SIGMOD '93

Practical Prefetching via Data Compression

(extended abstract)

*Kenneth M. Curewitz**
Digital Equipment Corp.
146 Main Street
Maynard, MA 01754

P. Krishnan†
Dept. of Computer Science
Brown University
Providence, RI 02912–1910

Jeffrey Scott Vitter‡
Dept. of Computer Science
Duke University
Durham, NC 27708–0129

Abstract

An important issue that affects response time performance in current OODB and hypertext systems is the I/O involved in moving objects from slow memory to cache. A promising way to tackle this problem is to use *prefetching*, in which we predict the user's next page requests and get those pages into cache in the background. Current databases perform limited prefetching using techniques derived from older virtual memory systems. A novel idea of using data compression techniques for prefetching was recently advocated in [KrV, ViK], in which prefetchers based on the Lempel-Ziv data compressor (the UNIX *compress* command) were shown theoretically to be optimal in the limit. In this paper we analyze the *practical* aspects of using data compression techniques for prefetching. We adapt three well-known data compressors to get three simple, deterministic, and universal prefetchers. We simulate our prefetchers on sequences of page accesses derived from the OO1 and OO7 benchmarks and from CAD applications, and demonstrate significant reductions in fault-rate. We examine the important issues of cache replacement, size of the data structure used by the prefetcher, and problems arising from bursts of "fast" page requests (that leave virtually no time between adjacent requests for prefetching and book keeping). We conclude that prediction for prefetching based on data compression techniques holds great promise.

*Support was provided in part by a Digital Equipment Corporation GEEP fellowship. Email curewitz@mast.enet.dec.com.

†Support was provided in part by the Defense Advanced Research Projects Agency under contract N00014–91–J–4052, ARPA order 8225, and by the Office of Naval Research. Email pk@cs.brown.edu.

‡Support was provided in part by a National Science Foundation Presidential Young Investigator Award with matching funds from IBM, by Air Force Office of Scientific Research grant number F49620–92–J–0515, and by a Universities Space Research Association/CESDIS associate membership. Email jsv@cs.duke.edu.

1 Introduction

Most computer memories are organized in a hierarchical manner. For example, a typical two-level memory consists of a relatively small but fast *cache* (such as internal memory) and a relatively large but slow memory (such as disk storage). The pages accessed by an application must be in cache. In the event that a requested page is not in cache, a *page fault* occurs and the application has to wait while the page is fetched from slow memory to cache. The method of fetching pages into cache only when a fault occurs is called *demand fetching*. The problem of *cache replacement* 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 which page the user will request 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, the page 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 seriously impeding performance in 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]. Independently to our approach, there has been recent work by Palmer and Zdonik, who use a pattern matching approach to 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]. Research projects in prefetching at a lower level of abstraction include a software approach in which the compiler reorders instructions and introduces explicit prefetching instructions to reduce the effect of cache misses [MLG], a hardware scheme of non-blocking and prefetching caches that lets 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].

The idea of using data compression techniques for prefetching was first advocated by Vitter and Krishnan [KrV, ViK]. The intuition is that data compressors typically operate by postulating (either implicitly or explicitly) a dynamic 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 we can prefetch as many pages as desired limited only by the cache size k (the *pure prefetching* assumption), Vitter and Krishnan show theoretically that any optimal character-by-character data compressor (for example, one derived from the Lempel-Ziv compressor for sequences of page accesses generated by a finite state Markov source) can be converted to a prefetcher that has an optimal fault rate. This is extended to worst-case page access sequences in [KrV].

In this paper we analyze the practical issues of using data compression techniques for prefetching. Although the pure prefetching assumption in [ViK] may be valid in some hypertext applications, in general the time between user page requests will not allow k prefetches at a time. It may actually be prudent in practice to prefetch less than k pages even if there is time (e.g., to avoid burning disk bandwidth). It is therefore imperative to meld good cache replacement techniques with good pure prefetchers, which we address.

The process of converting character-based data compressors to pure prefetchers is quite simple. However, the practical issues in prefetching are *much different* from the ones in data compression; in prefetching, time and memory issues are more significant. In this paper we look at these problems of practical prefetching and develop solutions for them.

