Fido: A Cache That Learns To Fetch

Mark Palmer
Stanley B. Zdonik

Department of Computer Science
Brown University
Providence, Rhode Island 02912

Technical Report No. CS-91-15
February 1991

Fido: a Cache that Learns to Fetch

Mark Palmer[1]
Stanley B. Zdonik

Brown University, Providence RI 02912

Keywords: OODB Performance, Buffering, Prefetching, Clustering, Adaptive Databases, Neural Nets

Abstract

Accurately fetching data objects or pages in advance of their use is a powerful means of improving performance, but this capability has been difficult to realize. Current OODBs maintain object caches that employ fetch and replacement policies derived from those used for virtual memory demand paging. These policies usually assume no knowledge of the future. Object cache managers often employ demand fetching combined with data clustering to effect prefetching, but cluster prefetching can be ineffective when the access patterns serviced are incompatible.

This paper describes Fido, an experimental *predictive cache* [PZ90] that predicts access for individuals during a session by employing an associative memory to assimilate regularities in the access pattern of an individual over time. By dint of continual training, the associative memory adapts to changes in the database and in the user's access pattern, enabling on-line access predictions for prefetching. We discuss two salient components of Fido:

1. MLP, a replacement policy for managing prefetched objects
2. Estimating Prophet, an associative memory that recognizes patterns in access sequences adaptively over time and provides on-line predictions used for prefetching.

We then present some early simulation results which suggest that predictive caching works well, especially for sequential access patterns, and conclude that predictive caching holds great promise.

1. Introduction

A major performance factor for current OODB systems is the cost of fetching objects from secondary storage into a local cache 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. Given the high relative cost of performing I/O, an ability to anticipate data movement across communication links and prefetch objects presents a potent means of improving performance.

Data caching in a workstation-server architecture improves performance, as described by Cattell et al [RKC87]. Chang and Katz [CK89] found that cache management policy has the largest effect on response time, followed by clustering, a topic of much interest [C+82],[S84],[HK90]. Clustering creates for data a locality of reference like that present in code, the property exploited by virtual memory managers.

However, a major purpose of databases is to allow data sharing. Clustering 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 finding an optimal partitioning can be computationally infeasible [N78].

We are finding that access patterns within isolated contexts are often predictable because of structure in data and determinism in programs. Similarities between sessions can be recognized and exploited to achieve accurate prefetching.

[1] now at Digital Equipment Corporation, CAD/CAM Technology Center, Chelmsford MA. Enet: palmer@cadse.dec.com

Predictive Caching

Predictive caching promises a more accurate means of prefetching. A predictive cache can supplement cache management strategies such as LRU or function as a primary cache. Fido's predictive cache model is a simple extension of common demand fetching strategies. A client cache manager is augmented with a "black box" component known as the Estimating Prophet. The prophet learns access patterns in 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 are used by the predictive cache manager to prefetch data. Each prediction indicates an 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, and tailors its predictions in each isolated 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 adjusts its predictions to reflect changes in usage quickly, incrementally, and in a uniform way.

Paper Structure

This paper introduces some concepts and terminology useful as a framework for studying predictive caching, and discusses the design issues encountered as well as properties we have discovered by experimenting with Fido. It does not present analysis of prediction optimality in all possible worlds.

Our current research focuses on two topics. The first topic concerns how to best incorporate predictive prefetch activity in cache management, given the costs of prediction and prefetching. The second topic involves studying various designs for the prediction component. Accordingly, this paper presents a a) model for predictive cache management and b) an associative memory design for a prophet that recognizes and predicts access sequences on-line.

Section 2 describes the way Fido fits into a typical OODB architecture. In section 3, a cache model for managing prefetches is presented. An method for recognizing and predicting access patterns is presented in section 4. Section 5 describes some interesting experiments in 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 validation and synchronization of cached objects against replicas in other caches and on secondary storage. However, these issues have also been addressed by others (e.g. [ABG90]. [N90], [G+88]), 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 ([RKC87], [CK89]). The operation of predictive caching per se does not rely on these simplifications. Predictive caching is most useful to distributed applications that:

- are data intensive, with high read/write ratios
- have structurally constrained or regular access patterns for which a good clustering is elusive
- 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 daa shared between applications

running independently on workstations. Several systems, such as Cactis [HK90], O_2 [VBD89], ORION-1SX [G+88], and Mneme [M90] are similar in this respect. In [DM90], the performance of several such variations is compared. The system to which we are adding Fido is Observer/ENCORE [FEZ90]. The primary components of this architecture are the database server, the client, and the prophet.

