BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-93-M4

"Practical Prefetching via Data Compression"

by

Kenneth Marion Curewitz

Practical Prefetching via Data Compression[1]

Kenneth Marion Curewitz
Department of Computer Science
Brown University
Providence, Rhode Island 02912-1910

Submitted in partial fulfillment of the requirements for the
Degree of Master of Science in the Department of Computer Science
at Brown University

May 1993

[1]An extended abstract appears in *Proceedings of the ACM SIGMOD, Washington, D.C., 1993.*

This research project by Kenneth Curewitz is accepted in its present form by the Department of Computer Science at Brown University in partial fulfillment of the requirements for the Degree of Master of Science.

[illegible]

Professor Jeffrey Scott Vitter
Advisor

March 12, 1993

Date

Contents

1 Introduction

The basic operations performed by a computer system comprise processing, storing, and transferring data. Recent increases in processor speed have outpaced those in communication speed, making latency between the processor and the storage element a system bottleneck. A method of improving the overall response time of a computer system is to anticipate a request for data by the processor and *prefetch* data that is likely to be accessed by the application in the near future. This report describes three deterministic, adaptive algorithms based on theoretically optimal algorithms and presents results of applying these techniques to actual program access traces.

1.1 Motivation and goals

In most computer systems, memory is hierarchical in nature, with a fast memory, or *cache*, at the top and slow memory such as disk storage below it in the hierarchy. An application requires that the pages it accesses be in cache. In the event that it is in cache a *page hit* occurs; otherwise a *page fault* occurs and the page is fetched from slow memory to cache. The application has to wait until this fetch is completed. The time that it takes to complete this task is called the *I/O latency.* The method of fetching pages into cache only when a fault occurs is called *demand fetching.* The problem of *caching* is to decide which pages to remove from cache to accommodate the incoming pages.

In many OODB applications and hypertext systems, users spend a significant amount of time processing a page, and the computer and I/O system are essentially idle during that period. If the computer system can predict the page the user will access next, it can fetch that page into cache (if it is not already in cache) *before* the user asks for it. Thus, when the user actually asks for the page, it is available instantaneously, and the user perceives a faster response time. This method of anticipating and getting pages into cache in the background is called *prefetching.*

Current database systems perform prefetching using techniques derived from older virtual memory systems. The I/O bottleneck is a limiting factor for the performance of large-scale databases, and the demand for improving response time performance is growing [Bra]. This has stimulated renewed interest in developing improved algorithms for prefetching [ChB, Lai, MLG, PaZb, RoL]. Independent to our approach, there has been recent work by Palmer and Zdonik, who use a pattern matching approach for prediction [PaZb], by Salem, who computes various first-order statistics for prediction [Sal], and by Laird, who uses a growing order Markov predictor [Lai]. Prefetching in a parallel environment is studied in [KoE].

The idea of using data compression techniques for prefetching was advocated by Vitter and Krishnan [ViK]. The intuition is that data compressors typically operate by postulating (either implicitly or explicitly) a probability distribution on the data to be compressed. Data expected with high probability are encoded with few bits, and unexpected data with many bits. Thus, if a data compressor successfully compresses the data, then its probability distribution on the data must be realistic and can be used for effective prediction.

Assuming that as many pages as desired can be prefetched limited only by the cache size k (the *pure prefetching* assumption), it is shown in [KrV] [ViK] that any theoretically optimal character-by-character data compressor (for example, one obtained from the Lempel-Ziv compressor) can be converted to a prefetcher that has an optimal hit rate. In [ViK], one prefetcher is shown to be optimal in the limit for sequences of page accesses generated by a Markov source (where the page accesses correspond to the arcs not the states). The result was generalized in [KrV] to show that a modified prefetcher was optimal among finite-state prefetchers for arbitrary worst-case sequences of page accesses.

The pure prefetching assumption is not valid in many real-world applications. In this report, we consider non-pure prefetching in which only limited and varying time is allowed for the prefetching.

We consider three data compressors that perform well in practice, and we build simple, deterministic, universal[2] prefetchers based on them. We run our simulations on page access sequences derived from the DEC OO7 benchmark [CDN], the Object Operations (OO1) benchmark [CaS], and from CAD applications used at Digital Equipment Corporation. We find that the fault rate (the ratio of number of page faults to the number of page accesses) decreases significantly in relationship to a paging scheme using just the least-recently-used (LRU) heuristic. The reduction in fault rate is also better than that of recent proposed schemes for prefetching [PaZb].

1.2 History of the project

My introduction to this area of research began in December of 1991 at meeting with Professor Jeff Vitter to discuss working with him on a master's project. He suggested looking at applying algorithms for prefetching based on data compression that he and P. Krishnan, a Ph.D. candidate, had devised. My previous experience with prefetching came from an operating systems course where we learned that some operating systems perform sequential prefetch of pages to improve system performance. This method of prefetching relies on the fact that sequential access is fairly common. For some applications like reading a file for sequential processing, data at location $i+1$ will be accessed directly after data at location i, but, in general, this is not always the case. The more general scheme proposed by Vitter and Krishnan [ViK] seemed to be quite elegant and intuitively sound, especially for database systems with access patterns generated by user's requests.

[2]A universal prefetcher makes no assumptions about the application or data representation. Older virtual memory prefetchers that prefetch pages in sequence, that is, prefetch page $i+1$ when page i was being accessed, are not universal. The usefulness of universality is extremely significant in current databases [Sal]. Any specific knowledge about the sequence of page accesses can be utilized to improve the performance further using the techniques of [FKL].

To form a solid basis in this area of research, I began learning about lossless data compression algorithms, looking at database systems that might benefit from prefetching, and by collecting page fault traces from database systems and applications that use a database (such as CAD applications). I then developed a simulation environment to gain some experience with the algorithms and data structures involved. The results were encouraging and are presented in Section 7.

There are a number of data compression algorithms ranging from simple to complicated. Their performance for compression is generally related to their complexity. We decided first to implement a prefetcher based on the well known Lempel-Ziv dictionary scheme devised in the late 1970s. Results of prefetching using this algorithm were quite positive and suggested promise for prefetching in a real-world environment. The next algorithm we simulated was based on one of the best compression schemes, prediction by partial match (PPM), a context-based scheme. Just as in data compression, PPM gave better results than the Lempel-Ziv approach.

After simulating prefetching in an "ideal" environment, it was time to think about system issues like the amount of time needed to make a prediction, how many predictions should be made at any point in time, and how much space the prediction model occupied. Up to this point, we had made the assumption that we had all of the time and space that we needed and that we could prefetch as much data as would fit in cache. As a basis for comparison to a relatively simple and compact prefetcher, I implemented a model of the access pattern based on a first order Markov model with a fixed size, sliding window passing over the access sequence.

We obtained traces from a CAD application and two sets of database benchmark suites. The CAD traces were used by Mark Palmer for evaluation of his prefetcher and give an indication of the relative performance of his scheme compared to ours. Results of prefetching based on the three algorithms were good and we submitted a paper to the ACM SIGMOD '93 conference which will be presented in Washington, D.C. May 1993 and which will be included in the conference proceedings.

The final feature added to the simulator was the ability to limit the number of pages prefetched at any one time by using the toss of two biased coins to determine if there was time to prefetch, and if so, how many pages could be prefetched. By biasing the coins appropriately, we could model several light, moderate, and heavy workloads in a system environment and study its effects on our prefetching scheme.

1.3 Paper structure

In Section 2 we describe the system environment. In Section 3 we look more closely at problems stemming from memory and time restrictions unique to prefetching. We propose solutions to these problems and bound their worst-case behavior. We describe our three prefetchers in Section 4. Our method of keeping the prefetcher's model intact when there is not time to prefetch is described in Section 5. In Section 6 we present our simulation environment. In Section 7 we give a brief description of the page access traces and present our simulation results. We discuss related work in Section 8, and present our conclusions in Section 9.

2 System environment

2.1 The client-server model

The familiar *client-server* model of computing has become quite popular with the advent of relatively inexpensive desktop workstations. Practical aspects of managing data accessed by workstations are resolved by having a centralized server responsible for and capable of this task. When the common data is a database and the clients are running a database application, the database system is made up of a server that manages the database and clients that access and modify the database. Clients and servers pass messages and data back and forth. A client can use local memory to cache data that it has requested and send modified data or log entries back to the server. Both the client and server are distinct processes that can either coexist on one computer system or can exist on separate systems. Typically, in large database systems the server runs on a dedicated machine equipped with a large amount of secondary storage and network connections to a number of client machines in a *distributed* database system. The client-server architecture is shown in Figure 1.

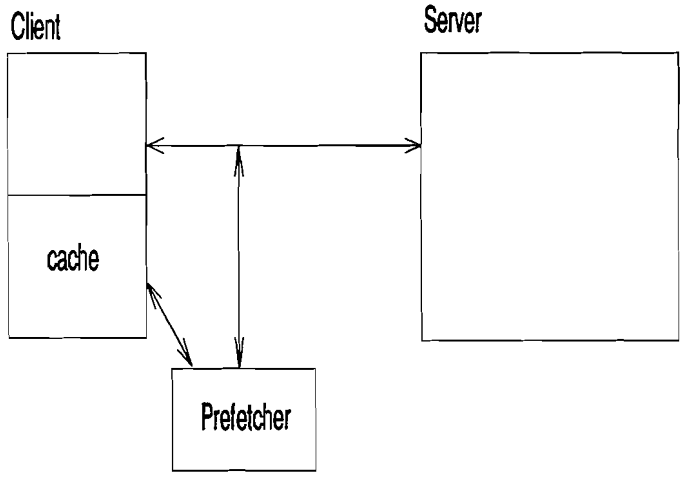

Figure 1: The client-server model

2.2 Slowdown associated with distribution

One limiting factor in the performance of a distributed database is the network *bandwidth*—the amount of data per unit time a network can handle [GrR]. Another important factor to consider is the *load* on the network—the number of messages and data being sent and received at any one time. Together, the bandwidth and load determine the *latency* associated with client-server communication. The benefits of distributed databases are easily understood. A drawback to the distributed approach is the relatively high cost of transferring data over a network connection. *Prefetching* reduces the effect of network latency by anticipating the client's future requests and making such requests when the network is idle.

2.3 Database support for prefetching

The server has the ability to handle *demand* read requests from the application and *prefetch* read requests from the prefetcher. The server gives priority to the application's requests, flushing prefetch requests in its queue when a demand request comes in. Such provisions are generally available in prefetching systems [GrR, PaZa].

The prefetcher can be either part of the application or an entity distinct from the application. It works by listening to the application's request sequence and making requests for data from the server.

Due to the diverse nature of a user's access patterns, the improvement in fault rate is best when each instance of an application (i.e., each user) on the client runs a copy of the prefetcher that takes into account only *its* access sequence.

3 System issues related to prefetching

Independent of the prefetching algorithm used to decide which pages to prefetch, a number of system issues must be considered when devising a prefetching *engine* for a database system. We consider those aspects closely related to the prefetcher and its associated data, and the management of the prefetched data itself. We describe our caching scheme in Section 3.1 and then look at some constraints placed on the prefetcher by the run-time environment.

