A Multi-Threaded Architecture for Prefetching in Object Bases

Carsten A. Gerlhof and Alfons Kemper

Universität Passau, Lehrstuhl für Dialogorientierte Systeme,
Fakultät für Mathematik und Informatik, D-94030 Passau, Germany,
[*gerlhof* | *kemper*]@db.fmi.uni-passau.de

Abstract. We propose a generic architectural framework, a *multi-threaded* run-time system for client/server architectures, which facilitates the integration, exchange and extension of various prefetching techniques. To demonstrate the viability of this architecture two prefetching techniques are incorporated: a *predictor-based* technique—which consists of a separate predictor component in the run-time system—and a *code-based* technique—which provides an explicit prefetch statement at the programming interface. Our quantitative analysis indicates that (static) code-based techniques are a promising alternative to expensive monitoring-based predictors.

1 Introduction

The fate of object-oriented database systems will largely depend on their performance compared to currently employed database technology. Database management systems bridge the gap between the two-level memory hierarchy by transferring objects between (volatile) main memory and (persistent) secondary storage. Unfortunately, the gap between disks and high-speed memories has widened. Therefore paging is now less effective than it used to be. As Wilkes [Wil92] already pointed out, there is an urgent need to reexamine memory management at a fundamental level. This holds definitely for client/server systems where the I/O costs are mainly determined by network latency and server utilization. To illustrate this problem we adapt an example from [KL91]. Consider a client/server system where the appropriate CPU costs for a regular buffer miss are 200 high-level statements. Assuming an application with 1000 statements and read or write operations every four statements, the application will be blocked over 30% of the time, even with 99% buffer hit ratio.

Prefetching is an optimization technique which loads pages in the database buffer *before* the application requests them. The goal of prefetching is to minimize buffer miss latency by overlapping computations and I/O. In the above example 500 program statements can be overlapped if we prefetch the fault pages timely, thus, we would save 30% running time in total. Prefetching techniques have already been proposed and investigated for virtual memory systems a long time ago [Jos70, BS76]. But in database management systems prefetching has not yet received enough attention. There are two reasons for that:

- For database applications prediction of page/object sequences is more complicated because of the overall complex structure of databases.
- Implementation of a software prefetching mechanism is not a trivial task. Only for microprocessors solutions are well-known [BC91, FP91, KL91, RL92].

Although some promising prefetching techniques have recently been proposed for database systems [WZ86, PZ91, CKV93], most of these techniques still have to prove their viability in a *real* system environment For this purpose, we propose a generic architectural framework, a *multi-threaded* run-time system for client/server architectures, which facilitates the integration, exchange and extension of various prefetching techniques. Threads can easily be exchanged or added to satisfy different requirements of different prefetching techniques. To demonstrate the viability of this architecture two prefetching techniques are incorporated: a *predictor-based* technique—which consists of a separate predictor component in the run-time system—and a *code-based* technique—which provides an explicit prefetch statement at the programming interface.

The paper is organized as follows. In Section 2 we first present a classification scheme for prefetching techniques and we give an overview of related work. Section 3 introduces the multi-threaded prefetching architecture and its implementation. In Section 4 benchmark results for a large variety of parameters for the two most promising prefetching techniques [MLG92, CKV93] are presented. The pros and cons of both approaches and some remaining open problems are summarized in Section 5. Section 6 concludes the paper.

2 Classification of Prefetching Techniques

Prefetching techniques can be classified along two dimensions: The *prediction mechanism* and the *unit of prediction*. Methods for predicting future references will be surveyed first.

Most of the prefetching techniques found in the literature use a separate *predictor component* to determine the application's behavior in the near future. Predictors can either be *strategy-based* [Jos70, Tri76, KE90, FP91] or *training-based* [BS76, Smi78, BC91, PZ91, CKV93] or *structure-based* [CK89, KGM91, Gem91]. *Strategy-based* predictors incorporate an explicit programmed strategy whereas *training-based* predictors require extensive *monitoring* for consolidating the statistics during run-time. Future references of an application are predicted by exploiting the reference patterns obtained in the past. *Structure-based* predictors are represented directly by the object net itself together with one common traversal (e.g., depth-first). In contrast, code-based techniques make use of an explicit non-blocking (i.e., *asynchronously*) working, prefetch statement at the user level, which the run-time system must provide. Generally, the insertion of such an instruction in the application's source code is left to the user. For compiler optimization, however, there already exist some promising proposals for automatically generating the prefetch statements in combination with *code motion*. But none of theses techniques—as far as the authors know—has yet been implemented and investigated for database systems.