Database Server Observer acts as a typeless back end to applications, managing all access to the secondary storage of the database. Observer maintains strong identity [KC86]. This property greatly aids predictive caching, because the preservation of identity simplifies recognition of regularly reoccurring parts of an access pattern. Object identity in Observer is provided via an external unique identifier (UID) which acts as an immutable handle for an object and is not recycled. Some OODBs assign meanings to individual bits of UIDs, however these are not of concern to the prophet, which interprets all UIDs as symbols. In Observer, objects can be clustered into segments and can migrate and be replicated between segments. Observer usually prefetches clusters called segments in response to a read, but this feature is not used when predictive prefetching is in progress.

To facilitate prefetching, the basic Observer read function is extended in several ways. A requester marks reads for either *demand* or *prefetch* processing. The read message supplies a list of UIDs to the server, which gets the identified objects from disk or a server-managed buffer and returns them to the requester. Observer ensures that outstanding prefetch requests do not delay demand requests by providing a *pre-emptive read* operation, for demand requests, that is serviced before other reads.

Client The workstation application interfaces to the server via the ENCORE client component, which acts as an interpreter, mapping the data model used by the application onto operations understandable by bserver. ENCORE implements object type semantics, executing methods and enforcing encapsulation, and is typically bound into the application's image. The client 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. It is the performance of this function that is greatly improved by prefetching. 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. These systems take various approaches to translating object references to memory addresses, but a common feature is that the application references each object via its UID. The client then uses a Resident Object Table (ROT) to obtain a pointer to the object's location in memory, and may "swizzle" the ROT entry by adding a pointer to it. The client also interfaces to the prophet component.

Estimating Prophet The job of a prophet component is to predict the next references to be made during a user session. The prophet can be configured as a separate process or embedded in the client image. It has two primary modes of operation: *training* and *prediction*. Given a small sample of the current access sequence, the prophet predicts which objects will be requested next. An individual prediction may indicate that alternate sequences are anticipated, by arranging identifiers according to expected order and likelihood of access. The prophet "learns" access patterns over time via training and becomes increasingly better at predicting accesses, 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 information the prophet stores about a single access pattern context is known as a *pattern memory*.

The prophet isolates access patterns specific to individual *contexts*. It provides context identifiers (CIDs) as a handle to associate patterns generated by the same source. A programmer can assign a CID to indicate the unique combination of user, application, and function that generated a particular access sequence. For example, an ECAD application might provide one function that graphically displays a circuit design, and another function to allow 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. An applicaton might establish different contexts corresponding to these two functions, even if invoked by the same user. With contexts, a designer can use knowledge

of an application to "focus attention" of the prophet, preventing conflicting access patterns from interacting in pattern memory. Contexts limit pattern memory size, speeding prediction. The prophet stores a pattern memory for each CID between sessions.

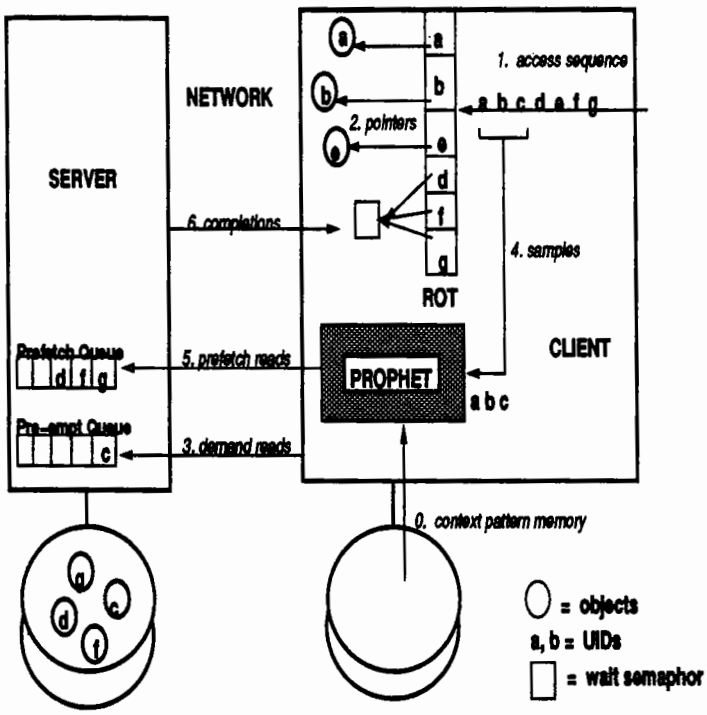


Figure 1 - Client, Server, and Prophet

Figure 1 shows how the prophet interacts with a client and server. UIDs are represented as letters and objects as circles. As the application begins a session, the prophet loads in (0) the pattern memory specific to the context. The application makes a sequence of references (1) to "**a b c** ...", handled by the client cache manager. The client converts the UIDs to memory addresses of objects by looking into the ROT. References to **a** and **b** return pointers (2) to those objects in the local cache. If an object (e.g. **c**) is accessed but is not in cache, the client issues a preemptive 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, the client 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, ROT entries implement an order that governs replacement decisions.

