Noname manuscript No.
(will be inserted by the editor)

Path and Cache Conscious Prefetching (PCCP)

Zhen He[1], **Alonso Marquez**[2]

[1] Department of Computer Science and Computer Engineering
La Trobe University
Bundoora, VIC 3086
Australia
e-mail: z.he@latrobe.edu.au

[2] Department of Computer Science
The Australian National University
Canberra, ACT 0200
Australia
e-mail: alonsomarquezes@yahoo.es

Abstract Main memory cache performance continues to play an important role in determining the overall performance of object-oriented, object-relational and XML databases. An effective method of improving main memory cache performance is to prefetch or pre-load pages in advance to their usage, in anticipation of main memory cache misses. In this paper we describe a framework for creating prefetching algorithms with the novel features of path and cache consciousness. Path consciousness refers to the use of short sequences of object references at key points in the reference trace to identify paths of navigation. Cache consciousness refers to the use of historical page access knowledge to guess which pages are likely to be main memory cache resident most of the time and then assumes these pages do not exist in the context of prefetching. We have conducted a number of experiments comparing our approach against four highly competitive prefetching algorithms. The results shows our approach outperforms existing prefetching techniques in some situations while performing worse in others. We provide guidelines as to when our algorithm should be used and when others maybe more desirable.

Keywords: prefetching, clustering, caching, databases.

1 Introduction

The current rate of performance improvement for CPUs is much higher than that for memory or disk IO. CPU performance doubles every 18 months while disk IO improves at only 5-8% per year. In addition, cheap disks mean databases will become bigger as database designers realise that more data can be stored [1]. A consequence of this is that disk IO is likely to be a bottleneck in an increasing number of database applications. It should also be noted, memory is also becoming a more prevalent source of bottleneck on modern DBMSs [2]. This paper is focused on reducing disk IO stalls rather than level 1 (L1) and level 2 (L2) cache misses. However, it is important to note that the methods described in this paper can be used in conjuction with prefetching techniques designed for L1 and L2 caches.

Throughout the rest of this paper we will use the word *cache* to denote the *main memory cache*. The IO bottleneck occurs when an application requests a page which is not cache resident. The result is the requested page must be loaded from disk while the application waits. This method of fetching pages into cache is referred to as *demand fetching*. However, if the Object Database Management System (ODBMS)[1] can predict which disk resident page is likely to be requested next, it can load that page in advance. This method of preloading user's requested pages in the background is referred to as *prefetching*. Prefetching allows disk IO to be overlapped with CPU, thus reducing disk IO stall time. In this paper the word *prefetching* refers to prefetches from disk.

Central to the design of prefetching algorithms is the design of the prediction engine. Most existing prefetching algorithms for ODBMSs use a context-model-based prediction engine [3–5]. Context-model-based prediction engines use historical access information to make future prefetching decisions. There are two problems with existing context-model-based prediction engines:

[1] From this point on, we will use the term 'ODBMS' to collectively refer to object oriented, object relational, and XML database management systems.

– The high storage cost of storing object-grained access statistics.
– The small time gap between prediction and reference of the next prefetched page.

Storing access statistics at the object-grain [4–8] can provide prediction engines with more precise statistical information of access patterns. However, the high storage cost of object-grained prefetching algorithms often limit their applicability. Large storage overheads leads to a reduced cache size for the data objects themselves. This in turn leads to reduced time between disk page requests. The consequence is less time is available for the prefetch algorithm to overlap CPU and disk IO.

Another problem with existing prediction engines for ODBMSs is the small time gap between prediction and reference of the next prefetched page. The consequence is that there is little potential overlap between IO and CPU. Existing prediction engines can only predict the next disk page request a few object references ahead of time. This limitation is becoming a bigger problem, since the rate of CPU improvement is much greater than that of disk IO. Therefore the CPU time it takes to process each object becomes much smaller relative to one disk IO[2]. This in turn leads to a smaller amount of overlap between CPU and disk IO for prefetchs started the same number of object references in advance. The path and cache conscious prefetching framework (PCCP) addresses both of these deficiencies in existing prefetching algorithms designed for ODBMSs.

During prediction engine training, PCCP minimizes statistics storage by storing short sequences of object references at key points (which we term 'feature points') in the reference trace. When these feature points are later encountered at prefetch time, the stored statistics are used to decide if a prefetch should be started. These feature points are selected to be sparsely spaced and early in terms of when the next prefetched page will be referenced.

There are two key concepts in PCCP, *path* and *cache* consciousness. Path consciousness refers to the careful selection of feature points so that the current path of navigation can be identified early and cheaply. In cache conscious prefetching, historical page access knowledge is used to guess which pages are likely to be cache resident most of the time (we term these pages 'resident' pages) and these pages are then ignored in the context of prefetching. Therefore cache conscious prefetching reduces the number of feature points to only those that occur during traversal of pages deemed to be 'non-resident', thereby reducing the total volume of statistics stored. An even more important result of cache consciousness is that only 'non-resident' pages are candidates for prefetching. The implication of this property is that prefetches can be started earlier (section 4.2 explains the reason for this behavior).

Two complementary techniques for addressing the disk IO bottleneck in ODBMSs are *clustering* and *buffer replacement*. *Clustering* is the arrangement of objects into pages so that objects accessed close to each other temporally are placed into the same page. This in turn reduces the total volume of IO generated. *Buffer replacement* involves the selection of a page to be evicted when the cache is full. Selection of the correct page for eviction results in a reduction in the total volume of IO generated by the system. In this paper we focus on developing new prefetching techniques. However, PCCP can use clustering and buffer replacement information to help make more informed prefetching decisions. We demonstrate the usefulness of this approach by creating and benchmarking the integrated prefetching algorithm (IP), an instance of PCCP.

We have conducted an extensive performance evaluation of six different prefetching algorithms: four existing prefetching algorithms PPM-1 [3], PPM-3 [3], PMC [4] and EPCM [9]; and two new algorithms created from PCCP. The results show PCCP algorithms outperforms PPM-1, PPM-3, and PMC in most situations. When compared to EPCM each algorithm has its own merits. PCCP algorithms are more robust to variations in prefetch threshold settings and use upto 20 times less storage overheads. However, EPCM produces less stall time than PCCP when EPCM is fine tuned to its optimal prefetch threshold and the cache size is large.

This paper makes the following contributions: it defines the PCCP prefetching framework which allows the creation of a new family of prefetching algorithms; validates the utility of PCCP by creating two new prefetching algorithms; and conducts an experimental study comparing the performance of the new prefetching algorithms with four existing prefetching algorithms.

The remainder of this paper is organised as follows. Section 2 provides a brief description of existing prefetching algorithms. A formal problem statement is stated in section 3. The PCCP framework and its benefits are described in section 4. Two new prefetching algorithms created using the PCCP framework are described in section 5. The experimental setup used to evaluate the prefetching algorithms are detailed in section 6. Experimental results comparing the two PCCP algorithms with four existing algorithms are reported in section 7. Lastly section 8 concludes the paper and indicates directions for future research.

2 Related Work

Gerlhof *et. al.* [10] identify two dimensions along which prefetching algorithms can be classified: prediction engine used; and granularity of prediction.

[2] Assuming the amount of computation per object remains the same.

Prediction engines are typically divided into four types: *strategy-based*; *structure-based*; *hint-based*; and *context-model-based*.

Strategy-based prefetching algorithms use explicitly programmed strategies to decide which objects to prefetch. The simplest example is the one block lookahead algorithm (OBL) [11], which upon a demand fetch[3], prefetches the next adjacent block. Another example is Thor's prefetching policy [12]. Thor divides objects into prefetch groups and whenever an object in a prefetch group is requested, all the objects in the group are fetched. More recently, Bernstein *et al.* [6] proposed a new strategy-based prefetching algorithm that fetches all objects in the structure context of the requested object. Examples of structure context include query results and collections. The general problem with strategy-based prediction engines is their lack of flexibility in catering for different paths of object graph navigation.

Structure-based prefetching algorithms obtain information from object structure. In Chang and Katz's [7] approach, objects are linked via three types of structural relationships: inheritance; configuration; and version history. The user specifies which type of relationship he/she is currently navigating under and this information is used in combination with the structural hierarchy graph to decide which objects to prefetch next. The problem with this approach is its reliance on user provided information. Knafla [8] proposes an approach in which different possible paths of navigation are first identified *using the object graph alone* and then as client navigation proceeds, the prefetch algorithm uses the current navigational context to determine which path is likely to be taken. All structure-based approaches assume objects will always be accessed by navigating references defined in the object graph, however, ad hoc queries do not navigate the object graph. Thus structure-based approaches perform poorly when ad hoc queries are used.

Hint-based prefetching algorithms [13] uses hints from applications to make prefetching decisions. Patterson *et al.* in [13] show how to use application disclosed access patterns (hints) to prefetch disk blocks in a file system setting. They also dynammically allocate file buffers among the three competiting demands of prefetching hinted blocks, caching hinted blocks for reuse and caching of recently used data for unhinted accesses. The drawback of hint-based approaches is the reliance on application provided hints. In the real world these hints often do not exist.

