Dynamic SEOF: An Adaptable Object Prefetch Policy for Object-oriented Database Systems

JUNG-HO AHN[1] AND HYOUNG-JOO KIM

OOPSLA Laboratory, Department of Computer Engineering, Seoul National University, Shilim-Dong Gwanak-Gu, Seoul 151-742, Korea
Email: {jhahn,hjk}@oopsla.snu.ac.kr

When accessing objects in an object-oriented database, performance can be greatly improved by prefetching objects efficiently. In this paper we present a new object prefetch policy, dynamic selective eager object fetch (SEOF), which prefetches objects only from selected candidate pages without using any high-level object semantics. Our policy considers both correlations and frequencies of fetching objects when selecting prefetch candidates. Unlike existing prefetch policies, dynamic SEOF 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. It also adjusts the degree of prefetching dynamically according to the working environment. We show the performance of the proposed policy through experiments over various multi-client system configurations.

1. INTRODUCTION

Object-oriented databases are widely being deployed in the next generation of telecommunications, Internet and financial applications around the globe due to their rich data model. One of the key issues in such applications is performance, because of their computationally complex data management. To meet their stringent requirements, there have been numerous publications concerning clustering and pointer swizzling, which are major issues in achieving high performance in object management [1, 2, 3, 4, 5, 6].

The most common approach to building an object-oriented DBMS is the data shipping architecture in the client–server environment, e.g. O2 [7], ORION-1SX [8]. 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 that need to be solved if object buffering is to be efficient. The object buffer should handle fragmentation as well as heavy memory allocation and deallocation, since it should be able to manipulate objects of various sizes. Moreover, a buffer consistency protocol may make object-buffer management more difficult [9].

It is also not easy for an object buffer to delimit the span of an object access. This is because the use of the `FIX`/`UNFIX` protocol for page-buffer management results in performance degradation of object access. Thus, efficient buffer replacement is very difficult, if not impossible.

Because of the difficulties related to object buffering, most previous projects [1, 9, 10, 11] have investigated efficient object prefetch policies rather than efficient object-buffer replacement algorithms, in order to increase the object-buffer hit ratio. An early article proposed a policy which exploits high-level object semantics in terms of inheritance and structural relationships [1]. Alternative approaches based on profiling or the learning of object access patterns have been studied in [9, 11].

A prefetch policy that uses high-level object semantics is likely to help the efficient retrieval of complex objects. However, this approach cannot predict general object access patterns which might result from invoking a method [12]. Moreover, it is difficult for an object manager fashioned in a byte server to understand all the semantics of objects because the byte server has no understanding of class, relationship or inheritance. In predictive schemes, it is expensive to keep profiling or learning object access patterns on every access context and it is difficult to realize these policies. In addition, all of the previous studies [1, 2, 5, 9, 13] took little notice of exhaustion and competition for memory and swap resources, since most experiments were performed on small databases. Further, performance issues in multi-client environments have rarely been studied.

We therefore set out to build a new object prefetch policy, dynamic selective eager object fetch (SEOF), 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.

[1]Current address: Samsung Electronics Co., Ltd. 12th Fl., Samsung Plaza Bldg. 263, Seohyeon-Dong, Pundang-Gu, Sungnam-Si, Kyounggi-Do, Korea.

The remainder of the paper is organized as follows. Section 2 discusses in further detail the techniques of object prefetch and related work. In Section 3, we introduce our dynamic SEOF algorithm and the rationale underlying it. Section 4 describes the simulation model and presents the experimental results. In Section 5, we briefly describe some issues in the implementation of our policy. Finally, conclusions from our study and some areas for future research are given in Section 6.

2. TECHNIQUES OF OBJECT PREFETCH

Most object-oriented database applications have a strong tendency to cache a large number of objects in virtual memory and perform extensive computations 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 [8, 9, 13].

To solve this problem, many object-oriented DBMSs are based on the dual-buffer architecture in which an object buffer functions above a page buffer [8, 9]. Examples are Itasca [14], Ontos [15] and Versant [16]. This partitioned buffering approach provides good space utilization by filtering unused objects from the page buffer, and also 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 references to an object is the most significant problem, since the system should know it in order to displace unused objects deliberately. Thus, adopting a replacement algorithm such as LRU to an object buffer might be a difficult task, if not impossible. In addition, paging or buffer replacement in object-oriented database applications, the working cycles of which are characterized by a *load-work-save* model, are less important than in traditional ones [4]. Taking this into consideration, many systems including O2, Objectivity/DB [17], Versant, Mneme [4] and EPVM [5] cache objects in virtual memory without object replacement and an underlying operating system takes all the 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 are not likely to be invalidated by other clients, because of the high read/write ratio and weak data sharing [1, 4, 11].

We can classify object prefetch policies into three categories according to how we select candidate objects for prefetching. The first is the aggressive eager prefetch scheme where all objects are extracted from a page or a segment together upon the first fetch of an object from a particular page or the segment. ORION and ENCORE [18] use this scheme. This policy allows 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 reduce the workload 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 heavy swapping as well as unnecessary copy overhead. Thus, this approach may induce significant performance degradation although it can be of benefit with small and well-clustered databases.

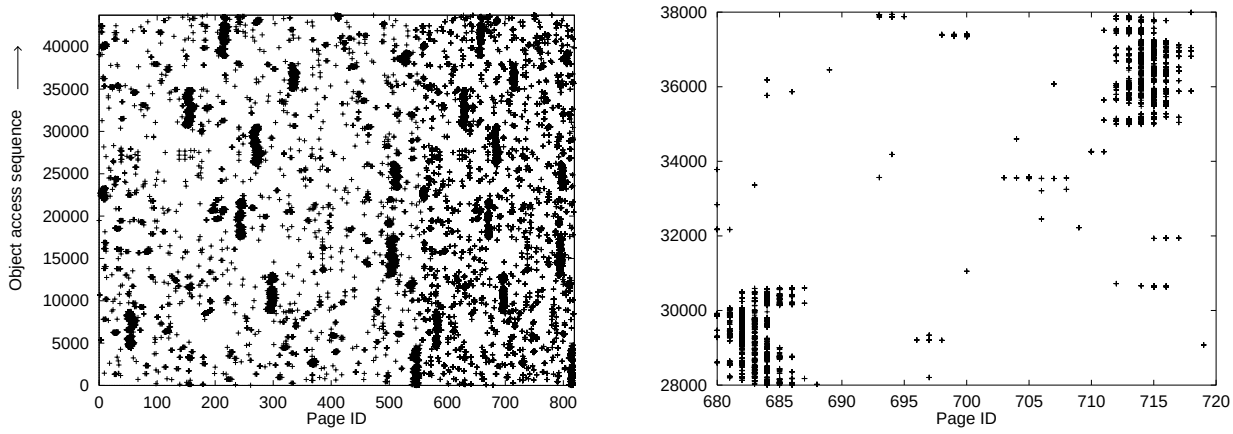
The second policy is equipped with advanced object semantics. That is, all of the objects linked with the requested one are fetched recursively on every object miss. Observing the access patterns of object-oriented database applications, Chang and Katz [1] proposed a run time clustering algorithm and a smart buffering policy which exploits knowledge about inheritance and structural relationships. Their smart buffering uses access hints provided by a user and object relationships to obtain all objects that will be used in advance. It also gives high priority to the pages related with the accessed objects, if they are already cached.

This policy can achieve 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 [9, 11]. That is, this policy may actually cause more unnecessary object fetching than the first one if only some of the relationships are utilized. It also cannot predict general object access patterns and it is difficult to implement this scheme on a byte server.

