BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-91-M12

A Data Cache that Learns to Fetch

by
Mark L. Palmer

A Data Cache that Learns to Fetch

Mark L. Palmer

Submitted in partial fulfillment of the requirements for the
Degree of Master of Science
in the Department of Computer Science at Brown University
March, 1991

A This submission by Mark L. Palmer is accepted in its present form
by the Department of Computer Science in partial fulfillment of the
requirements for the Degree of Master of Science.

Date 3/12/91

Stanley B. Zdonik
Stanley B. Zdonik
Advisor

A Data Cache That Learns to Fetch

Mark Palmer
Digital Equipment Corporation
2 Elizabeth Drive CTC2-2/D10
Chelmsford, MA 01829

Stanley B. Zdonik
Brown University
Computer Science Dept.
Providence, RI 02912

Abstract

This paper describes **Fido**, a *predictive cache* [Palmer 1990] that anticipates access by employing an associative memory to recognize regularities in access pattern for each isolated access context over time. Continual training adapts the associative memory contents to database and access pattern changes, allowing online access predictions for prefetching. We discuss two salient components of Fido - **MLP,** a replacement policy for managing prefetched objects, and **Estimating Prophet**, the component that recognizes patterns and predicts access. We then present some early simulation results which suggest that predictive caching works well and conclude that it is a promising method.

1 Introduction

A major performance factor for current OODB systems is the cost of fetching data from secondary storage as needed. In workstation-server architectures, this cost is compounded as data crosses several I/O boundaries on its path from the server's secondary storage to an application's memory. Caching in workstation-server OODB architectures improves performance, as described by [Rubenstien 1987]. Chang and Katz [Chang 1989] found that cache management policy has the largest effect on response time, followed by clustering, a much-studied topic [Chan 1982], [Stamos 1984], [Hudson 1990]. Current OODB cache designs are derived from virtual memory demand pagers, which often prefetch clusters, but assume no explicit knowledge of the future. The goal of clustering is to force a locality of reference on data like that inherent in code execution, the property exploited by demand paging.

However, a major purpose of databases is to allow data sharing and integration of diverse applications. Cluster prefetching alone can be inherently ineffective when users mix conflicting patterns in accessing the same data. Devising a clustering that is both fair and efficient then becomes problematic and expertise-intensive, and finding an optimal partitioning can be computationally unfeasible [Niamir 1978]. Thus a user-specific yet general purpose complement to cluster prefetch is of interest.

We are finding that access within individual contexts can be predicted because of structure in data and determinism in programs, irrespective of inherent or arranged locality of reference. Fido automatically recognizes and exploits patterns that emerge over time within each access context to provide accurate prefetching under mixed access paradigms, for which the best clustering may be a compromise.

1.1 Predictive Caching

A predictive cache can supplement existing cache management strategies, such as demand paging with cluster prefetching, or could function as a primary cache. Fido's predictive cache model is an extension of common demand fetching strategies. The cache manager is augmented with a component consisting of an associative memory and several pattern recognition routines. This "black box" is known as the Estimating Prophet. The prophet learns access patterns for isolated contexts over time well enough to predict access within each context. During a database session, the prophet monitors client-server communication and generates access predictions, which Fido then uses to prefetch data. Each prediction indicates an explicit expected order and likelihood of access, information that is exploited by the cache manager's replacement policy.

This solution addresses the difficulties described above and has several desirable properties. The prophet gains experience with each access pattern individually, tailoring its predictions within each access context according to the history of that context, without considering other usage patterns. Also, the prophet monitors access sequences in a non-invasive way, requiring no knowledge of data model or schema. Its operation is automatic and invisible to the database administrator, requiring little in the way of time, intuition, or expertise to operate. By continually monitoring access sequences, the prophet can adjust its predictions to reflect changes in usage quickly, incrementally, and in a uniform way.

1.2 Paper Structure

This paper introduces concepts and terminology useful as a framework for studying predictive caching, and discusses design issues identified by running the simulation experiments described. It does not attempt formal analysis of optimality for all possible worlds.

Our current research focuses on two topics. The first topic concerns how to best incorporate predictive prefetch activity into cache management, given costs of prediction and prefetching. The second topic involves comparing various prophet designs to assess predictive accuracy and efficiency. Accordingly, this paper presents a) a model for predictive cache management and b) a prophet design that uses associative memory to recognize and predict access sequences on-line.

Section 2 describes how Fido fits into a workstation-server OODB architecture. Section 3 presents a cache management model that incorporates prediction. The method of recognizing and predicting access patterns is discussed in section 4. Section 5 describes some interesting experiments simulating a predictive cache using actual access traces. Section 6 discusses related work, and some conclusions are offered in Section 7.

2 Architectural Overview

The purpose of this section is to summarize just those aspects of the target database architecture essential to illustrate how Fido works. This description covers a subset of functions provided by the database system, and entails some simplifying assumptions. First, methods for managing a cache of variable-sized objects are orthogonal to the issues of accuracy and costs of prefetching examined in this paper, so one running assumption is that objects are of uniform size. Second, the distributed system architect must consider methods of validating and synchronizing cached objects against replicas. These issues have been addressed by others (e.g. [Alonso 1990], [Wilkinson 1990], [Garza 1988]), and are not considered here. We assume that prefetched objects are locked and validated in cache as if they had been requested. The target design applications will usually have low contention for write locks, since these applications often have high read/write ratios ([Cattell 1987], [Chang 1989]). The functioning of a predictive cache per se does not rely on these simplifications.

Predictive caching will be most useful to distributed applications that:

- are data intensive, with high read/write ratios
- use navigational access patterns which each imply different data clusterings
- create and delete medium-granularity objects at a rate slow enough to permit tracking of changes
- preserve some degree of object identity.

In general, CAD applications have many of these characteristics. Object identity is maintained by OODBs, which support data sharing between such applications.