Context-model-based prefetching algorithms uses preceding events to model the next event. Examples algorithms include the Fido[5], PMC[4], PPM[3,14], type-level-based prefetching[15], PCM[16] and EPCM[9].

The Fido algorithm [5] prefetches by identifying and matching sequences of object accesses and storing them as patterns in pattern memory. However, their pattern memory mechanism of storing access sequences is prohibitively expensive. The PMC prefetching algorithm [4] uses discrete-time Markov chains (DTMC) to model object level access patterns. Since DTMC only allow the prediction of future accesses based on the current state, they only incorporate path information of length one. This approach is expensive in terms of statistics storage cost (using DTMC to model *object-level* transitions) and makes predictions late (short time between prediction and when the next prefetched page is referenced).

The algorithms in [3,14] use the principles of data compression for training and prediction. The intuition is that data compressors typically operate by predicting the dynamic probability distribution of the data to be compressed. If the data compressor successfully compresses the data, then its predicted probability distribution must be correct and can then be used for prediction in prefetching. One such algorithm is the predicition-by-partial-match (PPM) prefetching algorithm [3]. PPM uses a predictor-based on the higher order Markov chains (HMC) model. Curewitz *et al.* [3] found that PPM-3 (HMC model of order 3) gave the best performance among PPM and Fido. The problem with PPM is the coarse-grain (page-grain) at which statistics are stored. This coarse granularity of prediction produces less accurate prediction engines when compared to PCCP algorithms (which uses a hybrid object/page grain prediction engine).

Vitter *et al.* in [14] prove that compression-based algorithms are optimal under the assumption that there is sufficient time between page requests to prefetch as many pages as needed and the cache size is the only constraint. However, in the real world this assumption is rarely valid.

Han *et al.* [15] proposed a *type-level-based* prefetching algorithm for ORDBMSs. In their algorithm, recurring access patterns at the type-level are first identified and then used for prediction. Type-level access patterns are patterns of attributes that are referenced when accessing the objects. The drawback of this approach is its dependency on type-level access locality. Many ODBMS applications issue short ad hoc queries to the database, these queries do not exhibit type-level access locality.

Kroeger *et al.* in [16] proposed an approach for prefetching files in a files system based on a partitioned context model(PCM). In this approach a context model uses a trie (tree based data structure) to store sequences of events efficiently. The storage cost is bound by $O(n^m)$, where m is the highest order tracked and n is the number of unique files. If the number of files is large this approach uses too much storage space. To overcome this drawback the trie is partitioned, so that each partition consists of a first order node and all of its descendants. The number of nodes in each partition is limited to a static number that is a parameter of the model. The effect is that space requirements become $O(n)$.

[3] When a page is loaded upon request and no sooner.

Kroeger *et al.* in [9] identified that the PCM approach started prefetches too late and thus causing there to be only a small amount of overlap between disk IO and CPU. Kroeger *et al.* in [9] improves on PCM by proposing the extended PCM(EPCM) algorithm, which extends the model's maximum order to between 75% and 85% of the partition size. PCM restricts how the model grows by only allowing one node to be created for each instance of a specific pattern. In addition, if the predicted nodes has a child that has a high likelihood of access, we can also predict that file. The result is an algorithm that is both storage efficient and allows predictions to start early. However, the work is done in the context of prefetching files in file systems. This differs from our aim of prefetching pages in ODBMSs. However, in the experimental part of this paper we have adapted EPCM to prefetch pages for ODBMSs. In this paper we adapt EPCM to prefetch at the page-grain instead of the object-grain due to the fact EPCM is most effective if it is configured to store long sequences of references, but long sequences of *object references* would be prohibitively large to store in an OODBMS. Although EPCM allows memory usage to be limited (by limiting partition size), this comes at the expense of not being able to store long sequences of references. Our experiment in Section 7.3 shows that the *page-grained* EPCM already uses upto 20 times the storage space of the PCCP algorithms.

The second dimension of prefetching algorithm classification is granularity of prediction. There are three typical grains of prediction: object-grain; page-grain; and attribute-grain. object-grained techniques [4–8] make predictions using object-level information. page-grained algorithms [3, 11] observe access patterns that occur at the page-level (popular in file systems research). Attribute-grained algorithms use patterns of attributes to make predictions, eg. Han *et al.* [15] type-level-based prefetching algorithm.

Cao Pei et al. in [17] study the implications of integrating *prefetching* and *buffer replacement* when prefect knowledge of future access sequence is known. They argued that prefetching too early maybe harmful since early prefetching results in early buffer replacement if the cache is full. Early buffer replacement can be harmful since new and better replacement opportunities may open up as the program proceeds. Using this observation they develop two new integrated prefetching and buffer replacement algorithms called *aggressive* and *conservative*. These strategies were found to reduce application running time by up to 50% compared to no prefetching. However, their research is not applicable to our work since we do not assume perfect knowledge of future access patterns. In addition, their study was done within the context of file systems whereas we focus on ODBMSs.

Cao Pei et al. in [18] improve on their work in [17] by presenting the design and implementation of integrating application-controlled caching, prefetching, and disk scheduling in the multiprocess environment. They improve disk access latency by submitting prefetches in batches so that the requests can be scheduled to optimize disk access performance. Another improvement is that they no longer assume perfect future knowledge. They also show the proposed approach leads to significant performance improvement on real applications. However, like their previous study it is done in the context of files systems instead of ODBMSs.

Kraiss *et al.* in [19] consider an integrated approach to vertical file migration between tertiary, secondary, and primary storage. One of their sub-problems is the prefetching of documents from tertiary storage to secondary storage. They use a continuous Markov chain model for prediction purposes. The reason they are able to use the computationally expensive *continuous* Markov chain model is that they are optimizing for the high latency of the tertiary storage. However, in our problem we are optimizing for the relatively lower latency of secondary storage and thus can not afford the overhead of a maintaining a *continuous* Markov chain model.

3 Problem Statement

Given an object base, a set of workloads that operate on the object base, a buffer replacement algorithm, an object to page mapping (maybe created by a static clustering algorithm[4]), we are interested in improving the throughput[5] of the system by reducing the total stall time. The system can either load a page in response to a cache miss (demand fetch), or it can load a page before it is referenced in anticipation of a miss (prefetch). Let us assume it takes F time units to load a page into memory. When a program tries to access a page that is not available in the cache, it stalls until the page arrives in the cache. The stall time is: F if the page is loaded on-demand; $F - i$ if the load was started i time units ago; or $F + (F - i)$ if the currently loading page (load initiated i time units ago) is not the requested page. The $F + (F - i)$ stall time is derived from $F - i$ time units spent blocked loading the wrong page (not the next requested page), plus the F time units spent loading the correct page.

In our problem we assume disk IO can only be processed sequentially, and therefore, concurrent disk IO is not considered. We leave the concurrent disk IO problem for future work. We also assume a single user environment. Therefore, we do not consider multiple user transactions.

To reduce stall time, the prefetching algorithms attempt to accomplish the following goals:

- **Increase prefetch accuracy:** Correctly predicting and loading the next disk page request. Frequent

[4] Static clustering algorithms, recluster the database offline.

[5] Long term average performance.

wrong prefetches may cause system performance with prefetching to drop below that of demand fetching.
- **Earlier prefetching:** Earlier anticipation and prefetching of the next disk page request results in more overlap between IO and CPU.
- **Low overheads:** CPU and storage costs for both prediction and prefetching need to be kept low in order to make prefetching profitable.

Note the first and second goals are often in conflict. In general the earlier the prefetching is started the greater the number of possible candidate pages to prefetch. Since only one of these pages is the correct page, more candidate pages usually results in lower prefetch accuracy.

4 Path and Cache Conscious Prefetching (PCCP)

In this section we first describe the concept of path and cache conscious prefetching (PCCP). We then demonstrate the benefits of PCCP via a concrete example. Lastly we define the PCCP framework.

4.1 The Concept

This section introduces the two key concepts, *path* and *cache conscious* prefetching. Both concepts rely on historical training data to gain insight into how the database is being used. The training data is then used during prefetching.

In path conscious prefetching, features in the object trace are remembered during training and used to identify the current path of navigation during prefetching. The current navigational path information can then be used to determine the next non-memory resident page to be prefetched. The goal of path conscious prefetching is to identify the current path of navigation as early and accurately as possible.

Cache conscious prefetching uses training data to divide the pages of the database into two types: *'resident'*; and *'non-resident'*. *'Resident'* pages are deemed to be always memory resident (however, in practice they will sometimes be non-memory resident but this should occur rarely). In contrast, *'non-resident'* pages are deemed to be always non-memory resident (again in practice these pages are sometimes in memory). Having divided the database into 'resident' and 'non-resident' pages, cache conscious prefetching is only interested in prefetching the 'non-resident' pages. Since 'resident' pages are almost always in memory, avoiding them completely will cost only a small number of prefetch opportunities. The benefits of such an approach are that prefetching can be started earlier and prefetching storage overheads are lowered (only storing statistics for 'non-resident' pages). The reasons this approach allows prefetching to be started earlier are described in section 4.2.