We look at three data compressors that perform well in practice and build simple, deterministic, universal[1] prefetchers based on them. We simulate our prefetchers on page access sequences derived from the Object Operations (OO1) benchmark [CaS], the OO7 benchmark [CDN], and from CAD applications used at DEC. We find that the page fault rate (number of page faults divided by the length of the access sequence) decreases significantly compared to that of demand fetching, in which the cache is organized using the least-recently used (LRU) heuristic or using the optimal offline algorithm, OPT [Bel] (in which the page evicted from cache is the one whose next access is furthest in the future). The reduction in fault-rate is also better than that of recent proposed schemes for prefetching [PaZb].

In Section 2 we describe the system environment. We describe our three prefetchers in Section 3. In Section 4 we look closely at problems stemming from memory and time restrictions unique to prefetching in some systems. We propose solutions to these problems and bound their worst-case behavior. In Section 5 we present our simulation environment. In Section 6 we give a brief description of the page access traces and present our simulation results. We present our conclusions in Section 7.

2 System Environment

We model the *client-server* paradigm of computing in which the client is the database user (or application) and the server manages the database. Clients make requests for data from the server and the server fulfills these requests. The client typically runs on a workstation with a modest amount of main memory (cache) and local secondary storage. Data used by an application must be in cache to be accessible. Secondary storage, which can be accessed faster than server storage, is used to store the local operating system, application programs, and is used as swap space by the workstation. The client is connected to the server over a network for communication.

The database is necessarily managed by the server because of its size, its distributed nature, and for consistency control. The server manages the database and handles requests from a number of clients. The obvious benefits of such a distributed system are well known. Prefetching reduces the effect of network latency by anticipating the client's future requests and making such requests when the network is idle.

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 client's requests, flushing prefetch requests in its queue when a demand request arrives. Such provisions are generally available in prefetching systems [GrR, PaZa].

The prefetcher can be either part of the application or a separate entity distinct from the application.

[1] 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].

It works by processing the sequence of the client's previous page requests and making requests for data from the server. If more specific information is available about the client's pattern of page requests, prefetching performance can be improved further. In this paper, though, we prefetch based only on previous page accesses.

Due to the diverse nature of 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 which takes into account only *its* access sequence.

3 Algorithms for Prefetching

Let α be the alphabet size (total number of pages in the database) and k be the cache size. 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 3.1–3.3 in their "generic" form, as pure 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 3.4 we look at the modification to the generic algorithm in which we must prefetch fewer than k pages at a time instant. This occurs when the time between page requests is small, or as mentioned earlier, when the prefetcher makes only $k_1 < k$ educated choices. This partial prefetching automatically introduces the problem of cache replacement; our decision strategy on which pages are 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 4.

3.1 Algorithm LZ

We denote the empty string by λ. Algorithm LZ [ViK] is based on the character-based version $\mathcal{E}$ 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, ..., x_c

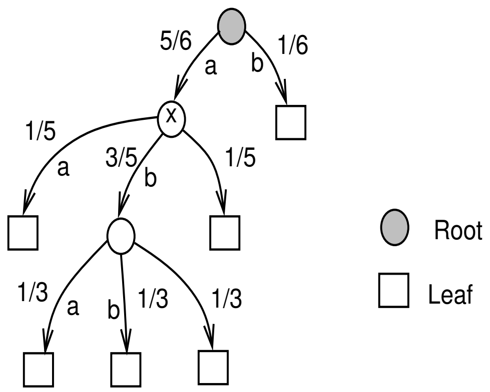

Figure 1: The parse tree constructed by the character-based encoder $\mathcal{E}$ for Example 1. Notice that since the substrings are prefix-closed, they can be represented by a tree in a natural way.

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 $\mathcal{E}$ 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[2] that our alphabet is $\{a, b\}$. We consider the page access sequence "$aaaababaabbbabaa\ldots$," which the Lempel-Ziv encoder parses 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 1. 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 five 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. □

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

[2] We use a binary alphabet in our examples only for ease of exposition. In general, $\alpha \gg 2$.

$\mathcal{E}$'s parse tree. (See Figure 1.) 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.

The data structure used for prediction is a tree with at most one pointer into each node. Instead of maintaining explicit probabilities on each transition, we instead maintain an (integer) count of the number of times the transition is "traversed." For example, in Figure 1, at node x we can store counts of 1, 3, and 1 at the three transitions (instead of the probabilities). The same comment holds for the PPM and FOM algorithms described below.