Fido is intended to operate in OODB systems where applications retrieve objects from secondary storage and cache them in local memory. This is known as a workstation-server, or interpreter/ storage manager architecture. The typical system consists of a central server machine responding to requests for data shared between applications running independently on workstations. Several systems, such as Cactis [Hudson 1990], O_2 [Velez 1989], ORION-1SX [Garza 1988], and Mneme [Moss 1990] are similar in this respect. In [DeWitt 1990], the performance of several such variations is compared. We wish to add Fido to Observer/ENCORE [Fernandez 1990]. The primary components of the architecture are the database server and the client, which includes the predictive cache.

2.1 Database Server

Observer acts as a typeless back end to applications, managing access to database secondary storage. Observer maintains strong object identity [Khoshafian 1986], which aids predictive caching - the preservation of identity simplifies recognition of reoccurring parts of an access pattern over time. Object identity in Observer is provided via an external unique identifier (UID) that acts as an immutable handle for an object and is not recycled. OODBs may assign meanings to individual bits of UIDs, these semantics are not of concern to Fido, which treats access sequences as strings of symbols.

A read message supplies a list of UIDs to Observer, which gets the identified objects from disk or a server-managed buffer and returns them to the requester. To facilitate prefetching, the basic Observer read function is extended in several ways. A requester marks reads for either *demand* or *prefetch* processing. Observer ensures that outstanding prefetch requests never delay other requests by providing a *pre-emptive read* operation, to be used by demand requests, that is serviced before other reads.

Observer objects can be clustered into segments and can migrate or be replicated between segments. Observer returns a segment when one object in the segment is referenced, thus segments are the usual unit of server-client communication. Segment prefetching may also be disabled, in which case the set of objects in a request is returned as the unit of network communication.

2.2 Client

A workstation application interfaces to Observer via the ENCORE client component, which acts as an interpreter, mapping the data model used by the application onto operations understandable by Observer. ENCORE implements object type semantics, executing methods and enforcing encapsulation, and is typically bound into the application's image. ENCORE also validates cache objects and supports other database functions related to persistence and distribution, but the operation of these is unrelated to the prefetch mechanism. The salient function of the client is that it allows the application to reference objects by UID, without knowledge of how or where objects are stored. The client ensures that referenced objects are ferried between the server and the application's local memory transparently. Clients take various approaches to translating object references to memory addresses, often using some form of Resident Object Table (ROT) to obtain a pointer to the object's location in memory, and may "swizzle" the ROT entry by adding the memory pointer .

ENCORE maintains a cache of currently used objects, sending demand requests to the server when the application references an object not in cache and "flushing" modified objects back to the server's secondary storage as needed. If the way objects are clustered into Observer segments does not suit the current access pattern, cache faults can increase network demand fetches, slowing response time performance.

2.3 Access Contexts

The client isolates access patterns according to *access contexts*. The prophet provides context identifiers (CIDs) as a handle for associating patterns generated by the same source. By default, ENCORE uses CIDs to indicate the combination of user and application that generates a particular access sequence, but context assignment may be controlled further by the programmer. For example, an application might provide one function that graphically displays a circuit design, and another function that allows ad hoc queries. The access pattern of the display function might be very predictable, allowing efficient learning, while sequences generated by ad hoc queries could be arbitrary and difficult to learn. A programmer might establish different CIDs corresponding to these two functions, even when invoked by the same user. With CIDs, a designer can use knowledge of an application to "focus attention" of the prophet, reducing pattern memory requirements at any given time and speeding prediction.

2.3.1 Fido Predictive Cache

A portion of ENCORE's client cache is allocated to the Fido predictive cache, which interfaces to the prophet to decide what to prefetch and the order in which to replace cached objects. The prophet can be configured as a separate service or as part of the client image. It has two primary modes of operation: *prediction* and *training*.

Given a sample of the latest sequence of access to Fido, the prophet predicts which accesses will occur next. An individual prediction may indicate that alternate sequences are anticipated by arranging identifiers according to expected order and likelihood of access.

In training mode, the prophet learns access patterns over time and becomes increasingly better at prediction, until it reaches a stable state where learning ceases. This state may be reached because the prophet is not encountering any new information or changes in access pattern, or because it has exceeded user-specified resource limits. The access pattern information for a single CID is known as a *pattern memory*. Fido stores each pattern memory between sessions.

Figure 1 shows how Fido fits in with the client and server; UIDs are represented as letters and objects as circles. As an application session begins, Fido loads (0) the pattern memory for the access context. The application generates a sequence of references to the client, which converts UIDs to object memory address via the ROT.

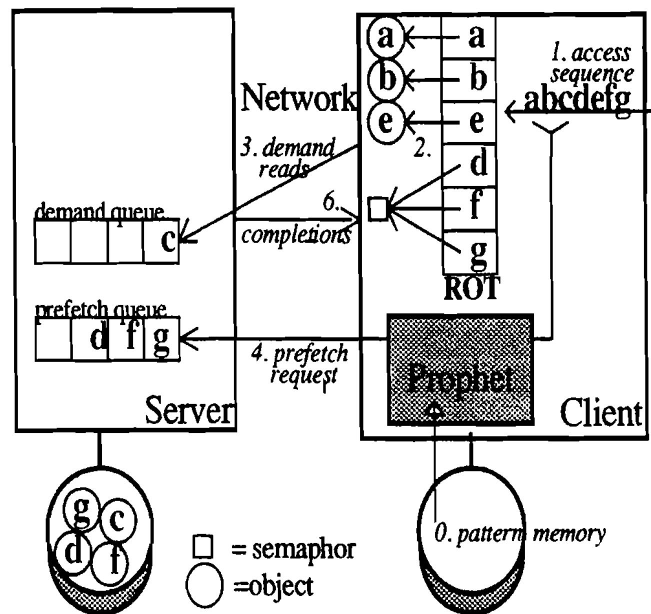

Figure 1: Server and Client with Predictive Cache

References to a and b return pointers (2) to those objects in the cache. If an accessed object (e.g. c) is not in cache, the client issues a pre-emptive demand read (3) to the server, and blocks. During the session, the prophet samples the current access sequence (4), recognizes the start of a known pattern, and completes it, predicting "d e f g". Since e is already in cache, Fido starts a prefetch request for (5) d, f, and g, entering "promises" for these objects into the ROT. Any access to d, f, or g before the prefetch request arrives blocks until I/O completion updates the promises, converting them to regular ROT entries. As described in the next section, the prophet's predictions also govern replacement ordering.