Path and cache conscious prefetching can be used in a complementary fashion. First, cache conscious prefetching is used to classify pages as either 'resident' or 'non-resident'. Then path conscious prefetching gathers prefetching statistics using a limited scope ('non-resident' pages only). This approach provides the benefits of both path and cache conscious prefetching.

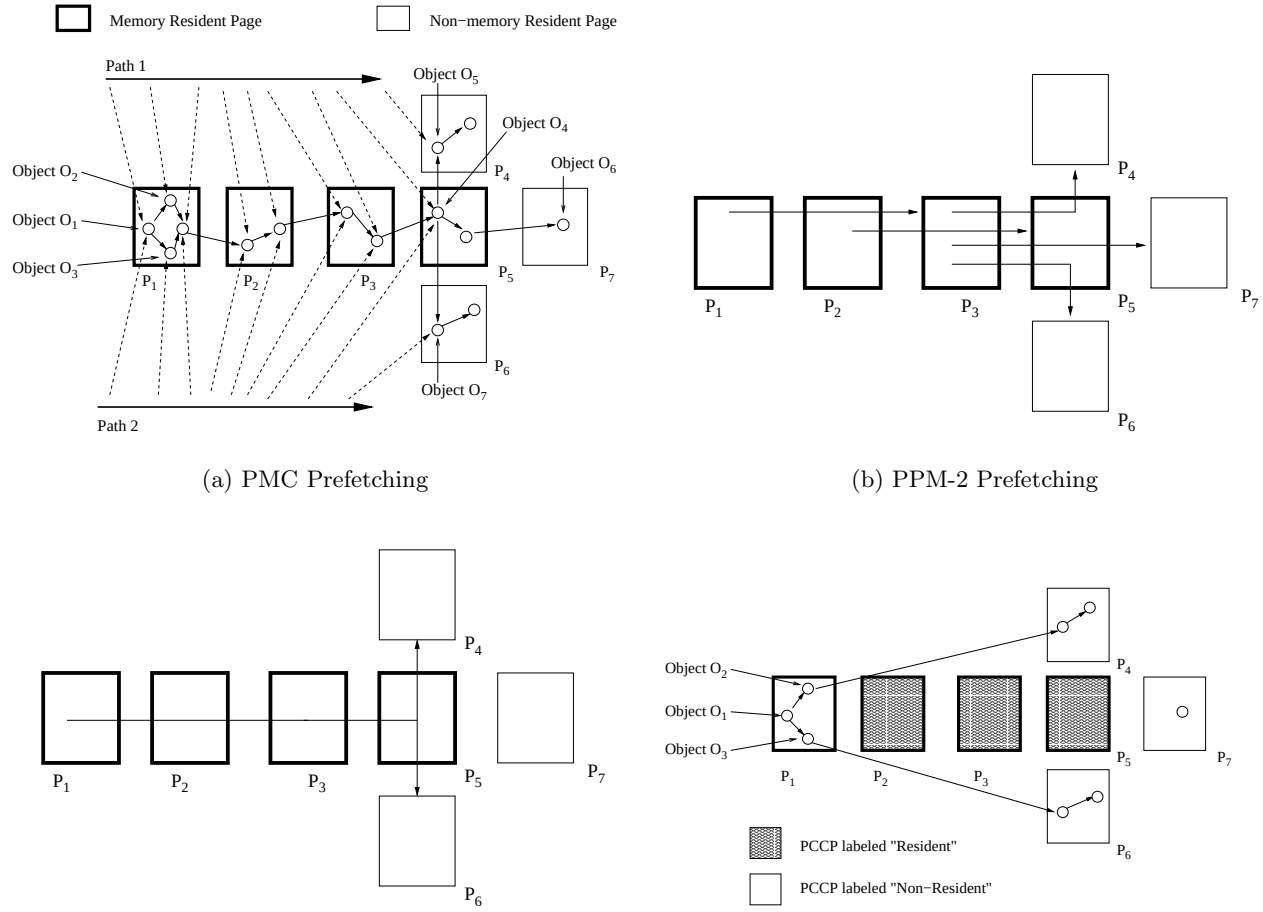
4.2 The Benefits of PCCP

In this section we describe the benefits of PCCP via a concrete example. In the example, PCCP is compared against three existing highly competitive context-model-based prefetching algorithms, PPM-2, PMC and a page-grained EPCM (see section 2 for a description).

Figure 1 contrasts the statistics collected by the prefetching algorithms, PMC, PPM-2, PCCP, and EPCM, given the same example object base navigations.

Figure 1(a) conveys both the illustration of two paths of navigation (path 1 and path 2) and PMC's object transition statistics. PMC uses an object level Discrete-Time Markov chains (DTMC) model to make predictions. This means PMC only stores object transition statistics between consecutive pairs of object references.[6] These statistics are stored in an object transition graph in which nodes represent objects and weights on edges represent probability of traversal. In order to avoid cluttering the figure, transition probabilities are not depicted. However, for the purposes of this example it is sufficient to assume that all probabilities are some number greater than zero but less than one. Now assume navigations starting from object O_1 always either follow path 1 or path 2. It should be possible to prefetch either page P_4 or P_6 depending on which object is referenced after object O_1. Page P_4 should be prefetched if the access sequence is O_1, O_2. Similarly the access sequence O_1, O_3 should predict page P_6. The problem with using PMC is that the statistics collected only capture the probability of transiting from one object to the next. This means that using PMC's statistics we can not prefetch with *total confidence* until we observe which path starting from O_4 is taken (from object O_4 there are three different possible objects to reference next). Therefore in this example PMC can not perform any prefetching if we need to have total confidence before prefetching a page. However, it is important to note that prefetching can be configured so that the pages with the highest probability of being referenced is prefetched, even if the confidence that the page will be referenced in less than 100%. In this case predictions can be made further into the future. However, in our example the page that would ultimately be prefetched by PMC would depend on the relative frequency of the following reference sequences O_4, O_5; O_4, O_6; and O_4, O_7 and not depend on whether the traversal

[6] Prefetching algorithms generally do not store longer sequences, due to high storage overheads.



(a) PMC Prefetching

(b) PPM-2 Prefetching

(c) Page-grained EPCM Prefetching

(d) PCCP Prefetching

Fig. 1 Illustration of the statistics collected for each prefetching algorithm, given the same example object base navigations.

has gone through O_1, O_2 or O_1, O_3. The result would be many wrong predictions. The reason is PMC uses only 1st order Markov Chain model. Using a higher order model at this fine-grain (object-grain) would be prohibitively expensive in terms of storage costs.

The same problem is encountered for PPM-2 prefetching (shown in figure 1(b)). PPM-2 prefetching collects only page level transition statistics. At page P_5 there are three possible next pages, P_4, P_6 and P_7. Therefore none of the pages can be prefetched with complete confidence. Again we can use a policy of prefetching the page with highest probability of being referenced. In this case the page prefetched would be the one with highest reference frequency from P_5. However, the traversal information that allows path 1 and path 2 to be distinguished is whether the object reference sequence O_1, O_2 or O_1, O_3 has been used. The reason this algorithm can not use this information is it does not consider object-grained access patterns.

The same problem is encountered for the page-grained EPCM prefetching algorithm (shown in figure 1(c)). Although the page-grained EPCM collections page transition information for a longer sequence of pages, it is still unable to predict the next page to be referenced until after page P_5 has been referenced. This is because it does not use object-grained access information to distingush whether the object reference sequence O_1, O_2 or O_1, O_3 has been used. It is important to note that this analysis is only valid for our *page-grained* adaptation of the EPCM prefetching algorithm. If an object-grained adaptation is chosen instead then EPCM can be prefetched after O_2 is accessed, since an object-grained EPCM can be configured to store long sequences of object references. Please see Section 2 for the reason we have adapted EPCM at the page-grain instead of object-grain.

Figure 1 (d) depicts the statistics that can be collected by PCCP algorithms. Assume in the case the cache conscious prefetching algorithm, PCCP has deemed pages P_2, P_3 and P_5 as memory resident and therefore are ignored for prefetching purposes. Then combining this knowledge and path conscious prefetching (where features in the object trace are used to distinguish be-

tween different paths), the statistics shown on Figure 1 (d) can be collected. In this example we define feature point as the first two objects referenced in each page. Using these statistics we can start the prefetch of page P_4 as soon as O_2 is referenced, since the statistics collected captures the knowledge that the sequence O_1, O_2 predicts page P_4. Similarly the prefetch of page P_6 can be started once O_3 is referenced.