3. Predictive Cache 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 cache management model for use with predictive prefetching.

Simulation has been helpful in understanding issues in designing a predictive cache, that include:

- an appropriate model for prediction
- ideal case operation, when many correct predictions are being generated
- faulting and space utilization behavior in the worst case, when no (or many, incorrect) predictions are made
- interactions between cache objects when a prefetch is accessed, on faults, and during eviction/replacement decisions
- computation and communication overhead for prediction and prefetching.

As illustrated in figure 1, Fido maintains a FIFO sampling window through which the current access sequence passes. On each reference to an object *o*, if *o* is not in cache, a demand read (fault) is started. Fido then gives the immediate past sequence history to the prophet, which may return an estimate of the immediate future. If so, Fido checks whether any predicted objects are already in cache, and begins a prefetch read for those which are not. When prefetched or demanded objects arrive, an asynchronous completion routine decides which objects to replace. The overall strategy of predictive cache management involves three goals:

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

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

Definitions

We use the following terms to define a predictive cache:

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

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.

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 o_i is expected before an access to 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. The prediction size, $\omega < p_{max}$, is ideally the cache size.

Prefetch request
A prefetch request π_i is the order-maintained subset of objects identified in prediction Π_i which are not in C at the time the prediction is made. The "head" of the prefetch request is the object most likely to be accessed first, while the "tail" is the object whose access is expected furthest in the future and is least likely: $\pi_i = \{\Pi_i - C\}$

MLP Replacement Policy
A cache manager decides which objects to replace with new objects by using 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.
- Our definition of P implies that old prefetch requests are evicted before new prefetch requests.
- On access to object *o*, promote *o* to the list head, i.e. - use the move-to-front (MF) rule [ST85], or remove *o* if *o* is consumed (i.e. - copied out).

The MLP eviction rule is adapted from the proven optimal replacement policy for demand paging, OPT [S90]. 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 *forseen* future. The main difference between MLP and OPT is that MLP uses *estimated, incremental* knowledge of the future, instead of the perfect prescience assumed by OPT.

Operation

We outline a simpler version of the algorithm presented in [PZ90] here. The ROT for C is modeled as a fixed size list that is scanned on reference. For efficiency, scans begin at the head of the most recent prefetch request. Items are inserted at the head and deleted from the tail. This behavior governs how objects are brought into and evicted from C, whether they be prefetch requests or 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 order. The object whose access is expected furthest in the future - and is least likely - enters and is evicted from C first. The figure below illustrates reference processing.

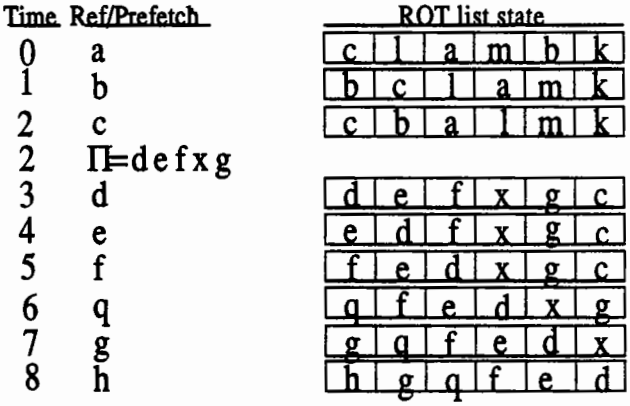


Fig 2: Processing Prefetch Requests

In this example, references are in the left column, with the ROT shown at right. UIDs are added at the head (left) of the list and removed from the tail. By time 2, references to **a**, **b**, and **c** cause reordering. and the sampling window contents, **abc**, match a known pattern, triggering a the prefetch request **defxg**. The prefetch request's arrival causes eviction of everything but **c**. At 7, an access to **q** (instead of **x**) faults, replacing **c** with **q**. At 8, **g** is moved up, leaving **x** to be evicted by the fault for **h**.

To characterize the operation of the predictive cache we briefly describe three cases:

1: Sequence Recognition (best case)

A prediction is made every p_{max} accesses. Each prediction request arrives, fills the list, and is then consumed from head to tail. This has the end effect of moving Π from P to R, and move-to-front leaves the list containing Π in reversed order.

2: Prediction Starvation

Within a session, long intervals occur during which the current sequence is unknown to the prophet, which then generates no prefetch requests. In this situation, each access to C results in a demand fault. Since each demanded object is inserted to the list head, the cache operates as an LRU cache using the move-to-front rule in the absence of predictions.