The purpose of prefetching is to increase the system performance by submitting read requests to the server when it is otherwise idle, thus *spreading out* the I/O activity in the computer system. The most important property of a good prefetcher is that it *not* degrade the system performance. A prefetcher can degrade system performance by making incorrect predictions and requests that replace relevant data in cache with data that will not be accessed in the near future.

The prefetcher makes processor (time) and data (space) requirements of the database system. It can degrade performance of the application when it takes too much time to make predictions, preempting processing of data by the application. The space used by the prefetcher effectively reduces the amount of cache space available to the database application. We would expect the prefetcher to make better use of system resources than the application would if these resources were dedicated to the application as cache space. We address a way to reduce the space requirements of a prefetcher in Section 3.3.1 and a way to model the application's temporal nature in Section 3.3.2.

Higher level considerations, such as the ACID[3] properties, logging, recovery, replication, authentication, authorization, and naming [GrR] are essential elements of a distributed database system and are assumed to be managed by the database system.

3.1 Caching issues

Caching issues automatically arise when we prefetch less data than fits in the cache; we need to decide which pages to evict from cache to make space for incoming pages. Any caching scheme is considered independently of the prefetching algorithm employed by the system. We can use the probabilities of the prefetcher to determine those pages to evict from cache, or adapt strategies like the MLP replacement strategy from [PaZb], or adapt well-known cachers like FIFO or LRU. In our simulations, we use a version of LRU suitably modified to handle prefetched pages. An important motivation for this scheme is simplicity.

The scheme we use works as follows: Prefetched items are put into cache as if they were demand-fetched. They are marked as most recently used items, with more probable pages marked as more recently used. Prefetched data replace the least

[3]Atomicity, Consistency, Isolation, and Durability

recently used pages which, if modified, are written back to disk by the database management system.

A number of additional issues related to caching arise in distributed database design and implementation. These include data sharing, replication, and transparency. While these are important to maintaining the integrity of the database they are the subject of current research and implemented in existing database systems. We are aware, but do not address these issues in this report.

3.2 Accounting for the space used by the prefetcher

In order to form a fair basis of comparison between a system with prefetching and a system without prefetching, we will run our system without prefetching with a larger data cache. The added space is made available by the absence of the prefetch data structure and is used to cache database pages.

3.3 Prefetching in a restricted memory environment

For some applications and system environments, it is reasonable to assume that resources required by the prefetcher do not exceed the capacity of the computer system, but we cannot expect all systems to exhibit such nice characteristics.

Similar problems arise in data compression, and several techniques are known for limiting data structure size in data compressors [Sto]. An explicit upper bound M is placed on the size of the data structure. The data structure is either frozen when its size reaches M, flushed and rebuilt when its size reaches M, or frozen when its size reaches $M/2$ (and a new data structure is built while the old one is used for prefetching). There are also more sophisticated techniques that use an LRU strategy on the data structure to maintain its size [BuB]. Our ongoing work studies these techniques in the prefetching context. However, explicit bounds on the data structure size degrade performance as is evidenced by a look at the results from Section 7 and comparing order 1 PPM with FOM.

Motivated by the above observation, we present the following new scheme to prefetch in a restricted memory environment.

3.3.1 Paging the data structure

The data structures used by our prefetchers are modified trees (see Figures 2 and 3). Each node of the tree maintains information about its children (their counts, addresses, etc.). This information is required to make predictions for the next access. It is reasonable to assume that every node of the tree (except maybe the root) fits in at most one page of memory. (This can be ensured by simple schemes.)

We maintain some nodes of the tree in cache using one of many heuristics (like LRU) to decide what to evict from cache. In particular, the root is always maintained in cache. We *page in a node of the tree* when it is required. This scheme works smoothly if each node is given its own page and at least two extra I/Os can be performed between two accesses (to write out the evicted node and read in the desired node).

It is more space-efficient to compact "small" nodes together into one page. In such cases, nodes may have to be moved when they threaten to overflow a page. For a pure tree data structure as in LZ (Figure 2), it can be verified that nodes can be reallocated to "less crowded" pages in a lazy fashion using one extra I/O for the movement, and no subsequent extra I/Os. In PPM, the node of the data structure can have many (vine) pointers into it. In this case, when a node moves, it leaves a "forwarding address", and when a vine pointer is traversed, this forwarding address pointer is "short-circuited", that is the pointer is updated in the predecessor so that the next time this path is traversed, the old node will have been eliminated from the path. In the worst case there may be one extra I/O per vine pointer per reallocation (although in practice our simulators show few reallocations and few short-circuitings of pointers).

3.3.2 Sequence of fast accesses

The scheme explained in Section 3.3.1 solves the limited memory problem by using disk space efficiently but creates a new "timing" problem of *fast accesses* (some number of page requests that occur so close to one another that they allow no data structure or prefetching I/O to be performed between them). When the data structure is always in cache, it can be updated every time even when there is no time to prefetch between page requests. If the data structure is paged, a sequence of fast accesses can force us to disregard important sequence information.

We propose the following strategy to cope with the problem of updating the data structure during fast accesses: In both LZ and PPM, the pages in the fast access sequence are used to increment the counts/probabilities *of the current node* (that is, the node used for prediction just before the fast access started). The intuition behind this scheme is that if the sequence of fast accesses is context-dependent then accumulating statistics at the current node will ensure the prefetching of the correct pages in the future before the start of the fast access subsequence. This is explored further in Section 5.

4 Algorithms for Prefetching

Let α be the alphabet size (total number of pages in the database) and k be the cache size in pages. In typical databases, α is large and $k \ll \alpha$.

In this section, we describe our three simple, deterministic prefetching algorithms based on practical data compressors. (An elegant discussion of the data compressors appears in [BCW].) We describe our prefetchers in Sections 4.1– 4.3 in their "generic" form: as prefetchers that can store their entire data structure in cache. These prefetchers make k suggestions for prefetch ordered by their relative merit. To make these suggestions the algorithms use $O(k)$ time. Sometimes the algorithms may have information to make $k_1 < k$ suggestions. In such cases, the remaining $k - k_1$ locations of cache are left undisturbed.

In Section 3.1 we look at the modification to the "generic" algorithm when we can prefetch less than k pages at each time instant. This occurs because the time between accesses is small (or as mentioned earlier, the prefetcher makes only $k_1 < k$ educated choices). This automatically introduces the problem of caching; our decision strategy on what is evicted from cache becomes important. It is implicit in our discussion that the page the application is working on is left undisturbed; hence the actual number of pages in cache is $k + 1$. Other changes to the "generic" algorithms in situations that arise in practice (for example, when the data structure cannot be stored entirely in cache) are discussed in Section 3.

4.1 Algorithm LZ

We denote the empty string by λ. Algorithm LZ of [ViK] is based on the character-based version of the Lempel-Ziv algorithm for data compression. The original Lempel-Ziv algorithm [ZiL] is a word-based data compression algorithm. The Lempel-Ziv encoder breaks the input string into blocks of relatively large length n, and it encodes these blocks using a block-to-variable code in the following way: It parses each block of size n into distinct substrings $x_0 = \lambda$, x_1, x_2, $\ldots$, x_c such that for all $j \geq 1$ substring x_j without its last character is equal to some x_i, for $0 \leq i < j$. It encodes the substring x_j by the value i, using $\lceil \lg j \rceil$ bits, followed by the ascii encoding of the last character of x_j, using $\lceil \lg \alpha \rceil$ bits.

The equivalent character-based algorithm builds in an online fashion a probabilistic model that feeds probability information to an arithmetic coder [HoV, Lan, WNC]. (The exact compression method is irrelevant for our current discussion and is omitted.) We show by an example how the probabilistic model is built.

Example 1 Assume for simplicity that our alphabet is $\{a, b\}$. We consider the page access sequence "*aaaababaabbbabaa*...". The Lempel-Ziv encoder parses this string as "$(a)(aa)(ab)(aba)(abb)(b)(abaa)\ldots$".

In the character-based version $\mathcal{E}$ of the Lempel-Ziv encoder, a probabilistic model (or parse tree) is built for each substring when the previous substring ends. The parse tree at the start of the seventh substring is pictured in Figure 2. There are five previous substrings beginning with an "a" and one beginning with a "b." The page "a" is therefore assigned a probability of 5/6 at the root, and "b" is assigned a probability of 1/6 at the root. Similarly, of the 5 substrings that begin with an "a," one begins with an "aa" and three begin with an "ab," accounting for the probabilities of 1/5 for "a" and 3/5 for "b" at node x, and so on. □

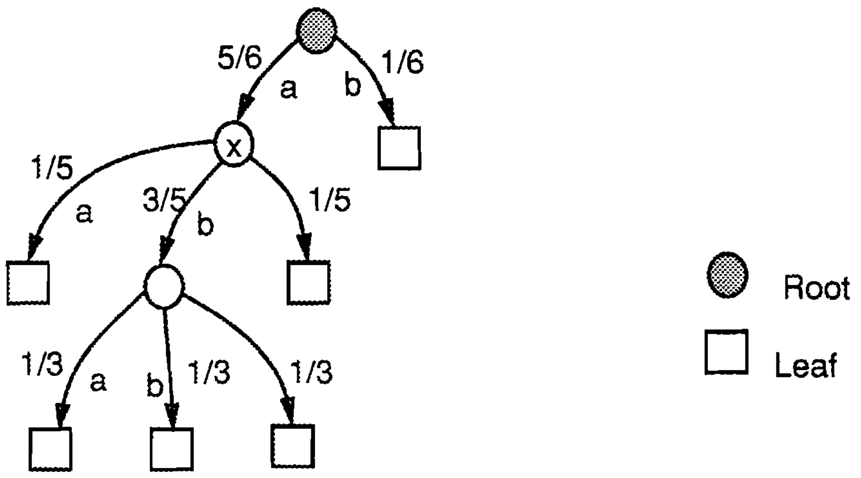

Figure 2: The parse tree constructed by the character-based encoder $\mathcal{E}$ for Example 1. Notice that since the substrings are prefix-closed, they can be represented by a tree in a natural way.

Our prefetcher LZ uses the probabilistic model built by the encoder $\mathcal{E}$ as follows: At the start of each substring, LZ's current node is set to be the root of $\mathcal{E}$'s parse tree. (See Figure 2.) Before each page access, LZ prefetches the pages with the top k estimated probabilities as specified by the transitions out of its current node. On seeing the actual page requested, LZ resets its current node by walking down the transition labeled by that page and gets ready to prefetch again. In addition, if the page is not in memory, a page fault is generated. When LZ reaches a leaf, it fetches in k pages at random. The next page request ends the substring, and LZ resets its current node to be the root. Updating the model can be done dynamically while LZ traverses it. At the end of n page accesses, for some appropriately large n, LZ throws away its model and starts afresh.

In our simulations, we use a heuristic in LZ (that parallels the Welsh implementation [BCW] of the Lempel-Ziv data compressor). While LZ is at a leaf ℓ, instead of fetching in k pages at random, it resets its current node to be the root (that is, it goes to the root one step early). However, it updates the node ℓ *and* the root based on the next access.

The data structure used for prediction is a tree with at most one pointer into each node.

