from the pages, but the reference is ignored during its stay in S_{in}.

In SEOF, S_{in} solves the correlated fetch problem by counting the repeated references within a short interval as one and S_{out} selects the frequently referenced page as a candidate for prefetch. Using these two queues, SEOF fetches objects eagerly only from pages which are likely to have many to-be used objects. The detailed algorithm is given in table 1.

$Thresh_{S_{in}}$ and $Thresh_{S_{out}}$ of SEOF are the important tuning parameters. Qualitatively, as $Thresh_{S_{in}}$ gets smaller and $Thresh_{S_{out}}$ gets larger, SEOF prefetches objects more aggressively. If $Thresh_{S_{in}}$ is large and $Thresh_{S_{out}}$ is small, SEOF takes no prefetching due to too strong test for candidate selection. The sensitivity of these parameters will be discussed more in section 4.

Our SEOF algorithm works independently with the page buffer management. However, the page buffer can employ S_{in} and S_{out} as its own buffer pools. How SEOF is incorporated with the page buffer management is potentially our next research topic.

4. Performance Evaluation

4.1. Simulation Model

The performance evaluation employed in this study is based on the page-server architecture using dual-buffering[4]. The conceptual structure of the simulation model is shown in figure 3. Several components are simulated in the model:

[3]Experiments with LRU queues gave us the similar results.

[4]The results of our experiments can be also adopted to the object-server architecture, where the results explain the evaluation of object prefetch from a server.

```
// when the reference to object o, which locates in page p, is invoked

if o is already in the object buffer then
    // do nothing (an object cache hit)
else
    if p is in S_out then
        fetch all uncached objects in p eagerly
        dequeue p from S_out
    else
        if p is in S_in then
            // do nothing (a correlated access)
        else
            if sizeof(S_in) ≥ Thresh_{S_in} then
                if sizeof(S_out) ≥ Thresh_{S_out} then
                    dequeue the first-come entry e_out from S_out
                end if
                dequeue the first-come entry e_in from S_in
                enqueue e_in into S_out
            end if
            enqueue p into S_in
        end if
        fetch object o
    end if
end if
return object o
```

Table 1. SEOF Algorithm

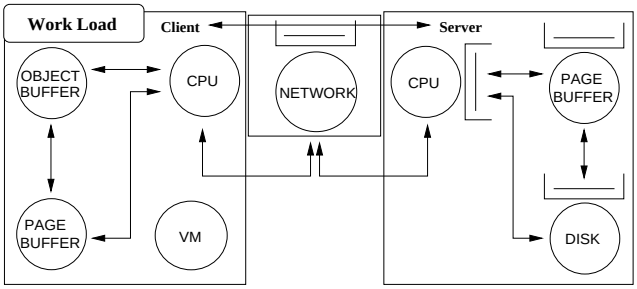


Figure 3. Simulation Model

Parameters	Value(*msec*)
`Part` processing time	5.0/`Part`
`Connection` processing time	0.5/`Connection`
Object copy time	0.003/object
Avg. Object buffer processing time	0.077
Avg. Page buffer processing time	0.025
Avg. Swap I/O time	16.0
Avg. Disk access time	17.0/8 K bytes
Network processing time	1.1 + 0.00075/byte
Network transfer time	4.5 M bits/sec

Table 2. Simulation Parameter Settings

`CPU`, `DISK`, `VM`, `PAGE BUFFER`, `OBJECT BUFFER`, and `NETWORK`.

`VM` component of the simulator is used for modeling page faults and swap I/Os. `VM` manages physical memory by page aging[1], where the pages that are no longer part of working set are aging at regular intervals. When there is no available physical memory, `VM` swaps out some oldest memory. In the simulator, a virtual memory block is 4 K bytes long. `PAGE BUFFER` component uses LRU for buffer replacement and all fetched objects are kept in `OBJECT BUFFER` until a transaction ends. All requests in the simulator are scheduled on first-come-first-serve basis and concurrency control is simulated by piggybacking page-level locks on page requests.

Table 2 presents the parameter settings used in the experiments. The values listed in the table were obtained by profiling our own object manager which is based on the page-server architecture.