3 Predictive Cache Management Model

As mentioned above, the client ferries objects between the server and the workstation's memory transparently to the application. The prophet generates predictions, causing objects to be prefetched. The client needs a strategy for managing prefetched objects. This section outlines a predictive cache management model. Simulation has been helpful in identifying design issues for predictive caching, such as:

- defining an appropriate model for prediction
- ideal case operation, when many correct predictions are being generated
- faulting and cache space utilization behavior in the worst case, when no predictions are being made
- interactions between cached objects when a prefetch is accessed or an object eviction is required
- computation and communication overhead for prediction and prefetching.

Fido maintains a FIFO sampling window on the current access sequence. On each access to an object *o*, if *o* is not in cache, a demand read is started. Fido then gives the sequence sample to the prophet, which may return a prediction of the immediate future. If so, Fido checks whether any predicted objects are already in cache, and starts a single prefetch request for those which are not. When the prefetched objects arrive, an asynchronous completion routine decides which cached objects to replace. The overall strategy of predictive cache management involves three goals:

1. quickly flushing erroneous prefetches from cache
2. avoiding wasted cache when no predictions are made
3. controlling the volume and cost of predictions.

The first two goals are addressed by Fido's replacement policy, and the last by heuristics - Fido monitors prediction performance and adjusts certain prophet parameters dynamically to throttle prediction rate balancing cost against accuracy. We will now introduce some terms, describe the replacement policy, and then portray Fido's operation under several conditions.

3.1 Definitions

The following terms define a predictive cache.

3.1.1 Predictive cache

A predictive cache C={R U P} is a union of two disjoint sets - a prefetch set P, and a set of referenced objects, R. C can hold *k* objects.

3.1.2 Prefetch set

A prefetch set P is the set of prefetch requests present in cache at any one time. Objects in older prefetch requests are considered less likely to be accessed than objects in more recent requests.

3.1.3 Prefetch request

A prefetch request π_i is the order-maintained subset of objects in prediction Π_i not in C at the time Π_i is made. The "head" of a prefetch request identifies the object most likely to be accessed first, while the "tail" identifies the object whose access is expected furthest in the future and is thus least likely: $\pi_i = \{\Pi_i - C\}$. A prefetch request is the intended unit of I/O between cache and secondary storage; objects identified are prefetched in a single request. In this sense, it resembles a page of objects assembled at the server. The difference is that its contents are determined dynamically instead of by static clustering, and have explicit ordering.

3.1.4 Prediction

A prediction $\Pi = o_1 \ldots o_\omega$ is a list of identifiers partially ordered by expected access sequence and fully ordered by probability of access. That is, an access to object o_i is expected before an access to object o_j for i<j, unless o_i and o_j are alternate possibilities, in which case an access to o_i is more likely than an access to o_j.

3.2 MLP Replacement Policy

A cache manager decides which objects to replace with new objects by implementing a replacement policy. Fido's replacement policy flushes erroneous prefetches from cache by ensuring that unused prefetches have lower priority than new prefetches or referenced objects. The Minimum Likelihood Prefetch (MLP) replacement policy stipulates:

- Within a prefetch request, evict the Minimum Likelihood Prefetch first. That is, prefetch eviction happens from tail to head of each prefetch request.
- The definition of P implies that old prefetch requests are evicted before new prefetch requests.
- On access to object *o*, promote *o* to most-recently-used status within C. If operating "beneath" a primary cache, swap *o* with *o'* evicted by the primary cache.

The MLP policy is adapted from the proven optimal replacement policy for demand paging, OPT [Mattson 1970]. OPT always replaces that item which is accessed furthest in the future, but operates off-line. MLP replaces objects which are *expected* to be referenced furthest in the *foreseen* future. One difference between MLP and OPT is that MLP uses *estimated, incremental* knowledge of the future, instead of perfect prescience as assumed by OPT. During periods when the prophet does not make predictions, however, the third rule causes Fido to operate as a demand LRU cache.

3.2.1 Replacement Ordering

The replacement ordering for objects in C can be modeled by operations on a fixed size list. Identifiers are inserted at the head and deleted from the tail. This behavior governs how objects are brought into and evicted from C, whether by prefetch requests or by faults. As an access sequence is processed, the prophet generates predictions, which result in prefetch requests. As prefetch requests arrive, they are inserted at the list head in reverse of their expected access order. The object whose expected access is furthest in the future, i.e. - that least likely to be used, enters into and is evicted from C first.

Time	Ref/Prefetch	Replacement List State					
0	a	c	l	a	m	b	k
1	b	a	c	l	m	b	k
2	c	b	a	c	l	m	k
2	Π=d e f x g	c	b	a	l	m	k
3	d	d	e	f	x	g	c
4	e	e	d	f	x	g	c
5	f	f	e	d	x	g	c
6	q	q	f	e	d	x	g
7	g	g	q	f	e	d	x
8	h	h	g	q	f	e	d

Figure 2: Replacement Example

The example in Figure 2 illustrates replacement priority as a prefetch request is handled and faults occur. For simplicity, the example assumes that Fido is operating as a primary cache. Successive references descend the left column, with the list state shown at right. Identifiers are added at the list head (left) and removed from its tail. By time 2, references to a, b, and c cause reordering, and the sampling window contents, abc, match a known pattern, triggering the prefetch request for defxg. This prefetch request arrives before another reference is made, causing eviction of everything but c. At 6, an access to q (instead of x) faults, replacing c with q. At 7, g is moved up, leaving x to be evicted by the fault for h.

3.3 Cache Behavior

Three prototypic cases combine to characterize the intended behavior of the cache during operation.

3.3.1 Sequence Recognition (best case)

A prefetch request of size k is made every k accesses, which arrives, fills the cache, and is then consumed from head to tail, moving π from P to R, and leaving the replacement list containing π in reversed order.

3.3.2 Prediction Starvation

Within a session, intervals occur during which the current sequence is unknown to the prophet, which generates no predictions. In this situation, the move-to-front [Tarjan 1985] rule produces the replacement behavior of a demand LRU cache processing the same sequence.