4.2 Algorithm PPM

Although the LZ prefetcher is optimal in the limit against the class of finite state prefetchers [KrV, ViK], convergence to optimality is slow. This motivates us to adapt the prediction by partial match data compressor (that performs better in practice for compression of text than the Lempel-Ziv algorithm) for prefetching.

A *jth-order Markov predictor* on page access sequence σ uses statistics of contexts of length j from the sequence to make its predictions for the next character.

Example 2 Let $j = 2$, and let the page access sequence σ be "*abbababab*." The next character is predicted based on the current context, that is on the last $j = 2$ characters "*ab*" of σ. In σ, an "*a*" follows an "*ab*" twice, and a "*b*" follows an "*ab*" once. Hence "*a*" is predicted with a probability of 2/3, and "*b*" is predicted with a probability of 1/3. If $j = 0$ (no context) , each character is predicted based on the relative number of times it appears in the access sequence. □

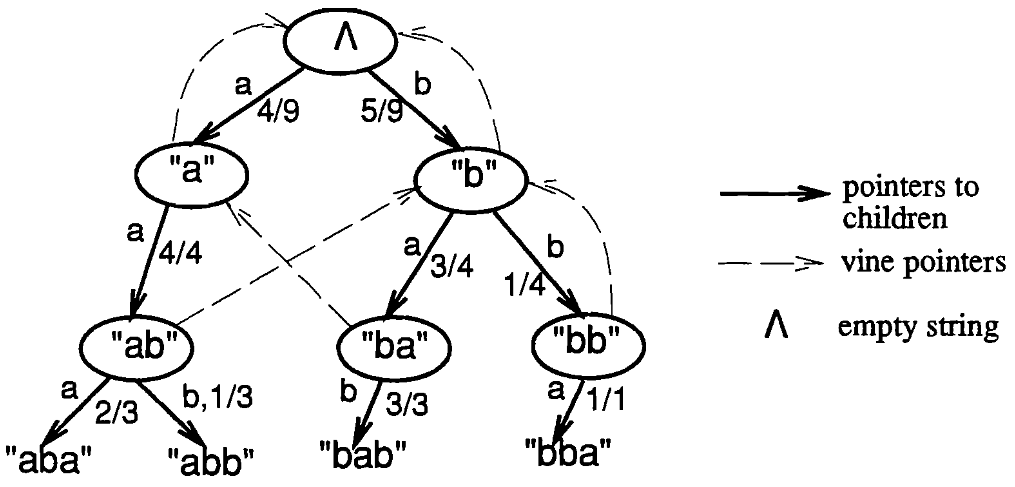

Figure 3: Snapshot of the data structure for PPM of order 2 when the access sequence is "*abbababab*." The vine pointer from node "*ab*" to node "*b*" indicates that if the current 2-character context is "*ab*," then the current one-character context is "*b*."

The PPM prefetcher has an *order* parameter m associated with it. A PPM prefetcher of order m maintains jth-order Markov predictors (on the page access sequence seen till now) for all $0 \leq j \leq m$. Let n_j be the number of pages predicted by the jth-order Markov predictor[4] (for the current context) that is not predicted by

[4]n_j can be less than k if, for example, the current context occurs rarely in the past.

any rth-order Markov predictor, $r > j$. Intuitively, PPM prefetches the k pages with the maximum k probabilities giving preference to pages predicted by higher order contexts. Formally, it executes the following loop:

```
for $j = m$ downto 0 do
begin
  $t_j := \min(n_j, k - n_m - n_{m-1} - \cdots - n_{j+1})$;
  $t_j := \max(t_j, 0)$;
  prefetch the pages with the maximum $t_j$ probabilities
    as given by the $j$th-order Markov predictor;
end
```

The above technique is based on the prediction by partial match algorithm for data compression using exclusion [BCW].

The various jth-order Markov predictors, $j = 0, 1, \ldots, m$, can be represented and updated simultaneously in an efficient manner using a forward tree with vine pointers [BCW]. An example of a forward tree with vine pointers is given in Figure 3. The data structure is "almost" a tree; there can be more than one edge into a node because of vine pointers. In our simulations we use PPM of order 3 and order 1.

4.3 Algorithm FOM

Algorithm FOM is a limited memory prefetcher designed so it can always fit in a small cache. It takes as parameter a quantity w, the window size. Algorithm FOM with window size w maintains a 1st-order Markov predictor on the page access sequence formed by the last w page accesses. (The 1st-order Markov predictor is explained in Section 4.2.) It prefetches the k pages with the maximum k probabilities as given by this 1st-order Markov predictor. We use $w = 1,000$ in our experiments reported in Section 7.2. We would expect FOM with $w = \infty$ to be "close to" PPM of order 1 in performance. (Note that unlike FOM, algorithm PPM of order 1 uses an additional order 0 context for prediction.)

4.4 Adaptive prefetching

Our prefetcher is *adaptive* meaning that after each demand request made by the application, the prefetcher updates its model to reflect the new state of the access sequence. Adaptive techniques are commonly used in practice for data compression [BCW]. Our prefetcher *continuously* adapts its model of the access sequence and predicts whenever possible.

4.5 Cold and warm startup

For simplicity, our implementation begins each simulation in *cold* startup mode, that is, the data structure for prefetching is empty. A *warm* startup should be considered for applications that would benefit by using information from previous runs to generate predictions for the current run. A warm startup would only require saving the prefetch data structure at the end of a database session and restoring it at the beginning of the next session. The data structures used by the algorithms described in this section can easily be saved and restored with little processing overhead.

4.6 Lazy addition of leaves in the tree

The sizes of the data structures for LZ and PPM tend to grow as the application runs. In Section 3.3.1 we discussed paging the prefetch data structure. In addition to employing this technique for managing a fairly large data structure, we use *lazy addition* of nodes in the tree to reduce the amount of space used by the prefetch data structure. Each internal node of the tree contains a pointer and reference count for each of its children. Until a leaf becomes an internal node, it contains no useful information and, as such, it is not allocated any space until a subsequent access through which it becomes an internal node. It is at this point we allocate space for it in a lazy fashion.

4.7 Prefetching Heuristics

Our prefetching algorithms are deterministic and make predictions based only upon past history. In some cases, it is desirable to stray from the predictions that the unmodified algorithm makes and make others based upon specific knowledge of the application program or simple observations about the access sequence or the prediction accuracy.

4.7.1 Sequential Access

It is possible that an access sequence exhibits a simple access pattern such as purely sequential or a slight variation that walks through the database with a "stride" of i. If this sequence does not repeat itself, the algorithms in this section will recognize the pattern and make correct predictions. In an application that is likely to exhibit sequential access patterns, it would be fairly simple to add a heuristic approach that recognizes the pattern and replaces the least likely page predicted by the prefetcher with the next page in the sequence. With such a heuristic and a sequential access pattern, the application will enjoy a decrease in fault rate.

4.7.2 Random access sequences

Unfortunately, our algorithms are not a panacea for prefetching in an arbitrary environment. It is possible to devise a nemesis sequence that causes the prefetcher to degrade the system's performance by having it prefetch pages that *will not* be used in the near future and having it evict pages that *will* be used in the near future. An approach to limit the degradation of such a sequence would be to monitor the number of hits associated with prefetched pages and to reduce or completely stop prefetching when this hit rate goes below some threshold.

4.8 Scalability

Vitter and Krishnan report that their algorithms are theoretically optimal in the limit for almost all sequences of page accesses against the class of finite state prefetchers [KrV, ViK]. The consequence of this property is that we expect the algorithms to perform better on longer access sequences such as we would find in an online environment. Given that the algorithm's performance improves as it becomes more experienced with an access pattern, we have to account for the increased size of the data structure and time to make predictions. Our paging scheme described in Section 3.3.1 proposes a way to manage a large prefetching model with a minimum of overhead.

5 Handling fast accesses in the algorithms

We described the notion of fast accesses in Section 3.3.2. A fast access occurs at the point when two or more pages are requested by the application without leaving time to perform prefetching. The implication for the action of the prefetcher is clear; no prefetching is done in order to stay out of the way of the application making demand requests for data. What happens to the model is not so obvious. Since the FOM model fits completely in memory, we must only worry about how the model is updated. On the other hand, if the relevant nodes of the data structures for LZ and PPM are not in cache, our model could become out of sync with the actual access pattern and degrade performance by making bad predictions.

When there is enough time to prefetch, the model is updated and the node corresponding to the demand requested page is made *current* (or nodes, in the case of PPM). The approach of Section 3.3.2 proposes making updates to that part of the model resident in cache, effectively modifying the data structure without moving to a new *current* node. The effect of not moving to the new node is that it removes the need for paging in a node of the prefetch data structure and takes the temporal property of the access sequence into account. It is our expectation that an application will display a consistent relationship between a set of pages accessed and the time used to process those pages. If that is the case, then our scheme should work especially well. This conclusion is further supported by the good prefetching performance of our scheme under an independent scenario, in which the locations and durations of the fast accesses are independent of the page accesses.

Each node of the LZ tree contains all of the information necessary to make predictions and updates based upon demand requests. The significance of keeping this information in the current node is seen in the case of fast accesses. At any point during processing, the "current node" of the prefetch data structure is by definition in cache; it is only when we move forward in the model that we may have to cause a page fault to bring in a new node.

Example 3 Consider the subsequence "$\ldots abba \ldots$" of an access sequence. In one extreme, the three requests following the first a can all be *normal* accesses that allow time to prefetch and update the prefetch data structure. At the other extreme, the requests could all come at once, leaving time neither to prefetch nor to page nodes in the data structure. We will show what happens in both of these cases by showing the state of the prefetch data structure before and after processing the access sequence as all *fast* or all *normal* accesses. Any combination of slow or fast accesses is accommodated by making the appropriate updates to the data structure and current node in a straightforward manner.

Assume that we are at the current node in the data structure after the page access a is processed. For clarity, we will only show the relevant nodes in the subtree of the data structure for LZ. The starting state for our data structure is shown in Figure 4a. The arc between nodes is labeled with the page identifier and reference count.

The model of the normal access pattern updates the reference count for page b as a child of the current node and we move to the appropriate new current node. Then we update the count for b as the child of the node b and move to the appropriate current node. Finally, update the count for a and move to the new current node. The updated data structure is shown in Figure 4b.