The other dimension concerns the *unit of prediction*. Many techniques used in the past—mainly in operating systems—predict sequences of *pages* whereas in the last years some techniques were proposed for database systems which try to predict sequences of *objects*.

The following table shows the related work according to the classification scheme:

Prediction mechanism	Unit of prediction	
	Objects	Pages
Predictor-based	[CK89, KGM91] [BC91, Gem91, PZ91]	[Jos70, BS76, Tri76, Smi78] [FP91, KE90, CKV93]
Code-based	[WZ86, KL91, CKP91] [RL92, CB92, MLG92]	UNIX *madvise*

Predictor-Based Prefetching Techniques. One essential characteristic of predictor based prefetching techniques is the *time of prefetching*. Basically, there exist two classes of prefetching policies: (1) prefetching at every reference and (2) prefetching only at buffer misses. The prediction component is represented by an universal predictor $\mathcal{P}$. We divide predictors in two classes: fast predictors and slow predictors. Fast predictors employ a simple table lookup or they are strategy-based [Jos70, BS76, Smi78, Tri76, KE90]. Slow predictors require more prediction time [CK89, KGM91, Gem91, PZ91, CKV93]. Therefore, in the latter case prediction should take place in a separate *coroutine* to overlap computation with I/O caused by the buffer miss.

Code-Based Prefetching Techniques. As already mentioned in [BC91, KL91] complex data structures like linked lists or binary trees are a hard challenge for code-based prefetching techniques because the reference stream is entirely data dependent. The same holds for object-oriented database applications where mainly path expressions (reference chains) are traversed. In the following example below we show how an iteration through a list of part objects can be supported by code-based prefetching:

(a) Original Code

```
for(i := 0; i < len; i++) begin
  list.n_th[ i ].access;
end for;
```

(b) Transformed Code

```
P1: prefetch(list.n_th[ k ]);
P2: for(i := 0; i < len − k; i++) begin
      prefetch(list.n_th[ i + k + 1 ]);
      list.n_th[ i ].access;
    end for;
P3: for(i := len − k; i < len; i++) begin
      list.n_th[ len − 1 ].access;
    end for;
```

Here, we prefetch one object k iterations ahead. Therefore, we split the loop in three parts [MLG92]: the first part (P1) does the initial prefetching, in the second part (P2) references and prefetching is done and the third part (P3) constitutes the trailer of the loop without any prefetching.

Prefetching Performance Measures. As the previous discussion motivates, the *prediction accuracy* [PZ91] would be an intuitive way to measure the performance of prefetching techniques. Unfortunately, there is no guarantee that prefetched objects/pages are resident in the buffer by the time they are really accessed. Therefore, a more realistic measure is the *reduction of buffer faults* [Jos70, BS76, Smi78, PZ91, CKV93]. But there are some disadvantages when considering only buffer faults:

- When a prefetched item is pending, the application *stalls* until the item is available. This waiting time reduces the total running time and is not reflected by buffer faults.
- The computation for prediction can be very expensive, especially for space-consuming predictors which do not fit entirely in main memory.
- Operating system implementation details (size of the internal socket buffers, implementation of TCP/IP protocol, asynchronous notification) have a great impact on running time because they affect the total overhead of prefetching; e.g., the more items arrive asynchronously the more the CPU overhead is increased.
- The additional network traffic caused by prefetching also affects a regular demand fetch. Therefore, prefetching can increase the costs for regular buffer faults [BC91, KL91].

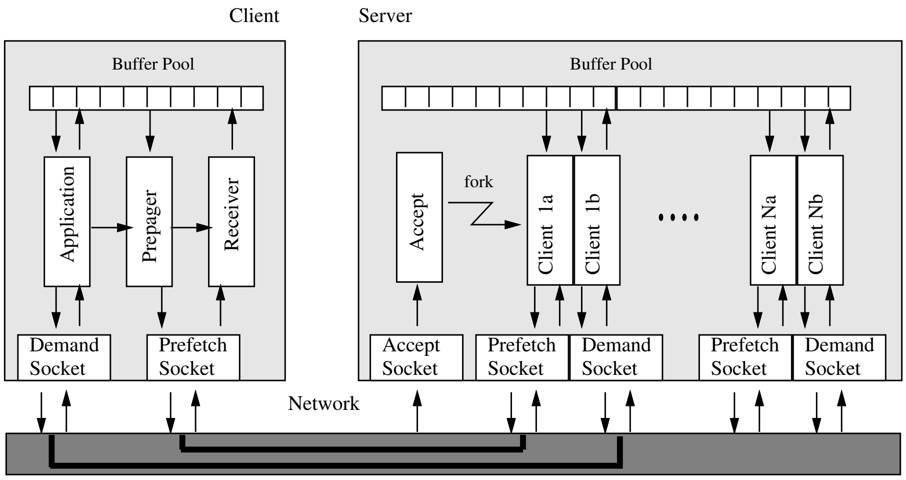
Summarizing the above arguments we conclude that buffer faults alone do not suffice to assess prefetching [KE90, KL91, MLG92]. Instead, the *total running time (RT)* of an application or the *speed-up factor* [FP91] seem to be the only practical evaluation criteria for prefetching. Consequently, in this paper we used the relative *savings in running time* which is computed as follows:

$$savings = \frac{(RT_{no\ prefetching} - RT_{with\ prefetching})}{RT_{no\ prefetching}} * 100$$

3 A Multi-Threaded Client/Server Architecture

In this section we focus on the general architecture for prefetching in object bases. First, we give an overview by outlining the new components which are essential for the realization of prefetching in a client/server environment. Afterwards, we discuss some intrinsic details of the basic threads and their interaction.

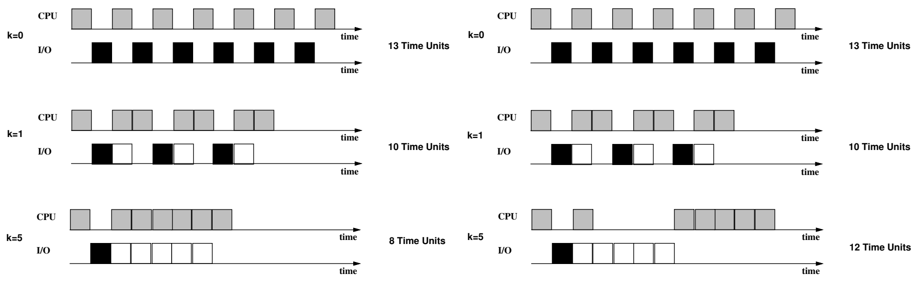
Lightweight processes (threads) are independent units of control within a regular (heavyweight) process. Context switches between threads are much faster than that of processes and cooperation primitives—like concepts for mutual exclusion—can be realized very efficiently at the user level, thus avoiding expensive calls to the operating system [Mue93]. In the following we present how threads can be used for prefetching and how they could be integrated in a page server architecture:



The server maintains two separate sockets with two associated threads for every connected client: one socket/thread for demand requests and one socket/thread for prefetch requests. In contrast to [PZ91], both sockets are given the same priority. We believe—and empirically verified—that the benefits from prefetching strongly depend on timely satisfaction of prefetch requests. Therefore, we don't want to run the risk that regular demands *overtake* prefetch requests. This becomes all the more likely if many clients are present and the demand requests are given higher priority. Otherwise, the gain achieved by prefetching becomes negligible.

At the client side predictor-based prefetching is implemented by three threads: the *application thread*, the *prepage thread*, and the *receiver thread*. The prepager is responsible for predicting and requesting pages. At buffer fault time the application thread notifies the prepager about the page number causing the fault and some special information about the current context (e.g., the working set). After the application thread has requested the faulting page from the server it will block on its next receive operation causing the prepager to be scheduled next. The prepager now computes the set of non-resident pages according to its page predictor $\mathcal{P}$ and sends the entire request to the server. Asynchronously incoming pages are handled by the receiver. For code-based prefetching we omitted the *prefetch thread* because, in this case, it seems more natural to let the prefetch statements directly be executed by the application thread. The buffer policy is similiar to [PZ91, CKV93]; i.e., prefetched pages are moved to the top of an LRU buffer as if they were demand fetched.

Because the implementation of the reception of a *set* of pages is more important than it seems to be on first glance, we will spend a little more attention on it. There are two major aspects: (1) The server has to maintain the ordering of page requests and (2) the receiver thread must give back control to the application thread as soon as it has received an entire prefetched page. Otherwise, the application thread cannot continue and, thus, computation cannot be overlapped with I/O. To explain this in more detail consider the next figure:



(a) Best case ordering

(b) Worst case ordering

For simplicity we assume that one I/O unit (black/white) can be overlapped with exactly one CPU unit (grey). In case of no prefetching ($k = 0$) the running time is 13 time units. If we prefetch $k = 5$ pages (white) at page fault time and the server keeps ordering (case (a)), these I/O units overlap with computation, thus 5 units of running time are saved in total. If the ordering is reverse, i.e., the next (faulting) page comes last and the last page comes first, the application would block until *all* prefetched pages are received (case (b)). This shows that receiving a set of prefetched pages must be *interruptable*. Otherwise, if we receive all pages en bloc this would have the same effect as worst case ordering.