We used the `traversal` operation of the OO1 benchmark as work load[5]. Two databases are used: the small database of 20,000 `Parts` and the large database of 200,000 `Parts`. With `Part` of 200 bytes and `Connection` of 32 bytes, the small and large database sizes are 6.4 M bytes and 64 M bytes respectively. The page size is 8 K bytes. We also used two alternative clustering factors, 90%-1% and 80%-5% in order to evaluate the performance with varying the goodness of clustering. These versions of database shall be referred to as `SMALL-DB-90-1`, `LARGE-DB-90-1`, `SMALL-DB-80-5`, and `LARGE-DB-80-5` respectively. Each client generates a single stream of object access by tracing the `traversal` operation with random seeds. In order to study the effects of prefetching in multi-client environments, we varied the number of clients from 1 to 20.

The simulator was coded in C++SIM[6], since we can use C++ directly without learning another language and it is available in public.

4.2. Simulation Results

The experiments were performed to compare three different prefetch policies: 1) EOF that fetches all objects eagerly from a page upon every the first object miss in the page, 2) LOF that fetches only one requested object at a time, and 3) SEOF we proposed. EOF and LOF set the page buffer size of the client to 8 K bytes and 4.3 M bytes respectively, which offered the best performance to each one in our

[5] We are preparing to evaluate our algorithm with real world applications.

additional experiment[6]. Based on [11], we ran SEOF with parameters $Thresh_{S_{in}}$ and $Thresh_{S_{out}}$ set to (120,120), (160,160), and (80,120), where $Thresh_{S_{in}}$ is ranged from 12.5% to 25% under the constraint of 10 M byte object buffer. On all three cases of SEOF, the size of the client page buffer is set to 2.7 M bytes. The server page buffer of 5.5 M bytes and 5 M byte client physical memory are used for all of the experiments.

We first present the results of the simulation for the OO1 benchmark databases clustered by 90%-1% factor.

Figure 4 shows the average elapsed time of running `traversal` 10 times. Not surprisingly, EOF shows the best performance over all other policies for `SMALL-DB-90-1`. This is due to the fact that the small database of 6.4 M bytes is such that most of the database can fit in the physical memory. That is, EOF can get the high object buffer hit ratio by prefetching the entire database while it experiences a small number of swappings.

The performance curves of SEOF on small database lie in the middle of EOF and LOF, but SEOF on the large database is very close to LOF. The reason is that the large working set of the large database prevents SEOF from prefetching enough objects to keep the hit ratio high. Of SEOF policies, SEOF(120,120) is slightly closer to LOF than other two SEOFs, although it is not easy to see in figure 4. This can be explained by their $Thresh_{S_{in}}$ and $Thresh_{S_{out}}$, as described in section 3.

Observing the performance as a function of the number of clients, we see that the experiments with two or three clients offer a reasonable performance improvement over one client. This is because with two or three clients, cached pages in the server are likely to be used by more than one client. However, as more clients are added, this advantage has been lost due to the contention against the server page buffer and the server tends to be overloaded.

Figure 4(b) also shows that LOF(and also SEOF) outperforms EOF in the range of 4 or less clients. This is due to the fact that naturally EOF suffers from heavy swappings, since it tends to prefetch many unneeded objects. However, as clients are added, a small number of page requests of EOF can pay off the swapping overhead by reducing the response time of the server. This fact is well reinforced by figure 5 which plots the total number of page requests and table 3 that represents the average response time of the server at the peak load. The server gives near-linear increase of the response time beyond 4 or 5 clients.

Considering the number of page requests, the response time of the server in SEOF is slightly worse than our expectation. This is due to the fact that prefetching by SEOF may result in the lower hit ratio of the server, since prefetching can make the cached pages in the server useless. In addition, the server response time in LOF for the small database increases a bit slowly because the server page buffer can cache almost the entire database.

Table 4 presents the size of the object buffer and the total number of swappings observed during our experiments. The ratio of actually used objects to (pre)fetched objects is also listed in parentheses. One interesting fact from this result is that LOF experiences more swappings for the small database than for the large database. This is explained by the fact that an object hit may require two swap I/Os: a swap out for making room and a swap in for reading a memory block in which the object is cached, while an object miss requires at most only one swap out. Thus, more object hits in the small database might induce more swappings.