3.3.3 Error Glut (worst case)

In this situation, some maximum number of predictions is made on every reference, but none are correct, potentially causing the cache to be full of useless objects.

The following heuristic adjusts certain prophet parameters to control cases 2 and 3 above:

- Guess rate: Let λ denote the ratio of objects prefetched divided by the number of references made at any given time during the session
- Accuracy: χ denotes the ratio of correct predictions to total predictions during the session.
- Efficiency: If α denotes the sample window size, and ω denotes the size of a prediction, the ratio $\upsilon=\omega/\alpha$ reflects how much sample is used in generating each prediction.

Guess rate, accuracy, and efficiency interact. Specifically, increasing the sample size lowers efficiency and guess rate, but raises accuracy. Fido monitors guess rate and efficiency by keeping running averages. Prediction starvation can occur if too much information is supplied in the samples, and error glut can happen if the sample size is too small. To control these situations, Fido adjusts sample size, α, according to $\chi\lambda$. If $\chi\lambda$ crosses a low or high threshold, Fido increments or decrements α accordingly to try to bring $\chi\lambda$ back into range.

4 Estimating Prophet

Previous sections of this paper have outlined a model for managing a cache containing prefetched objects, but have assumed an ability to predict references. This section presents a design intended to illustrate how the current prophet learns to predict. The prophet learns access patterns in training mode and recognizes them in prediction mode. The client records reference traces from each session within each access context. Training mode processes each reference trace - normally (but not *necessarily*) off-line, between sessions, and incrementally improves pattern memory for each access context. Prediction mode is used to generate prefetch requests, as described in sections 2 and 3.

The intuitive explanation of how training and prediction work is that the present sequence acts as a cue - when the prophet is presented with a sequence that is *similar* to some previously encountered situations, it recalls the *consequences* of those previous situations - this is analogous to the way organisms determine present behavior according to past experience. Thus both training and prediction rely on an ability to quickly but inexactly retrieve previous sequences using information about the present sequence as a key. We will first discuss this inexact retrieval capability, and then outline its function in training and prediction.

4.1 Associative Memory

The prophet stores and retrieves access order information in an inexact manner using a *nearest-neighbor* associative memory. Much work has been done on associative memory architectures ([Potter 1987], [Kanerva 1988]) some of which provide the most biologically plausible neural net models. Research by Anderson and others has shown how such memories can be constructed from elements that imitate the functioning of neurons in cortex [Anderson 1990], and evidence suggests that cognition may indeed operate this way. Nearest-neighbor models map data units with k elements to points in k-dimensional pattern space, defining similarity metrics in k dimensions. Similar patterns are near each other in pattern space, and equivalence classes are defined by radius values. The following terms apply to the nearest-neighbor based pattern memory model.

4.1.1 Pattern Memory

A pattern memory consists of t *unit patterns*, $\xi_0, \ldots, \xi_t$, pairwise at least r distant in pattern space. Each unit pattern defines an *equivalence class* - all observed sequences within the pattern-space sphere having ξ at its center and radius r are considered equivalent to ξ. This capacity for inexactness is important for two reasons.

First, it means that pattern memory is lossy - it ignores minor variations in patterns encountered over time, only using resources to represent significant differences.

Second, it allows useful predictions to be made even if the current access sequence does not exactly match what has appeared before, as when new identifiers appear after recent updates to the database.

4.1.2 Unit Pattern

A unit pattern $\xi=<o_1...o_\sigma>$ is a list of identifiers that acts as a partial approximation of access pattern. Each unit pattern divides into a *prefix* of variable length α and *suffix* of length $\sigma-\alpha < k$. The prefix acts as a key for the suffix during prediction. One can think of the prefix as an observed antecedent, and of the suffix as its consequence. Each suffix usually contains the prefix of some other unit pattern(s), creating inexactly linked chains of unit patterns that approximate observed alternate sequences (see Figure 4). Also stored with each unit pattern are ratings of its historical frequency of occurrence and average predictive accuracy. Newly created unit patterns are of a uniform maximum length, but may be shortened over time by training.

4.1.3 Distance in Pattern Space

The measure of dissimilarity between two unit patterns is the count of columnwise unequal identifiers.

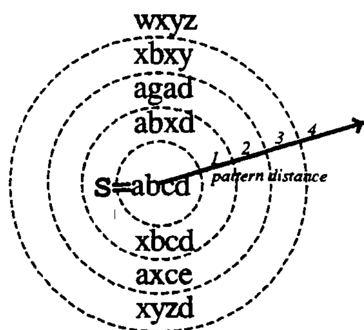

Figure 3: Unit Patterns in Pattern Space

In Figure 3, one sample S and a set of unit patterns, all of equal lengths, are arranged according to increasing distance from S. The nearest neighbors to S are the unit patterns closest to S in pattern space. S is considered equal to all unit patterns closer than r to S.

4.2 Resource Costs

The current prophet implementation is described further in [Palmer 1990]. Two of its properties are:

- A non-repeating sequence of length l can be stored in pattern memory using $O(l)$ space, creating $t=l/\sigma$ unit patterns.
- All nearest neighbors of a sample of length α can be found in $O(\alpha \log(t) + f)$, where f is an expected number of neighbors.

Pattern memory space requirements of the implementation are adequate for problem sizes of current interest (i.e. - infrequently-repeating strings of $O(10^5)$ identifiers). The size of pattern memory can be controlled by user-set resource limits, and by establishing multiple access contexts per application. Neighbor finding is fast enough for predictions involving thousands of unit patterns, since sample size is typically small. We expect to be able to further reduce prophet resource requirements through continued research.

4.3 Training Mode

After each session, the client saves a reference trace for each access context invoked, then runs the prophet in training mode to process the saved traces. The training algorithm adapts the contents of pattern memory over time so that only common unit patterns are retained in pattern memory. Training reinforces unit patterns that appear frequently, and represses sequences that appear sporadically, or which consist of obsolete information. Unit patterns "compete" for space in pattern memory over time based on their ability to generate prefetch requests contributing to overall system speedup. This causes pattern memory to self-organize, focus on regularly reoccurring phenomena, and evolve an internal generalization of each access pattern. The training algorithm employs an "evolutionary" strategy consisting of two phases - credit assignment and adaptation.