In our implementation a prefetch request is forced to be executed in any case. Therefore, pages are given different *states* according to a traffic light sequence: Resident pages are *green*, prefetched pages which are pending are assigned the color *red*. If the application tries to access a page in state *red*, the page's state changes to *yellow* and the application thread is blocked. As soon as the page becomes available the state changes to green, the receiving thread unblocks the waiting thread and gives up control in favor of the application thread.

The multi-threaded architecture is implemented with the LWP-package included in the SunOS 4.1.3 operating system. The implementation of prefetching in UNIX is currently affected by several problems: (1) There is still no efficient operating system support for threads, (2) the UNIX (stream) file system completely lacks support for *multiple* asynchronous notifications of incoming data and (3) the socket interface lacks operations to manipulate the internal buffer. As already pointed out in [Sto81] modern operating systems should evolute towards supporting database management functions more efficiently. We stress that an efficient implementation of prefetching will strongly depend on the capabilities of the underlying operating system.

4 Assessment of Prefetching Techniques

In this section we present some typical results of our extensive benchmarking. Our goal was to assess the basic prefetching principles—i.e. predictor-based and code-based techniques—and not to investigate the quality of the prediction mechanism. The following aspects are subject to a closer investigation: (1) the impact of clustering on prefetching, (2) the effect of physical versus logical OIDs on prefetching, (3) the effect of varying computation peaks, (4) the overhead of prefetching induced by high locality, and (5) the network traffic in a multi-client environment.

4.1 Benchmark Environment

Prefetching is affected by many parameters. Therefore, it is very important to use a simple benchmark for obtaining meaningful results. More complicated benchmarks have irregular structures (making prediction difficult) or are indeterministic (making interpretation of results difficult).

The structure of our database constitutes a list object containing 10000 objects. The size of the objects are uniformly 300 bytes, the page size is 4096 bytes. We performed a linear scan through all elements of the list. Note that after every time an element in the list is accessed, we return to the list object to obtain the reference to the next element. Thus, the trace looks like this: $id_0, id_1, id_0, id_2, \ldots, id_0, id_n$, where $id_1, \ldots, id_n$ are the OIDs of the element objects of the list id_0. The quality of the following techniques is compared:

- predictor-based prefetching using a Markov predictor for pages [CKV93],
- code-based prefetching using the technique proposed in [MLG92], and
- no prefetching, applying only a standard LRU buffer policy.

The Markov predictor constitutes a *page transition graph* which is constructed by monitoring the page accesses only of the element objects.[1] E.g., if we assume that two objects fit into a page and the database is clustered, then we would get the following (linear) sequence of pages: $S_0, S_0, S_1, S_1, \ldots$ If we have no clustering—that is the objects are randomly distributed over the pages—then the sequence will form a general graph, thus, making prediction more difficult. In our predictor-based technique we prefetched 1 page ahead and in the code-based prefetching we prefetched 1 object 1 iteration ahead, if not stated otherwise. The clustering algorithm used in this paper is a *greedy graph partitioning* (*GGP*) algorithm which yields good-quality clustering under acceptable running time for object bases [GKKM93].

4.2 Prefetching under Different Clusters

In this experiment we investigate the influence of different storage characteristics on prefetching assuming physical OIDs. Because the time between two object references is very crucial for prefetching (for overlapping with I/O), we plotted the *inter-reference time* (*IRT*) in microseconds against the x-axis. The inter-reference time is artificially generated by an appropriate **loop**-statement in our applications. Having no significant inter-reference time ($IRT \approx 0$) often means that—apart from the fact that there is a constant system overhead—in case

[1] If we had monitored the list object, too, we would have needed a second-order Markov predictor.