The model of the fast access pattern behaves in a completely different manner. The current node never changes during a sequence of fast accesses. The reference counts for a and b are incremented in the same current node. The current node, by definition, is always accessible to the prefetcher in cache. Since a node fits on a page, no page faults are required to update the data structure and our model is in step with the access sequence. The updated data structure is shown in Figure 4c. $\square$

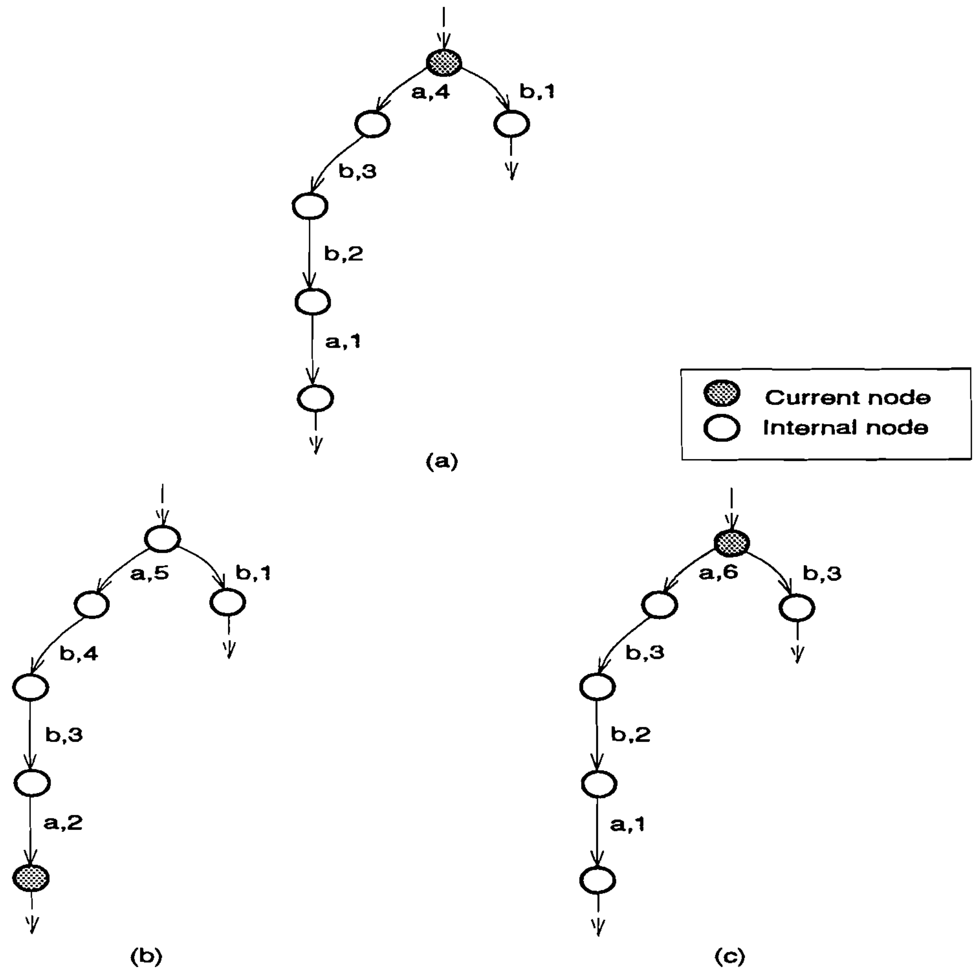

Figure 4: Effect of updating the data structure for LZ for the subsequence "...*abba*...". Figure (a) shows a subset of the inital data structure. Figure (b) shows the same data structure updated after a series of *normal* accesses. Figure (c) shows the data structure updated after a series of *fast* accesses. The arc between nodes is labeled with the page identifier and reference count.

6 Simulation environment

In this section we describe the simulation environment we developed to evaluate each of our prefetching algorithms. We first look at the assumptions we make for simulation and then describe the method used for simulations.

6.1 Modeling run-time constraints

6.1.1 Paging the data structure

Associated with each node of the data structure is a logical page number used for caching the nodes of the tree. We page the data structure just as we page the actual database, evicting (and writing out) the least-recently-used page and replacing it with the page containing the node needed by the prefetcher.

6.1.2 Sequence of fast accesses

Fast access sequences preclude any paging activity by the prefetcher. While we cannot prefetch any pages or page the prefetch data structure, we do update the prefetcher's model as described in Section 5.

In order to simulate with fast accesses, we need either traces with detailed timing information and system timing information or, alternatively, a probabilistic approach to decide when and if prefetching can occur, and if it can occur, how much data can be prefetched. The former approach is difficult to obtain with reliable timing information for purposes of prefetching. The latter approach is far simpler and more widely applicable and was our method of choice, although it removes the relationship between the previous context and the occurrence of fast accesses we expect in practice. As a result, the simulations can be expected to offer a conservative estimate of prefetching performance.

6.2 Simplifying assumptions

We bound the complexity of the simulator with the following assumptions about the application being analyzed [5] :

- Data items can either be pages or objects and are generically referred to as pages for purposes of discussion.
- Pages have a name, address, and a location. We make the assumption that this information remains fixed during a run of prefetcher. This restriction obviates the requirement to resolve an objects location with the use of a name server.
- Pages are assumed to be of a fixed size to allow uniform fetching.

[5] Similar assumptions are common and have been made by others doing related work [PaZb].

- Our simulator uses a parameter d, that determines the maximum number of pages to prefetch at each step. When the prefetcher makes $d_1 < d$ predictions, up to d_1 pages are prefetched. We refer to simulation results using this constraint as *uniform* prefetching results.

- In order to experiment with our methods of handling fast accesses, we implement a probablistic model to determine the maximum number of I/O transactions, r, that can occur between any two demand fetches. One *step* of our algorithm comprises prefetching up to r pages and fetching (if necessary) the page requested by the application and updating the requested page's LRU rank in the cache. Two probabilities[6] p and q, $(0 \leq p, q \leq 1)$ which a user specifies, simulate a workload in the computer system on which the application is running. Setting p and q to a real number close to zero simulates a heavily loaded system while setting p and q to a real number close to one simulates a lightly loaded system. The load on the system is inversely proportional to the amount of prefetching that can occur in the system.

  The first probability, p, determines the likelihood of a "head" for the toss of one coin.[7] The second probability, q, gives the likelihood of a "head" for the toss of the other coin. The first coin is flipped, and with probability p, at least one page is prefetched at this time instant; the second coin is then flipped repeatedly until a "tail" occurs. The number of "heads" determines how many additional pages can be prefetched up to the limit d. Given that the first coin is a "head," the expected number of pages to be prefetched is $1/(1-q)$, assuming we can prefetch any number of pages. Since we can only prefetch a number of pages equal to the cache size k minus 1 (assuming that the application is working with one page of cache), we can express the expected number of pages prefetched, given that the first coin is a "head" as

$$\sum_{1 \leq t \leq k-1} \min\{t, k-1\} q^{t-1}(1-q)$$

6.3 Simulation method

Three simulators are used to perform prefetching. In each simulator, we have the ability to perform uniform prefetching or prefetching with fast accesses.

The prefetcher's model contains information necessary for monitoring its paging behavior. For both LZ and PPM, the root is always kept in cache and the rest of the structure can be paged in and out on demand. In the FOM algorithm, the whole data structure is assumed to fit in memory and is therefore not paged. It is instructive to know the amount of paging activity needed by the prefetch data structure for

[6]We use the C library pseudo-random number generator, "drand48()", to realize our probabilistic model.

[7]We know that when the probability of a head is not 50% then the coin is said to be *biased*.

processing prefetch requests and updating the model to get an idea of the overhead associated with the prefetch engine. We assume that the prefetch data structure can be paged to local disk as opposed to a disk on the server to reduce the paging cost but we realize that paging may reduce the benefit of prefetching. Our results show us that the number of page faults incurred by the prefetcher for its data structure is very small compared to the size of the access trace.

Common to the simulators are the cache routines that implement our modified LRU cache described in Section 3.1. Specific to each simulator is the code that implements the prefetcher and its associated model (or data structure). From least to most complex, the predictors rank: FOM, LZ, and PPM. Not surprisingly, their performance corresponds directly to their complexity.

The simulator measures the cache fault rate for an access sequence using each of the prediction algorithms and for any value of d from 0 up to k. The number d_1 is computed on-the-fly by using the two probabilities specified for a given run of the simulator and the random number generator provided by the system. Statistics about the number of faults, the size of the prefetch data structure, and the average number of predictions at each step are reported. Such information will be useful in designing an efficient online prefetcher.

For each page access sequence σ, we simulate each of our prefetchers from Section 4 on σ, prefetching d_1, $0 \leq d_1 \leq d$ pages at each step.

6.3.1 Uniform prefetching

The simulator reads a sequence of numbers. A number specifies the page (or object) identifier requested by the application. The prefetcher's job is to make predictions based upon previous page requests using one of the algorithms described in Section 4. The simulator requires a parameter that specifies the maximum number of prefetches that occur between any two demand fetches.

The number of pages prefetched, d_1, is the larger of d and the number of predictions available in the model.

The prefetcher executes the following loop:

```
for each time step do
begin
  prefetch up to $d$ most probable pages
  get the actual page requested by the user into cache (if not in cache)
  update the LRU rank of the actual page requested by the user
  update and advance the data structure to reflect the actual request
  update the prefetcher's statistics
end
```

6.3.2 Prefetching with fast accesses

The simulator reads a sequence of pairs of numbers. The first number is the page (or object) identifier requested by the application and the second is the number of pages we can prefetch before fulfilling this demand request. The prefetcher's job is to make predictions based upon previous page requests using one of the algorithms described in Section 4. If the number of pages allowed to be prefetched at any point is zero, then there are no pages prefetched and the model is updated based upon the demand request. The model is *not* advanced, that is, the current substring (in the case of Lempel-Ziv, or context for PPM) is kept current. By not advancing our model, we distinguish between normal and fast accesses.

The number of pages prefetched, d_1, is determined by flipping two biased coins as described above and by the prefetch data structure. From Section 3.1 it follows that when $d = 0$ the prefetcher works as an LRU cache. This provides a basis for comparison against our prefetcher.

The prefetcher executes the following loop:

```
for each time step do
begin
  prefetch up to $r$ most probable pages
  get the actual page requested by the user into cache (if not in cache)
  update the LRU rank of the actual page requested by the user
  update the data structure to reflect the actual request
  advance the data structure's current pointer if $r > 0$
  update the prefetcher's statistics
end
```

7 Experimental results

This section presents the results of simulating our prefetcher on access traces generated by a CAD application, the Object Operations Benchmark (OO1), and the DEC OO7 benchmark written at the University of Wisconsin [CDN]. We first describe the access traces and then present our results.

7.1 Description of the traces

The following characterizes the traces that were used to test our prefetching algorithms[8]:

CAD1, CAD2: These are object ID (UID) traces from a CAD tool written at Digital's CAD/CAM Technology Center in Chelmsford MA. We include them here as a comparison to the Fido [PaZb] algorithm that analyzed prefetching on the same traces.

The references represent UIDs requested by the application using tool functions: invocation, zoom in and out, select ICs, and setting filters that remove certain parts of the board display (e.g. runs and junctions). The circuit design data contained 100,000 objects, but only 10,000 or so could fit in the "usable window" at once. The first trace, CAD1, has 73,767 accesses and the second, CAD2, has 147,345 accesses.

Database benchmarks: The OO1 database benchmark, also known as the "Sun Benchmark", was run on the DEC Object/DB[9] product to generate page fault information for all phases of the benchmark. The more interesting phases include traversal of the structure in both the forward and reverse directions. The OO1 benchmark tests aspects of a DBMS that are critical in computer-aided software engineering (CASE) and computer-aided design (CAD) applications [CaS] engineering applications.