This example demonstrates how PCCP can start a prefetch much earlier than PMC, PPM-2 and page-grained EPCM. In our simulation study (section 7) we found that situations similar to this example occur often. Frequently, many consecutive 'resident' pages references occur before a 'non-resident' page is referenced; and the first object referenced in a page can uniquely identify the next disk page referenced.

4.3 The PCCP Framework

In this section we describe the PCCP framework. The PCCP framework allows the definition of a family of prefetching algorithms which all possess path and cache consciousness. PCCP prefetching algorithms use context-model-based prediction engines and store statistics at both the page and object-grains. page-grained statistics are used to classify database pages as either memory 'resident' or 'non-resident'. object-grained statistics used for feature point selection are only stored for 'non-resident' pages. This approach uses less memory than existing object-grained algorithms that store statistics for both 'resident' and 'non-resident' pages. When compared to the page-grained prefetching algorithms, the inclusion of object-grained feature points for 'non-resident' pages results in the earlier identification of high probable paths of navigation.

In order to define a PCCP prefetching algorithm, the following steps must be followed:

- ***Define 'resident' / 'non-resident' page metric:*** Cache conscious prefetching requires the classification of database pages as either memory 'resident' or 'non-resident'. In this step, a metric is used to rank pages in terms of likelihood of being memory resident at any moment in time. Example metrics include: frequency of page references; sum of past memory residency durations; and hot/cold page classification information given by the C3 clustering algorithms (see section 5 for a PCCP prefetching algorithm that uses this metric). Database pages are sorted according to this metric in descending order and the first *MEM_RES_PAGES* pages are classified as memory 'resident', the remaining pages are classified as memory 'non-resident'. A possible basis for choosing *MEM_RES_PAGES* is via physical memory size, e.g. *MEM_RES_PAGES* multiplied by page size should equal 90% of physical memory.
- ***Define feature point selection algorithm:*** In this step, an algorithm is defined for finding feature points in the trace. Feature points are object sequences occurring at special points in the trace. A feature point can span one or more pages. During prediction engine training, feature points are identified and stored, together with the page that the feature point predicts. For example, a feature point selection algorithm that picks the first two object references occurring in a page as a feature point stores the following statistics: at every page reference, the object ID of the first two objects referenced is stored (in sequential reference order) together with the page ID of the next page reference.
- ***Define prefetch threshold:*** If the probability of next navigating to a particular 'non-resident' page is greater than the prefetch threshold (*PREF_THRESHOLD*), that page is prefetched. The prefetch threshold is user defined.

At prefetch time the prediction engine looks for feature points occurring in 'non-resident' pages. When one is found, the corresponding training data is loaded and used to find the next 'non-resident' page with the largest probability of being referenced. If that page's probability of reference is greater than *PREF_THRESHOLD*, the page is prefetched.

5 Two New Concrete PCCP Algorithms

In this section we describe two new prefetching algorithms created from the PCCP framework called heat-based prefetching(HP) (where 'heat' is simply a measure of access frequency[7]) and integrated prefetching (IP). The IP algorithm integrates clustering information into prefetching statistics. These two algorithms are derived from the following PCCP design decisions:

- ***'resident' / 'non-resident' page metric:*** We define two alternative 'resident' / 'non-resident' page metrics.
  - *Heat-based (HB):* In this approach we use page heat as the 'resident' / 'non-resident' page metric. This is based on the observation that in general, frequently referenced pages are less likely to be evicted at buffer replacement time. The HP prefetching algorithm uses this 'resident' / 'non-resident' page metric. Figure 2 shows the algorithm used by HP to label pages as 'resident' or 'non-resident'.
  - *Clustering-based (CB):* In this approach we use clustering information to determine whether a page is 'resident' or 'non-resident'. More specifically, clustering information from the C3-GP [22] clustering algorithm is used. C3-GP first divides the

[7] This term has been extensively used in the existing literature[20, 21].

```
Determine_Resident_Pages_HP(LP: list of pages)

1. Sort pages in LP in decreasing order of heat.
2. Label first MEM_RES_PAGES pages of LP as
   "resident pages" and label the remaining pages
   as "non-resident".
```

Fig. 2 Alogrithm used by the HP prefetching algorithm for labeling pages.

```
Determine_Resident_Pages_IP(LP: list of pages)

1. Label C3-GP clustering algorithm's hot pages as
"resident pages".
2. MEM_RES_PAGES = number of hot pages created
by the C3-GP clustering algorithm.
```

Fig. 3 Alogrithm used by the IP prefetching algorithm for labeling pages.

database into hot and cold regions, then clusters objects of each region into pages separately. In this approach we classify all pages in C3-GP's hot region as 'resident' pages and the remaining pages as 'non-resident'. The IP prefetching algorithm uses this 'resident' / 'non-resident' page metric. Figure 3 shows the algorithm used by IP to label pages as 'resident' or 'non-resident'.

- ***Feature point selection algorithm:*** We have used the following criteria to choose a feature point selection algorithm: the feature points should identify a high probable path navigation as early as possible; and there should be a minimum amount of information stored per page in order to decrease memory usage. In order to accomplish this we define a sequence of N consecutive entry objects as a feature point. Where, 'entry object' refers to the first object referenced in each page. During prediction engine training, the object IDs of N consecutive entry objects are stored (in sequential reference order), together with the probabilities of navigating to the next 'non-resident' page. This effectively means we use a higher order Markov chain model, in which the current state is defined by N consecutive entry objects of 'non-resident' pages and the next state is defined as a set of 'non-resident' pages and their probability of reference.

Figure 4 shows the algorithm used by both HP and IP to train the prediction engine. Figure 5 shows the algorithm used by both HP and IP to determine when and which to pages to prefetch.

```
Train_Prediction_Engine(S: sequence of object
                        references, N: number of
                        consecutive entry objects
                        needed to define a feature point)

1. Initialise an N^th-order Markov Chain MC.
2. For each object reference r in S begin
3.     If r is an entry object in a "non-resident" page then
4.         Use r to incrementally train MC
5.     End if.
6. End for.
7. return MC.
```

Fig. 4 Algorithm used to train the prediction engine. The same algorithm is used for both HP and IP prefetching algorithms.

```
Prefetch_Pages(S: sequence of object references,
               P: prediction engine)

1. For each object reference r in S begin
2.     If r is an entry object of a "non-resident" page then
3.         Given r as input to P, predict next "non-resident"
           page reference np with highest probability.
4.         If np is currently not in memory and probability
           of referencing np > PREF_THRESHOLD then
5.             Prefetch np.
6.         End if.
7.     End if.
8. End for.
```

Fig. 5 Algorithm used by both HP and IP to prefetch pages.

5.1 Algorithm Analysis

In this section we analyze the time and space complexity of the HP and IP prefetching algorithms. The time complexity of HP for the page labeling algorithm (Figure 2) is $O(n)$ for a sequence of n object references, since the heat of a page is incremented every time an object is referenced. The time complexity of IP for the page labeling algorithm (Figure 3) is $O(m)$, where m is the number of pages in the database. This is because we need to label each page according to label assigned by the GP-C3 clustering algorithm.

The following time complexities apply to both HP and IP prefetching algorithms since they both use the same techniques. The time complexity of the prediction engine training algorithm (Figure 4) is $O(e^N)$, where e is the number of objects in the database and N is the order of the Markov Chain used. This is due to the space needed to update the N^{th}-order Markov Chain. In practice the time complexity is much lower than e^N since many objects are not entry objects and only entry objects in 'non-resident' pages are considered. In our experiments in Section 7 we show small N values (i.e. 1 and 3) provides good results. The time complexity of the page

prefetching algorithm (Figure 5) is $O(n)$ for a sequence of n object references. This is because it takes a constant time computation to find the next "non-resident" page reference np with highest probability (line 3) for each new object referenced.

Both HP and IP prefetching algorithms have a space complexity of $O(e^N)$. This is due to the space needed to store the N^{th}-order Markov Chain. In practice the space complexity is much lower than e^N since many objects are not entry objects and only entry objects in 'non-resident' pages are considered when growing the Markov Chain.

Parameter Description	Value
System class	Centralised
Disk page size	4096 bytes
Buffer size	varies
Buffer replacement strategy	LRU
Pre-fetching policy	varies
Object initial placement	Optimised sequential
Object think time	1 ms
Disk seek time	6.5 ms
Disk latency	4.3 ms
Disk transfer time	0.5 ms

Table 1 VOODB parameters.

6 Experimental Setup

In this section we report the experiemtal setup used to evaluate the performance of the prefetching algorithms. The prefetching algorithms are benchmarked with the Object Clustering Benchmark (OCB) [23] using the Virtual Object Oriented Database simulator, (VOODB) [24].

6.1 Simulator Setup

VOODB is based on a generic discrete-event simulation framework. Its purpose is to allow performance evaluations of OODBs in general, and optimisation methods such as clustering in particular. VOODB simulates all of the traditional OODBMS components such as, the transaction manager, object manager, buffer manager, IO subsystem, *etc.* The correctness of VOODB has been validated for two real-world OODBs, O_2 [25] and Texas [26].