The third approach is the predictive one, which predicts which objects will be used through profiling or learning access patterns. In the profile-based policy [9], sophisticated buffering hints are stored in a profile to prefetch objects more precisely. As an alternative, Palmer and Zdonik [11] 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 very much accuracy.

Kemper and Kossmann [13] made another effort to improve the performance of object access. They tried to adapt to more complex and variable object patterns by mixing page-based buffering and object-based buffering simultaneously. This combined buffer management technique caches an object as placed in its home page and later copies the object to the object buffer pool if the home page should be replaced. However, it is very difficult to move a fetched object lazily, when the address of the object is used directly by languages like C or C++.

Object prefetch has also been studied in many previous researches on swizzling, since these two topics are closely related [2, 3, 4, 5, 6]. Although these studies made observations mainly about various swizzling techniques, their results exhibit the evaluation of object prefetch schemes indirectly.



(a) (b)

FIGURE 1. Access pattern of OO1 `traversal` operation.

3. DYNAMIC SELECTIVE EAGER OBJECT FETCH POLICY

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

$$\begin{aligned}&\textit{Access Cost}\\&\quad = O_{\text{hit}} \times C_{O_{\text{hit}}}\\&\qquad + O_{\text{miss}} \times (P_{\text{hit}} \times C_{P_{\text{hit}}} + P_{\text{miss}} \times C_{P_{\text{miss}}} + C_{O_{\text{miss}}})\end{aligned}$$

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, $C_{O_{\text{hit}}}$ ($C_{O_{\text{miss}}}$) is the access cost for a hit (missed) object and $C_{P_{\text{hit}}}$ ($C_{P_{\text{miss}}}$) is the access cost for a hit (missed) page. This is explained by the steps in accessing an object: if an object is missed, the page which has the object is fixed and then the object is copied from it. In the same way, if a page is missed, the page should be fetched from a server or a disk device.

Here, $C_{P_{\text{miss}}}$ is much greater than any other cost since it includes the cost of client–server interactions. Therefore, decreasing P_{miss} and O_{miss} is the key for improving the performance of object access.[2] First, an efficient page buffer replacement policy can raise the page buffer hit ratio. Several alternatives are possible including LRU, LFU, LRU-K [19] and 2Q [20]. In object-oriented DBMSs, these replacement polices may be as good as in traditional DBMSs.[3]

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 have already been reviewed in the previous section.

In this section, we first present the basic SEOF method and then we extend the basic version into the dynamic one, which adjusts its behavior dynamically according to working environment.

3.1. Basic policy

Now, we derive our new object prefetch policy, SEOF. 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 [21], 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 and the sequence of object accesses, respectively. Figure 1a, which shows the access pattern when running `traversal` 10 times, indicates the two distinct localities of page accesses on each traversal: one is for accessing `Parts` and the other is for `Connections`. Figure 1b scales up a portion of Figure 1a.

Given this access pattern, we found that a page that contains many objects accessed by the `traversal` operation is continuously referenced at regular intervals. Figure 1b shows some examples: pages 681, 682, . . . , 685. On the other hand, a page that is accessed intermittently

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

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

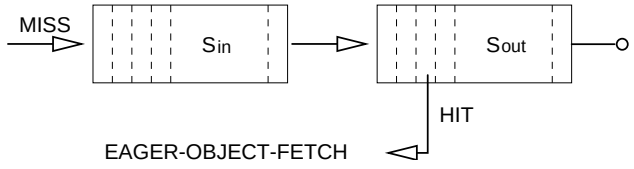


FIGURE 2. S_{in} and S_{out} in SEOF.

is considered to have only a few objects to be fetched. This observation allows us to select the page that has been accessed often as a candidate from which objects are eagerly prefetched.

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

To solve the problem described above, correlations between object fetches should be factored out. Until now, the correlation problem has been considered only by several page buffer management schemes. Frequency-based replacement algorithms exclude the surge of references to a page by not incrementing the reference count if the page has been referenced repeatedly in a short interval [22]. From this, the policy does not give high priority to a page that is accessed repeatedly in a short interval and yet relatively infrequently referenced overall. The problem of correlated references is also mentioned in LRU-K [19] and 2Q [20] in a similar manner.

This factor can be incorporated into the object prefetch policy, where a page is not selected as a candidate if the page is referenced several times only in a short interval. Since a page with correlated references seems to contain only a few objects to be used, fetching only the requested objects would save buffer space. Furthermore, the page is very likely to be hit since the time interval between fetches of objects from the page is short.

As such, our SEOF algorithm is based on the two observations addressed below.

- A 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,[4] S_{in} and S_{out}, as shown in Figure 2. The two queues have lengths $Thresh_{S_{\text{in}}}$ and $Thresh_{S_{\text{out}}}$, respectively. On every fixing of a page for a missed object, SEOF places the page in S_{in}, if the page is not already in either S_{in} or S_{out}. When S_{in} becomes larger than $Thresh_{S_{\text{in}}}$, the first-come entry of S_{in} is moved to S_{out}. The length of S_{out} is maintained in the same way. If the fixed page is in S_{out}, SEOF fetches all uncached objects from the page, but the reference is ignored during its stay in S_{in}.