In our simulations, we use a heuristic for LZ that parallels the Welsh implementation [BCW] of the Lempel-Ziv data compressor. While LZ is at a leaf, 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 transition counts for both the leaf node *and* the root.

3.2 Algorithm PPM

Although the LZ prefetcher is theoretically optimal in the limit [KrV, ViK], convergence to optimality is slow. This motivates us to adapt for prefetching the prediction-by-partial-match (PPM) data compressors, which perform better in practice for compression of text than the Lempel-Ziv algorithm.

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 encountered so far 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. Note that if $j = 0$, each character is predicted based on the relative number of times it appears in the access sequence. □

A PPM prefetcher of order m maintains jth-order Markov predictors (on the page access sequence seen till now) for all $0 \leq j \leq m$. It prefetches the k pages with the maximum k probabilities giving preference to pages predicted by higher order contexts. In our simulations we use PPM of order 3 and order 1.

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]. (Details of data structure management are omitted in this abstract.) The data structure is "almost" a tree; there can be more than one edge into a node because of vine pointers.

3.3 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 3.2.) It prefetches the k pages with the maximum k probabilities as given by this 1st-order Markov predictor. We use $w = 1000$ in our experiments reported in Section 6.2. 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.)

3.4 Cache Replacement Issues

Cache replacement issues automatically arise when we prefetch less than k pages; we need to decide which pages to evict from cache to make space for incoming pages. Any cache replacement algorithm can be suitably modified to work with the "generic" prefetchers described earlier. In particular, we can use the probabilities of the generic prefetcher to determine what to evict from cache, or adapt strategies like the MLP replacement strategy from [PaZb], or adapt well-known cache replacement algorithms like FIFO or LRU. In our simulations, we use a version of LRU suitably modified to handle prefetched pages. 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 recently used pages which, if modified, are written back to disk (a write-back policy).

4 Restricted Memory Environment

Our descriptions of the algorithms in Section 3 assume that the data structures of the prefetcher fit in cache. In some applications this is justified. However, we cannot expect all systems to have this facility.

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 one is built while the old one is used for prefetching. There are also more sophisticated techniques that use an LRU-type strategy on the data structure to maintain its size [BuB]. Our ongoing work studies these techniques in the prefetching context. (We shall see later in Section 6 though that order-1 PPM performs better than FOM; this suggests that placing explicit bounds on the data structure size degrades performance.)

We present the following new scheme to prefetch in a restricted memory environment. For brevity we mention the basic ideas and omit the details.

4.1 Paging the Data Structure

The data structures used by our prefetchers are essentially trees (see Figure 1). 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 of the 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 several "small" nodes into a single page and to allocate only "big" nodes to a page by themselves. 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 1), 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 back a "forwarding address," and when a vine pointer is traversed, this forwarding address pointer is "short-circuited." In the worst case there may be one extra I/O per vine pointer per reallocation (although in practice we see few reallocations and few short-circuiting of pointers). Simulations show that this technique significantly reduces paging for the data structure.

4.2 Sequence of Fast Page Requests

The scheme explained in Section 4.1 solves the limited memory problem by using disk space efficiently but creates a new "timing" problem of *fast page requests* (page requests that arrive quickly so that no I/O can 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 page requests σ can force us to disregard important sequence information.

We have proposed and investigated the following strategy to cope with this problem: In both LZ and PPM, the counts for the pages accessed in the fast sequence σ are incremented at *the current node* (that is, the node used for prediction just before σ started). We explain our scheme with an example for the LZ algorithm. (A similar scheme is used with the PPM algorithm.)

Example 3 Consider a subsequence "*abba*..." of an access sequence. Let the relevant nodes in the subtree for the LZ data structure be as shown in Figure 2a. If the subsequence of page requests is "slow" (i.e., if there is sufficient time to prefetch between accesses), the data structure would look as in Figure 2b after this subsequence.

Consider now the case where the page requests in the subsequence are fast. The current node does not change during the subsequence of fast accesses. The reference counts for a and b are incremented at the current node, which is accessible to the prefetcher in cache. By assumption, a node fits on a page, so no page faults are required to update the data structure. The updated data structure is shown in Figure 2c. □

The intuition behind this scheme is that if the sequence of fast accesses is context-dependent, accumulating statistics at the current node will aid in prefetching the correct pages in the future before the start of a fast access subsequence. By this updating strategy we encapsulate information at a node about not just the next page request but a sequence of future fast page requests.