VOODB is implemented on top of the discrete-event simulation package for C++ (DESP-C++) [27]. DESP-C++ is a validated simulation package which performs 20 to 1000 times faster than competing simulation packages like the 'queuing network analysis package 2^{nd} generation' (QNAP2). The high performance of DESP-C++ allows us to test more complex workloads and system settings.

VOODB's prefetching simulation framework was not fully developed. Consequently we extended the simulator to allow full support for our prefetching algorithms. The extended simulator is validated against example traces we created and computed the IO stall time for. The VOODB parameters used for the experiments in this chapter are shown on table 1. The parameters used are based on typical system settings and hardware characteristics, eg. disk speeds.

6.2 Benchmark Environment

This paper uses the OCB benchmark to evaluate prefetching performance. The OCB benchmark is initially designed for benchmarking clustering algorithms but its rich schema and realistic workloads makes it particularly suitable for benchmarking prefetching algorithms too. The OCB database has a variety of parameters which makes it very user-tunable. A database is generated by setting parameters such as total number of objects, maximum number of references per class, base instance size, number of classes, *etc.* Once these parameters are set, a database conforming to these parameters is randomly generated. The database consists of objects of varying sizes. In the experiments reported in this chapter, a total of 100, 000 objects are generated. The objects varied in size from 50 to 1600 bytes and the average object size is 232 bytes. The total database size is 23 MB. Although this is a small database size, we also use small cache sizes (1MB and 11MB) to keep the database to cache size ratio large. Since we are interested in the caching behavior of the system, the database to cache size ratio is a more important parameter than database size alone. The OCB database parameters used are shown on table 2 (a).

In the experiments we start with an empty cache which fills up quickly due to the small cache size ratio and the large number of transactions run (10000). Thus, in our experiments many page evictions occur. The buffer replacement algorithm we have used is the popular least recently used(LRU) algorithm.

The OCB workload used in this study included simple, hierarchical and stochastic traversals [23]. The simple traversal performs a depth first search starting from a randomly selected root object. The hierarchical traversal picks a random root object and a random reference type and then always follows the same reference type up to a pre-specified depth. The stochastic traversal selects the next link to cross at random. At each step, the probability of following reference number N is p(N) = $\frac{1}{2^N}$. Stochastic traversals approach Markov chains, which are known to simulate real query access patterns well [28]. Each transaction involved execution of one of the three traversals. The OCB workload parameters used are shown on table 2 (b).

We introduce skew into the way traversal roots are selected. Roots are partitioned into hot and cold regions. In all experiments the hot region is set to 3% of the size

Parameter Description	Value
Number of classes in the database.	50
Maximum number of references, per class.	10
Instances base size, per class.	50
Total number of objects.	100000
Number of reference types.	4
Reference types random distribution.	Uniform
Class reference random distribution.	Uniform
Objects in classes random distribution.	Uniform
Objects references random distribution.	Uniform

(a) OCB database parameters

Parameter Description	Value
Simple traversal depth.	2
Hierarchy traversal depth.	4
Stochastic traversal depth.	4
Transaction root selection distribution.	Hot/Cold
Simple traversal selection probability.	0.3
Hierarchical traversal selection probability.	0.35
Stochastic traversal selection probability.	0.35
Number of transactions.	100000

(b) OCB workload parameters

Table 2 Parameters used for OCB.

of database and has an 80% probability of access.[8] These settings are similar to those used in related work. Gray and Putzolu [29] cites statistics from a real videotext application in which 3% of the records got 80% of the references. Carey *et al.* [30] used a hot region size of 4% with a 80% probability of being referenced in the HOTCOLD workload used to measure data caching tradeoffs in client/server OODBMSs. Franklin *et al.*[21] used a hot region size of 2% with a 80% probability of being referenced in the HOTCOLD workload used to measure the effects of local disk caching for client/server OODBMSs.

6.3 Result Generation

The results are generated via four steps. The first *clustering training* step runs the database and collects clustering statistical data. The second *clustering* step uses the training data with the clustering algorithm to rearrange objects. The third *prefetching training* step runs the newly clustered database to collect prefetching statistical data. The fourth *evaluation* step runs the prefetching algorithm with the newly clustered database to measure the performance of the system. In the experiments we generate different random traces for the third and fourth steps, using the same hot region and query type.

6.4 Clustering Algorithms Used

Most of the experiments in this chapter include two sets of results, one set uses the C3-GP [22] clustering algorithm and the other set uses a combination of three clustering policies, greedy graph partitioning (GGP) [31], Wisconsin greedy graph partitioning WGGP [32], and no clustering.

C3-GP is a member of the C3 [22] family of cache conscious clustering algorithms. The C3-GP clustering algorithm works in two phases [22]. In the first phase objects are sorted into decreasing order via heat, then cut to produce a 'hot' and 'cold' region. The hot region is almost equal to the size of memory. In our experiments we set a C3-GP hot region size of 90% size of memory[9]. In the second phase objects of each region are further partitioned into pages by the greedy graph partitioning clustering algorithm (GGP). C3-GP has been shown to outperform the highly competitive GGP in many situations [22].

GGP [31] and WGGP [32] are both members of the graph partitioning family of clustering algorithms. Before the appearance of the C3 family of clustering algorithms, graph partitioning algorithms [32] were widely accepted as the best performing existing clustering algorithms [31,32]. GGP and WGGP both use the simple Markov chain model (SMC) to cluster objects accessed frequently together in time into the same page. GGP and WGGP both operate greedyly, the way they differ is the way they form partitions. GGP starts by placing each object into a different partition and then iteratively joins partitions. In contrast, WGGP starts with an empty partition and then iteratively fills it up before moving onto the next partition, where the process is restarted with the remaining un-partitioned objects.

6.5 Prefetch Algorithms

The prefetching algorithms shown in the result graphs of this chapter are labeled as follows:

[8] That is, there is a 80% probability that the root of a traversal is from the hot region.

[9] This settings has been found to produce the best C3-GP clustering performance [22].

– **DM**: demand fetching;
– **PPM-1**: 1st order PPM prefetching algorithm [3];
– **PMC**: Knafla's [4] object-grained statistical prefetching;
– **PPM-3**: 3rd order PPM prefetching algorithm [3];
– **PCCP-IP1**: 1st order IP prefetching, see section 5;
– **PCCP-IP3**: 3rd order IP prefetching, see section 5;
– **EPCM**: page-grained enhanced PCM algorithm [9];
– **PCCP-HP1**: 1st order HP prefetching, see section 5;
– **PCCP-HP3**: 3rd order HP prefetching, see section 5;

The reason we report the results for 3rd order PCCP algorithms is that in our experiments we have found HP3 and IP3 give best results among the PCCP variants. This is similar to the conclusions made by Curewitz *et al.*[3], in which they found the third order PPM-3 gives best performance among the PPM variants.

The IP1 and IP3 algorithms require the use of clustering information from the C3-GP clustering algorithm, to classify pages as 'resident' and 'non-resident'. IP1 and IP3 classify pages occurring in C3-GP's hot region as 'resident' and the remaining pages as 'non-resident'. HP1 and HP3 rank database pages in terms of frequency of page reference. Once ranked, HP1 and HP3 classify the first *MEM_RES_PAGES* pages as being 'resident', where *MEM_RES_PAGES* multiplied by page size equals 50% of the memory size. HP1 and HP3 classify the remaining pages as being 'non-resident'.

We have configured the PPM algorithms to prefetch only one page at each step. This is for two reasons: we do not assume there is concurrent disk IO (see the problem statement in Section 3); and the time between disk page requests is normally quite small and the cache size /workload size ratio is small (a very common case in real systems).

In each experiment[10] the prefetch threshold[11] is set to 0.9 for every prefetch algorithm other than EPCM. The prefetch threshold for EPCM is set to 0.4. We have tested different settings (at 0.1 increments) for each prefetching algorithm and found that the best setting is 0.4 for EPCM and 0.9 for the other prefetching algorithms.

For EPCM we have set the maximum order to 45 and the partition size to 64. These settings have given best results for the experiments in [9] and we have found the same is true for our experimental setup.

6.6 Performance Metric

The results reported are in terms of ratio of prefetching stall time over demand fetching stall time, i.e. the amount of time the system is idle waiting for a page to load when using prefetching over when using demand fetching. This metric provides the reader with an idea of how much stall time is reduced by using prefetching instead of just demand fetching.

[10] Except for the experiment where the prefetch threshold is varied.

[11] The minimum probability of being the next disk-resident page to be referenced.

7 Experimental Results

In this section we report the results of experiments conducted to compare the performance of four existing prefetching algorithms designed for ODBMSs and four new algorithms produced using the PCCP framework.