[4]Experiments with LRU queues gave us similar results.

```
{when the reference to object o, which is located in page
p, is invoked}

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

ALGORITHM 1. Selective eager object fetch.

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 prefetching. Using these two queues, SEOF fetches objects eagerly only from the pages which are likely to have many objects which will be used. The detailed algorithm is given in Algorithm 1.

Our SEOF algorithm works independently of 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.

3.2. Dynamic policy

$Thresh_{S_{\text{in}}}$ and $Thresh_{S_{\text{out}}}$ of SEOF are the important tuning parameters, since the sizes of the two FIFO queues decide the degree of object prefetching. Qualitatively, as $Thresh_{S_{\text{in}}}$ gets smaller and $Thresh_{S_{\text{out}}}$ gets larger, SEOF prefetches objects more aggressively. On the other hand, SEOF does less prefetching because the test for candidate selection is too strong, when $Thresh_{S_{\text{in}}}$ is large and $Thresh_{S_{\text{out}}}$ is small.

In order to test the sensitivity of SEOF to its parameters, we conducted experiments with various values of $Thresh_{S_{\text{in}}}$ and $Thresh_{S_{\text{out}}}$. The `traversal` operation was repeated on the small OO1 database with $Thresh_{S_{\text{in}}}$ and $Thresh_{S_{\text{out}}}$

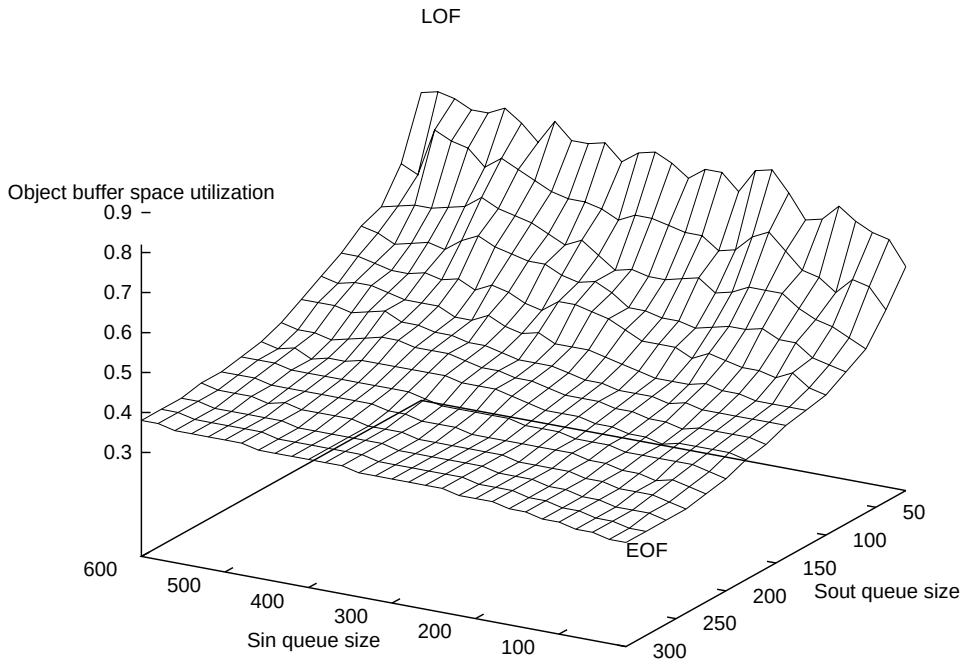


FIGURE 3. Sensitivity of SEOF to $Thresh_{S_{in}}$ and $Thresh_{S_{out}}$.

ranging from 20 to 600. The results are plotted in Figure 3. 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 above, SEOF moves closer to a non-prefetching policy 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 the aggressive eager prefetching algorithm. SEOF was insensitive to $Thresh_{S_{out}}$ values larger than 300 and that portion of the results is not plotted. This is because the maximum interval between accesses to a given page was small in these experiments. The labels LOF and EOF in Figure 3, lazy object fetch and eager object fetch respectively, represent the results for the non-prefetching policy and the aggressive eager prefetching policy, respectively.

Although SEOF can cover a wide range of prefetching behavior, as shown in Figure 3, it is not easy to find optimal values for $Thresh_{S_{in}}$ and $Thresh_{S_{out}}$. In addition, no single value of $Thresh_{S_{in}}$ and $Thresh_{S_{out}}$ could give the best performance in all working environments. Therefore, it is necessary to adjust $Thresh_{S_{in}}$ and $Thresh_{S_{out}}$ dynamically according to such environment factors as the workloads of the system components, such as the server or the network, and contention for the resources of the client, such as physical memory and swap space. We therefore developed the dynamic version of SEOF, which adjusts $Thresh_{S_{in}}$ and $Thresh_{S_{out}}$ dynamically according to the working environment.

As we showed in our previous work [10], the degree of optimal prefetching depends on the system workload. When a server or a network is overloaded, it is beneficial to reduce

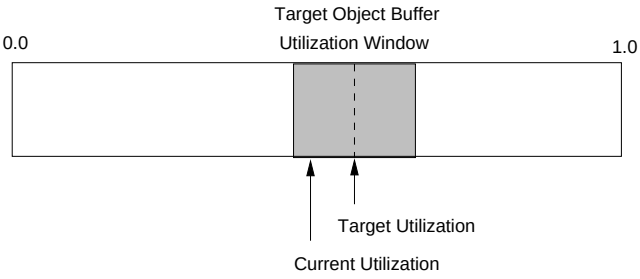
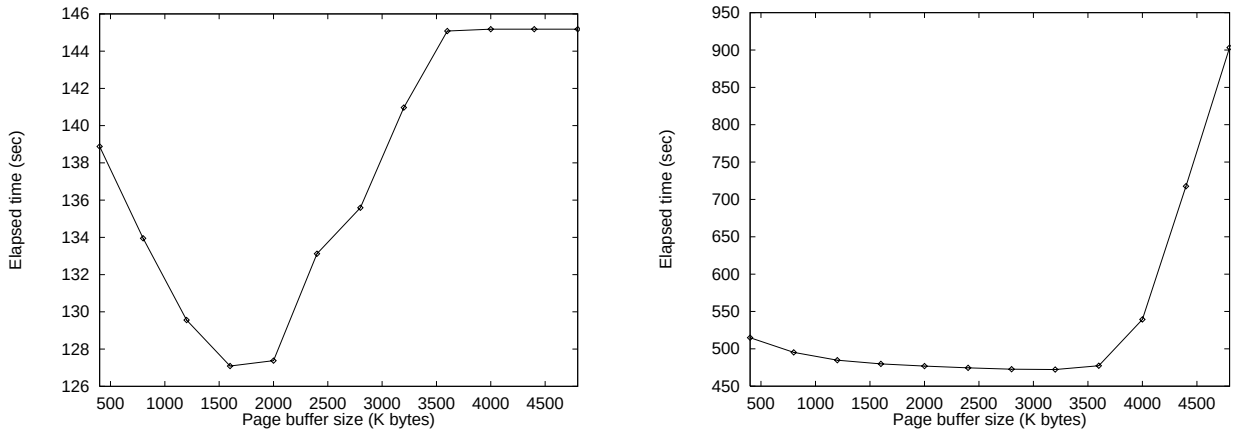


FIGURE 4. Object buffer utilization window.

client-server interactions by prefetching objects aggressively and caching many objects in clients. That is, aggressive prefetching could reduce the average response time of page requests to the server and thus, improve the overall system performance. However, when the number of clients is small and the system workload is low, the advantage of eager prefetching is lost because of the heavy swapping that results from fetching too many objects. That is, swap I/O becomes a major bottleneck, since the server response time is short. In this case, system performance could be improved by prefetching fewer objects.

Therefore, dynamic SEOF determines the degree of prefetching based on the current system workload, and then adjusts $Thresh_{S_{in}}$ and $Thresh_{S_{out}}$ accordingly.

Our dynamic SEOF algorithm can be sketched as follows. First, periodically or after some fixed number of runs of the SEOF algorithm, dynamic SEOF determines the target degree of prefetching by examining the current system workload. Here, the degree of object prefetching can be expressed numerically by the utilization of objects in the



(a) Small DB

(b) Large DB

FIGURE 5. Effects of the page buffer size.

buffer. That is, the ratio of actually used objects to (pre) fetched objects measures the aggressiveness of prefetching. For instance, the non-prefetching policy used all the fetched objects, so its utilization is 1.0, but as the aggressiveness of prefetching gets stronger the utilization of objects in the buffer decreases. Thus

$$\text{new target utilization} = F'(\text{current system workload}),$$

where the function F' is inversely proportional to the system workload. Any function may be used for F' if it can reflect the system workload well. For example, we can consider a simple function that increases the target utilization by $\Delta_{\text{utilization}}$ when the system workload is lower than a $LowThreshold_{\text{workload}}$ and decreases the target utilization by $\Delta_{\text{utilization}}$ if the system workload is higher than a $HighThreshold_{\text{workload}}$.

However, the target utilization should have a limited range in order to prevent resource exhaustion. As mentioned in our previous work [10], it might not be feasible to run aggressive prefetching in a real environment, especially for large database applications, since it may require too much swap space by prefetching many unnecessary objects blindly. Thus, dynamic SEOF should set minimum target utilization according to the resources of the client, and should not allow a target utilization below that minimum. The restrictions on object buffer usage will be discussed further in Section 4.