Table 4 also reveals that EOF might be infeasible to run in a real environment especially for large database applications[7]. That is, considering 5 M bytes of physical memory, EOF may not be allowed to use about 40 M bytes of swap space. This is because EOF tends to prefetch too many unnecessary objects blindly, as noted in section 2. Consequently, we can consider EOF as a theoretically optimal policy, although it does not give the best performance in some parts of the experiments. From this consideration, we define the relative improvement provided by SEOF as follows:

$$\text{Relative Improvement} = \frac{LOF_{elapsed} - SEOF_{elapsed}}{LOF_{elapsed} - EOF_{elapsed}}$$

This shows the improvement over LOF(no-prefetching policy) as a fraction of the difference between EOF and LOF. Table 5 lists the results. The negative means that LOF performs better than SEOF and EOF. Please note that the negative values are larger than their actual performance gap, because the difference of EOF and LOF is very small at the intersection of two curves.

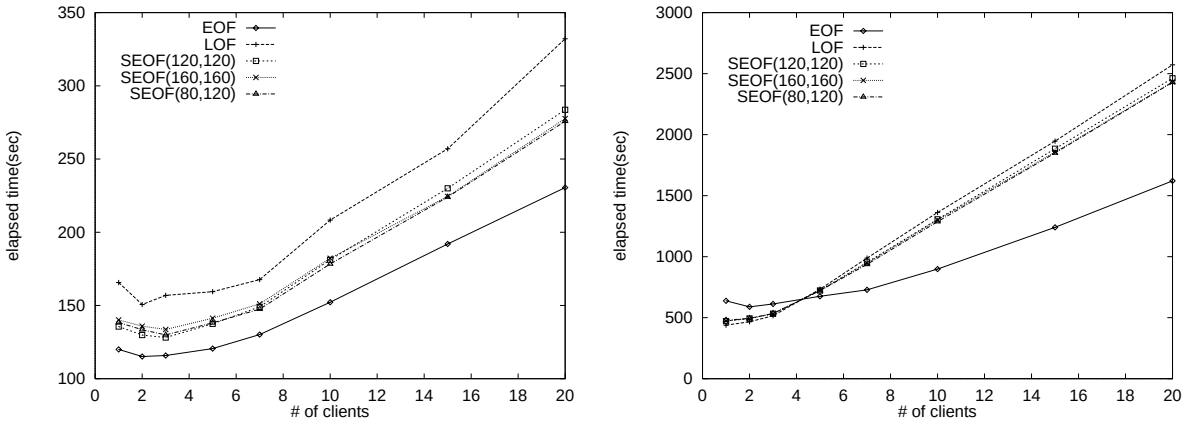
Next, we consider the effects of clustering on the performance of prefetch polices.

Figure 6 plots the average elapsed time of the traversal on the databases clustered by 80%-5% factor. These graphs show similar results with those for 90%-1% clustering factor. However, with increasing the number of clients, the performance of LOF degrades more sharply than the previous result. The reason is that the server is saturated more quickly because of the low buffer hit ratio in clients.

Comparing the results of SEOF with the previous ones, SEOF moves more closely to EOF in the experiments with the small database but behaves more similarly to LOF for the large database. This is explained as follows. With 80%-5% clustering factor, the small database has fewer cycles