7.1 Varying Cache Size

In this experiment we measure the effect of varying cache size on the performance of the prefetching algorithms. Two sets of results are collected for this experiment, the first set uses the C3-GP clustering algorithm and the second set reports an average of the results from using three different clustering policies GGP, WGGP and no clustering. The effects of the three clustering algorithms are also individually reported in the appendix. Note PCCP-IP1 and PCCP-IP3 results are only shown for C3-GP results (figure 6 (a)) since PCCP-IP1 and PCCP-IP3 require clustering information from C3-GP to classify 'resident' and 'non-resident' pages.

The prefetching results shown on figure 6 (a) depict the PCCP algorithms performing better than the other algorithms when cache size is small when the C3 clustering algorithm is used. When the cache size is large (at beyond 6 MB), almost the entire working set fits in memory. Thus almost all the pages in the working set are classified as 'resident' by the PCCP algorithms. Since PCCP algorithms only prefetch 'non-resident' pages and there are none of them in the working set, no prefetching is performed. Hence, the performance of PCCP algorithms rapidly degrades to that of demand fetching at these large cache sizes.

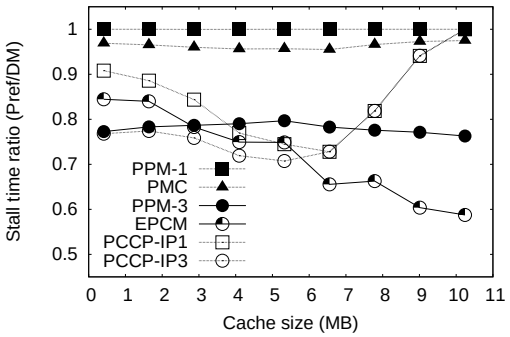
The results for using the GGP, WGGP and no clustering algorithms shown on figure 6 (b). The results show that EPCM begins outperforming PCCP algorithms after a cache size of 1 MB. The reason for this is that unlike the C3 clustering algoirthm the clustering algorithms used for this experiment do not separate pages into hot and cold pages. This means that the PCCP algorithms which works best when resident and non-resident pages are very distinct can not perform at its peak. The result is EPCM which does not need to distinguish between resident and non-resident pages performs relatively better than PCCP algorithms in this experiment rather than when the C3 clustering algorithm is used.

There are two main observations that can be made from figure 6 (b). First, at large cache sizes the PCCP algorithms in figure 6 (b) exhibit a milder rate of performance degradation than figure 6 (a). The cause lies in the way the clustering policies work. The clustering policies GGP, WGGP and no clustering (used for 6 (b)),

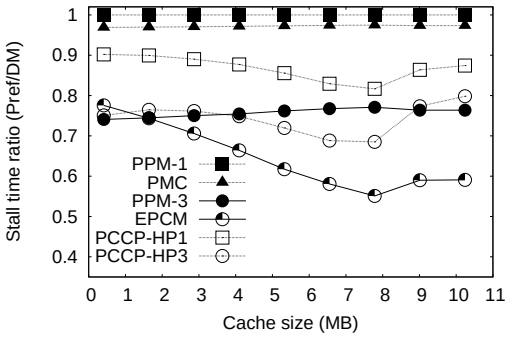
are not designed to produce pages of homogeneous heat whereas C3-GP is designed to produce pages of homogeneous heat. The result is that the clustering policies GGP, WGGP and no clustering need a larger cache size to fit the entire working set in memory (the working set is spread across more pages). Hence, given the same cache size, in figure 6 (b) PCCP classifies more pages that contain objects of the working set as 'non-resident'. Since PCCP algorithms only attempt to prefetch 'non-resident' pages, figure 6 (b) gives PCCP algorithms more opportunities for prefetching.

Second, in figure 6 (b) PCCP-HP3 outperforms PCCP-HP1 consistently by a large margin. This contrasts with figure 6 (a) in which the two PCCP algorithms perform about the same after cache size of 6.5 MB. Unlike C3-GP, the clustering policies GGP, WGGP and no clustering do not extract hot objects from cold pages. The result is that 'non-resident' pages may contain hot objects (which often have a large fan out). In these conditions a 'non-resident' page (used by PCCP algorithms for prefetch prediction purposes) may contain many different paths of navigation. Hence, under these conditions, PCCP-HP3, which identifies navigational paths based on more historical reference information, can more accurately identify the current path of navigation when compared to PCCP-HP1.

The reason for the poor performance of PPM-1 is that it only uses the current state to predict the next state (SMC model). Furthermore, it stores prediction information at the page-grain. The combination of the two drawbacks makes it very difficult to accurately distinguish between different paths of navigation early enough for prefetching. This problem is compounded by the rich schema and workloads (creating many different paths of navigation intersecting in a multitude of places) used in our experiments.



(a) Using C3-GP clustering



(b) Using GGP, WGGP and no clustering

Fig. 6 Results of varying cache size. The results on the right report an average of the results from using three different clustering policies GGP, WGGP and no clustering. The appendix shows the results of using the clustering policies individually.

7.2 Varying Clustering Algorithm

In this experiment, we examine the effect that varying clustering algorithms has on prefetching algorithm performance. The results are shown on figure 7. The cache size was set to 6 MB. For each prefetching algorithm, the results of no clustering and three different clustering algorithms are reported in the following order: no clustering; the Wisconsin greedy graph partition algorithm (WGGP) [32]; the greedy graph partitioning algorithm (GGP) [31]; and the C3-GP clustering algorithm. PCCP-IP1 and PCCP-IP3 prefetching algorithms are used when the C3-GP clustering algorithm is used. PCCP-HP1 and PCCP-HP3 are used for the remaining clustering policies.

The results show that the PCCP-IP3/HP3 algorithms shows similar or better performance than all existing prefetching algorithms (except for EPCM) for all clustering algorithms tested, including no clustering. The reason that EPCM outperforms PCCP is that it stores very long page transition information, but that comes at a large storage cost (this will be demonstrated in Section 7.3).

7.3 Statistics Storage Costs

In this experiment we have examined the statistics storage requirements of the prefetching algorithms. The results show the number of statistics data values stored instead of the size of the data structures needed. We define a data value as a statistic that needs to be stored (e.g. the heat of an object). The reason for measuring the number of data values instead of storage size is that there are many different existing data structures which have various speed to space trade-offs[12], including some

[12] Data structures used for the prefetching algorithms tested will require the index key to be a combination of a pair of IDs, and thus simple data structures like arrays are precluded.

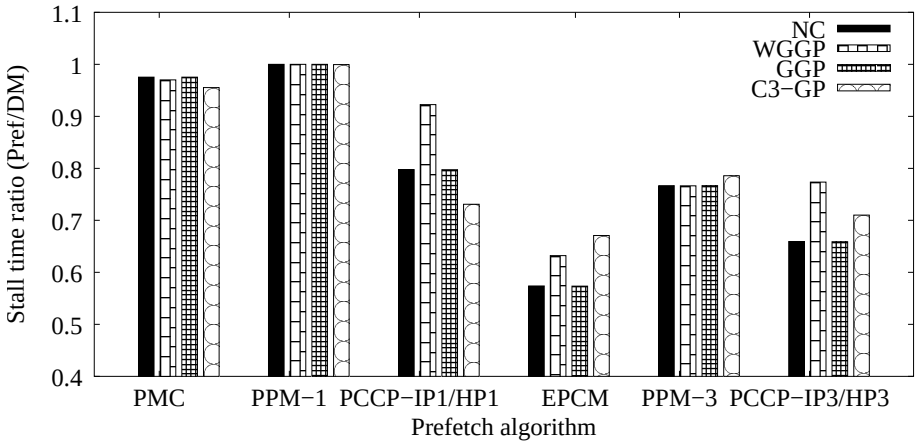


Fig. 7 The impact of varying clustering algorithm. Each prefetching algorithm is tested against no clustering and three different clustering algorithms. The results are shown in the following order: NC (no clustering); WGGP; GGP; and C3-GP.

that limit statistics space consumption by flushing and rebuilding the data structures once a size limit has been reached. However, all of the data structures will offer better speed and storage size performances when the number of data values stored is smaller.

The results are shown on figure 8. The cache size used in this experiment is 6 MB. PCCP algorithms require the least space for storing data values. PCCP derives its cost savings mainly from restricting the storage of statistics to only 'non-resident' pages. In addition, the low statistics storage requirements of path conscious prefetching (storing short feature points) also helps to reduce the storage costs of the PCCP algorithms. These results show that path and cache conscious information (used by PCCP algorithms) can be stored efficiently.

The results show PCCP algorithms stores upto 20 times less statistics than EPCM. The reason for this is EPCM stores long chains of page transitions statistics. Whereas PCCP algorithms only stores transition statistics of upto length of 3 (for PCCP-IP3 and PCCP-HP3) and that is further restricted to only 'non-resident' pages.