The DEC OO7 benchmark, developed at the University of Wisconsin [CDN], tests critical aspects of object-oriented database systems not covered by other benchmarks. This suite of tests was also run on the DEC Object/DB product used for the OO1 tests. This benchmark includes tests and reports the performance of an object oriented database in the following key areas:

1. Pointer traversal.
2. Application-DBMS coupling.
3. Complex object support and long data items.
4. Updates and Recovery.

[8]The traces were provided as part of the DEC-ERP grant 1139.

[9]DEC Object/DB is a trademark of Digital Equipment Corporation, Maynard MA.

5. Path indexing.
6. Caching and clustering.
7. Queries and optimization.
8. Concurrency control.
9. Relationships and versioning.

The benchmark performs *traversals*, *associative queries*, *insert/delete operations*, and *multiuser tests* [CDN]. We tested our prefetcher running with traces from the traversal and associative query portions of the benchmark.

7.2 Prefetch results for uniform prefetching

For every access sequence, we simulate for a fixed cache size k each of our algorithms and represent the results in easy-to-read graphical form. The y-axis denotes the fault rate[10] and the x-axis denotes the parameter d (the number of pages prefetched at each time step) that varies from 0 to k. When $d = 0$, the fault rate generated is exactly the fault rate of an LRU cache and is a basis for comparison with our prefetcher. Representative graphs of the results are shown in Figures 5, 6, 7, 8, 9, and 10.

7.3 Prefetch results with fast accesses

This section contains the results of prefetching using our probabilistic model for determining the amount of I/O activity between any two demand fetches by the database system (as described in Section 5).

Multiple simulation runs, using different seeds in the random number generator, produced little variation in the results. We present our results of running Algorithm PPM order 3 on the traces CAD1 and OO7_T1 in Figures 11 and 12. The cache size used is 10 pages, the probability, p, is fixed for each graph and displayed in the title. The value of the probability q ranges from 0.0 to 1.0 on the x-axis and the fault rate is shown on the y-axis . Two curves are shown, "Prefetch" for PPM order 3, and "LRU" for caching without prefetching.

The results suggest that the load on the system is inversely proportional to the improvement gained by prefetching and that, even under heavy load, a system with prefetching outperforms one without. Our results confirm the validity of our methods for modeling fast accesses in the algorithms.

7.4 Prefetch overhead statistics

We present statistics of the data structure size and the total number of page faults incurred during the simulation run.

[10]The fault rate f, is ratio of page faults to the total number of pages accessed in the trace, $0 \leq f \leq 100$.

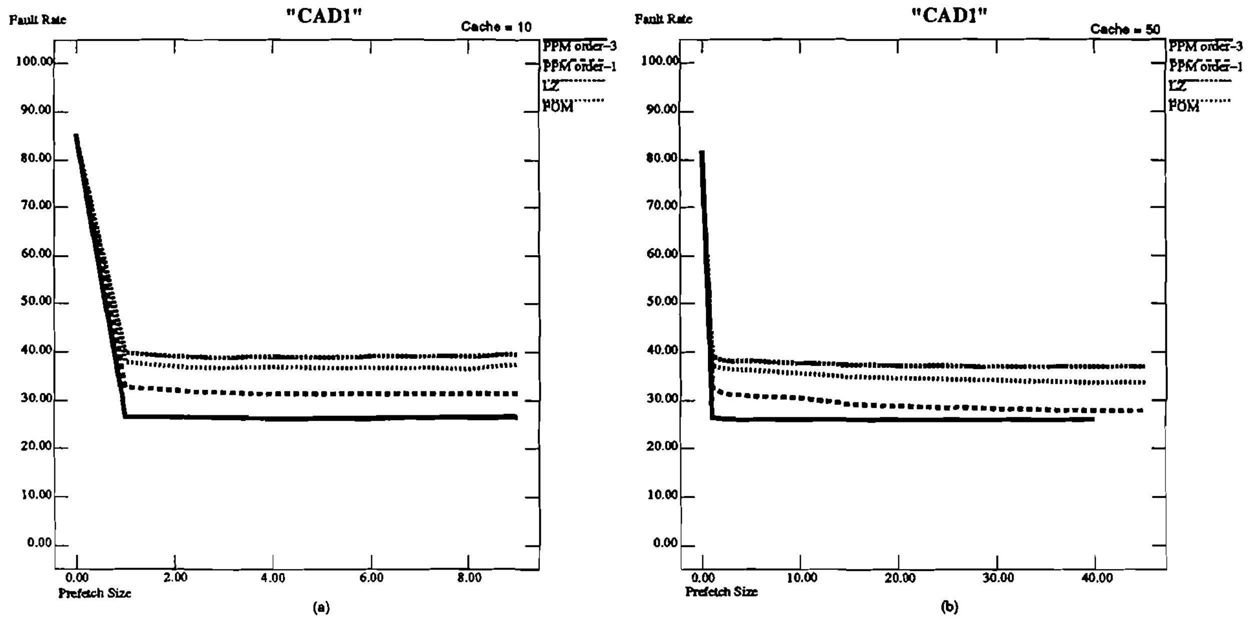

Figure 5: The fault rate for prefetching d objects ($0 \leq d \leq k$) for a fixed cache size. (a) $k = 10$. (b) $k = 50$ for the trace CAD1. There are 73768 object references and 15430 distinct objects in trace CAD1.

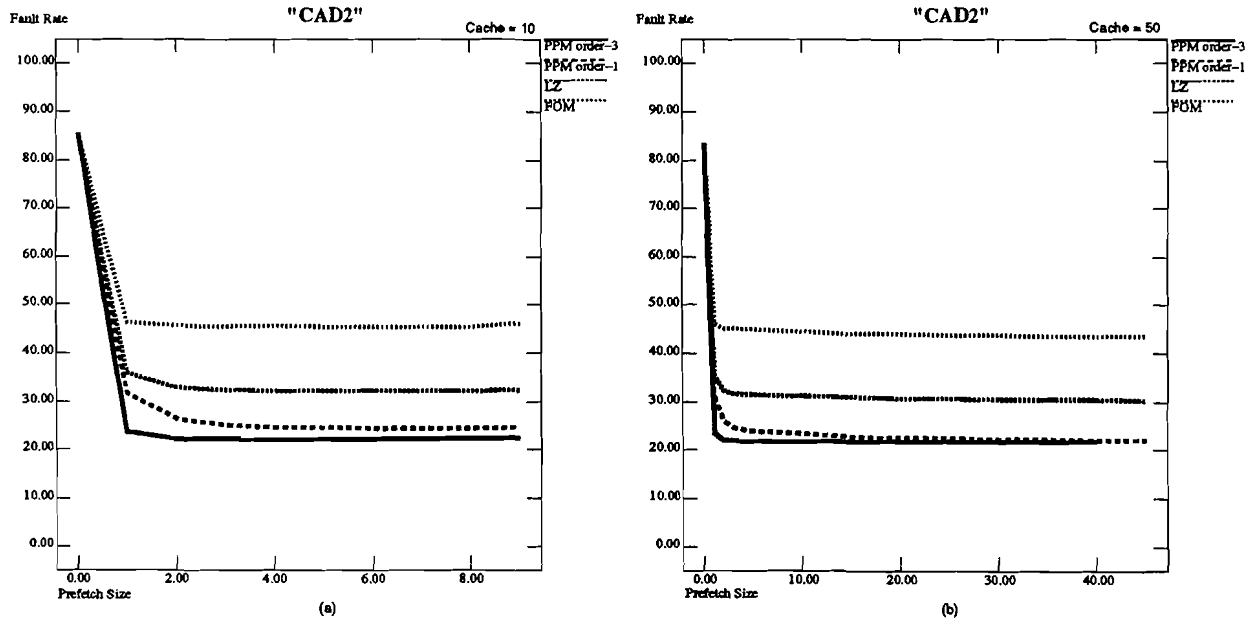

Figure 6: The fault rate for prefetching d objects ($0 \leq d \leq k$) for a fixed cache size. (a) $k = 10$. (b) $k = 50$ for the trace CAD2. There are 147344 object references and 15430 distinct objects in trace CAD2.

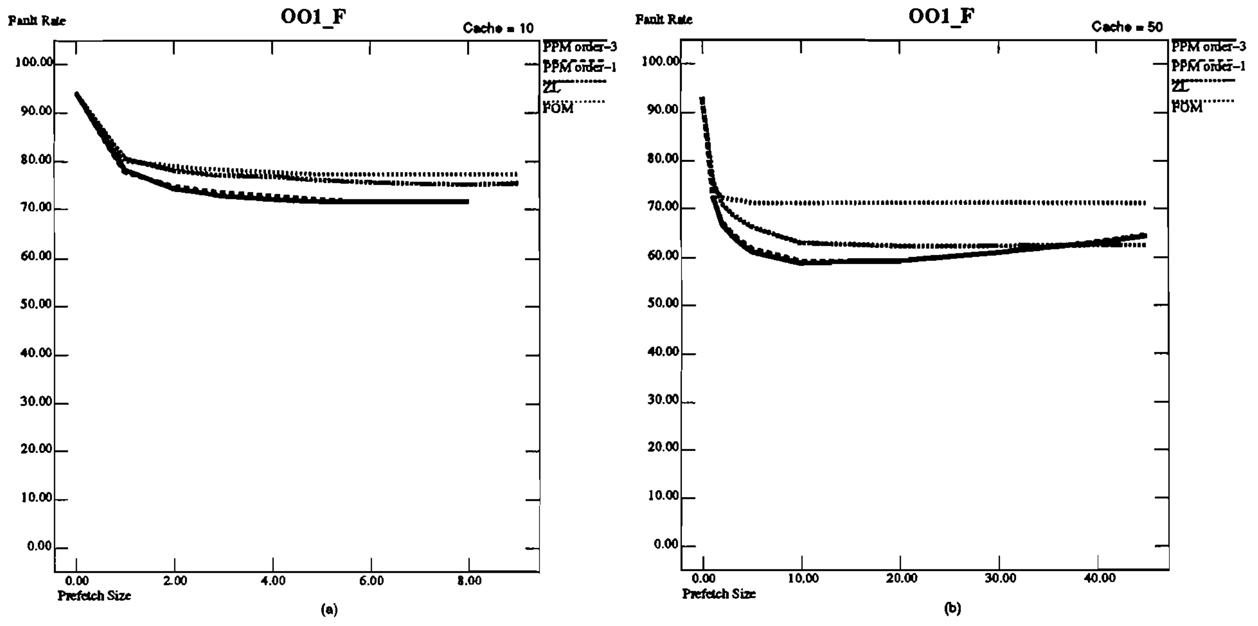

Figure 7: The fault rate for prefetching d objects ($0 \leq d \leq k$) for a fixed cache size. (a) $k = 10$. (b) $k = 50$ for the trace OO1_F. There are 11719 page references and 526 distinct pages in trace OO1_F.