5 Simulation Environment

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

5.1 Simplifying Assumptions

We bound the complexity of the simulator with the following assumptions about the application being analyzed: We assume that pages do not change their identity during a run and that they are of fixed size. As a rule of thumb, the cache size is chosen to be about 1/100 to 1/1000 of the number of distinct pages in the trace. Most of our simulations are performed on page

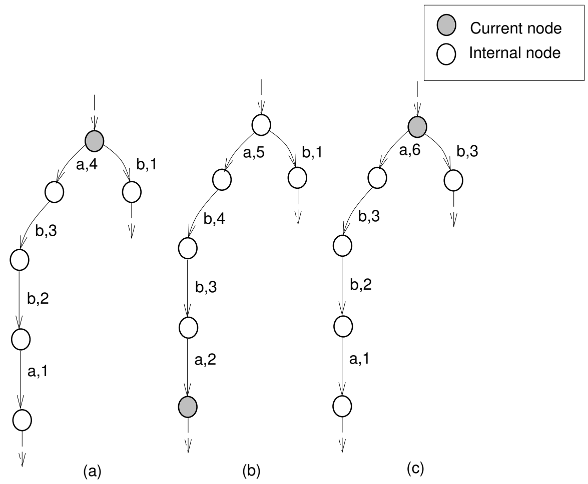

Figure 2: Effect of updating the LZ data structure for the subsequence "*abba* ...". The transition between nodes is labeled with the page identifier and the reference count. (a) Before the subsequence. (b) After the subsequence assuming slow page requests. (c) After the subsequence assuming fast page requests.

access traces. We also perform one set of simulations on object reference traces to aid in comparison with other prefetchers.

5.2 Simulation Method

Each trace (described in more detail in Section 6.1) is a sequence σ of page numbers accessed by a database. We perform two types of simulations on each page access sequence:

Uniform Prefetching. For each page access sequence σ, we simulate each of our prefetchers from Section 3 on σ, prefetching d pages at each prefetch step. From Section 3.4 it follows that when $d = 0$ the prefetcher works as an LRU cache. This provides a basis for comparison against our prefetcher.

We measure the page fault rate for an access sequence using each of the prediction algorithms and for each value of d from 0 up to k. Statistics about the number of faults and the size of the prefetch data structure when it is allowed to grow unbounded are reported.

To analyze the situation when the data structure is paged using our strategy from Section 4.1, we associate with each node of the data structure 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. The fault rate statistics are the same as without paging. We additionally report the number of data structure page I/Os. (Strictly speaking, when we page the data structure, prefetching d pages at each time step implies that we prefetch d pages and do any required data structure I/Os.)

Fast Page Request Prefetching. Fast page requests preempt any prefetching at a step in the execution of the simulator. We use our strategy from Section 4.2 to deal with fast requests. In order to simulate fast requests, we need either traces with detailed 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. Reliable timing information for purposes of prefetching is difficult to obtain. A probabilistic approach is simpler and more widely applicable and was our method of choice. Unfortunately, it removes the relationship between the previous context and the occurrence of fast accesses we expect in practice and thus it provides a *conservative* estimate of prefetching performance. In practice, we expect that our prefetching algorithms will perform even better.

We supply our simulator with the (raw) page access sequence σ (used in the uniform prefetching case) and two probability parameters p, q, $0 \leq p, q \leq 1$. The parameters p and q are used to simulate a workload in a computer system. At each access, the simulator tosses a (biased) coin that lands a "head" with probability p. A "head" signifies that prefetching can be done. If the first coin lands a "head," the second (biased) coin (that lands a "head" with probability q) is repeatedly tossed until we get a "tail" or get $k-1$ "heads." The number of "heads" from the second coin plus one gives the number of pages we can prefetch at this time instance. Setting p and q to a real number close to zero simulates fast page requests while setting p and q to a real number close to one simulates a lightly loaded system. (The expected number of pages prefetched at each time step is $p(kq^k + \sum_{1 \leq t \leq k} tq^{t-1}(1-q))$.)

In this context we simulate only the LZ and the PPM algorithms. (The FOM data structure can always be updated since its data structure is always in cache.)

6 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 OO7 benchmark written at the University of Wisconsin [CDN]. We first describe the access traces and then present our results. In Section 6.4 we analyze the results; this also gives the intuition for the particular format in which the results are presented.