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.

Fido adjusts several parameters as a heuristic to control cases 2 and 3 above:

- Guess rate : The ratio of prefetches retrieved divided by the number of references made at a given time during the processing of a sequence is denoted λ.
- Accuracy: χ denotes the ratio of correct predictions to total predictions at any given time
- 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 predictions.

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

4. Estimating Prophet

Previous sections of this paper have outlined a model for managing a cache with prefetched objects, but have assumed the ability to predict references. This section presents a design intended to illustrate how a prophet can learn 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 context. The prophet training algorithm processes each reference trace - normally (but not necessarily) offline, between sessions, and incrementally improves pattern memory for that 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 analagous 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 partial information about the present sequence as a key, or address. We will first discuss this inexact retrieval capability, and then outline its use in training and prediction.

4.1 Pattern Memory

The prophet stores and retrieves access order infomation in an inexact manner by using a *nearest-neighbor* associative memory. Nearest-neighbor models map data units with k elements to points in k-dimensional space, defining similarity metrics in k dimensions. Similar patterns are near each other in this pattern space, and equivalence classes are defined by hypersphere radius. Much work has been done on architectures for associative memory [P87], [K88], some of which provide the most biologically plausible neural net models. Research by Anderson and others has shown how such memories can be constructed using elements that imitate the functioning of neurons in cerebral cortex [AR90], and evidence suggests that cognition may indeed operate this way.

Although we are exploring several designs for a fast pattern memory, the following concepts are generally useful for the nearest-neighbor models:

Definitions:

Pattern Memory
A pattern memory consists of t *unit patterns*, $\xi_0, ..., \xi_t$, that are at least r distant in pattern space.

Unit Pattern
A unit pattern $\xi=<o_1...o_\sigma>$ is a list of UIDs that acts as a partial, approximate representation of the access pattern. 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. Each unit pattern is divided into a *prefix* of size α and *suffix* of size ω. The prefix acts as a key for the suffix during prediction, and a suffix usually contains a prefix of some other unit pattern, creating inexactly linked chains of unit patterns to represent approximate sequences [see figure 4]. Also stored with each unit pattern are ratings of its historical likelihood and predictive accuracy.

Distance:
The measure of distance between two unit patterns is equal to the number of columnwise mismatched UIDs.

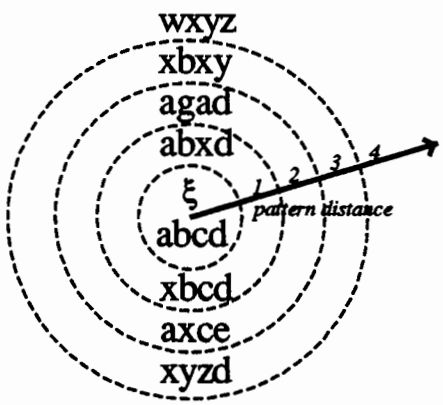


Figure 3: One unit pattern, ξ and a set of sample sequences, arranged according to increasing distance from ξ. The nearest neighbors of a sample S are the unit patterns closest to S in pattern space. All samples closer than r are considered equal to ξ

The current prophet implementation is described in [PZ90]. We recall two of its properties here:

- 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 f length k can be found in $O(k \log(t) + f)$ where f is an expected number of neighbors.

4.2 Training Mode

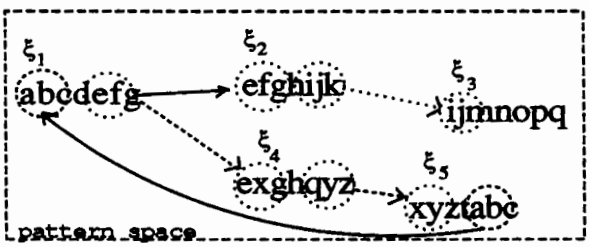
After each session, the client saves a reference trace for the context, then invokes the prophet's training algorithm to process this training trace. The training algorithm adapts the contents of pattern memory over time so that only common unit patterns are retained in pattern memory. Training reinforces 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 predict. This causes pattern memory to focus on regularly reoccuring phenomena and to evolve an internal prototype, or generalization, of each access pattern. The training algorithm employs an "evolutionary" strategy consisting of two phases - credit assignment and adaptation.