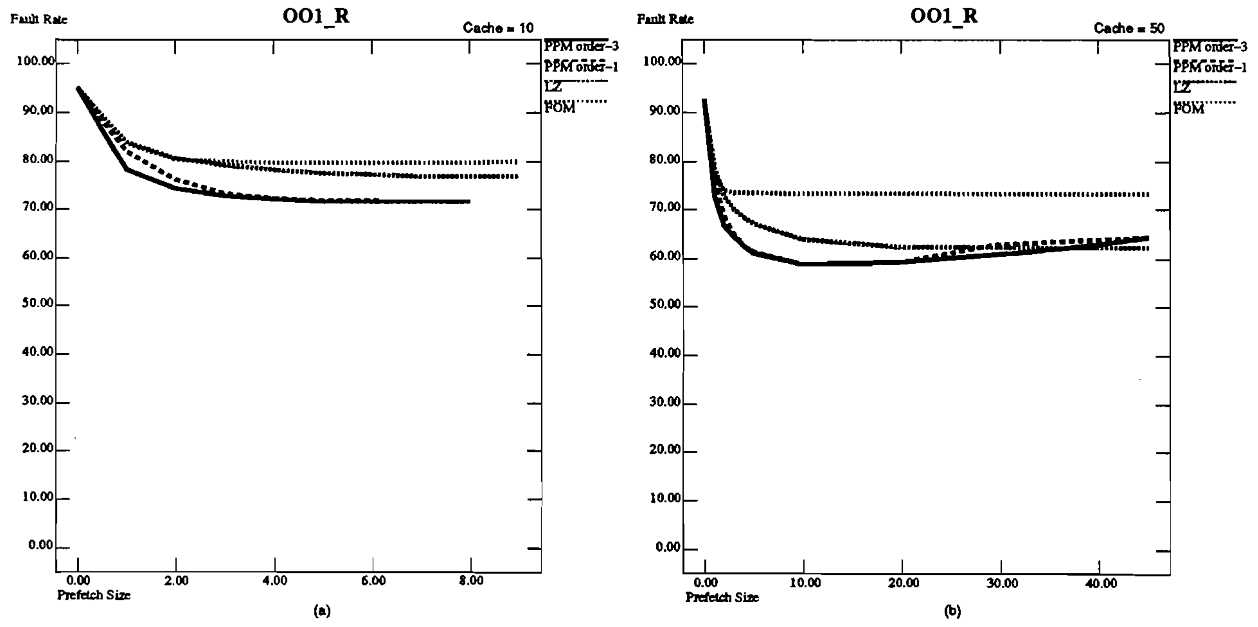

Figure 8: The fault rate for prefetching d objects ($0 \leq d \leq k$) for a fixed cache size. (a) $k = 10$. (b) $k = 50$ for the trace OO1_R. There are 11700 page references and 534 distinct pages in OO1_R.

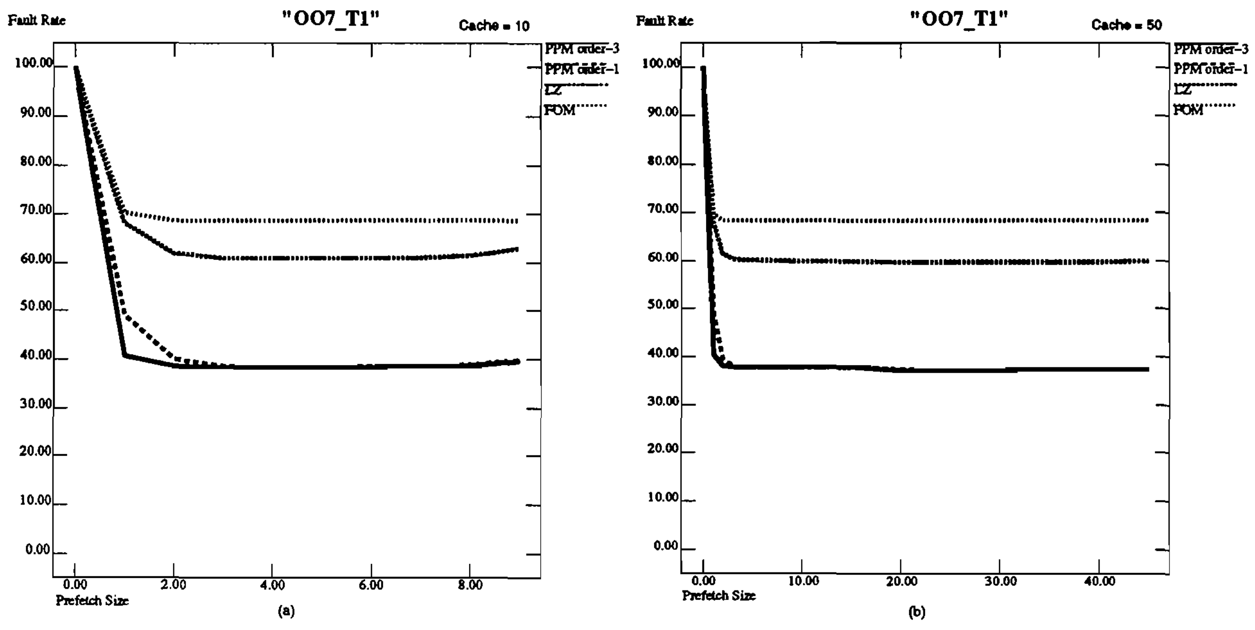

Figure 9: The fault rate for prefetching d pages ($0 \leq d \leq k$) for a fixed cache size. (a) $k = 10$. (b) $k = 50$ for the trace OO7_T1. There are 28103 page references and 6033 distinct pages in trace OO7_T1.

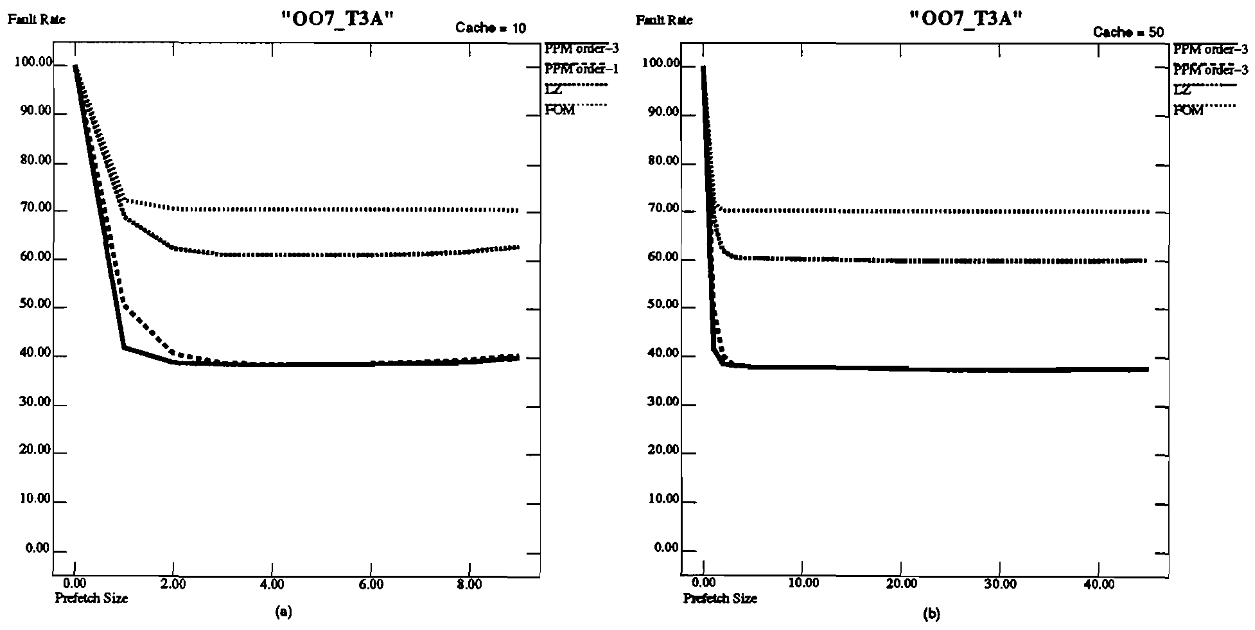

Figure 10: The fault rate for prefetching d pages ($0 \leq d \leq k$) for a fixed cache size. (a) $k = 10$. (b) $k = 50$ for the trace OO7_T3A. There are 30127 page references and 6260 distinct pages in trace OO7_T3A.

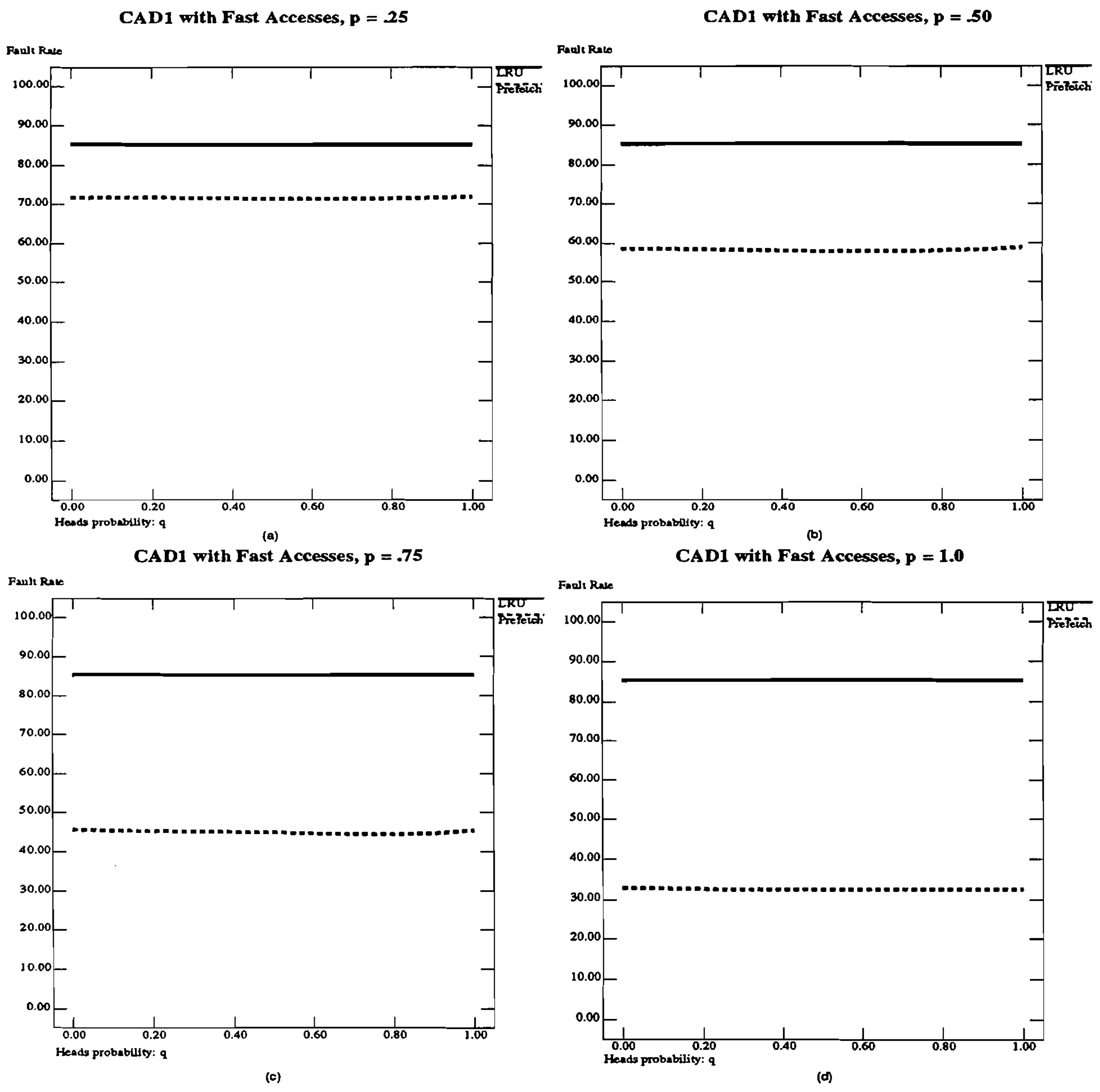

Figure 11: The fault rate for prefetching with the fast access model for a cache size of 10 on the trace CAD1 using Algorithm PPM order 3. The probability q is shown on the x-axis and the fault rate is shown on the y-axis. There are 73768 object references and 15430 distinct objects in trace CAD1.