Trace name	Pages accessed	Unique pages	LRU	OPT
CAD1	73,768	15,430	.853	.809
CAD2	147,344	15,430	.833	.825
OO1_F	11,719	526	.941	.891
OO1_R	11,700	534	.952	.911
OO7_T1	28,103	6,033	.999	.994
OO7_T3A	30,127	6,260	.999	.994
OO7_T4	1,529	1,521	.994	.987

Table 1: Trace files and fault rates for LRU and OPT demand caching for cache size $k = 10$.

Trace name	FOM	LZ	PPM-1	PPM-3
CAD1	.378	.398	.328	.267
CAD2	.464	.358	.315	.236
OO1_F	.791	.806	.778	.766
OO1_R	.842	.838	.820	.783
OO7_T1	.702	.682	.492	.407
OO7_T3A	.723	.689	.505	.418
OO7_T4	.994	.994	.994	.994

Table 2: Fault rates of uniform prefetching for cache size $k = 10$, when prefetching $d = 1$ page.

6.1 Description of the Traces

We used CAD and database traces[3] to test our prefetching algorithms. Statistics are given in Table 1.

CAD1 and CAD2 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 OO1 database benchmark, also known as the "Sun Benchmark," was run on the DEC Object/DB[4] 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].

The 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: pointer traversal, application-DBMS coupling, complex object support and long data items, updates and recovery, path indexing, caching and clustering, queries and optimization, concurrency control, and 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 query portion of the benchmark.

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

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

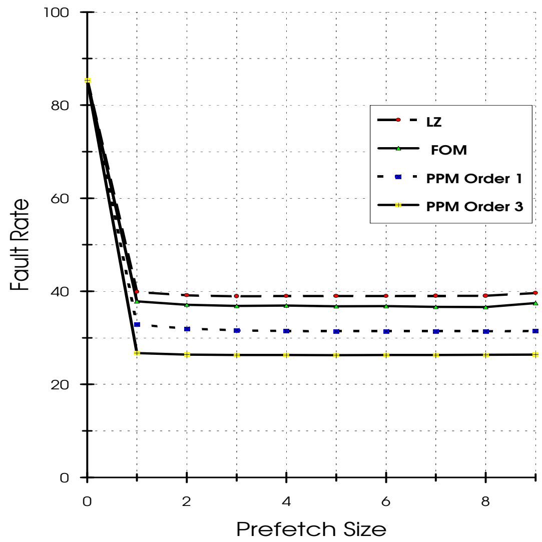

Figure 3: The fault rate for prefetching d objects ($0 \leq d < k$) for a fixed cache size $k = 10$ for the trace CAD1.

6.2 Prefetch Results for Uniform Prefetching

The simulation method for uniform prefetching was described in Section 5.2. We depict graphically the performance of our prefetchers on trace CAD1 in Figure 3 and on trace OO7_T1 in Figure 4. The y-axis denotes the fault rate 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.

In the CAD1 trace, any prefetcher that predicts only pages previously accessed must have a fault rate of at least $15,430/73,768 \approx 21\%$, by Table 1. The PPM-3 fault rate of 26.7% is therefore close to best possible. The graphs look similar for the other traces (except OO7_T4) and the performance numbers are given in Table 2. Our prefetchers' data structure size (when the

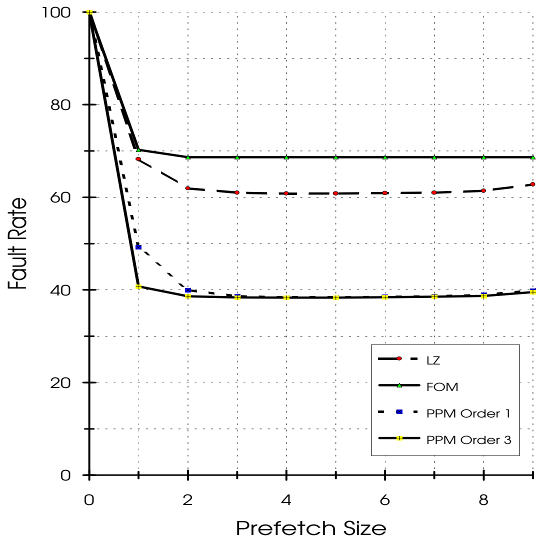

Figure 4: The fault rate for prefetching d objects ($0 \leq d < k$) for a fixed cache size $k = 10$ for the trace OO7_T1.