4.3.1 Credit Assignment

This phase assigns credit to all unit patterns that contribute to predicting the training trace. Some stored unit patterns recur only infrequently, while others become obsolete as updates to the database cause new identifiers to appear and others to disappear. Also, the lengths of unit patterns are initially uniform, producing arbitrary sampling boundaries. Infrequent, obsolete, or poorly chosen samples produce unit patterns that do not function well as predictors, and which can congest pattern memory with useless information.

Credit assignment begins by simulating a prediction run along the training trace l. Each time a prediction occurs at a point i in l, accuracy and frequency ratings of each unit pattern ξ contributing to the prediction are updated. Accuracy is assessed for ξ by counting the number of identifiers in the suffix of ξ that appear within a lookahead interval, usually k, ahead of i in l, in any order. If the accuracy of ξ falls below a threshold, it may be because errors occurred at the end of ξ's suffix. If so, the length of ξ is decreased and ξ is re-rated. Frequency of occurrence is then updated for all contributing unit patterns.

4.3.2 Adaptation

Each time an application runs, it can reveal more of its total access pattern. To recognize new parts of an access pattern, the algorithm again scans the training trace, shifting it through a sample window of unit pattern size, matching each sample against pattern memory, and skipping ahead by σ when an equivalence is found.

Any subsequences of unit pattern length that do not fall within an existing equivalence class and which were not predicted well during credit assignment are added as new unit patterns to pattern memory, timestamped, and rated. All unit patterns are then ranked according to rating and length. Unit patterns with the lowest ratings and shortest lengths are pruned, until pattern memory fits within space allotted to it. The pattern memory is then ready to be saved or to be used for prediction mode.

4.4 Prediction Mode

The task of prediction mode is to quickly recognize similarities between the current access sequence sample and stored unit pattern prefixes, and combine their associated suffixes. During a session, the sampling window contents are given to the prophet's PREDICT routine on each access. PREDICT finds the nearest neighbors of the sample. One can think of prediction as a navigation through pattern space. Training initially overlaps successive unit patterns, and when re-training does not change these links, consumption of one prefetch request generates a match with the next unit pattern prefix. In this example, access to efg in the suffix of ξ_1 matches the prefixes of ξ_2 and ξ_4.

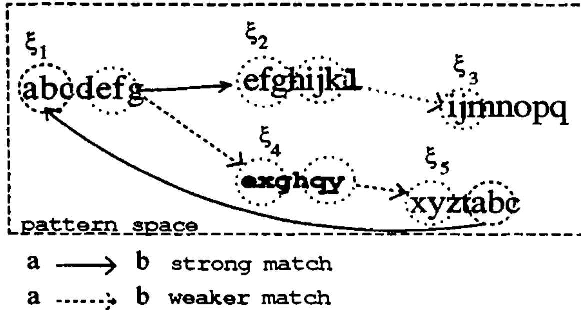

Figure 4: Linked Unit Patterns

PREDICT constructs an ordered union of suffixes as the prediction, Π. As it copies the suffix of each ξ to Π, it uses the rating of each unit pattern to place UIDs from the "best" ξ at the head of Π, and avoids duplicates. Thus, UIDs of multiple unit pattern suffixes are interleaved in the prediction output, with UIDs from the best matches and predictors appearing first. The following example shows **efg** matching unit patterns ξ_2 and ξ_4 above, causing an interleaving of their suffixes into Π.

SAMPLE:	e f g	
ξ_2	e f g h i j k l	$\chi = .4$
ξ_4	e x g h q y	$\chi = .3$
OUTPUT, Π:	h i q j y k l	

Note that the prophet finds multiple matches for a sample. Fido's model of prediction permits parallel possibilities. Since cache memory is cheaper than I/O time [Gray 1987], Fido spends cache space to save I/O, prefetching alternate possibilities (limited to a constant factor) simultaneously into P. For example, suppose that after sequence **e f g h**, an access to **i** is .4 likely, but an access to **q** is .3 likely. Fido's prefetch request includes both **i** and **q**, giving a combined hit probability of .7 - and assuming that one of **i** or **q** will go unused. MLP replacement then quickly reclaims space wasted by erroneous prefetches by evicting unused prefetches first.

5 Experiments

We have been experimenting with predictive caching, using Fido as a framework for exploration. Our first simulations examined aspects of prediction, prefetch and faulting behaviors, and we are using the results to fit predictive cache operations to the actual I/O subsystem.

5.1 Resilience to Noise

Other users make unpredictable updates to a database, changing the set of UIDs to be learned and predicted. One measure of prediction performance is the rate at which prediction accuracy degrades as updates increasingly disrupt pattern recognition. UID changes appear as "noise" in the access sequence during prediction. While the update rate of OODB applications is slower than for transaction processing, it is reasonable to expect[1] that 10 or 20 percent of the UIDs could change between sessions. Experiments with an early (also nearest-neighbor) pattern memory [Palmer 1990] revealed a property of resilience to create/delete noise. One experiment ran as follows. An access simulator produced a string, *l*, of 600 random UIDs, used to train the prophet and produce a pattern memory. The following process was repeated until the original *l* contained 30% noise:

1. Mutate 2% of *l* by deleting or inserting new UIDs at uniformly random points, maintaining *l*'s length
2. Simulate prediction along *l* without first re-training, then plot final guess rate and accuracy against total percent noise in *l*.

We observed that accuracy and guess rate degraded linearly as noise increased, that guess rate declined more quickly than accuracy, and that the relative rates could be varied by adjusting sample size, i.e. - efficiency. The result appears in Figure 5.