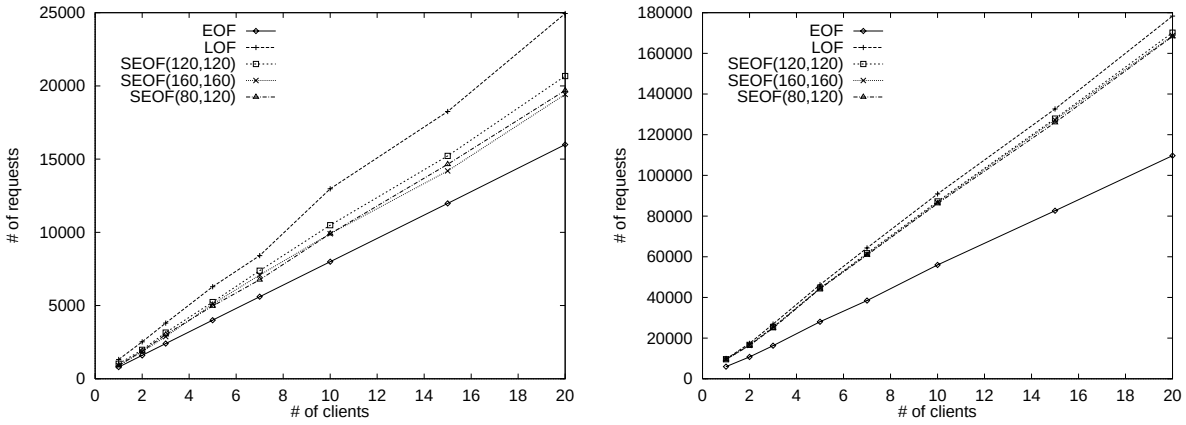
[6] In this experiment, we examined the memory resource contention by the object buffer and the page buffer. This results confirm our intuition that the large page buffer may suffer from heavy swappings, although it gives high buffer hit ratio.

[7] The system should be able to displace unused objects at any time in order to use EOF policy. However, object replacement cannot be easily done with C or C++ language binding.



(a) SMALL-DB-90-1 (b) LARGE-DB-90-1

Figure 4. Average Elapsed Time (90%-1% clustering factor)



(a) SMALL-DB-90-1 (b) LARGE-DB-90-1

Figure 5. Total Number of Page Requests (90%-1% clustering factor)

Strategies \ # of Clients		1	2	3	5	7	10	15	20
SMALL-DB -90-1	EOF	0.036	0.036	0.038	0.049	0.073	0.113	0.172	0.223
	LOF	0.036	0.038	0.041	0.054	0.081	0.122	0.172	0.229
	SEOF(120,120)	0.036	0.038	0.040	0.054	0.079	0.120	0.174	0.227
	SEOF(160,160)	0.036	0.038	0.041	0.054	0.081	0.122	0.179	0.228
	SEOF(80,120)	0.036	0.038	0.040	0.054	0.079	0.120	0.174	0.227
LARGE-DB -90-1	EOF	0.036	0.040	0.047	0.064	0.092	0.136	0.207	0.286
	LOF	0.036	0.044	0.048	0.074	0.102	0.146	0.220	0.295
	SEOF(120,120)	0.035	0.044	0.048	0.073	0.102	0.145	0.220	0.295
	SEOF(160,160)	0.036	0.044	0.048	0.073	0.102	0.145	0.218	0.296
	SEOF(80,120)	0.035	0.044	0.048	0.073	0.102	0.145	0.218	0.296

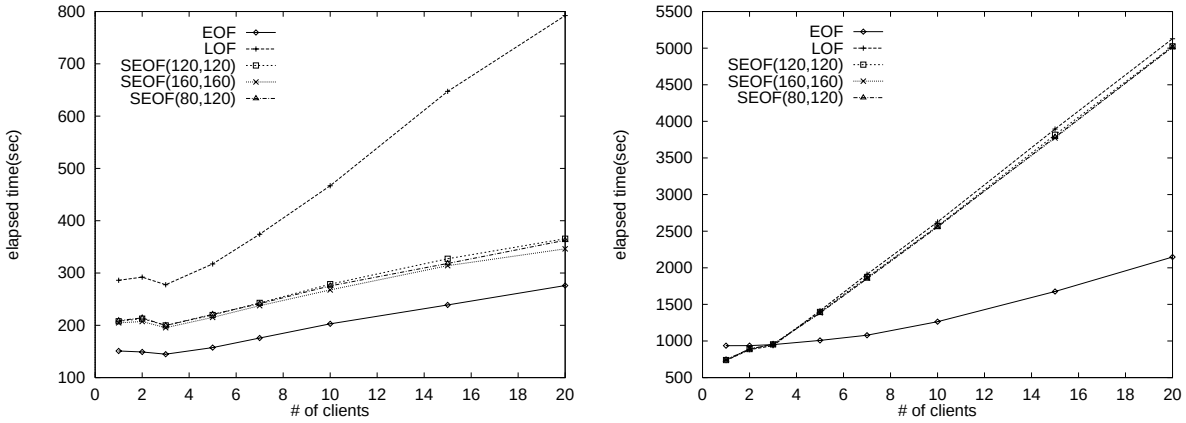
Table 3. Average Response Time of the Server(msec) (90%-1% clustering factor)