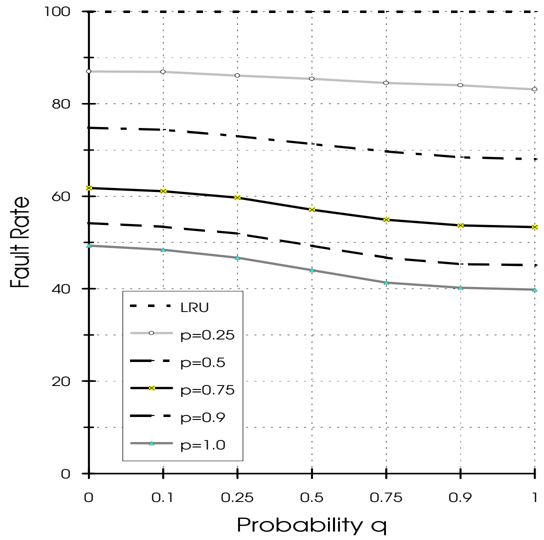

Figure 5: The fault rate for prefetching with the fast page request model for a cache size $k = 10$ on the trace OO7_T1 using algorithm PPM order 3.

Trace name	LZ	PPM-1	PPM-3
CAD1	28,513	32,871	69,986
CAD2	44,000	35,886	40,664
OO1_F	1,792	8,807	28,127
OO1_R	1,902	8,842	28,837
OO7_T1	13,479	17,462	45,486
OO7_T3A	14,161	18,695	49,650
OO7_T4	1,525	3,048	6,108

Table 3: Uniform prefetching memory use in terms of number of nodes for Algorithms LZ, PPM-1, and PPM-3.

Trace name	LZ	PPM-1	PPM-3
CAD1	27,961	42,050	69,478
CAD2	43,448	68,215	92,139
OO1_F	1,240	14,894	35,048
OO1_R	1,350	15,540	36,144
OO7_T1	12,927	23,251	1,272
OO7_T3A	13,609	25,768	1,170
OO7_T4	973	131	12

Table 4: Data structure page I/Os for Algorithms LZ, PPM-1, and PPM-3.

data structure is allowed to grow unbounded) is given in Table 3. In Table 4 we give the number of data structure I/Os performed when 10 out of $k = 20$ cache pages are used for storing the prefetch data structure.

6.3 Prefetch Results with Fast Page Requests

Our method for simulating with fast page requests was described in Section 5.2. 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 trace OO7_T1 in Figure 5. The cache size used is 10 pages. The x-axis denotes the probability q (that ranges from 0.0 to 1.0) and the y-axis denotes the fault rate. The lines represent the fault rate curves for different values of p; one of the lines gives the fault rate performance of LRU (our comparison base).

6.4 Analyzing the Results

For each of our traces, our prefetchers achieve a significantly reduced fault rate than that of a pure LRU cache and even of the OPT caching strategy. (As seen from Table 1, the fault rates reduce by about 60% for the CAD application traces, by about 15%–20% for the OO1 traces, and by about 50%–60% for the OO7 traces.)

In most cases it takes only a small number of predictions (one or two) to greatly reduce the fault rate of the application. (This is graphically visible in Figures 3, 4 and is true for other traces as well; this is the reason we give numbers for only $d = 1$ in Table 2.)

We find that the algorithms' prefetching performance relative to one another parallels their relative performance for data compression in general: $F_{PPM} < F_{LZ} < F_{FOM}$ (where F_A is the *fault rate* for algorithm A).

k	LRU	LZ	PPM-3
10	.853	.398	.267
50	.817	.391	.264

Table 5: The effect of different cache sizes on prefetching performance for trace CAD1.

Increasing the cache size by a significant factor of 5 (from say, 10 to 50) does not lower the fault rate much. (See Table 5. Although it is shown for only trace CAD1 in Table 5, it is true for other traces also.) 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 still hold. In addition, the number of data structure I/Os is not much (see Table 4); for LZ and PPM-1, for example, that is typically a fraction of a data structure I/O per page request.

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.

The cache replacement strategy used in conjunction with the uniform prefetcher is extremely relevant. Our cache replacement scheme performs very well as seen. Some other cache replacement strategy may give even better improvements.