A surprising result is that PCCP-IP3 and PCCP-HP3 store around the same number of data values as their single page counterparts (PCCP-IP1 and PCCP-HP1). For the purposes of explaining this behavior let us assume n navigations passing through an entry object (first object in a page to be referenced) goes to n different target pages (next page referenced). Further assume the n navigations each originate from different previous page entry objects. Under these conditions, PCCP-IP3 and PCCP-HP3 store the same number of data values as their single page counterparts. We now explain why this type of navigational pattern occurs often in our experiments. It is due to a combination of the cache conscious feature of PCCP algorithms and trace characteristics. PCCP's cache conscious feature excludes 'resident' pages (pages more likely to contain hot objects) from prefetching statistics. Due to trace characteristics it is often the hot objects that have high fan outs. Thus the effect is hot objects that have high fan out (which produces large number of diverging paths of navigation) are excluded from prefetching statistics.

7.4 Varying Prefetch Threshold

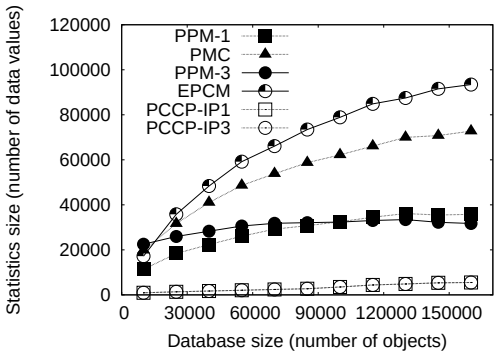
In this experiment we vary the prefetch threshold of each prefetching algorithm. The prefetch threshold is a user defined parameter that specifies the minimum probability required before a prefetch is allowed to occur. The cache size is set to 6 MB.

The results are shown on figure 9. EPCM shows the greatest sensitivity to prefetching threshold. This is undesirable since it means EPCM's prefetch threshold needs to be fine tuned before it can yield optimal performance. PPM-1 and PMC are also sensitive to changes in the prefetch threshold, especially at low threshold values. In contrast, the PCCP algorithms and PPM-3 are not sensitive to the prefetch threshold.

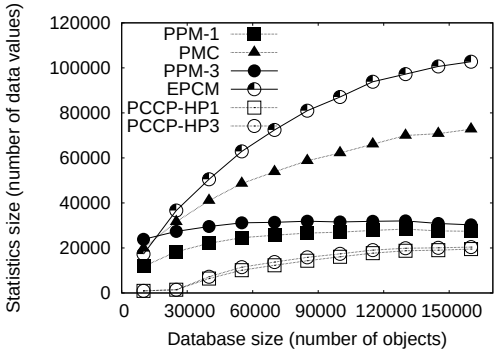
8 Conclusion

This paper proposes PCCP, a new ODBMS prefetching framework. The framework allows the definition of a family of prefetching algorithms which possesses the properties of path and cache consciousness. In order to demonstrate the usefulness of the PCCP framework, we have used it to create four new prefetching algorithms. We have conducted an extensive experimental study comparing the four new prefetching algorithms with four existing prefetching algorithms.

The results show that PCCP algorithms outperforms the existing prefetching algorithms PPM-1, PPM-3, and PCM in a variety of situations. When comparing PCCP

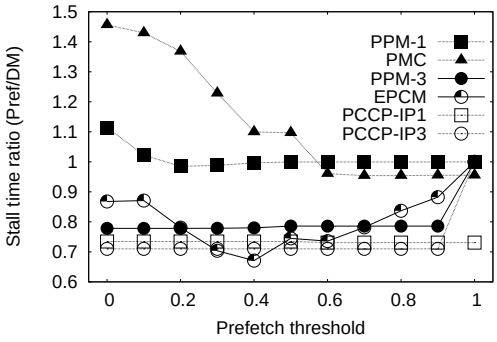


(a) Using C3-GP clustering

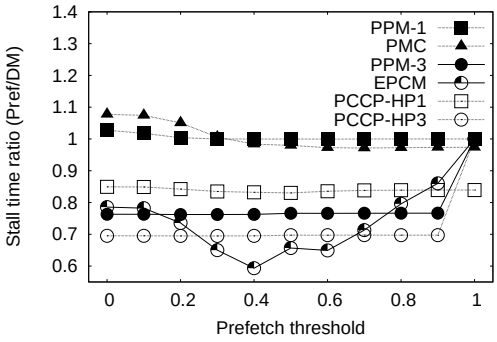


(b) Using GGP, WGGP and no clustering

Fig. 8 Statistics storage cost results. The results on the right report an average of the results from using three different clustering policies GGP, WGGP and no clustering. The appendix shows the results of using the clustering policies individually.



(a) Using C3-GP clustering



(b) Using GGP, WGGP and no clustering

Fig. 9 Results of varying the prefetch threshold. The results on the right report an average of the results from using three different clustering policies GGP, WGGP and no clustering. The appendix shows the results of using the clustering policies individually.

against EPCM, we found PCCP is better in terms of producing less stall time for small buffer sizes, uses less storage overheads and is more robust to prefetch threshold settings. However, EPCM produces less stall time than PCCP when EPCM's prefetch threshold is fine tuned to its optimal setting and the cache size is large.

When choosing between EPCM and PCCP we recommend that PCCP be used if memory space is very constrained, since PCCP stores less statistics and produces less stall time when cache size is small. EPCM should be used if memory size is large, since EPCM produces less stall time for this situation.

We identify four directions of future work. First, is to develop and test new PCCP algorithms. Second, is a more in depth investigation of how clustering statistics can be used by the prefetching algorithm to improve performance. Third, heat may not be the ideal 'resident' / 'non-resident' metric for database scenarios such as long running transactions (customer account history preparation transaction), where a page maybe needed in the cache for quite a long time but is not frequently referenced. A direction of future work is to try other methods to define the 'resident' / 'non-resident' metric such as total duration of references for such scenarios.

Acknowledgment

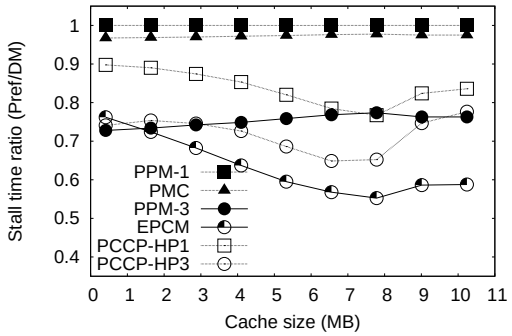
We would firstly like to thank Stephen Blackburn for his careful proof reading of this paper, he made numerous grammatical corrections. We would also like to thank Jerome Darmont for making the sources of VOODB and OCB freely available. These tools have helped our experimental work tremendously.

References

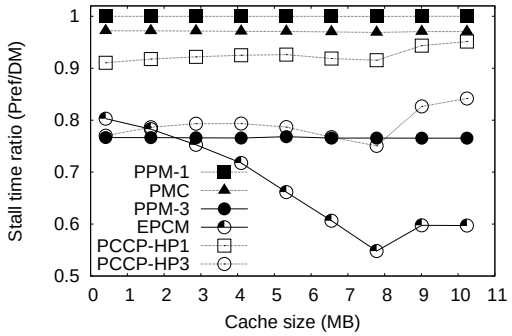
1. Knafla, N. (1999) *Prefetching Techniques for Client/Server, Object-Oriented Database Systems*. Ph.D. thesis, University of Edinburgh.