After the target utilization is determined, dynamic SEOF compares the current utilization with the target utilization and then adjusts $Thresh_{S_{\text{in}}}$ and $Thresh_{S_{\text{out}}}$ in order to approach the target utilization. Therefore, $Thresh_{S_{\text{in}}}$ and $Thresh_{S_{\text{out}}}$ are a function of the target utilization of objects in the buffer,

$$\text{new } \{Thresh_{S_{\text{in}}}, Thresh_{S_{\text{out}}}\} = F''(\text{target utilization, current utilization}).$$

Similar to the previous function F', we can consider a function F'' that increases $Thresh_{S_{\text{in}}}$ and decreases $Thresh_{S_{\text{out}}}$ by Δ_{queue} when the current utilization is lower than the target utilization. If the current utilization is higher than the target utilization, the function F'' decreases $Thresh_{S_{\text{in}}}$ and increases $Thresh_{S_{\text{out}}}$ by Δ_{queue}. However, dynamic SEOF should not be too sensitive to variations of the object utilization. For this problem, we can use a window concept as shown in Figure 4.

That is, the function F'' allows a certain amount of difference between the target utilization and the current utilization by adjusting $Thresh_{S_{\text{in}}}$ and $Thresh_{S_{\text{out}}}$ only if the current utilization is outside the target utilization window.

In a similar way, the initial values of $Thresh_{S_{\text{in}}}$, $Thresh_{S_{\text{out}}}$ and the target utilization may be determined from the system workload at start time.

Finally, the size of the page buffer of the client should be considered in order to prevent resource contention at clients. We performed additional experiments to examine the memory resource contention by the object buffer and the page buffer. We ran the `traversal` operation on the OO1 database clustered by 90%–1% factor varying the page buffer size from 400 to 4800 K bytes. The results, as shown in Figure 5, confirm our intuition that a large page buffer may suffer from heavy swapping, although it gives high buffer hit ratio.

In addition, the optimal size of the page buffer depends on the degree of object prefetching. That is, if the prefetching is aggressive—when the utilization of objects in the buffer is low—only a small page buffer is needed, since many objects are cached in the object buffer and the small page buffer may reduce resource contention. For instance, the eager prefetching policy requires only one page buffer frame, since it prefetches all the objects in a page and caches them in the object buffer. On the other hand, when only a small number of objects are prefetched, a large page buffer may improve the system performance by increasing the number of page buffer hits. Dynamic SEOF therefore adjusts the

Dynamic SEOF
run the basic SEOF algorithm
if adjusting $Thresh_{S_{in}}$ and $Thresh_{S_{out}}$ is required **then**
  calculate the new target utilization of objects in the buffer
  calculate the new $Thresh_{S_{in}}$ and $Thresh_{S_{out}}$
  resize the S_{in} and S_{out} queue according to the new $Thresh_{S_{in}}$ and $Thresh_{S_{out}}$
  calculate the new page buffer size and resize the page buffer
end if

Function F'
{Calculate the new target utilization of objects in the buffer}
get the current system workload
if the current system workload $>$ $HighThreshold_{\text{workload}}$ **then**
  move the current target object buffer utilization window left by $\Delta_{\text{utilization}}$
else if the current system workload $<$ $LowThreshold_{\text{workload}}$ **then**
  move the current target object buffer utilization window right by $\Delta_{\text{utilization}}$
end if

Function F''
{Calculate the new $Thresh_{S_{in}}$ and $Thresh_{S_{out}}$}
if the current utilization is in the right side of the target utilization window **then**
  decrease $Thresh_{S_{in}}$ and increase $Thresh_{S_{out}}$ by Δ_{queue}
else {the current utilization is in the left side}
  increase $Thresh_{S_{in}}$ and decrease $Thresh_{S_{out}}$ by Δ_{queue}
end if

Function F'''
{Calculate the new page buffer size}
a new page buffer size $\Leftarrow$ (the current utilization $\times\ \alpha + \beta$) $\times$ the maximum page buffer size

ALGORITHM 2. Dynamic SEOF.

page buffer size dynamically according to the degree of object prefetching. As we mentioned above, the degree of aggressiveness can be measured as object buffer utilization. Thus, dynamic SEOF resizes the page buffers of clients according to current utilization of objects in the buffer. That is,

$$\text{new page buffer size} = F'''(\text{current utilization}).$$

We can use a function F''' that returns a fraction of the maximum page buffer size according to the current utilization.

The detailed algorithm for dynamic SEOF is given in Algorithm 2.[5]

4. PERFORMANCE EVALUATION

4.1. Simulation model

The performance evaluation employed in this study is based on a page-server architecture using dual-buffering.[6] The conceptual structure of the simulation model is shown in Figure 6. Several components are simulated in the model: `CPU`, `DISK`, `VM`, `PAGE BUFFER`, `OBJECT BUFFER` and `NETWORK`.

The `VM` component of the simulator is used for modeling page faults and swap I/Os. `VM` manages physical memory by page aging [23], where the pages that are no longer part of working set are aged at regular intervals. When there is no available physical memory, `VM` swaps out some of the oldest pages. In the simulator, a virtual memory block is 4 Kbytes long. The `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 a first-come-first-served basis and concurrency control is simulated by piggybacking page-level locks on page requests.

Table 1 presents the parameter settings used in the experiments. The values listed in the table were derived from measurements of our own object manager, which is based on the page-server architecture.