Credit Assignment
This phase assigns credit to unit patterns involved in predicting the training trace. Some unit patterns appear infrequently. Updates to the database cause new UIDs to appear and others to disappear, invalidating other unit patterns. Also, the sampling boundaries of unit patterns are initially arbitrary. Infrequent, obsolete, or poorly chosen samples produce unit patterns that do not function well as predictors, but which might occupy pattern memory, were it not for selection. Credit assignment proceeds by simulating a prediction run on the training trace. Each time ξ contributes to a prediction at point i in the training trace l, its prediction accuracy and frequency ratings are updated. Accuracy, χ, is assessed for ξ by counting the number of UIDs in the suffix of ξ that appear within a lookahead interval, usually p_{max}, ahead of i in l, in *any position*. Note that this method of rating removes positional sensitivity from the rating. ξ gets the same χ rating for any of $\omega!$ possible permutations of its suffix in the lookahead interval.

Adaptation
Each time an application runs, it reveals more of its total access pattern. To recognize new parts of an access pattern, the training algorithm again scans the training trace. Each portion is matched against the pattern memory, and any sample that does not fall within an existing equivalence class and which was not predicted well during the selection phase is added to pattern memory, timestamped, and rated. All unit patterns are then ranked according to rating, and patterns with the lowest ratings are pruned, until pattern memory fits within its allotted space.

4.3 Prediction Mode

The task of the prediction algorithm is to quickly recognize similarities between the current sequence sample and prefixes of stored unit patterns, and recall their associated suffixes. During a session, the sampling window contents are given to the prophet's PREDICT routine on each reference. PREDICT finds the nearest neighbors of the sample. One can think of prediction as navigation through pattern space. Training initially overlaps successive unit patterns, and when re-training does not change this relationship, it generally causes consumption of one prefetch request to generate a match with the next unit pattern prefix. In this example, the sample **efg** would matche the end of ξ_1, strongly matching the prefix of ξ_2, and loosely linking to ξ_4.



a ⟶ b strong match

a ⋯⋯› b weaker match

PREDICT constructs an ordered union of suffixes as the prediction, Π. As it copies the suffix of each ξ to Π, it considers the rating associated with each ξ, places UIDs from the "best" ξ at the head of Π, and eliminates 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 the sample matching unit patterns ξ_2 and ξ_4, and interleaving suffixes into Π.

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

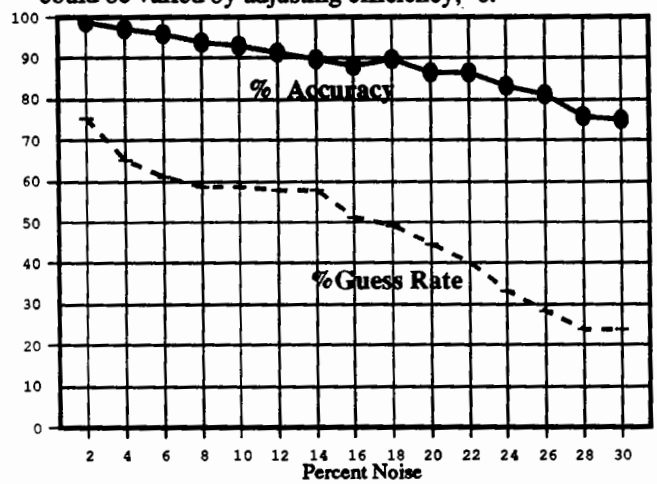
Note that the prophet uses a nearest-neighbor model to find *multiple* matches for a sample because Fido's model of prediction permits parallel possibilities. Since cache memory is cheaper than I/O time [G87], Fido spends cache space to save I/O, prefetching alternate possibilities (limited to a constant 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 prefetches both **i** and **q**, giving a combined hit probability of .7 - and "expects" that one of **i** or **q** will go unused. MLP replacement then quickly reclaims space wasted by erroneous prefetches, evicting unused prefetches first.

5. Experiments: A: Resilience to Noise

Other users make unpredictible updates to a database, causing change in the set of UIDs being learned and predicted. One measure of prediction performance is the rate at which accuracy degrades as changes increasingly disrupt pattern recognition. UID changes appear as "noise" in the access sequence during prediction. 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 objects may change between sessions. Experiments with an early (also nearest-neighbor) pattern memory [PZ90] revealed a property of resilience to create/delete noise. One experiment ran as follows: an access sequence simulator produced a single sequence of 600 UIDs, which was then used to train the prophet. The following process:

1. perform equal amounts of UID insert and deletions in 2% increments at uniformly random points in the original string, maintaining its length.
2. simulate prediction, without re-training, and plot final guess rate and accuracy at each noise level.

was repeated until the original sequence contained 30% noise. We observed that accuracy,χ, and guess rate,λ, degraded linearly with increasing noise. λ degraded more quickly than χ, and these relative rates could be varied by adjusting efficiency, υ.