of no clustering "staccato-like" sequences of page faults will be generated. The main observation in Fig. 1(a) is that both prefetching techniques perform better than LRU—in almost all cases. With good clustering code-based techniques suffer from unnecessary requests for already resident objects; thus, increasing running-time. As IRT increases the code-based technique is able to overlap I/O with more computations and, finally, outperforms the predictor-based technique. Note that overlaps are well indicated by a lower gradient compared to the LRU curve. Remember that in the predictor case we prefetch only at page fault times whereas in the code-based technique prefetching is independent from any page accesses; therefore, reducing memory latency for every page fault uniformly. It is just these little savings accumulated step by step that makes the code-based technique superior at the end. The maximal savings in running-time are 27% for the predictor-based ($IRT = 0$) and 10% for the code-based technique ($IRT = 1500$).[2] Fig. 1(b) exhibits that techniques that are based on prediction of pages are very sensitive to clustering. Here, the Markov graph formed by the cluster is not precise enough to predict the exact pages. The need for a Markov Predictor of an order higher than 1 is evident. But it is difficult to determine the best order for all applications and all clusters in advance. In contrast, the prediction accuracy of the code-based technique is 100% and nearly every prefetch request results in requesting a non-resident page, thereby eliminating unnecessary prefetching requests. However, for applications with *high locality* this does not hold. The maximum savings for the code-based technique are 28% ($IRT = 1500$). But Fig. 1(b) also indicates that at $IRT = 0$ we have a small penalty. Therefore, the curve of the code-based technique needs a more detailed explanation. Let IPT be the inter-page-fault time, let T_{Pref} denote the CPU overhead induced by the prefetch statement and let T_{Fault} denote the time for a buffer miss. Then, we classify the effect of prefetching into three cases:

- Case 1: $IPT < T_{Pref} \leq T_{Fault}$. There is a loss of $T_{Pref} - IPT$.[3]
- Case 2: $T_{Pref} < IPT \leq T_{Fault}$. There is a gain of $IPT - T_{Pref}$.
- Case 3: $T_{Pref} < T_{Fault} \leq IPT$. There is a gain of $T_{Fault} - T_{Pref}$.

In the case of an inter-page-fault time of zero, i.e., no locality and no significant inter-reference time, there cannot be any gain through prefetching because there is no CPU time to overlap. This corresponds to Case 1. Case 2 occurs as soon as there are some computations which exceed the prefetch CPU overhead ($T_{Pref} \leq IPT$). Then, we have a reduction in running-time. Additionally, due to random buffer hits some prefetch requests could be executed completely; therefore, we save the next request message to the server and the running time is further reduced. However, as soon as the maximum overlap is reached increasing IRT also increases running-time.

[2] Note that with increasing inter-reference time, computations dominate over I/O—thus, reducing the savings in running-time achievable by prefetching.

[3] Because the code-based techniques do not prefetch exactly at page fault times, taking the IPT corresponds to the best case.

4.3 Prefetching With Logical Object Identifiers

To investigate the impact of the additional overhead induced by mapping logical OIDs to physical addresses we conducted the same experiment as in the preceding section for an object model utilizing logical OIDs (Fig. 2). Compared to Fig. 1 the overhead for code-based techniques substantially increases when relying on logical OIDs. This is exactly what we expected. Under good clustering, the benefits of code-based techniques—compared to LRU—are turned into the opposite: the running-time increases by up to 50%. However the gradient is much lower than for the LRU curve. This indicates that the "break-even point" becomes more unfavorable in terms of inter-reference time. For random clustering I/O waiting time dominates. Thus, the additional computations induced by the mapping can easily be compensated for. Note that for $IRT = 0$ there is, in reality, not an inter-reference time of zero because the mapping consumes some CPU time. Additionally, we have some random buffer hits (remember that we have poor clustering but a large buffer), thus the mapping causes a further prolongation of the inter-page-fault time. This is indeed the reason why here is—compared to Fig. 1(b)—a saving in running-time because the extra computations suffice to execute some overlapping prefetches. On the other hand, the predictor-based technique is not affected by the translation of logical OIDs because the prediction is independent of that mapping. This is, of course, only true for page-oriented predictors. When the predictor is object-oriented it would suffer just as severely as the code-based technique.

4.4 Prefetching under Varying Computation Peaks

The benefits of prefetching are based on overlapping computations with I/O; i.e., the *inter-page-fault time* must be "well behaved". In our experiments we controlled the inter-page-fault time by varying the inter-reference time. In real applications the inter-page-fault time is generally not constant, e.g., it is highly influenced by the buffer size and the locality of an application. In practice, this will reduce the performance gain achieved by prefetching. To study these effects we introduced *cycles* of computation peaks. We varied the inter-reference time for the i-th reference according to the following formula:

$$\mathrm{IRT}(i) = \begin{cases} \mathit{cycle} * \mathit{peak} & : \text{ if } i \bmod \mathit{cycle} = 0 \\ 0 & : \text{ otherwise} \end{cases}$$

Fig. 3 shows the running-time for random clustering, physical OIDs, and for $\mathit{peak} = 500, 1000, 1500$. We omitted the predictor-based technique here because the last experiments showed no improvement. As expected, computations cannot as easily be overlapped as in the uniform case. For all peak levels a substantial increase in running-time can be observed with increasing cycle length. But in most cases the running-time still remains below the corresponding LRU curve. Under good clustering no significant differences can be observed between the code-based and the predictor-based technique, because many subsequent hits on the same page leave the inter-page-fault time nearly invariant.