We used the `traversal` operation of the OO1 benchmark as the work load. Two databases are used: the medium database of 50,000 `Parts` and the large

[5] In Algorithm 2 we give the functions F', F'' and F''', which were used in our experiments. However, any good functions can be used with our algorithm.

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

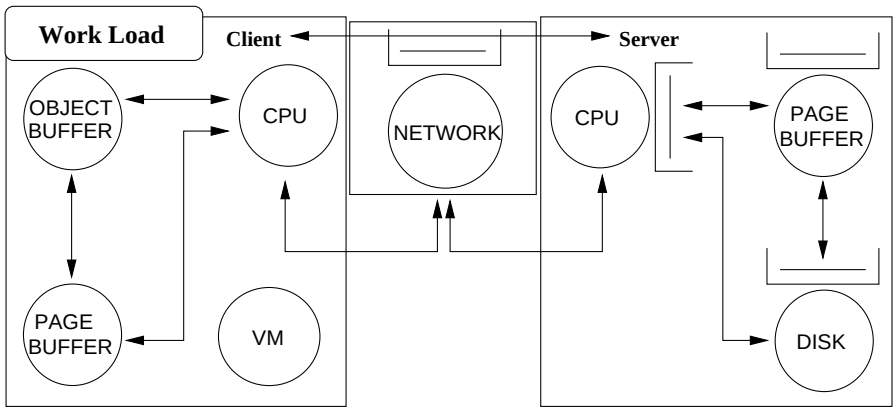


FIGURE 6. Simulation model.

TABLE 1. Simulation parameter settings.

Parameters		Value (ms)
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 kbytes
Network processing time		$1.1 + 0.00075$ byte^{-1}
Network transfer time		4.5 Mbits s^{-1}
OO1	`Part` processing time	5.0/`Part`
	`Connection` processing time	0.5/`Connection`
OO7	Object access time	0.5/object

database of 200,000 `Parts`.[7] With `Part` of 200 bytes and `Connection` of 32 bytes, the medium and large database sizes are 16 Mbytes and 64 Mbytes, respectively. The page size is 8 Kbytes. We used two alternative clustering factors, 90%–1% and 80%–5% in order to evaluate the performance with varying degrees of clustering. Each client generates a single stream of object accesses 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. We also performed the simulation with the `traversal 1` operation of the OO7 benchmark [24]. We used the small and medium databases of the OO7 benchmark, where each atomic part object is connected to three other atomic parts. The databases are 4 Mbytes and 25 Mbytes, respectively.

In all experiments, we pre-ran each operation twice so that the dynamic SEOF policies get the initial values of queue sizes and the target utilization by collecting the system workload. The simulator was coded in C++SIM [25], since we can use C++ directly without learning another language, and it is publicly available.

[7] We eliminated the results on the small database of 20,000 `Parts` from this paper because the database is small enough to fit in physical memory and so gives little information on our algorithm. The results using the small database can be found in [10].

4.2. Simulation results

Experiments were performed to compare three different prefetch policies:

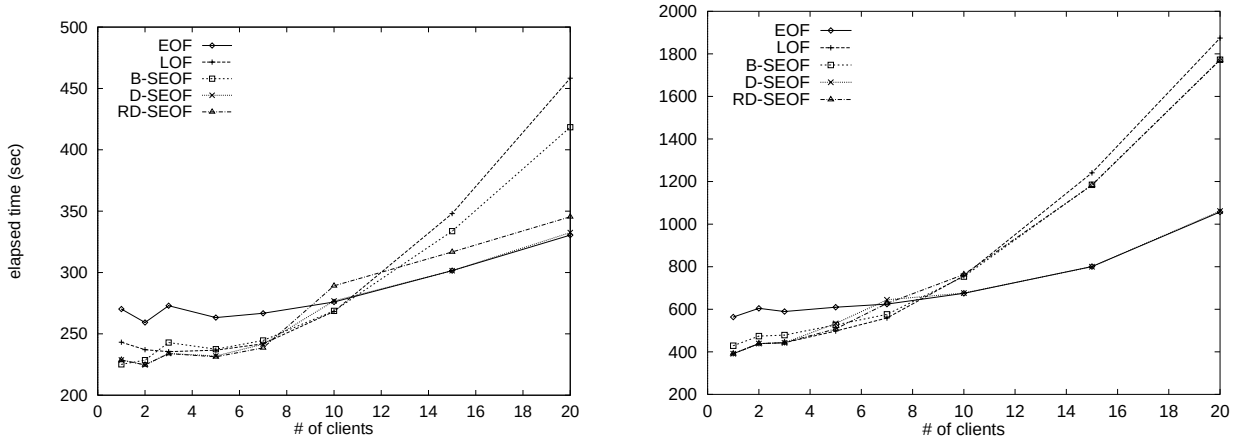
(i) EOF that fetches all objects eagerly from a page upon the first object miss in the page;
(ii) LOF that fetches only one requested object at a time; and
(iii) the SEOF policies we proposed: the basic SEOF (B-SEOF) and its two derivatives, dynamic SEOF (D-SEOF) and restricted dynamic SEOF (RD-SEOF).

RD-SEOF works like D-SEOF but it limits the range of utilizations of objects in the buffer in order to prevent resource exhaustion.

EOF and LOF set the page buffer size of the client to 8 Kbytes and 4.3 Mbytes, respectively. This offered the best performance for these two cases in our experiments. In the experiments we ran B-SEOF varying the queue sizes. However, in this paper, we have presented only one of the results, which gives reasonable performance under the constraint of 10 Mbytes object buffer space. Its $Thresh_{S_{\text{in}}}$ and $Thresh_{S_{\text{out}}}$ were 120 and the size of the client page buffer was set to 2.7 Mbytes. In the experiments with the two dynamic versions of SEOF, we used only the network utilization as the system workload, since it was the major bottleneck of the system performance in our experiments. The high and low system workload thresholds were 0.55 and 0.3, respectively, and $\Delta_{\text{utilization}}$ was set to 0.1. We resize $Thresh_{S_{\text{in}}}$ and $Thresh_{S_{\text{out}}}$ when the current utilization is beyond 20% of the target utilization[8] and Δ_{queue} was 8. In RD-SEOF, the minimum object buffer utilization was configured to limit the object buffer space up to about 12 Mbytes. A server page buffer size of 5.5 Mbytes and 5 Mbytes of client physical memory were used for all of the experiments.

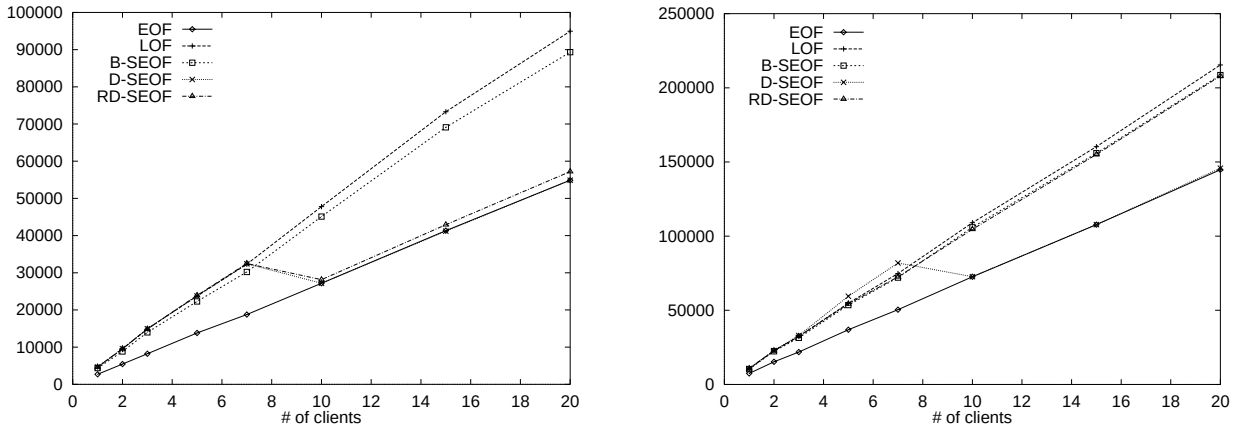
We first present the results of the simulation for the OO1 benchmark databases clustered by 90%–1% factor.

[8] We varied the window size according to the target utilization, since we thought that the degree of change in the utilization depended on it. However, the experiments with static window sizes gave similar results.



(a) Medium DB (b) Large DB

FIGURE 7. Average elapsed time (s) (OO1, 90%–1%).



(a) Medium DB (b) Large DB

FIGURE 8. Total number of page requests (OO1, 90%–1%).

Figure 7 shows the average elapsed time of running `traversal` 10 times. As we expected, non (or less) prefetching policies outperform the eager prefetching policy in the range of 10 or fewer clients. This is because the eager prefetching policy (like EOF) suffers naturally from heavy swapping, since it tends to prefetch many unneeded objects. However, as clients are added, the small number of page requests in the eager prefetching policy can pay off the swapping overhead by reducing the response time of the server. This fact is reinforced by Figure 8, which plots the total number of page requests and Table 2, which represents the average response time of the servers. In Figure 7, we can 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 is lost because the server tends to become overloaded due to contention against the server page buffer.

Observing the performance of SEOF policies, the curve of B-SEOF on the medium database lies in the middle of EOF and LOF, but B-SEOF is very close to LOF on the large database. The reason is that the large working set of the `traversal` operation prevents B-SEOF from prefetching enough objects to keep the hit ratio high. However, D-SEOF gives the best performance at almost all configurations. D-SEOF prefetches small numbers of objects when the system workload is low, but it prefetches objects more aggressively as the system becomes more overloaded. This behavior of D-SEOF can also be found in Table 3 which represents the size of the object buffer used during experiments as well as Figure 8 and Table 2. These results demonstrate the ability of dynamic SEOF to adjust the queue sizes well according to the system workload.