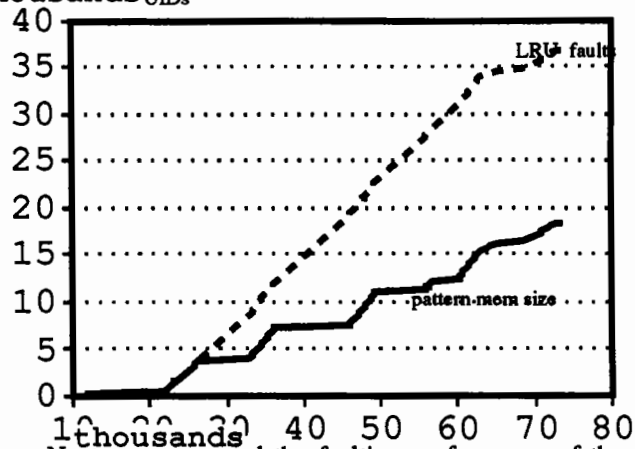
[1] based on conversations with CAD tool developers at Digital

B: Predictive Cache Simulation

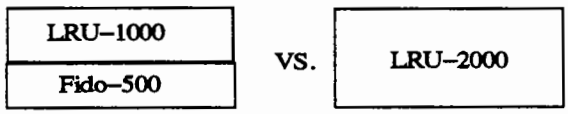
We have been experimenting with predictive caching, using Fido as a framework for exploration. Working with developers at Digital's CAD/CAM Technology Center in Chelmsford MA, we obtained reference traces from a CAD tool that seemed to perform mostly sequential access - spending most of its time waiting to fault through a graphics display list, especially during invocation, when the whole design is displayed. Two traces were obtained, each representing 5 to 10 minutes of using tool functions: invocation, zoom in and out, select ICs, and setting filters that removed certain parts of the board display (e.g. runs and junctions). In observing the display, possibilities for distinct functional contexts suggested themselves, but we treated the tool as a black box generator, and did not establish different prediction contexts. The circuit design data contained 100,000 objects, but only 10,000 or so could fit in the "usable window" at once. The first trace, T1, had 73,767 UIDs, and the second, T2, had 147,345. The first session was kept short, so we could notice the effect that training for a first short session had on caching during the next, longer session.

Using these traces, we simulated a predictive cache of 500 elements handling faults from an LRU cache of 1000 elements. One effect of this arrangement is that all prophet computation occurs on LRU faults - during time that would otherwise be spent on I/O, thus there is no computation added to normal LRU operation. We simulated this by training on the first session's fault sequence and predictively caching the next session's fault sequence, running T1 through an 1000-element LRU cache to produce sequence LRU-1000(T1). This fault sequence was about 21% shorter than the original access trace, evincing re-use. The LRU-1000(T1) sequence was then used to train, and the resulting pattern memory used to predictively cache the faults generated from trace T2. We measured faults accumulating, looking for the effects of prefetching. A sampling window of size α=5 was used, a unit pattern size of 250, and an overlap of 5. The hypersphere equivalence radius used during training was .4 - i.e., if 60% of the UIDs in two unit patterns differed, they were considered distinct.

First, we measured the growth of pattern memory during training on the LRU-1000(T1) trace. Pattern memory size is measured in UIDs, as are fault quences. The result is shown below. The pattern memory grew in steps, as the fault total grew linearly. This was encouraging - during the plateaus in learning, few new patterns were being found - indicating recurrent sequences in the trace.



Next, we compared the faulting performance of the LRU-1000/Fido combination to that of a strict LRU cache of 2000 elements - which occupied about the same space as would the LRU/Fido cache, accounting for space to store prophet pattern memory and the prefetch cache C, as in the following diagram:



The T2 trace was run through an LRU simulation of a 2000-element cache, producing LRU fault sequence LRU-2000(T2). We graphed the accumulation of LRU-2000(T2) faults against LRU-1000+Fido-500(T2) faults during the T2 session:

THE ATTACHED PAGE 12

PROVIDES THE 3 GRAPHS THAT

MUST BE SUPERIMPOSED HERE

This faulting behavior was as we had hoped. LRU fault sequences that were learned from session T1 reappeared and were predicted/prefetched, supressing faults. This is particularly noticable during the initialization sequence. After one short training run, we had reduced total faults by about half. Places where faults rise at the same rate as the LRU faults indicate sequences not yet known to the prophet.

6. Related Work

Previous work in adaptive databases ([N78], [C76], [HC76]) in the mid 1970s explored means of automatically configuring the physical and secondary access structures of databases according to usage. These solutions involved running statistical analysis procedures periodically to generate schema reconfigurations or choose indices. The physical structures did not adapt to changes in usage patterns between reconfigurations, and data was unavailable during reconfiguration. The statistical procedures required customization to each database schema, resulting in brittleness and lack of generality. The adaptive mechanisms were not transparent to the database administrator and attempted to optimize access over all users rather than for each user individually.