4.5 Prefetching under High Locality

In the previous subsections we only investigated prefetching under strict (logical) sequentiality. In this experiment we used a nested loop join over two lists of 300 elements. Because of the repeated scanning of the inner list, very high locality is provided. We simulated different scenarios, i.e., joining small or large lists, by using two buffer sizes: a small buffer (75% of the working set) and a large buffer (125% of the working set). Here, the working set basically consists of the set of pages in the inner list. Dependent on the buffer size we used three prefetching schemes for the code-based technique: prefetching for both lists, prefetching only for the outer loop in the case of a small buffer, and prefetching only for the inner loop in the case of a large buffer. The other remaining cases make no sense. We omitted this variant for the predictor-based technique, since the monitored sequences are independent of the buffer size.[4] The inter-reference time was set to 0 because a comparison inside a join loop is very fast. The savings in running-time are shown in the following table:

Working Set	Predictor-based	Code-based		
		Both	Inner	Outer
125%	-23%	-63%		+18%
75%	+17%	-17%	-16%	

Only in 2 out of 6 cases prefetching achieves savings in running-time. Surprisingly, the predictor-based technique suffers under high locality—there is a prolongation of running-time by 23%. The reason is that in this benchmark the prediction always suggests pages which are already resident: because we prefetch only on page fault times there cannot be any gain because the entire inner list fits into the buffer. The next page fault can only occur in the outer loop. The overhead induced under high locality is more evident for the code-based technique: the negative "savings" by 63% indicates that prefetching here must be considered harmful. Only after elimination of unnecessary prefetch requests, i.e. for the inner loop, the code-based technique can achieve reasonable savings (18%). In case of no locality, we have similar results to Fig. 1(a): the predictor-based technique outperforms the code-based technique at $IRT = 0$.

4.6 Prefetching in a Multi-Client Environment

The cost of a buffer fault essentially depends on the network traffic and on the number of clients connected to the server. In this experiment we focus on the performance of prefetching in presence of many heavy I/O-weighted clients. Therefore, we ran up to 5 additional special-purpose clients in parallel which produced 10000 page requests without any computations in between two page accesses. We used the same configuration as in Fig. 1(a) with $IRT = 1500$. To

[4] Note that there is no (trivial) mapping of monitored sequences to high-level statements.

study the effect of requesting a set of pages, i.e., lower tag-along costs for the additionally requested pages, we prefetched up to 15 pages in this experiment. For the code-based technique we prefetched $k_o = 1, 195$ objects per iteration ahead, the predictor-based technique prefetched $k_p = 1, 15$ pages ahead. Assuming that 13 objects fit into a page, prefetching 195 objects is comparable to prefetching 15 pages in the predictor case. Fig. 4 shows that both prefetching techniques perform clearly better than LRU. Consider the $k_o = k_p = 1$ case first. The predictor-based technique clearly outperforms the code-based technique when more than one client is connected to the server. Prefetching one object one iteration ahead does not suffice to cover buffer miss latency. Remember that in this experiment we have high temporal locality through good clustering. If the code-based technique discovers a non-resident object, it is usually too late. The predictor-based technique "looks" more than one object ahead because it is based on entire pages. Subsequent buffer hits on the current demand page suffice to hide latency. Prefetching a large set of objects/pages is more advantageous in a multi-client environment. The reason can be found in the low tag-along costs for a single set-request. From this experiment we conclude that the inter-reference time is not as important as in the dedicated single-client case. Here, the inter-page-fault time is mainly influenced by the increased server response time.

5 Discussion

In the preceding section we benchmarked the two most promising prefetching techniques in our experimental system architecture: a predictor-based page-oriented technique and a code-based object-oriented technique. Both techniques can substantially reduce the running-time of an application. However, the positive effect cannot be obtained in all cases. We conclude that the advantage of one technique may turn out as the disadvantage of the other technique and vice versa.

The most important parameter on prefetching is clustering. Prefetching tries to fill the "holes" still left by clustering. The first goal, however, should be to maximize locality by clustering. If this does not suffice to achieve reasonable performance—e.g. when clustering according to many diverging access patterns clustering is less effective than supporting only a single application—then prefetching is the method of choice. As our preceding quantitative analysis indicates, the degree of locality is an important hint for the applicability of prefetching techniques. For applications with high locality, when running on a well clustered database, prefetching often achieves only negligible gains. The most cost effective factor here is the residency check.