TABLE 2. Average response time of the server (s) (OO1, 90%–1%).

	No. of clients							
Strategies	1	2	3	5	7	10	15	20
Medium DB								
EOF	0.035	0.033	0.034	0.035	0.038	0.043	0.063	0.096
LOF	0.034	0.034	0.034	0.037	0.039	0.047	0.071	0.112
B-SEOF	0.033	0.033	0.034	0.036	0.039	0.047	0.071	0.112
D-SEOF	0.034	0.034	0.034	0.036	0.039	0.045	0.064	0.099
RD-SEOF	0.034	0.034	0.034	0.037	0.039	0.045	0.066	0.100
Large DB								
EOF	0.036	0.037	0.038	0.042	0.047	0.059	0.099	0.162
LOF	0.036	0.038	0.040	0.047	0.054	0.078	0.138	0.211
B-SEOF	0.035	0.037	0.040	0.046	0.053	0.076	0.135	0.208
D-SEOF	0.035	0.038	0.040	0.046	0.053	0.060	0.100	0.162
RD-SEOF	0.036	0.038	0.040	0.046	0.053	0.075	0.133	0.204

TABLE 3. Object buffer size (Mbytes) (OO1, 90%–1%).

	No. of clients							
Strategies	1	2	3	5	7	10	15	20
Medium DB								
EOF	13.0	12.8	13.1	13.0	12.9	13.0	13.1	12.9
LOF	2.1	2.1	2.2	2.2	2.1	2.2	2.1	2.1
B-SEOF	4.9	5.4	5.5	5.3	5.4	5.3	5.4	5.2
D-SEOF	2.1	2.1	2.2	2.1	2.2	13.0	13.0	13.0
RD-SEOF	2.1	2.1	2.2	2.1	2.2	12.8	13.0	12.8
Large DB								
EOF	37.0	39.3	37.7	38.0	37.9	38.0	37.6	37.9
LOF	2.8	2.8	2.8	2.8	2.8	2.8	2.7	2.8
B-SEOF	11.3	10.0	10.6	10.5	10.2	10.4	10.2	10.3
D-SEOF	2.8	2.8	2.8	4.1	5.6	38.0	37.5	38.2
RD-SEOF	2.8	2.8	2.8	4.3	7.0	9.1	9.6	10.3

In the medium-sized database, dynamic versions of SEOF (D-SEOF and RD-SEOF) show better performance than LOF when the number of clients is small. This can be explained by the fact that, although the object buffer spaces used by these policies are similar, LOF experienced more swapping since it used more page buffer space than the dynamic SEOF policies.

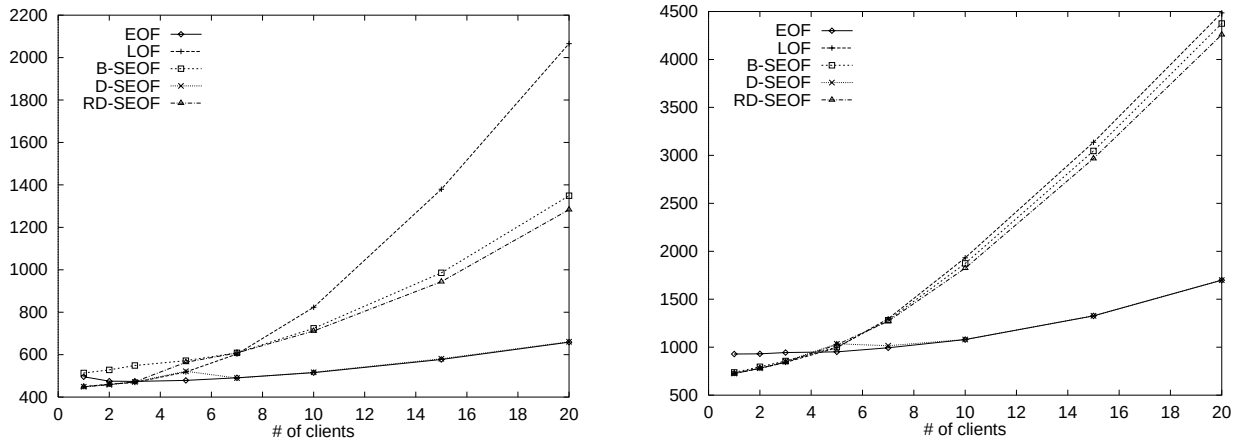
Lastly, the restricted version is less dynamic than D-SEOF because of the buffer space restriction, and thus its performance is somewhat inferior to D-SEOF when the system workload is high. However, when the system workload is low, the restriction on object buffer space does not affect the behavior of RD-SEOF since it works like LOF; thus it performs as well as D-SEOF.

Compared to B-SEOF, RD-SEOF shows better performance than the basic policy in the medium database. However, the performance of the two policies is almost the same in the large database, since the restricted version was allowed to use an object buffer space as large as that of B-SEOF. Table 3 shows clearly the limitation on object buffer space of RD-SEOF.

The surge of the performance curve of RD-SEOF at 10 clients with the medium database can be explained by the fact that RD-SEOF uses more client page buffer space than other policies like EOF or D-SEOF, resulting in more swapping.

Not surprisingly, the size of the buffer space used by each policy is proportional to the aggressiveness of object prefetching and the static versions like EOF, LOF, and B-SEOF did not change the buffer usage regardless of the system workload. Unlike these policies, the dynamic versions, like D-SEOF and RD-SEOF, prefetched more objects and thus required more object buffer space as the number of clients increased.

In our experiments, we tuned the system workload thresholds so that the dynamic versions of SEOF worked aggressively, since the system workload (especially network load) was increased sharply as the number of clients



(a) Medium DB

(b) Large DB

FIGURE 9. Average elapsed time (s) (OO1, 80%–5%).

increased. Thus, these dynamic versions changed their behavior substantially.

Table 3 also reveals that it might be infeasible to run eager prefetching policies like EOF or D-SEOF in a real environment especially for large database applications.[9] That is, a system with 5 Mbytes of physical memory may not allow EOF or D-SEOF to use about 40 Mbytes of swap space. This is because these policies tend to prefetch too many unnecessary objects blindly, as noted in Section 2. This is also the reason why we include the restricted version of dynamic SEOF in our simulations, which can limit the aggressiveness of object prefetching according to client resources.

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

Figure 9 plots the average elapsed time for the traversal on the databases clustered by a factor of 80%–5%. These graphs show similar results to those for a clustering factor of 90%–1%. However, with increasing numbers 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 at the clients. The point of intersection of LOF and EOF moves left for the same reason.

Comparing the results for B-SEOF with the previous ones, it moves closer to EOF in the experiments with the medium database, but it behaves more like LOF for the large database. This is explained as follows. With a clustering factor of 80%–5%, the medium database has fewer cycles between `Part` objects, while almost all of the working set of the medium database can still be cached as before. Thus, B-SEOF with the poorly clustered medium database selects more pages as candidates for prefetching. On the other hand, the low clustering factor in the large database causes only a few objects in a page to be used. As a result, B-SEOF selects fewer pages as candidates on the poorly clustered large database. This behavior is also explained in Table 4, which represents the size of the object buffer used during the experiments for a clustering factor of 80%–5%.