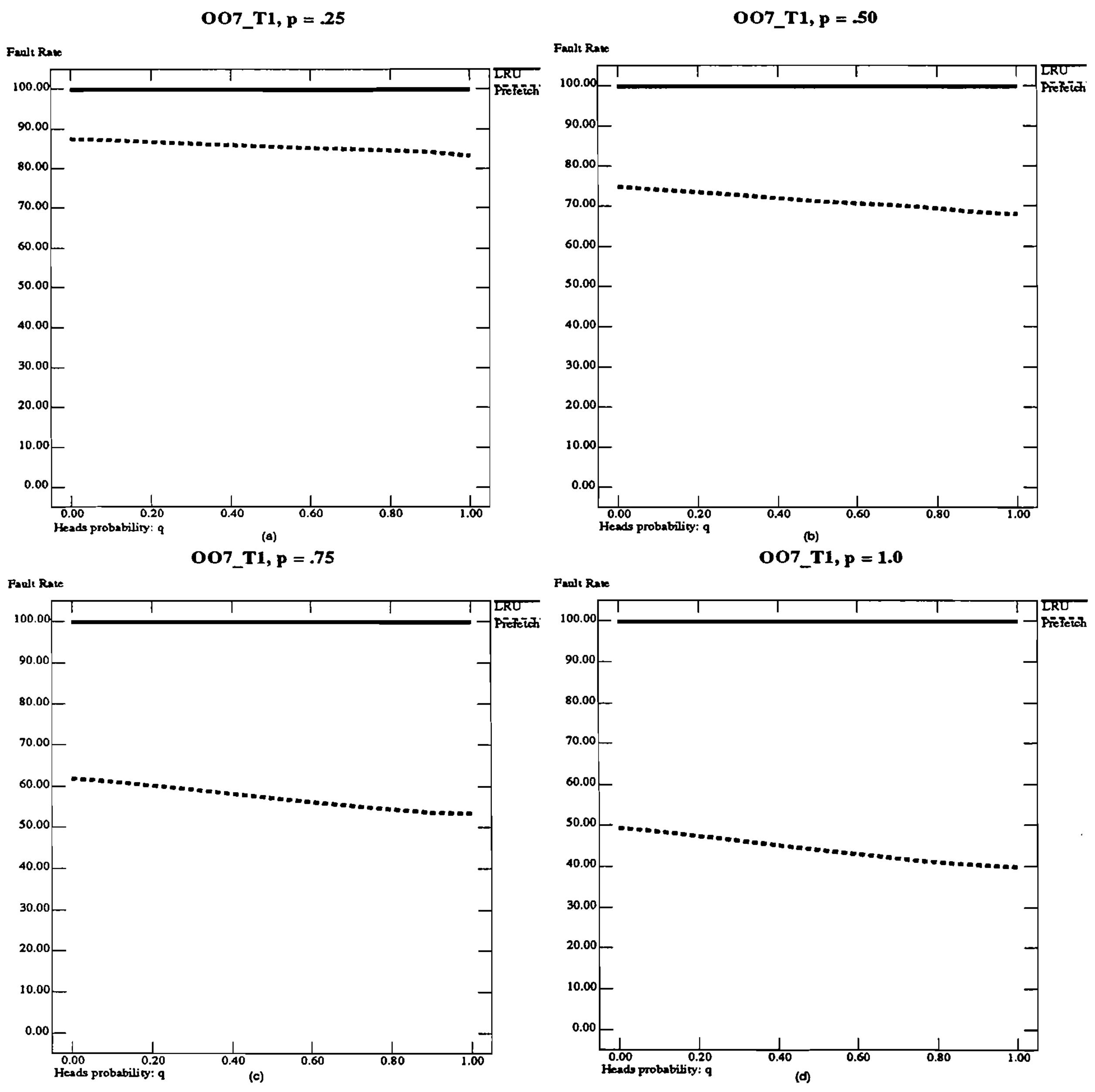

Figure 12: The fault rate for prefetching with the fast access model for a cache size of 10 on the trace OO7_T1 using algorithm PPM order 3. The probability q is shown on the x-axis and the fault rate is shown on the y-axis. There are 28103 page references and 6033 distinct pages in trace OO7_T1.

7.4.1 Algorithm LZ

The tree (or more correctly, trie) data structure used by the LZ algorithm grows as the access pattern is processed. In our simulations, we allow this data to grow as large as necessary and control its resource utilization by paging nodes on secondary storage. We found that even with a modest amount of space dedicated in cache to this data structure, its paging activity was small compared to the length of the access sequence. Some typical overall data structure sizes are contained in Table 1.

7.4.2 Algorithm PPM

As a fair basis of comparison to LZ, we used the raw size (number of bytes needed to represent the model) of the data structure built by LZ for a given trace and bounded the raw size of the PPM data structure by that amount. When the PPM model exceeded that amount, it was flushed and restarted. This method was used for simplicity and is only one of the many available (see Section 3.3). Directly after the model is flushed we expect the predictor to perform poorly but that it will resume making good predictions fairly quickly (in relation to the length of the trace). The following table contains the memory usage, page fault statistics and number of times the prefetch data structure was restarted for the sample traces. The memory limit was set equal to the memory used by LZ in as shown in Section 7.4.1. Ten pages of cache were reserved for paging the prefetch data structure. The page size is 8,192 bytes. The results are shown in Table 1.

7.4.3 Algorithm FOM

The data structure for the first order Markov predictor is bounded by the window size, w. It will occupy the maximum amount of space when the window contains w distinct pages, requiring one state node for each page. In the other extreme, if all of the references contained in the window refer to the same page, there will only be one stat node in the graph.

7.5 Analyzing the results

The results of the simulations are analyzed in detail in this section. The global goal of prefetching is to reduce the overall fault rate obtained by running without prefetching as seen by the application. A more subtle point is to assure that the prefetcher's activities do not incur undue delay at any one point during the application. This information is more difficult to discern from our simulation runs but we will report the results we do have in this area.

LZ

Trace name (total accesses)		Number of nodes in the data structure	Data structure page faults
CAD1	(73,768)	28,513	27,961
CAD2	(147,344)	44,000	43,448
OO1_F	(11,719)	1,792	1,240
OO1_R	(11,700)	1,902	1,350
OO7_T1	(28,103)	13,479	12,927
OO7_T3A	(30,127)	14,161	13,609
OO7_T4	(1,529)	1,525	973

PPM Order 1

Trace name (total accesses)		Number of nodes in the data structure at end of run	Data structure page faults	Data structure restarts
CAD1	(73,768)	32,871	42,050	0
CAD2	(147,344)	35,886	68,215	0
OO1_F	(11,719)	8,807	14,894	0
OO1_R	(11,700)	8,842	15,540	0
OO7_T1	(28,103)	17,462	23,251	0
OO7_T3A	(30,127)	18,695	25,768	0
OO7_T4	(1,529)	3,048	131	0

PPM Order 3

Trace name (total accesses)		Number of nodes in the data structure at end of run	Data structure page faults	Data structure restarts
CAD1	(73,768)	69,986	69,478	0
CAD2	(147,344)	40,664	92,139	0
OO1_F	(11,719)	28,127	35,048	1
OO1_R	(11,700)	28,837	36,144	1
OO7_T1	(28,103)	45,486	1,272	0
OO7_T3A	(30,127)	49,650	1,170	0
OO7_T4	(1,529)	6,108	12	0

Table 1: Uniform prefetching memory use and page fault statistics for Algorithms LZ and PPM

7.5.1 Improvement over LRU

For each of our traces, our prefetchers achieve a significantly reduced fault rate than that of a pure LRU cache. (The fault rates reduce by about 60% for the CAD application traces and by about 20%–30% for the OO1 traces.) The number of faults is related directly to the number of I/Os, and hence good prefetching is extremely significant in reducing the time taken by the application to complete its task. Improvements in fault rate of $x\%$ translate to speedups of roughly $x\%$.

7.5.2 Relationship to data compression performance

One algorithm's prefetching performance relative to the others parallels their relative performance for data compression: $F_{PPM} < F_{LZ} < F_{FOM}$ (where F_A is the *fault rate* for algorithm A).

7.5.3 Benefit realized with few prefetches

In most cases it takes only a small number of predictions (one or two) to greatly reduce the fault rate of the application.

7.5.4 Cache size less significant in decreasing fault rate

Increasing the cache size by a significant factor of 5 (from 10 to 50 in the figures in this section) does not lower the fault rate much. Hence LRU with a larger cache can be compared to our prefetcher with a smaller cache (with the remaining cache space used for storing the in-core prefetch data structures), and the gains in fault rate seen in the above figures still hold.

The true test of a prefetcher is when the cache size is small. We have simulated using a cache size that is roughly 1/100–1/1000 of the number of distinct pages in the trace.

7.5.5 Simple cache replacement performs well

The caching strategy used in conjunction with the uniform prefetcher is extremely relevant. (This is also suggested by the slight increase in fault rate with increasing d in Figures 7b, 8b.) Our caching scheme performs very well as seen. Some other caching strategy may give even better improvements.

7.5.6 Comparing results with Fido

For comparison with Fido [PaZb], we simulated our algorithms on the same trace (CAD2) with the same cache sizes for LRU (2,000) and the prefetcher (1,500) as used in [PaZb]. Fido decreased the fault rate from 45.8% to about 23.5%. Our improvement (for PPM of order 1) was from 45.8% to 18.2%.

7.5.7 Comparing results with sequential prefetching

For comparison with popular heuristics, we analyzed the OO1 traces using sequential prefetching (that is prefetching page $i + 1$ after a request to page i). We found that such an approach decreases the cache fault rate only minimally (by 5%).

7.5.8 Comparing results with the optimal prefetcher

There are 73,768 object references and 15,430 distinct objects in trace CAD1, yielding a lower bound of $15,430/73,768 = 20.8\%$ on the fault rate of *any* universal prefetcher that does not prefetch pages not previously accessed. The 26.7% fault rate of the PPM algorithm of order 3 is close to this lower bound.