The prediction of the access behavior of non-standard database applications is very difficult. In further experiments we observed that the prediction accuracy for page-oriented predictors drops significantly when the page capacity (in objects) is large and, in addition, the *degree of sharing* is high or when the database is accessed through many different *entry points*. Accessing many shared objects through different objects within the same application makes monitoring less accurate.

Our analysis yields that *no* prefetching technique performs superior for *all* cases. Further pros and cons of both techniques are summarized in the table below.

Predictor-Based Page-Oriented	**Code-Based Object-Oriented**
+ Adaptive and variable prefetching via feedback at run-time possible.	− Fix compile-time made scheduling distance to hide memory latency.
−− Prediction accuracy very sensitive towards cluster quality.	+ Independent of storage structure.
− Prefetching only possible for monitored sequences.	++ No monitoring. Prefetching possible for all program paths.
+ Prediction of long sequences very easy.	− Only limited look-ahead.
+ Small overhead in case of high locality and large buffers.	− Many unnecessary prefetch statements. High overhead.

6 Conclusion

In this paper we distinguished two dimensions for classifying prefetching techniques: (1) the prediction mechanism and (2) the unit of prediction. Along the first dimension we distinguished *predictor-based* versus *code-based* techniques, along the second dimension we distinguished *object-oriented* versus *page-oriented* prefetching techniques. To prove the viability of prefetching techniques we proposed a generic architectural framework, a *multi-threaded* run-time system architecture, which facilitates the integration, exchange and extension of prefetching techniques. We outlined the realization of the two most promising techniques, a predictor-based page-oriented technique and a code-based object-oriented technique, in our architecture. Our extensive quantitative analysis—only a part of which could be presented in this paper—indicates that *no* prefetching technique performs superior in *all* cases. However, (static) code-based techniques are a promising alternative to (dynamic) predictor-based techniques:

- monitoring sequences of pages/objects requires an extensive "training" phase during which the statistics have to become consolidated.
- static code-based prefetching enables prefetching for all program paths, especially when a training phase is not possible, e.g. for CIM applications bound to technical processes.

In conclusion, we showed that prefetching in database management systems is feasible by today's software technology.

Acknowledgement This work was supported by the German Research Council DFG under contract Ke 401/6-1. We thank T. Jansen for his help in the implementation and extensive benchmarking of the various prefetching techniques.

References

[BC91] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proc. of Supercomputing*, pages 176–186, 1991.

[BS76] J.-L. Baier and G.R. Sager. Dynamic improvement of locality in virtual memory systems. *IEEE Trans. Software Eng.*, 2(1):54–62, 1976.

[CB92] T.-F. Chen and J.-L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proc. on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), pages 51–61, Oct 1992.

[CK89] E. E. Chang and R. H. Katz. Exploiting inheritance and structure semantics for effective clustering and buffering in an object-oriented DBMS. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 348–357, Portland, OR, May 1989.

[CKP91] D. Callahan, K. Kennedy, and A. Portfield. Software prefetching. In *ASPLOS*, pages 40–52, Santa Clara, California, Apr 1991.

[CKV93] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *ACM SIGMOD*, pages 43–53, Washington, DC, May 1993.

[FP91] J. W. C. Fu and J. H. Patel. Data prefetching in multiprocessor vector cache memories. In *Proc. of the Intl. Symp. on Computer Architecture*, pages 54–63, May 1991.

[Gem91] GemStone. *Product Overview.* Servio Logic Corporation, 1420 Harbor Bay Parkway, Suite 100, Alameda, CA 94501, U.S.A., Feb 1991.

[GKKM93] C. Gerlhof, A. Kemper, C. Kilger, and G. Moerkotte. Partition-based clustering in object bases: From theory to practice. In *Proc. of the Intl. Conf. on Foundations of Data Organization and Algorithms (FODO)*, volume 730 of *Lecture Notes in Computer Science (LNCS)*, pages 301–316, Chicago, Illinois, Oct 1993, Springer-Verlag.

[Jos70] M. Joseph. An analysis of paging and program behaviour. *Computer Journal*, 13(1):48–54, 1970.

[KE90] D. F. Kotz and C. S. Ellis. Prefetching in file systems for MIMD multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 1(2):218–230, Apr 1990.

[KGM91] T. Keller, G. Graefe, and D. Maier. Efficient assembly of complex objects. In *ACM SIGMOD*, pages 148–158, Denver, CO, May 1991.