Dynamic versions of SEOF work well, as in the previous experiments. However, RD-SEOF works more like the B-SEOF policy with the medium database, but in the large database the performance gap between these two policies becomes wider than in the previous result. The reason is that in the medium database, B-SEOF uses more buffer space because of the low clustering factor, but with the poorly clustered large database, RD-SEOF prefetches more objects than B-SEOF because of the heavy workload.

The total number of page requests and the average response time of the server for a clustering factor of 80%–5% are not given in the paper, since the results are consistent with those of the previous experiment.

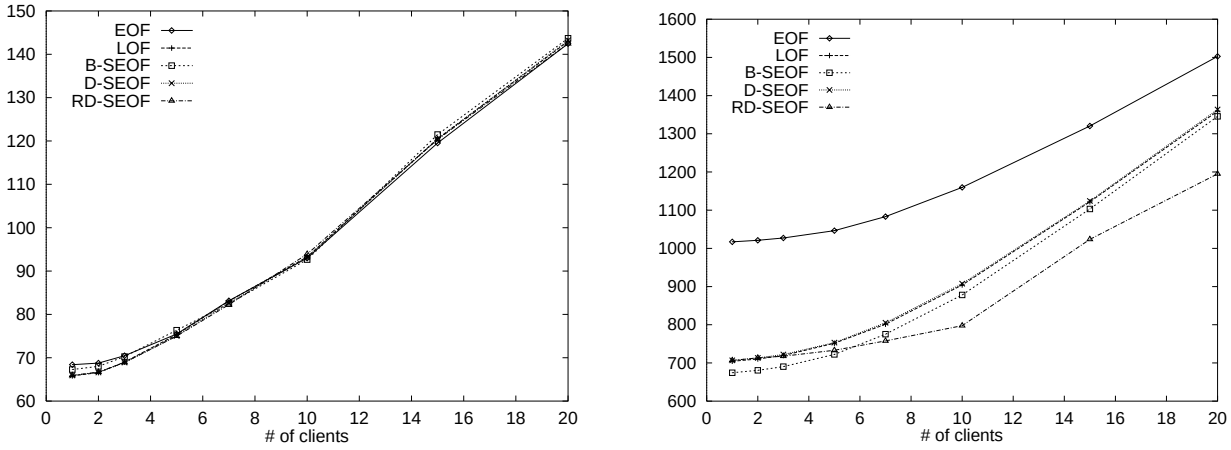
We also evaluate our algorithm with the OO7 `traversal 1` operation. The elapsed time of running a cold traversal appears in Figure 10. We do not give the results for the hot traversal, which consists of first running a cold traversal and then running the same traversal three more times, since the results for the hot traversal are similar to those for the cold traversal.

For the small database, all policies have a similar performance because of the high degree of locality in the traversal operation. In addition, the working set is small enough to be cached in its entirety in the client memory, and thus, there is no difference in the response time to page requests among policies. In this traversal, the performance of EOF was worse than other policies because of object copy overhead, although it is not easy to see in the figure. That is, most objects in the small database can be fetched from the client page buffer since the client caches all of the small database and thus, the response times to fetch one object are similar for all policies. However, EOF incurred a copy

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

TABLE 4. Object buffer size (Mbytes) (OO1, 80%–5%).

	No. of clients							
Strategies	1	2	3	5	7	10	15	20
Medium DB								
EOF	14.0	14.0	14.0	14.0	14.0	14.0	14.0	14.0
LOF	2.5	2.6	2.6	2.6	2.6	2.6	2.6	2.6
B-SEOF	8.4	8.9	9.1	8.8	9.1	9.0	8.9	8.9
D-SEOF	2.5	2.6	2.6	2.6	14.0	14.0	14.0	14.0
RD-SEOF	2.5	2.6	2.6	12.7	12.5	12.8	12.5	12.4
Large DB								
EOF	48.7	48.2	49.0	48.1	48.5	48.2	48.3	48.6
LOF	3.0	2.9	2.9	2.9	2.9	2.9	2.9	2.9
B-SEOF	7.2	7.3	7.3	7.8	7.5	7.5	7.5	7.5
D-SEOF	3.0	3.0	3.0	3.8	48.4	48.2	48.5	48.6
RD-SEOF	3.0	2.9	3.1	8.6	9.1	8.6	8.9	8.8



(a) Small DB

(b) Medium DB

FIGURE 10. Average elapsed time (s) (OO7, cold).

overhead by prefetching many unneeded objects, degrading the performance.

Even for the medium database, LOF also has an advantage over EOF for the same reason, that is, clients experienced high cache hit ratios, over 90%, because of the high degree of locality of traversal. However, EOF went from bad to worse, since EOF suffered from heavy swapping by prefetching too many objects (see Figure 11).

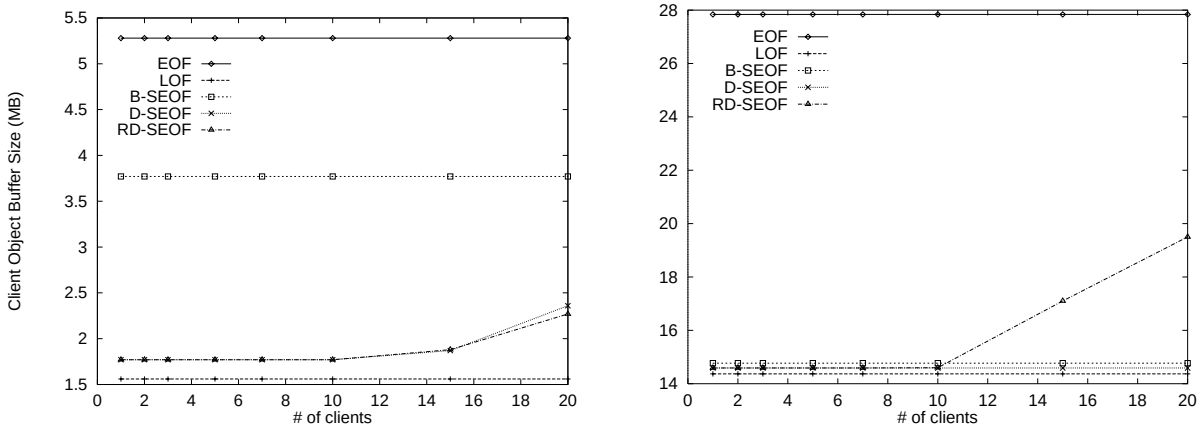
On the other hand, all versions of SEOF show better performance than LOF. This is because SEOF policies reduced the response time for page requests by prefetching (not too many) objects without incurring heavy swapping. Among SEOF policies, the performance of B-SEOF is better than the dynamic versions in the experiments with small numbers of clients, since the basic version prefetches more objects than the other versions, as shown in Figure 11, which plots the object buffer usages of each policy. Surprisingly, RD-SEOF shows the best performance in the case of seven or more clients. This can be explained by the fact that RD-SEOF starts the traversal by prefetching objects more aggressively, since the system workload is heavier than in the case of D-SEOF.[10]

5. IMPLEMENTATION

Our SEOF algorithm can be employed easily without changing an existing object manager. The SEOF algorithm requires only two FIFO queues, S_{in} and S_{out}, and a few interfaces to manage these queues, which would be called only by the object buffer manager. Figure 12 shows an example of the implementation of SEOF algorithm and the calling sequence from the object buffer manager.

For dynamic versions of SEOF, the resizing function

[10] In the experiments with a medium database, we tuned the restricted dynamic SEOF to use object buffer space up to 20 Mbytes, since the total size of accessed objects is about 14 Mbytes.



(a) Small DB (b) Medium DB

FIGURE 11. Object buffer size (Mbytes) (OO7, cold).

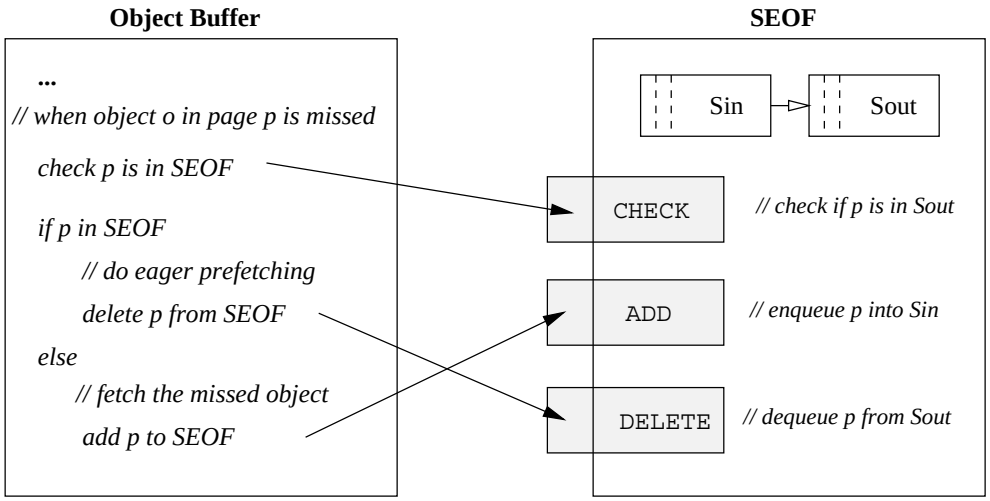


FIGURE 12. The calling sequence for the SEOF policy.

should be included and it should be called periodically or following some number of calls to the CHECK function, where the queue sizes are adjusted dynamically according to the system workload. Information on the system workload can be acquired from the underlying operating system or can be computed using the number of clients or the number of requests to the server.

6. CONCLUSION AND FUTURE RESEARCH

In this paper we have developed a new object prefetch policy, dynamic selective eager object fetch, which prefetches objects only from selected candidate pages without using any high-level object semantics and which adjusts its behavior dynamically according to the working environment. Our policy is based on two observations:

(i) the page which has been referenced repeatedly in a short interval seems to have only a few objects to be fetched; and

(ii) 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, dynamic SEOF utilizes the memory and the swap space of clients efficiently without resource exhaustion. It is also easy to implement.

The results of our experiments indicate that object prefetch can improve overall performance significantly, although performance may suffer from heavy swapping caused by prefetching too many unneeded objects. Dynamic versions of SEOF offered the best performance over almost all configurations. In addition, the restricted version of dynamic SEOF limited its object buffer space well.