Several advances in technology have combined to renew possibilities for adaptive database work. First, OODBs maintain object identity, which makes learning patterns for a set of data objects much easier. Second, recent work in neural networks has produced models for self-organizing systems which are less ad hoc, more robust, and better understood than previously. These models of cognition (e.g. [AR90], [K88]) provide a rich set of tools able to "make sense" out of patterns, sometimes by forming internal representations during learning [LF87]. In addition, neural models are generally quite robust, adapting and functioning well in the presence of noise. Also, parallel hardware for associative memory [P87] presents the possibility of vastly increasing the size of pattern memory that can be processed on-line. Much more computational power is now available to perform computation for learning. Lastly, these models operate as adaptive black boxes, and so are inherently non-invasive in observing system interactions.

To date, however, neural networks have found limited practical application. One unifying goal of our work is to mate appropriate learning technology with various system components to exploit regularities in deterministic operation. In this endeavor, approximate solutions - such as those developed by neural models - can have high payback.

Our use of associative memory for prediction is not new. Kanerva [K88] describes a *k-fold Memory* able to predict events generated by *k*th-order stochastic processes. Kanerva's method addresses memory words by content, storing pattern ξ_t at location ξ_{t-1} to represent order-1 transitions.

In the area of priority-based buffer managment, [JCL90], [CD85], [ABG90] 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 on expected order of access.

7. Conclusions

We began by noting an inherent conceptual conflict between data clustering and data sharing, and introduced the concept of predictive caching to provide better response time performance for conflicting but regular access requirements. We then described the design of a simple pattern memory for predicting sequences which has well-defined resource costs and scaling properties, adapts to changes in access pattern and data over time, and improves in prediction accuracy.

Fido's current pattern memory handles noise and inexact input well, and learns to predict simple access patterns. Yet sequential patterns present the worst problems for traditional demand caching. For example, the LRU fault trace for a given non-repeating sequence equals that sequence. Such patterns may be more common in data-intensive applications than one might expect. Chang and Katz [CK89] observed that most access patterns in the CAD tools they studied were "predictable". Navigational access predominates in design applications, occuring during verification scans and during complex object expansion. However, navigation is often performed in a deterministic manner, resulting in bursts of sequential access. A benchmark in [DM90] employs a "scan query" that reads all complex objects in a set, using breadth-first traversal to expand each complex object. Presumably this benchmark would produce the same sequential patterns over time by expanding complex objects in the same way each time.

Predictive caching is a very promising technique. In a broad sense, it automatically assimilates and isolates information about context-specific access order regularities, and exploits this information to avoid I/O. Our early results indicate that the costs of maintaining this access order information and retrieving it on-line are worth the I/O saved. We plan to continue this work, studying prediction optimality, using the Fido framework to incorporate improvements to the prophet, and by implanting Fido into a commercial system - perhaps a virtual memory pager.

Acknowledgements

The author is very grateful for the help and patience of Patrice Tegan, Misty Nodine, Jim Anderson, Yi-Jing Lin, and Dave Langworthy at Brown 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 Digital's GEEP office for making it possible, and to Bob Weir and Barnacle for inspiration.

References

[ABG90] 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.

[AR90] J. Anderson, E. Rosenfeld, Neurocomputing. MIT Press, 1988.

[C76] A. Chan, "Index Selection in a Self-Adaptive Relational Data Base Management System," SM Dissertation, MIT, September 1976.

[C+82] 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, Septembr 1982.

[CD85] H. Chou, D. DeWitt, "An Evaluation of Bufer Management Strategies for Relational Database Systems," Proceedings of the 11th VLDB Conference, Stockholm, Sweden August 1985.

[CK89] E. Chang, R. Katz, "Exploiting Ineritance and Structure Semantics for Effective Clusterng and Buffering in an Object-Oriented DBMS," Proceedings ACM-SIGMOD International Conference on Management of Data, Portland, OR, June 1989.

[DM90]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.

[FEZ90] M. Fernandez, A. Ewald, S. Zdonik, "Ob-Server: A Storage System for Object-Oriented Applications," Tech Report CS-90-27, Brown University, November 1990.

[G87] 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.

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

[HC76] M. Hammer, A. Chan, "Acquisition and Utilization of Access Patterns in Relational Data Base Implementation," Pattern Recognition and Artificial Intelligence, Academic Press, 1976.

[HK90] S. Hudson, R. King, "Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System," ACM Transactions on Database Systems, 1990.

[JCL90] R. Jauhari, M. Carey, M. Livny, "Priority-Hints: An Algorithm for Priority-Based Buffer Management," Proceedings of the 16th VLDB Conference, Brisbane, 1990.