Strategies	SMALL-DB-90-1 Object buffer size	SMALL-DB-90-1 Swap I/Os	LARGE-DB-90-1 Object buffer size	LARGE-DB-90-1 Swap I/Os
EOF	5.5 M bytes (26.8%)	382.1	38.1 M bytes (6.5%)	16717.8
LOF	1.5 M bytes (100%)	2223.9	2.8 M bytes (100%)	792.8
SEOF(120,120)	3.6 M bytes (37.8%)	966.8	9.5 M bytes (24.5%)	3146.0
SEOF(160,160)	3.9 M bytes (35.0%)	1270.3	10.1 M bytes (23.1%)	3336.3
SEOF(80,120)	3.7 M bytes (37.8%)	1104.7	10.5 M bytes (22.1%)	3497.3

Table 4. Object Buffer Size and Swap I/Os (90%-1% clustering factor)

Strategies \ # of Clients		1	2	3	5	7	10	15	20
SMALL-DB -90-1	SEOF(120,120)	0.66	0.59	0.70	0.57	0.50	0.48	0.41	0.48
	SEOF(160,160)	0.56	0.42	0.57	0.47	0.44	0.47	0.50	0.54
	SEOF(80,120)	0.60	0.48	0.66	0.55	0.54	0.53	0.51	0.55
LARGE-DB -90-1	SEOF(120,120)	-0.17	-0.22	-0.17	0.19	0.13	0.12	0.09	0.12
	SEOF(160,160)	-0.18	-0.22	-0.18	0.17	0.16	0.13	0.13	0.15
	SEOF(80,120)	-0.20	-0.23	-0.20	0.20	0.20	0.16	0.14	0.15

Table 5. Relative Improvement (90%-1% clustering factor)



(a) SMALL-DB-80-5 (b) LARGE-DB-80-5

Figure 6. Average Elapsed Time (80%-5% clustering factor)

between Part objects, while the working set covers almost of the small database still as before. Thus, SEOF in SMALL-DB-80-5 selects more pages as candidates for prefetch. On the contrary, the low clustering factor in the large database causes only a few objects in a page to be used. As a result, SEOF selects fewer pages as candidates for the poorly clustered large database.

This behavior is also explained by table 6 that represents the size of the object buffer and the total number of swappings observed during the experiments for 80%-5% clustering factor. A surprising result for the small database, that EOF and SEOF utilize the object buffer space better than the previous experiment, can be explained by the same reason described before: a small number of cycles and the small database size. For the large database, SEOF gets the high buffer utilization due to its closeness to LOF and the blindness of EOF decreases its utilization. These results show that SEOF has good adaptability to both the effectiveness of clustering and database size.

The total number of page requests and the average response time of the server for 80%-5% clustering factor are not given in the paper, since the results are consistent with those of the previous experiment. We show the relative performance improvement obtained by SEOF in table 7.

Strategies	SMALL-DB-80-5 Object buffer size	Swap I/Os	LARGE-DB-80-5 Object buffer space	Swap I/Os
EOF	5.6 M bytes (36.3%)	2185.7	48.6 M bytes (5.7%)	33367.6
LOF	2.0 M bytes (100%)	7111.6	2.9 M bytes (100%)	995.0
SEOF(120,120)	4.6 M bytes (43.2%)	5568.1	7.3 M bytes (28.7%)	2739.1
SEOF(160,160)	4.7 M bytes (41.0%)	5477.2	7.9 M bytes (26.9%)	3070.0
SEOF(80,120)	4.6 M bytes (43.7%)	5661.9	7.7 M bytes (27.1%)	2940.4