[KL91] A. C. Klaiber and H. M. Levy. An architecture for software-controlled data prefetching. In *Proc. of the Intl. Symp. on Computer Architecture*, pages 43–53, May 1991.

[MLG92] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *ASPLOS*, pages 62–73, Oct 1992.

[Mue93] F. Mueller. A library implementation of POSIX threads under UNIX. In *USENIX*, San Diego, CA, Winter 1993.

[PZ91] M. Palmer and S. B. Zdonik. Fido: A cache that learns to fetch. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 255–264, Barcelona, Sep 1991.

[RL92] A. Rogers and K. Li. Software support for speculative loads. In *ASPLOS*, pages 38–50, Oct 1992.

[Smi78] A. J. Smith. Sequentiality and prefetching in database systems. *ACM Trans. on Database Systems*, 3(3):223–247, 1978.

[Sto81] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, Jul 1981.

[Tri76] K. S. Trivedi. Prepaging and applications to array algorithms. *IEEE Trans. on Comp.*, 25(9):915–921, Sep 1976.

[Wil92] M. V. Wilkes. The long-term future of operating systems. *Communications of the ACM*, 35(11):23–24/112, Nov 1992.

[WZ86] H. Wedeking and G. Zörntlein. Prefetching in realtime database applications. In *ACM SIGMOD*, pages 215–226, Washington, D.C., 1986.

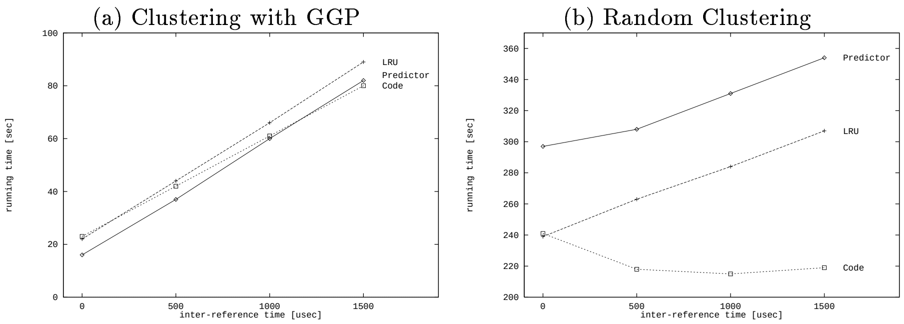


Fig. 1. Performance of Prefetching Techniques with Physical Object Identifiers

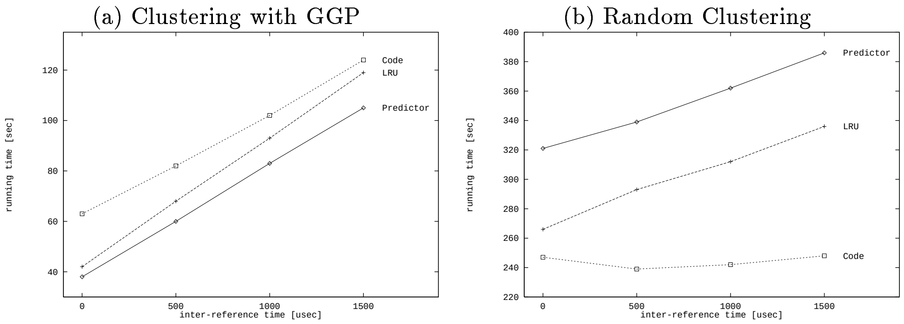


Fig. 2. Performance of Prefetching Techniques with Logical Object Identifiers

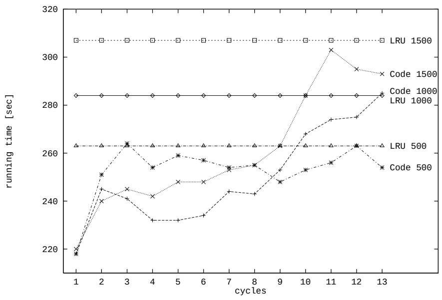


Fig. 3. Performance of the Code-Based Technique under Varying Computation Peaks

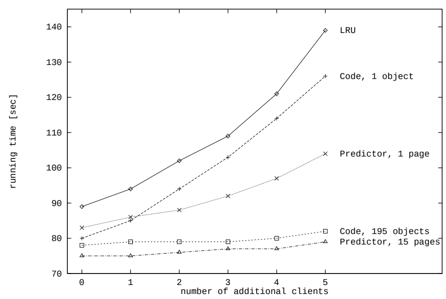


Fig. 4. Performance of Prefetching Techniques in a Multi-Client Environment