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 better; from 45.8% to 18.2%. (In Fido, 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 cache replacement strategy in Fido uses prediction information to both "promote" and "demote" cached pages; this idea can be used in our approach also.)

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%). Some traces (e.g., OO7_T4) are entirely sequential with almost all pages seen only once. In such cases simple heuristics would work well. We observe that heuristics can be melded with our prefetchers to get added performance benefits.

The fast page request prefetching results (see Figure 5) 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. These results confirm the validity of our methods for modeling fast page requests in the algorithms. The negative slope of the lines suggest that making more than one prefetch at each time step (if possible) has added benefits. This justifies the argument presented at the end of Section 4.2.

7 Conclusions

We started with the theoretical result from [KrV, ViK] that using data compression for prefetching is optimal in the limit. 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 some hypertext systems, needs to be relaxed while looking at general databases. Motivated thus, we converted three practical data compressors to get three practical prefetchers. We simulated our prefetchers on page access traces generated from the OO1 and OO7 benchmarks and from CAD applications at DEC. We observed significant improvements in hit 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 (when there is insufficient time between accesses to update the paged data structures in a normal way).

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 paper 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 and vice-versa. We expect that the problems encountered in this task are similar to the ones addressed in the current paper.

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 hope of reducing the average number of I/Os needed to retrieve objects. There has been extensive work in clustering (e.g., [TsN] and references therein). 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 cache replacement 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. We would like to thank Mark Palmer from Digital for his support in providing us with access traces and for many useful discussions and comments. We would also like to thank Stan Zdonik and Dave Langworthy at Brown for sharing their thoughts and practical experience with us.

References

[Bel] L. A. Belady, "A Study of Replacement Algorithms for Virtual Storage Computers," *IBM Systems Journal* 5 (1966), 78–101.

[BCW] T. C. Bell, J. C. Cleary & I. H. Witten, *Text Compression*, Prentice Hall Adv. Ref. Series, 1990.

[Bra] J. T. Brady, "A theory of productivity in the creative process," *IEEE CG&A* (May 1986).

[BuB] S. Bunton & G. Borriello, "Practical Dictionary Management for Hardware Data Compression," Department of Computer Science, University of Washington, FR-35, 1991.

[CDN] M. J. Carey, D. J. DeWitt & J. F. Naughton, "The OO7 Benchmark," *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, this proceeding.

[CaS] R. G. G. Cattell & J. Skeen, "Object Operations Benchmark," *ACM Transactions on Database Systems* 17 (March 1992), 1–31.

[ChB] T. F. Chen & J. L. Baer, "Reducing Memory Latency via Non-blocking and Prefetching Caches," *ASPLOS-V*, Boston, MA (October 1992).

[FKL] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator & N. E. Young, "Competitive Paging Algorithms," CMU, CS–88–196, November 1988.

[GrR] J. Gray & A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, Inc., 1993.

[HoV] P. G. Howard & J. S. Vitter, "Analysis of Arithmetic Coding for Data Compression," *Information Processing and Management* 28 (1992), 749–763, invited paper in special issue on data compression for images and texts.

[KPR] A. R. Karlin, S. J. Phillips & P. Raghavan, "Markov Paging," *Proceedings of the 33rd Annual IEEE Conference on Foundations of Computer Science* (October 1992).

[KoE] D. F. Kotz & C. S. Ellis, "Prefetching in File Systems for MIMD Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems* 1 (April 1990), 218–230.

[KrV] P. Krishnan & J. S. Vitter, "Optimal Prefetching in the Worst Case," *manuscript* (November 1992).

[Lai] P. Laird, "Discrete Sequence Prediction and its Applications," AI Research Branch, NASA Ames Research Center, manuscript, 1992.

[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 & A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," *ASPLOS-V*, Boston, MA (October 1992).

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

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

[RoL] A. Rogers & K. Li, "Software Support for Speculative Loads," *ASPLOS-V*, Boston, MA (October 1992).

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

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

[TsN] M. M. Tsangaris & J. F. Naughton, "On the Performance of Object Clustering Techniques," *Proc. of the 1992 ACM SIGMOD International Conference on Management of Data* , San Diego, California (June 1992).

[ViK] J. S. Vitter & P. Krishnan, "Optimal Prefetching via Data Compression," *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science* (October 1991), also appears as Brown Univ. Tech. Rep. No. CS–91–46.

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

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