Table 6. Object Buffer Size and Swap I/Os (80%-5% clustering factor)

Strategies \ # of Clients		1	2	3	5	7	10	15	20
SMALL-DB -80-5	SEOF(120,120)	0.58	0.56	0.59	0.61	0.66	0.71	0.78	0.83
	SEOF(160,160)	0.60	0.59	0.62	0.64	0.69	0.75	0.82	0.86
	SEOF(80,120)	0.57	0.54	0.59	0.60	0.67	0.73	0.81	0.83
LARGE-DB -80-5	SEOF(120,120)	-0.03	-0.14	-1.22	0.04	0.05	0.04	0.03	0.03
	SEOF(160,160)	-0.04	-0.18	-1.15	0.07	0.06	0.04	0.05	0.04
	SEOF(80,120)	-0.05	-0.16	-1.19	0.08	0.07	0.05	0.05	0.04

Table 7. Relative Improvement (80%-5% clustering factor)

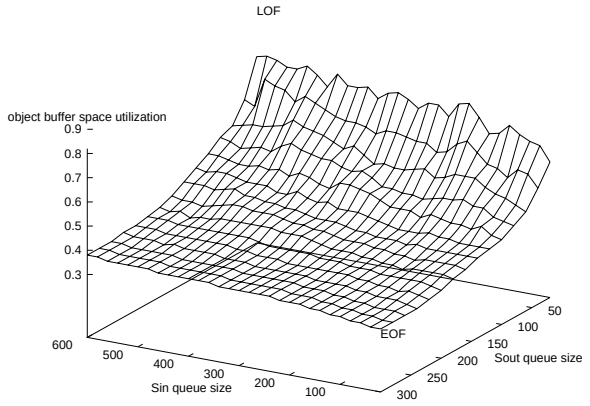


Figure 7. Sensitivity of SEOF to $Thresh_{S_{in}}$ and $Thresh_{S_{out}}$

In order to test the sensitivity of SEOF to its parameters, we conducted more experiments with various $Thresh_{S_{in}}$ and $Thresh_{S_{out}}$. The `traversal` operation was repeated on `SMALL-DB-90-1` for $Thresh_{S_{in}}$ and $Thresh_{S_{out}}$ ranging from 20 to 600. The results are plotted in figure 7. The X and Y axes are $Thresh_{S_{in}}$ and $Thresh_{S_{out}}$ respectively and the Z axis shows the ratio of used objects to (pre)fetched objects.

As described in section 3, SEOF moves closer to LOF as $Thresh_{S_{in}}$ gets larger and $Thresh_{S_{out}}$ gets smaller. With small $Thresh_{S_{in}}$ and large $Thresh_{S_{out}}$, SEOF performs very similarly to EOF. SEOF is insensitive to $Thresh_{S_{out}}$ of beyond 300 and that portion of results is not plotted. This is because the maximum interval of accesses to a same page is small. LOF and EOF labeled in figure 7 represent the results of the each named polices.

5. Conclusion and Future Remarks

In this paper we have developed a new object prefetch policy, Selective Eager Object Fetch which prefetches objects only from selected candidate pages without using any high level object semantics. Our policy is based on two ideas: 1) The page which has been referenced repeatedly in a short interval seems to have only a few objects to be fetched. 2) If there are frequent non-correlated references to a page, the page is likely to have many objects to be used. Unlike existing prefetch policies, SEOF utilizes the memory and the swap space of clients efficiently without resource exhaustion.

The results of our experiments indicate that object prefetch can improve overall performance significantly, although it may suffer from heavy swappings caused by prefetching too many unneeded objects. The relative improvement obtained by SEOF over LOF as a fraction of the difference between EOF and LOF is from 41% to 70% for the well clustered small database. For the large database, SEOF offers the improvement of up to 20%.

The additional experiments with low clustering factor confirm that SEOF has good adaptability to both the effectiveness of clustering and database size. SEOF tends to be closer to LOF with decreasing the clustering factor and SEOF behaves as EOF when the database is well clustered and small. We also investigated the sensitivity of SEOF to fixing its parameters.