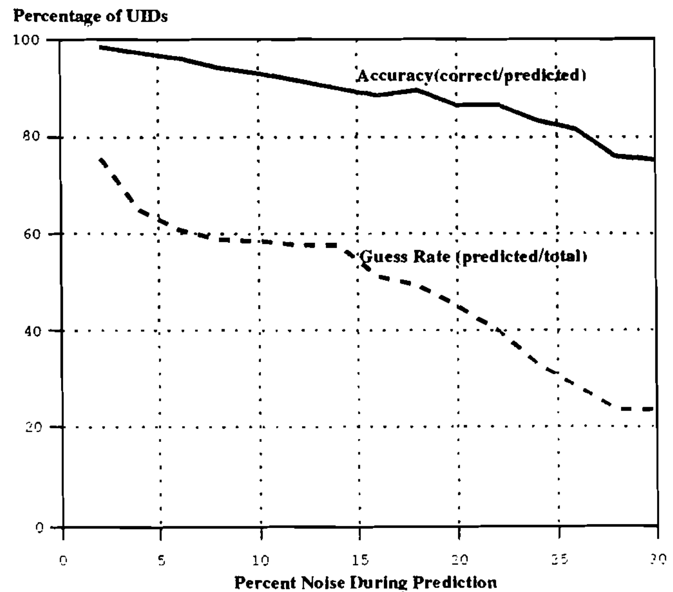

Figure 5: Prediction Accuracy and Guess Rate vs. Noise

[1] based on conversations with Digital CAD tool developers

5.2 Predictive Cache Simulation

We next wished to simulate the functioning of training and predictive caching using an access trace. Developers at Digital's CAD/CAM Technology Center provided virtual address reference traces from a CAD tool that seemed to perform navigational access, spending most of its time waiting to fault through a graphics display structure, especially during invocation, when the whole design was displayed. Two traces were obtained, each recording 5 to 10 minutes of tool use: invocation, zoom in and out, selecting ICs, and setting filters to remove certain parts of the board display (runs and junctions). In observing the display, possibilities for establishing distinct access contexts became obvious, but we treated the tool as a black box generator, using a single access context for training and prediction. The circuit design data contained 100,000 objects, but only 10,000 or so could fit in the graphics "usable window" at once. The first trace, T1, had 73,767 identifiers and T2 had 147,345. The first session was kept short, so we could notice the effect that training after a first short session had on caching during the next, longer session.

We simulated a Fido cache of 500 elements handling faults from an ENCORE demand LRU cache. Recall that Observer allows a segment of objects to be prefetched in response to a read, or up to k identified objects to be returned in a single prefetch request, either way saving k-1 network I/Os. However, the server may complete a segment fetch faster that its equivalent prefetch request. We wanted to isolate effects due only to prediction, so we did not simulate segment prefetch in the LRU cache and made no assumptions about service rates for prefetch requests or segment prefetch.

Placing Fido below a primary client cache would ensure that prophet computation only occurred during primary cache faults, incurring no prediction overhead for hits to the LRU cache. Prophet computation would begin after and complete well before each Fido fault, while each hit in the Fido cache would save one fault I/O at the cost of at most one prediction computation.

One question was whether cache space spent storing a pattern memory would pay for itself or would be better spent increasing LRU cache size. To find out, we trained the prophet on session T1's fault sequence, measuring growth of pattern memory, then used the result to predictively cache the next session's fault sequence. First, we simulated a 1000-element LRU cache using T1, to produce sequence LRU-1000(T1). This fault sequence was about 21% shorter than T1, evincing some re-use. LRU-1000(T1) was then input to train Fido. The sampling window size was α=5, with unit pattern size of 250. The equivalence radius used during training was .4 - i.e., 60% of the UIDs in two unit patterns had to differ before they were considered distinct.

During training, pattern memory grew in steps but the fault total grew linearly. During learning plateaus, few new patterns were being found - indicating recurring sequences in the fault trace. The result is shown in Figure 6 (pattern memory size estimated in identifiers).

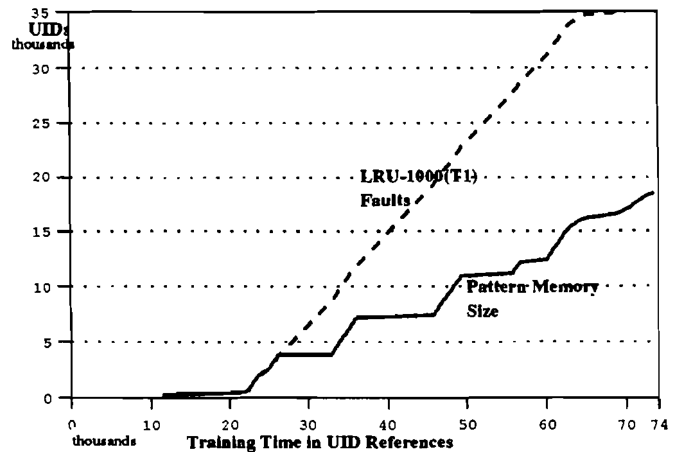

Figure 6: Pattern Memory Growth During Training

Next, we compared the faulting performance of the LRU-1000/Fido-500 combination to that of a strict LRU-2000 cache for session T2. If object sizes averaged 140 bytes and UIDs had 32 bits, the LRU-2000 would occupy about the same space as the LRU/Fido cache, including space to store the pattern memory from T1 in the Fido cache, as in the following diagram:

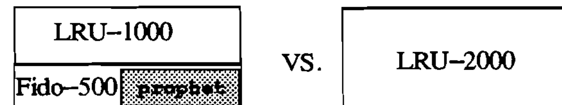

Figure 7: Configurations Requiring Similar Space

We measured faults accumulating, looking for the effects of prefetching. An LRU-2000 cache was simulated with trace T2, producing fault sequence LRU-2000(T2). The accumulation of LRU-2000 and LRU-1000 faults is compared to LRU-1000/Fido-500 faults during session T2 below:

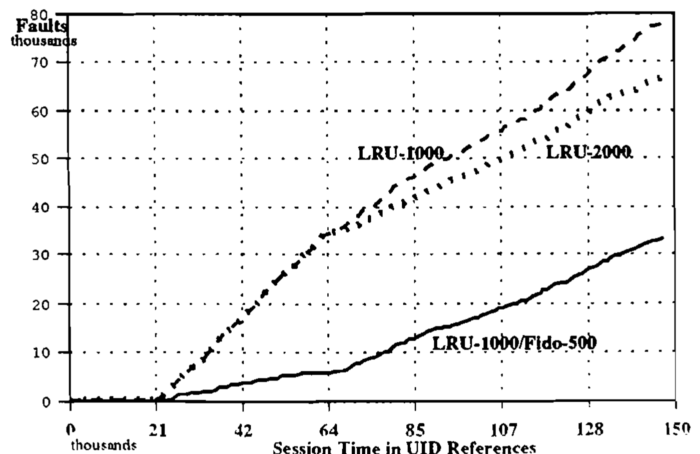

Figure 8: Fault Accumulation per Client Configuration

The faulting behavior was as we had hoped. LRU-1000 fault sequences learned from session T1 reappeared in session T2 and were prefetched, suppressing faults. This was particularly noticeable during the first part of T2 (Figure 9 provides a closer view). After one training run. we had reduced total faults by about half. Places where faults rise at the same rate as LRU faults indicate sequences not yet known to the prophet .

6 Related Work

Previous adaptive database work in the mid-1970s ([Niamir 78], [Chan 1976], [Hammer 1976]) explored methods of automatically adapting the physical and secondary access structures of databases according to use. These methods differ from ours in that they require running statistical analysis procedures periodically to reconfigure a schema or to choose indices. The structures did not adapt to changing usage patterns between reconfigurations, and data was unavailable during reconfiguration. The statistical procedures required customization to handle each database schema, resulting in brittleness and lack of generality. These adaptive mechanisms were not transparent to the database administrator, and attempted to optimize over all users rather than for each access context individually.

Although the current prophet is not implemented as a neural net, we are interested in using appropriate learning technology to detect and exploit regularities in the operation of low-level system components. When the best way of exploiting a context-specific pattern is unknown, approximate solutions such as those developed by neural models can have high payback.

Recent research in neural networks has produced self-organizing systems that are less ad hoc, more robust, and better understood than previously, enabling their use for adaptive database work. Computational models of cognition [Anderson 1990] provide a rich set of tools to "make sense" out of patterns, forming internal representations during learning [Rumelhart 1986], [Lapedes 1987]. The concept of solving a problem by using a black box that "programs itself" to produce a desired behavior from examples of the behavior is known as extensional programming [Cottrell 1988]. In Fido's case, the desired behavior is sequence prediction, but we are not yet sure of an optimal algorithm for it, making extensional programming attractive. [Moody 1989] discusses other potentially useful models. Hardware for parallel associative memory [Potter 1987] presents the possibility of vastly increasing the pattern space that can be processed on-line. Much more computational power is now available to learning algorithms. Neural models are also quite robust, adapting and functioning well in the presence of noise. Lastly, since these models operate as black boxes, they are inherently non-invasive in observing system interactions.

Our use of associative memory for prediction is not new. [Kanerva 1988] describes a *k-fold Memory* able to predict events generated by *k*th-order stochastic processes (e.g. a Markov process is a 1st-order stochastic process). A Kanerva memory addresses words by content, storing a pattern ξ_t at location ξ_{t-1} to represent order-1 transitions.

In the area of priority-based buffer management, [Jauhari 1990], [Chou 1985], [Alonso 1990] have used access pattern information to manage buffers, but use it chiefly to make replacement decisions for demand paging schemes, not directly for prefetching. Typically, qualitative "hints" about page priority are supplied, not detailed information about expected access order.

7 Conclusions

We began by noting a conceptual conflict between data clustering and data sharing, then introduced a prefetching method that promises to improve response time performance for conflicting but regular access requirements. We described a pattern memory for predicting sequences that has well-defined resource costs and scaling properties, adapts to changes in access pattern and data, and improves in prediction accuracy over time. This should result in client response time performance that improves over time, depending on how prefetch request processing actually maps to I/O. Since no semantics are involved in processing strings of identifiers, many variations are possible. For example, Fido might be used in both server and client to prefetch pages or clusters by using appropriate identifiers during training and prediction. Although predictive caching can prefetch independently of clustering, it can complement rather than replace clustering. In fact, a trained prophet should be able to provide useful clustering hints.

Fido's current pattern memory handles noise and inexact input, and learns to predict navigational access well. Navigational access can cause severe problems for a demand cache. For example, the fault trace produced by a demand cache of size *k* for a sequence *l* that is non-repeating within *k* references equals *l*. Such patterns may be fairly common in data-intensive applications. [Chang 1989] observed that the access patterns of CAD tools they studied were "predictable". Navigational access predominates in design applications, occurring during verification scans and during complex object expansion. However, navigation is often performed in a deterministic manner, resulting in bursts of non-repeating access. The benchmark in [DeWitt 1990] employs a "scan query" that reads all complex objects in a set, using breadth-first traversal to expand each complex object. Presumably this benchmark could produce the same sequences over time by expanding complex objects in the same way each time.

Predictive caching is a very promising method. Fido automatically assimilates and isolates context-specific access order regularities and exploits this information to avoid I/O. Our early results suggest that faults saved are worth the costs of maintaining access pattern information and retrieving it on-line. We hope to continue studying predictive optimality and efficiency, and to add a predictive cache to a real system.

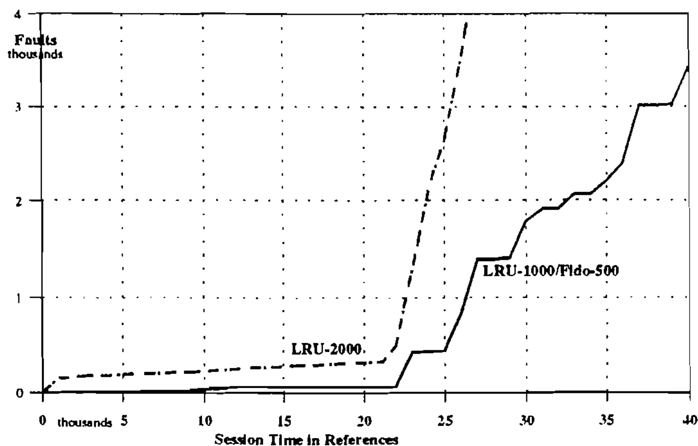

Figure 9: Fault Suppression Early in Session T2

7.0.1 Acknowledgments

The authors are very grateful for the help and patience of Patrice Tegan, Marian Nodine, Jim Anderson, and Dave Langworthy in reading paper drafts. Also, thanks to H.C. Wu of Digital's Chelmsford CAD/CAM Technology center, who obtained the CAD tool traces, to Bob Weir and Barnacle for inspiration, and to Digital's Graduate Engineering Education Program for making it possible. Views expressed herein are the authors' own, not those of Digital Equipment Corporation.