8 Related work in prefetching

Fido was developed by Palmer and Zdonik [PaZb] as a prefetcher for databases. It uses a pattern matching approach to prediction. The predictor is trained on an access sequence, the model is frozen, and it is used for prefetching on access traces from similar applications. (This is in contrast to our adaptive approach which continuously learns and predicts for each access sequence.) The MLP caching strategy in Fido ranks prefetched data differently from demand fetched data; this idea can be used in our approach too. Simulations performed on object traces (the CAD1 and CAD2 traces from Section 7) using a cache size of 1500 objects gave impressive improvements in fault rate of almost 22.3% for Fido. Our prefetcher on the same traces and for the same cache size gives improvements of 27.6%.

Work at the NASA Ames Research Center by Philip Laird [Lai] uses a transition directed acyclic graph (TDAG) as a sequence-learning tool for discrete sequence prediction. The TDAG can be applied to dynamically optimize Prolog programs or for maintaining a cache for a database on mass storage. TDAG approximates an unbounded-order PPM model with limited data structure. The TDAG algorithm when used for compression performs comparably to the Lempel-Ziv data compressor (the Unix compress program). In practice, the prediction by partial match algorithm compresses better than Lempel-Ziv [BCW]. Since better compressors typically yield better prefetchers as we report in Section 7.5 (Observation 7.5.2), we expect that the PPM prefetcher should compare favorably to TDAG. We are in touch with Dr. Laird to compare our implementation with his under similar conditions and access sequences.

Other interesting work in prefetching done by Salem uses various first order statistics of the access frequency of database objects to discover "hot spots" in a database. Objects that are found to be *hot* are kept close at hand with the expectation that they will be referenced frequently. We expect such an approach to perform similar to FOM given its similarity.

Research projects in prefetching at a much lower level of abstraction include a software approach in which the compiler reorders instructions to reduce the effect of cache misses [MLG], a hardware scheme of non-blocking and prefetching caches that let processing continue when a cache miss occurs, blocking only when the missed data is actually needed [ChB], and a combined hardware and software approach which uses an optimizing compiler and *speculative loads* to issue read requests in anticipation of a demand request [RoL].

9 Conclusions

We started with the theoretical result from [KrV, ViK] that using data compression for prefetching is a promising technique. We observed that the practical issues in prefetching in databases are much different from the practical issues in data compression, and the pure prefetching assumption made in [KrV, ViK], although valid for hypertext systems needs to be relaxed while looking at general databases. Motivated by this, we converted three practical data compressors to get three practical prefetchers. We simulated our prefetchers on page access traces generated from the DEC OO7 benchmark, the OO1 benchmark and from CAD applications at DEC. We observed a significant decrease in fault rate in comparison to using an LRU cache, and in comparison to other good prefetchers.

General predictors (except the simplest ones) can be expected to require nontrivial data structures, and these may not fit in cache for some applications. We looked at the data structures used by our algorithms, and suggested techniques for paging in the data structures efficiently with a minimum number of I/Os. We have also proposed a solution to the problem of fast accesses and found that our method for dealing with this system constraint was valid and produced good prefetching results.

An interesting result of our simulations is that the prefetching performance of our prefetchers is directly related to the compression ability of the data compressors they are derived from; in particular, algorithm PPM performs better than LZ for both compression and for prefetching. This suggests strongly that the vast research being done in developing good data compressors can be used to develop good prefetchers. The importance of the current report also lies in its attempt to unite two seemingly different practical fields of research. There is a note of caution required since the issues in data compression are different from the ones in prefetching; significant work is required to convert a data compressor to a prefetcher or vice-versa. We expect that the problems encountered in this task are similar to the ones addressed in the current report.

Another interesting result of our simulations is that the biggest benefit of prefetching usually comes from the first page prefetched, and that subsequent pages prefetched do not appreciably reduce the fault rate.

Another important way to achieve better response time is to use clustering. Clustering is in a way dual to prefetching. Clustering algorithms attempt to improve the performance of database systems by placing related sets of objects on the same page in the hopes of reducing the number of average I/Os needed to retrieve objects. There has been extensive work in clustering [BeD, Sta, TsNa, TsNb]. It would be interesting to see the combination of clustering and prefetching on response-time performance. Using prefetch data structures for clustering could also be considered.

There are many open problems that this work motivates, both theoretical and practical. Can our strategy of using LRU with prefetching be shown to be optimal in some reasonable models? Otherwise, is there some other provably optimal caching strategy that can be blended with prefetchers? We expect that recent work on caching models in [KPR] may be relevant. Can our techniques be extended for prefetching in parallel environments?

Acknowledgements

Thanks to Digital Equipment Corporation's GEEP program for making it possible for me to attend Brown University.

On the technical side, I'd like to extend thanks to my advisor Professor Jeff Vitter, Ph. D. candidate P. Krishnan and former GEEP participant and current member of Digital's Database Group, Mark Palmer. I would also like to thank my technical advisor at Digital, John Kirk, for providing much needed advice and assistance and a even few free lunches during my stay at Brown.

Thanks to my father, Ken Curewitz (Sr.), for helping me make this document readable and as error free as possible.

Finally and most importantly, thanks to my wife Shawn for all of her support and endurance and to my two daughters, Danielle and Alexandra, for keeping life in perspective (and keeping me from sleeping too late!).

A Implementation details

We implemented our simulator in ANSI C on a SUN Sparcstation 1 running SunOS Release 4.1. The simulator also runs without modification (after recompilation) on a DECstation 5000/20 under Ultrix V4.2.

We present the actual C code that defines the data structures used by the prefetchers in the following sections and finally include the code used to define the cache and its management.

A.1 Algorithm LZ

The LZ data structure is a trie. Each node in the data structure contains in it all of the information necessary to make predictions. The fixed size part of the node is the below defined "tnode" and the variable size part is an array of "child_info" elements. The root is allocated a large number of "child_info" elements and other nodes of the tree are allocated a small number. When the "child_info" arrays become full, they are reallocated to larger sizes and moved to another page if necessary.

```
typedef unsigned int event;        /* distinguish what an event is */

typedef struct tnode {             /* a node in the tree            */
  struct tnode *parent;            /* this node's parent            */
  unsigned int max_children;       /* maximum number of children    */
  unsigned int num_children;       /* current number of children    */
  unsigned int ds_page;            /* data structure page allocated on */
  struct child_info *child_info;   /* the array of children         */
} t_node;

typedef struct child_info {
  event pageno;                   /* the page number                      */
  int   event_num;                /* when this page last occurred         */
  unsigned int refcnt;            /* number of times it's been referenced */
  t_node *child;                  /* pointer to child with this info      */
} c_info;
```

A.2 Algorithm PPM

The PPM data structure is a slight modification of that used by data compression. Each node in the tree is an "eventnode", its children are pointed to by the "eventset" structure. Vine pointers are realized by the "prev" pointer in "eventnode".

```
typedef    struct {
    uns    totalcnt;          /* the number of events which follow */
    point  list;              /* list storing the event records    */
} eventset;

typedef    struct {
    event    eventnum;        /* the "event" is the page or object */
    uns      count;           /* count is its frequency count      */
    point    next;            /* the right sibling in tree         */
    point    prev;            /* the left sibling in tree          */
    eventset foll;            /* list of following events          */
    unsigned int ds_page;     /* "data structure" page we're on    */
} eventnode, *eventptr;
```

A.3 Algorithm FOM

The FOM data structure consists of "window_elements" and "state_nodes". There is a circular array of "window_elements" that points to the "state_nodes". "State_nodes" point to other "state_nodes" (or themselves) to realize a first order Markov model.

```
typedef struct state_node {     /* model a state node               */
  event  pagenum;               /* the page number of this state    */
  int    in_degree;             /* the count of pointers to state   */
  int    out_degree;            /* number of next states            */
  int    max_out_degree;        /* maximum of above                 */
  struct state_node **next_states;
  int    *next_count;           /* freq counts for next states      */
} state;

typedef struct window_element {  /* an element in the window        */
  event pagenum;                 /* the page number of this event   */
  state *node;                   /* the corresponding state node    */
  int   next_occur;              /* next occurrence of this event   */
} w_element;
```

A.4 Cache

A.4.1 Cache data structures

The cache is organized as an array of integers with one slot for each page in the cache. The number of cache entries is kept in the variable "d_cache_size". To implement LRU, another array, "d_lru_rank" is used to hold the LRU rank of each page in cache. The variable "d_acc_seq" contains one greater than the highest LRU rank of any cache entry.

```
int *d_cache;                  /* the cache                           */
unsigned int d_cache_size;     /* a default size for the cache        */
unsigned int *d_lru_rank;      /* the sequence of a cache entry for LRU */
unsigned int d_acc_seq;        /* access sequence for LRU             */
```

A.4.2 Cache management routines

The following routine headers are included to give the reader an idea how the cache is managed. The header comments describe the interface.

```
/*
  Check to see if "pageno" is in cache.
  If it is, do nothing, otherwise, put it in the cache.
  if it was already in cache, return 0 :  cache hit
  if a the page is put into cache, return 1 : i.e. it faulted
*/
int check_update_cache(event pageno, int *cache,
       int cache_size, uint *lru_rank,
       uint *acc_seq)

/*
  in_cache() returns the index of an entry in the cache if it's found,
  otherwise it returns NOTFOUND
  the implementation is a linear search through the array.
*/
int in_cache(int pageno, int *cache, int cache_size)

/*
  return the value of the next slot in the cache to insert an entry
*/
int next_cache_slot(int cache_size, uint *lru_rank)
```

[Lan] G. G. Langdon, "An Introduction to Arithmetic Coding," *IBM J. Res. Develop.* 28 (March 1984), 135–149.

[MLG] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1992).

[PaZa] M. Palmer and S. Zdonik, "Predictive Caching," Brown University, CS–90–29, November 1990.

[PaZb] M. Palmer and S. Zdonik, "Fido: A Cache that Learns to Fetch," *Proceedings of the 1991 International Conference on Very Large Databases* (September 1991).

[RoL] A. Rogers and K. Li, "Software Support for Speculative Loads," *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1992).

[Sal] K. Salem, "Adaptive Prefetching for Disk Buffers," CESDIS, Goddard Space Flight Center, TR–91–64, January 1991.

[Sta] J. W. Stamos, "Static grouping of small objects to enhance performance of a paged virtual memory," *ACM Transactions on Computer Systems* 2 (May 1984), 155–180.

[Sto] J. A. Storer, *Data Compression Methods and Theory*, Computer Science Press, 1988.

[TsNa] M. M. Tsangaris and J. F. Naughton, "On the Performance of Object Clustering Techniques," *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data* (June 1992), 144–153, Also appears as University of Wisconsin Madison Technical Report number 1090-1992.

[TsNb] M. M. Tsangaris and J. F. Naughton, "A stochastic aproach for clustering in object stores," *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data* (May 1991), 12–21.

[ViK] J. S. Vitter and P. Krishnan, "Optimal Prefetching via Data Compression," *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science* (October 1991), 121–130, Also appears as Brown University Technical Report No. CS–91–46.

[WNC] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic Coding for Data Compression," *Communications of the ACM* 30 (June 1987), 520–540.

[ZiL] J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable-Rate Coding," *IEEE Transactions on Information Theory* 24 (September 1978), 530–536.