2. Ailamaki, A., Dewitt, D. J., Hill, M. D., and Wood, D. A. (1999) DBMSs on a modern processor: Where does time go? *The 25th VLDB conference*, September, pp. 266–277.
3. Curewitz, K. M., Krishnan, P., and Vitter, J. S. (1993) Proceedings of practical prefetching via data compression. *ACM SIGMOD Conf. on Management of Data*, 26-28 May, pp. 43–53.
4. Knafla, N. (1998) Analysing object relationships to predict page access for prefetching. *Proceedings of Eighth Int. Workshop on Persistent Object Systems: Design, Implementation and Use (POS-8)*, pp. 160–170.
5. Palmer, M. and Zdonik, S. B. (1991) Fido: A cache that learns to fetch. *Proc. of the 17th Int. Conf. on Very Large Data Bases*, Septmeber, pp. 255–264.
6. Bernstein, P. A., Pal, S., and Shutt, D. (1999) Context-based prefetching for implementing objects on relations. *25th International Conference on Very Large Data Bases*, Sept, pp. 327–338, Morgan Kaufmann.
7. Chang, E. E. and Katz, R. H. (1989) Exploiting inheritance and structure semantics for effective clustering and buffering in an object-oriented DBMS. Clifford, J., Lindsay, B. G., and Maier, D. (eds.), *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989*, pp. 348–357.
8. Knafla, N. (1997) A prefetching technique for object-oriented databases. *Advances in Databases, 15th British National Conf. on Databases*, pp. 154–168.
9. Kroeger, T. M. and Long, D. D. E. (2001) Design and implementation of a predictive file prefetching algorithm. *Proceedings of the 2001 USENIX Annual Technical Conference*, pp. 105–118.
10. Gerlhof, C. A. and Kemper, A. (1994) A multi-threaded architecture for prefetching in object bases. *International Conference on Extended Database Technology (EDBT)*, pp. 351–364.
11. Joseph, M. (1970) An analysis of paging and program behaviour. *The Computer Journal*, **13**, 48–54.
12. Liskov, B., Adya, A., Castro, M., Day, M., Ghemawat, S., Gruber, R., Maheshwari, U., Myers, A., and Shira, L. (1996) Safe and efficient sharing of persistent objects in thor. *Proc. of the ACM SIGMOD/PODS96 Joint Conf. on Management of Data*, pp. 318–329.
13. Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. (1995) Informed prefetching and caching. *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pp. 79–95.
14. Vitter, J. S. and Krishnan, P. (1996) Optimal prefetching via data compression. *Journal of the ACM*, **43**, 771–793.
15. Han, W., Whang, K., Moon, Y., and Song, I. (2001) Prefetching based on the type-level access patterns in object-relational DBMSs. *Proceedings of IEEE international Conference on Data Engineering (ICDE 2001)*, pp. 651–660.
16. Kroeger, T. M. and Long, D. D. E. (1999) The case for efficient file access pattern modeling. *Proceedings of the 7th workshop on hot topics in operating systems (HotOS-VII)*.
17. Cao, P., Felten, E. W., Karlin, A. R., and Li, K. (1995) A study of integrated prefetching and caching strategies. *ACM SIGMETRICS*, pp. 188–197.
18. Cao, P., Felten, E. W., Karlin, A. R., and Li, K. (1996) Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, **14**, 311–343.
19. Kraiss, A. and Weikum, G. (1998) Integrated document caching and prefetching in storage hierarchies based on markov-chain predictions. *The VLDB Journal*, **7**, 141–162.
20. Carey, M. J., Franklin, M. J., and Zaharioudakis, M. (1994) Fine-grained sharing in a page server OODBMS. *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 359–370.
21. Franklin, M. J., Carey, M. J., and Livny, M. (1993) Local disk caching for client-server database systems. Agrawal, R., Baker, S., and Bell, D. A. (eds.), *Proceedings of the VLDB Conference*, pp. 641–655.
22. He, Z. and Marquez, A. (2001) Cache conscious clustering C3. *12th International Database and Expert Systems Applications Conference (DEXA 2001)*, September, pp. 815–825.
23. Darmont, J., Petit, B., and Schneider, M. (1998) OCB: A generic benchmark to evaluate the performances of object-oriented database systems. *International Conference on Extending Database Technology (EDBT)*, March, pp. 326–340, LNCS Vol. 1377 (Springer).
24. Darmont, J. and Schneider, M. (1999) VOODB: A generic discrete-event random simulation model to evaluate the performances of oodbs. *The 25th VLDB conference*, September, pp. 254–265.
25. Deux, O. (1991) The O_2 system. *Communications of ACM*, **34**, 34–48.
26. Singhal, V., Kakkad, S. V., and Wilson, P. R. (1992) Texas: An efficient, portable persistent store. *5th International Workshop on Persistent Object Systems*, pp. 11–33.
27. Darmont, J. (2000) DESP-c++:a discrete-event simulation package for c++. *Software Practice and Experience*, **30**, 37–60.
28. Tsangaris, M. M. and Naughton, J. F. (1992) On the performance of object clustering techniques. *In Proceedings of the ACM SIGMOD conference on Management of Data*, pp. 144–153.
29. Gray, J. and Putzolu, G. R. (1987) The 5 minute rule for trading memory for disk accesses and the 10 byte rule for trading memory for CPU time. *In Proceedings of the ACM SIGMOD conference on Management of Data*, pp. 395–398.
30. Carey, M. J., Franklin, M. J., Livny, M., and Shekita, E. J. (1991) Data caching tradeoffs in client-server DBMS architectures. Clifford, J. and King, R. (eds.), *In Proceedings of the ACM SIGMOD conference on Management of Data*, pp. 357–366.
31. Gerlhof, C., Kemper, A., Kilger, C., and Moerkotte, G. (1993) Partition-based clustering in object bases: From theory to practice. *In Proceedings of the International Conference on Foundations of Data Organisation and Algorithms (FODO)*, pp. 301–316.
32. Tsangaris, E.-M. M. (1992) *Principles of Static Clustering For Object Oriented Databases*. Ph.D. thesis, University of Wisconsin-Madison.

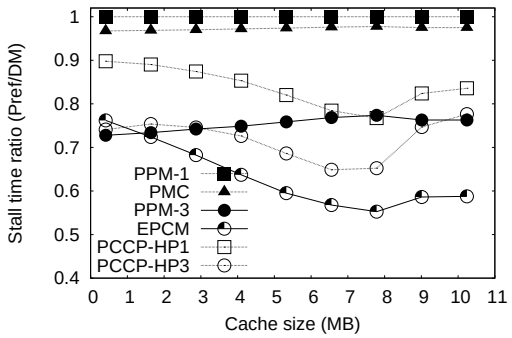
Appendix



(a) Using GGP clustering

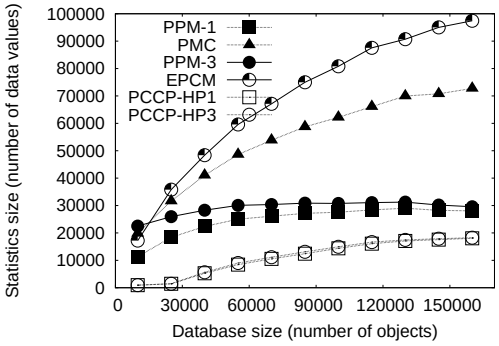


(b) Using WGGP clustering

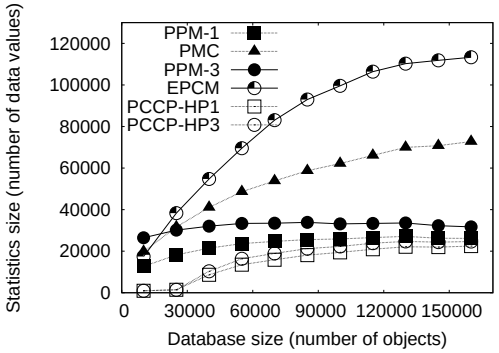


(c) Using no clustering

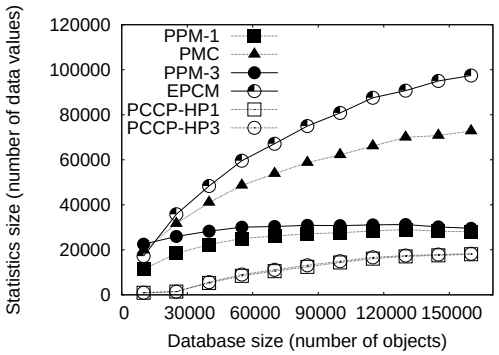
Fig. 10 Varying cache size results.



(a) Using GGP clustering

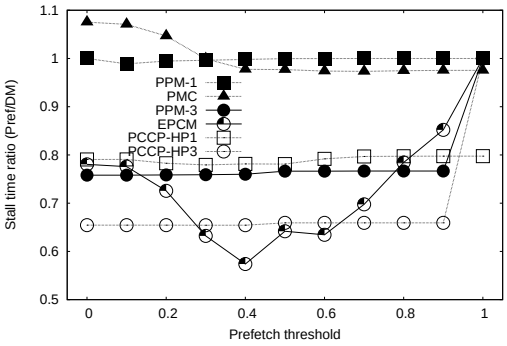


(b) Using WGGP clustering

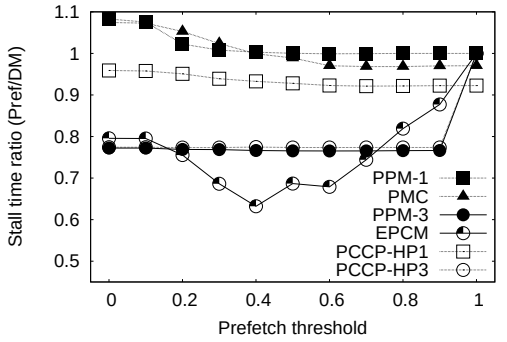


(c) Using no clustering

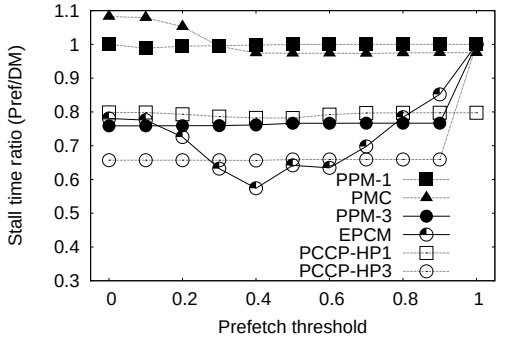
Fig. 11 Statistics storage size results.



(a) Using GGP clustering



(b) Using WGGP clustering



(c) Using no clustering

Fig. 12 Varying prefetch threshold results.