The experiments with a low clustering factor confirm that the dynamic SEOF policy has good adaptability to both the effectiveness of clustering and database size. It also shows good performance with the OO7 benchmark. From these results, we are sure that our dynamic SEOF is general enough to be used in real environments. Furthermore, our algorithm induces little overhead, since it does not

require any new information except to manage the two FIFO queues.

We are currently implementing the proposed dynamic SEOF policy on top of our ODMG-93 compliant object-oriented DBMS, SOP.[11] In the future, we would like to extend our prefetch policy so that a series of unused prefetched objects can be displaced efficiently. We will also develop functions (like F' and F'') that can adjust $Thresh_{S_{in}}$ and $Thresh_{S_{out}}$ smoothly according to access patterns as well as the system workload. We conjecture that these two methods could substantially improve the performance of dynamic SEOF.

ACKNOWLEDGEMENT

This work was partially supported by the Korea Ministry of Science and Technology under project 'A Development on Internet Transaction Processing Technology using OOT'.

REFERENCES

[1] Chang, E. E. and Katz, R. H. (1989) Exploiting inheritance and structure semantics for effective clustering and buffering in an object-oriented DBMS. In Clifford, J., Lindsay, B. G. and Maier, D. (eds), *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Portland, OR, pp. 348–357. ACM Press.

[2] Hosking, A. L. and Moss, J. E. B. (1993) Object fault handling for persistent programming languages: a performance evaluation. In Paepcke, A. (ed.), *Proc. OOPSLA'93*, *ACM SIGPLAN Notices*, **28**, 288–303. ACM Press.

[3] Kemper, A. and Kossmann, D. (1993) Adaptable pointer swizzling strategies in object bases: design, realization, and quantitative analysis. In *Proc. Int. Conf. on Data Engineering*, Vienna, Austria, pp. 155–162. IEEE Computer Society.

[4] Moss, J. E. B. (1992) Working with persistent objects: to swizzle or not to swizzle. *IEEE Trans. Software Eng.*, **18**, 657–673.

[5] White, S. J. and DeWitt, D. J. (1992) A performance study of alternative object faulting and pointer swizzling strategies. In Yuan, L.-Y. (ed.), *Proc. Int. Conf. on Very Large Data Bases*, Vancouver, Canada, pp. 419–431. Morgan Kaufmann.

[6] Wilson, P. R. and Kakkad, S. V. (1992) Pointer swizzling at page fault time: efficiently and compatibly supporting huge address spaces on standard hardware. In *Proc. 1992 Int. Workshop on Object Orientation in Operating Systems*, Paris, France, pp. 364–377. IEEE Computer Society.

[7] Bancilhon, F., Delobel, C. and Kanellakis, P. (eds) (1992) *Building an Object-Oriented Database System: The Story of O2*. Morgan Kaufmann Publishers, San Mateo, CA.

[8] Kim, W., Garza, J. F., Ballou, N. and Woelk, D. (1990) Architecture of the ORION next-generation database system. *IEEE Trans. Knowledge and Data Engineering*, **2**, 109–124.

[9] Cheng, J. R. and Hurson, A. R. (1991) On the performance issues of object-based buffering. In *Proc. Int. Conf. on Parallel and Distributed Information Systems*, Miami, FL, pp. 30–37. IEEE Computer Society.

[10] Ahn, J.-H. and Kim, H.-J. (1997) SEOF: an adaptable object prefetch policy for object-oriented database systems. In Gray, A. and Larson, P.-Å (eds), *Proc. Int. Conf. on Data Engineering*, Birmingham, UK, pp. 4–13. IEEE Computer Society.

[11] Palmer, M. and Zdonik, S. B. (1991) Fido: a cache that learns to fetch. In Lohman, G. M., Sernadas, A. and Camps, R. (eds), *Proc. Int. Conf. on Very Large Data Bases*, Barcelona, Spain, pp. 255–264. Morgan Kaufmann.

[12] Dewitt, D. J. and Maier, D. (1990) A study of three alternative workstation–server architectures for object-oriented database systems. In McLeod, D., Sacks-Davis, R. and Schek, H.-J. (eds), *Proc. Int. Conf. on Very Large Data Bases*, Brisbane, Australia, pp. 107–121. Morgan Kaufmann.

[13] Kemper, A. and Kossmann, D. (1994) Dual-buffering strategies in object bases. In Bocca, J. B., Jarke, M. and Zaniolo, C. (eds), *Proc. Int. Conf. on Very Large Data Bases*, Santiago de Chile, Chile, pp. 427–438. Morgan Kaufmann.

[14] IBEX Object Systems, Inc. (1995) *ITASCA Technical Summary Release 2.3*.

[15] Ontos, Inc. (1996) *ONTOS Product Description*.

[16] Versant Object Technology Corp. (1996) *Versant OODBMS Release 4*.

[17] Objectivity, Inc. (1995) *Objectivity/DB Technical Overview Version 3*.

[18] Hornick, M. F. and Zdonik, S. B. (1987) A shared, segmented memory system for an object-oriented database. *ACM Trans. Office Inf. Syst.*, **5**, 70–95.

[19] O'Neil, E. J., O'Neil, P. E. and Weikum, G. (1993) The LRU-K page replacement algorithm for database disk buffering. In Buneman, P. and Jajodia, S. (eds), *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Washington, DC, pp. 297–306. ACM Press.

[20] Johnson, T. and Shasha, D. (1994) 2Q: a low overhead high performance buffer management replacement algorithm. In Bocca, J. B., Jarke, M. and Zaniolo, C. (eds), *Proc. Int. Conf. on Very Large Data Bases*, Santiago de Chile, Chile, pp. 439–450. Morgan Kaufmann.

[21] Cattell, R. G. G. and Skeen, J. (1992) Object operations benchmark. *ACM Trans. Database Syst.*, **17**, 1–31.

[22] Robinson, J. T. and Devarakonda, M. V. (1990) Data cache management using frequency-based replacement. In *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, Boulder, CO, May 22–25, pp. 134–142.

[23] Bach, M. J. (1986) *The Design of the UNIX Operating System*. Prentice-Hall, Englewood Cliffs, NJ.

[24] Carey, M., DeWitt, D. J. and Naughton, J. F. (1993) The OO7 Benchmark. In Buneman, P. and Jajodia, S. (eds), *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Washington, DC, pp. 12–21. ACM Press.

[25] University of Newcastle-upon-Tyne (1995) *C++SIM User's Guide, Public Release 1.5*. http://cxxsim.ncl.ac.uk

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