We are currently implementing the proposed SEOF policy on top of our ODMG-93 compliant object-oriented DBMS, SOP[8]. In the future, we would like to extend our prefetch policy in a way that a series of unused prefetched objects can be displaced efficiently. We are also interested in finding an algorithm of fixing $Thresh_{S_{in}}$ and $Thresh_{S_{out}}$ dynamically according to access patterns and memory space utilization. We conjecture that these two methods could improve the performance of SEOF substantially.

Acknowledgments We greatly appreciate the SNU OOPSLA Lab researchers who have been involved in the SOP project during 1992-1995.

References

[1] M. J. Bach. *"The Design of the UNIX Operating System"*. Prentice-Hall Inc, 1986.

[2] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *"Building an Object-Oriented Database System: The Story of O2"*. Morgan Kaufmann Publishers, Inc., 1992.

[3] J. bing R. Cheng and A. R. Hurson. "On The Performance Issues of Object-Based Buffering". In *Proc. Int'l. Conf. on Parallel and Distributed Information Systems*, Dec. 1991.

[4] R. G. G. Cattell and J. Skeen. "Object Operations Benchmark". *ACM Trans. Database Syst.*, 17(1), Mar. 1992.

[5] 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 International Conference on Management of Data*, May 1989.

[6] Department of Computing Science, University of Newcastle upon Tyne. *"C++SIM User's Guide"*, public release 1.5 edition.

[7] D. J. Dewitt and D. Maier. "A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems". In *Proceedings of the International Conference on Very Large Data Bases*, Aug. 1990.

[8] M. F. Hornick and S. B. Zdonik. "A Shared, Segmented Memory System for an Object-Oriented Database". *ACM Trans. Office Inf. Syst.*, 5(1), Jan. 1987.

[9] A. L. Hosking and J. E. B. Moss. "Object Fault Handling for Persistent Programming Languages: A Performance Evaluation". In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 1993.

[10] IBEX Object Systems, Inc. "ITASCA Technical Summary Release 2.3", 1995.

[11] T. Johnson and D. Shasha. "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm". In *Proceedings of the International Conference on Very Large Data Bases*, Sept. 1994.

[12] A. Kemper and D. Kossmann. "Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis". In *Proceedings of the International Conference on Data Engineering*, Apr. 1993.

[13] A. Kemper and D. Kossmann. "Dual-Buffering Strategies in Object Bases". In *Proceedings of the International Conference on Very Large Data Bases*, Sept. 1994.

[14] W. Kim, J. F. Garza, N. Ballou, and D. Woelk. "Architecture of the ORION Next-Generation Database System ". *IEEE Transactions on Knowledge and Database Engineering*, 2(1), Mar. 1990.

[15] J. E. B. Moss. "Working with Persistent Objects: To Swizzle or Not to Swizzle". *IEEE Trans. Softw. Eng.*, 18(8), Aug. 1992.

[16] Objectivity, Inc. "Objectivity/DB Technical Overview Version 3", 1995.

[17] E. J. O'Neil, P. E. O'Neil, and G. Weikum. "The LRU-K Page Replacement Algorithm For Database Disk Buffering". In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1993.

[18] Ontos, Inc. "ONTOS Product Description", 1996.

[19] M. Palmer and S. B. Zdonik. "Fido: A Cache That Learns to Fetch". In *Proceedings of the International Conference on Very Large Data Bases*, Sept. 1991.

[20] J. T. Robinson and M. V. Devarakonda. "Data Cache Management Using Frequency-Based Replacement". In *Proceedings of the ACM SIGMETRICS Conference*, May 1990.

[21] Versant Object Technology Corp. "Versant OODBMS Release 4", 1996.

[22] S. J. White and D. J. DeWitt. "A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies". In *Proceedings of the International Conference on Very Large Data Bases*, Aug. 1992.

[23] P. R. Wilson and S. V. Kakkad. "Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware". In *Proc. 1992 Int'l Workshop on Object Orientation in Operating Systems*, Sept. 1992.

[8] SOP(SNU OODBMS Platform) consists of object storage system, OQL processor, schema manager, DBPL preprocessor, and several visual tools.