7.0.2 References

[Alonso 1990] R. Alonso, D. Barbara, H. Garcia-Molina, "Data Caching Issues in an Information Retrieval System," ACM Transactions on Database Systems, Vol. 15, No. 3, September 1990.

[Anderson 1990] J. Anderson, E. Rosenfeld, Neurocomputing. MIT Press, 1988.

[Chan 1976] A. Chan, "Index Selection in a Self-Adaptive Relational Data Base Management System," SM Dissertation, MIT, September 1976.

[Chan 1982] A. Chan, A. Danberg, S. Fox, W. Lin, A. Nori, and D. Ries, "Storage and Access Structures to Support a Semantic Data Model," Proceedings of the 8th Conference on VLDB, September 1982.

[Chang 1989] E. Chang, R. Katz, "Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS," Proceedings ACM-SIGMOD International Conference on Management of Data, Portland, OR, June 1989.

[Chou 1985] H. Chou, D. DeWitt, "An Evaluation of Buffer Management Strategies for Relational Database Systems," Proceedings of the 11th VLDB Conference, Stockholm, Sweden August 1985.

[Cottrell 1988] G. Cottrell, P. Munro, D. Zipser, "Image Compression by Back Propagation: an Example of Extensional Programming," Advances in Cognitive Science, Vol 3, Norwood, NJ, 1988.

[DeWitt 1990] D. DeWitt, D. Maier, "A Study of Three Alternative Workstation-Server Architectures for Object Oriented Database Systems," Proceedings of the 16th VLDB Conference, Brisbane, 1990.

[Fernandez 1990] M. Fernandez, A. Ewald, S. Zdonik, "ObServer: A Storage System for Object-Oriented Applications," Tech Report CS-90-27, Brown University, November 1990.

[Garza 1988] J. Garza, H. Chou, W. Kim, D. Woelk, "ORION Object Server - Architecture and Experiences," MCC TR ACA-ST-423-88, 1988.

[Gray 1987] J. Gray, "The 5 minute Rule for Trading Memory for Disc Accesses and the 10 Byte Rule for Trading Memory for CPU Time," Proceedings ACM-SIGMOD International Conference on Management of Data, San Francisco CA, May 1987.

[Hammer 1976] M. Hammer, A. Chan, "Acquisition and Utilization of Access Patterns in Relational Data Base Implementation," Pattern Recognition and Artificial Intelligence, Academic Press, 1976.

[Hudson 1990] S. Hudson, R. King, "Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System," ACM Transactions on Database Systems, 1990.

[Jauhari 1990] R. Jauhari, M. Carey, M. Livny, "Priority-Hints: An Algorithm for Priority-Based Buffer Management," Proceedings of the 16th VLDB Conference, Brisbane, 1990.

[Kanerva 1988] P. Kanerva, Sparse Distributed Memory. MIT Press, 1988.

[Khoshafian 1986] S. Khoshafian, G. Copeland, "Object Identity," ACM Proceedings on the Conference on Object-Oriented Programming Systems, Languages, and Applications, Portland, OR, 1986.

[Lapedes 1987] A. Lapedes, R. Farber, "Nonlinear Signal Processing Using Neural Networks: Prediction and System Modelling," Tech Report, Los Alamos National Lab Theoretical Division, July 1987.

[Mattson 1970] R. Mattson, J. Gecsei, D. Slutz, I. Traiger, "Evaluation Techniques for Storage Hierarchies," IBM Systems Journal 9, 1970.

[Moss 1990] J.Moss, "Design of the Mneme Persistent Object Store," ACM Transactions on Information Systems, Vol 8 No. 2, April 1990.

[Moody 1989] J. Moody, "Fast Learning in Multiresolution Hierarchies," Advances in Neural Information Processing, Morgan-Kaufmann, Los Altos, CA, 1989.

[Niamir 1978] B. Niamir, "Attribute Partitioning in a Self-Adaptive Relational Database System," MS thesis, MIT/LCS/TR-192, January 1978.

[Palmer 1990] M. Palmer, S. Zdonik, "Predictive Caching," Tech Report CS-90-29, Brown University, November 1990.

[Potter 1987] T. Potter, "Storing and Retrieving Data in a Parallel Distributed Memory System," PhD Thesis, State University of New York at Binghamton, 1987.

[Rubenstien 1987] W.Rubenstien, M. Kubicar, R. Cattell, "Benchmarking Simple Database Operations," Proceedings ACM-SIGMOD International Conference on Management of Data, San Francisco, May 1987.

[Rumelhart 1986] D. Rumelhart, G.. Hinton, R. Williams, "Learning Internal Representations by Error Propagation," Parallel Distributed Processing Vol 1: Explorations in the Microstructure of Cognition, MIT Press, 1986.

[Stamos 1984] J. Stamos, "Static Grouping of Small Objects to Enhance Performance of a Paged Virtual Memory," ACM Transactions on Computer Systems, Vol 2, No. 2, May 1984.

[Tarjan 1985] D. Sleator, R. Tarjan, "Amortized Efficiency of List Update and Paging Rules," Communications of the ACM 28, February 1985.

[Velez 1989] F. Velez, G. Bernard, and V. Darnis, "The O_2 Object Manager: An Overview," Proceedings of the 15th International Conference on VLDB, Amsterdam, 1989.

[Wilkinson 1990] K. Wilkinson, M. Neimat, "Maintaining Consistency of Client-Cached Data," Proceedings of the 16th VLDB Conference, Brisbane, 1990.