[KC86] S. Khoshafian, G. Copeland, "Object Identity". ACM Proceedings on the Conference on Object-Oriented Programming Systems, Languages, and Applications, Portland, OR, 1986.

[LF87] A. Lapedes, R. Farber, "Nonlinear Signal Processing Using Neural Networks: Prediction and System Modelling," Tech Report, Los Alamos National Lab Theoretical Division, July 1987.

[M90] J.Moss, "Design of the Mneme Persistent Object Store," ACM Transactions on Information Systems, Vol 8 No. 2, April 1990.

[N78] B. Niamir, "Attribute Partitioning in a Self-Adaptive Relational Database System". MS thesis, MIT/LCS/TR-192, January 1978.

[K88] P. Kanerva, Sparse Distributed Memory. MIT Press, 1988

[P87] T. Potter, "Storing and Retrieving Data in a Parallel Distributed Memory System," PhD Thesis, State University of New York at Binghamton, 1987.

[PZ90] M. Palmer, S. Zdonik, "Predictive Caching," Tech Report CS-90-29, Brown University, November 1990.

[RKC87] W.Rubenstien, M. Kubicar, R. Cattell, "Benchmarking Simple Database Operations", Proceedings ACM-SIGMOD International Conference on Management of Data, San Francisco, May 1987.

[S84] 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.

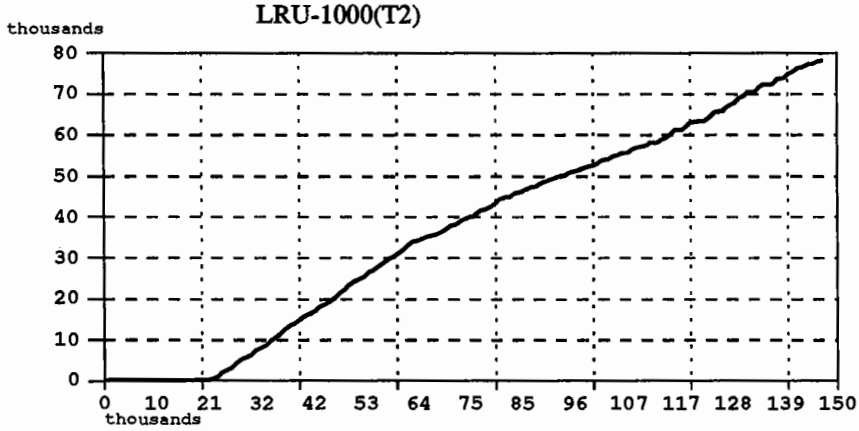
[S90] H. S. Stone, Computer Architecture, Addison-Wesley Publishing Co., 1990.

[ST85] D. Sleator, R. Tarjan, "Amortized Efficiency of List Update and Paging Rules", Communications of the ACM 28, February 1985. pur

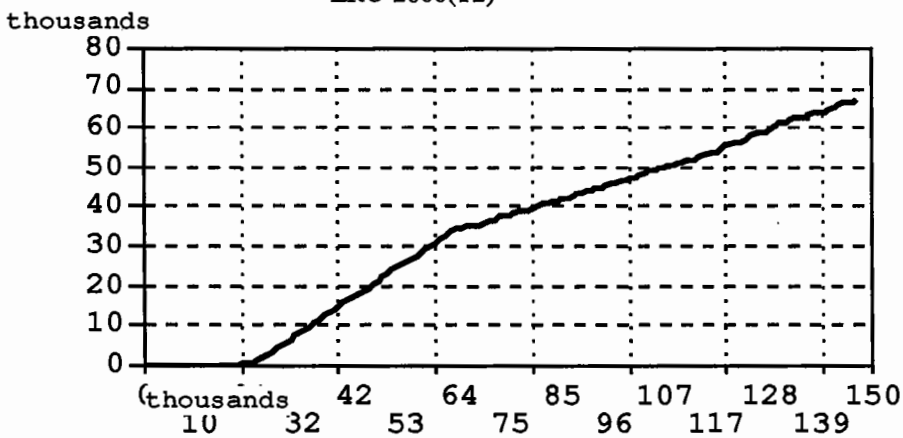
[VBD89] F. Velez, G. Bernard, and V. Darnis, "The O_2 Object Manager: An Overview," Proceedings of the 15th International Conference on VLDB, Amsterdam, 1989.

[WN90] K. Wilkinson, M. Neimat, "Maintaining Consistency of Client-Cached Data," Proceedings of the 16th VLDB Conference, Brisbane, 1990.

LRU-1000(T2)



LRU-2000(T2)



LRU-1000+Fido-500(T2)