A Strategy Selection Framework for Adaptive Prefetching in Data Visualization

Punit R. Doshi, Geraldine E. Rosario, Elke A. Rundensteiner and Matthew O. Ward
Computer Science Department, Worcester Polytechnic Institute, Worcester, MA 01609
{punitd|ger|rundenst|matt}@cs.wpi.edu *

Abstract

Accessing data stored in persistent memory represents a bottleneck for current visual exploration applications. Semantic caching of frequent queries at the client-side along with prefetching can improve performance of such systems. However, a prefetching setup that only uses one prefetching strategy may be insufficient because (1) different users have different exploration patterns, and (2) a user's pattern may be changing within the same session. To solve this, existing research focuses on refining a single prefetching strategy. We, on the other hand, now propose a novel framework wherein prefetching strategies are adaptively selected over time across and within one user session. This work is the first to study adaptive prefetching in the context of visual data exploration. Specifically, we have implemented our proposed approach within XmdvTool, a freeware visualization system for multivariate data, and evaluated it using real user traces. Our results confirm that our approach improves system performance by dynamically selecting the most appropriate combination of prefetching strategies that adapts to the user's changing patterns.

1. Introduction

1.1. Addressing Needs in Data Visualization

Whether the domain is stock market data, scientific measurements, or the distribution of sales, visualization is becoming an increasingly popular technique for data exploration. When presented with visual depictions of the data, humans can often easily detect interesting patterns as well as outliers, which may be more difficult to identify and rate as relevant with automated techniques [18]. Interactive visual navigation tools play an important role in aiding users to find their way through large data sets. Significant effort has thus been spent on developing effective methods to display and visually explore information [2, 17, 11].

Most visualization tools still execute on data that is first fetched from the file system and loaded entirely into main memory. However, as typical sizes of data sets become larger (on the order of giga-bytes or more), current data sets can no longer be held entirely in main memory. We thus must scale visual tools to work with large data sets without sacrificing the near real-time responses required to service user's navigation requests. Even a small movement in the user's navigation tool may mean executing a new query to retrieve the selected data, potentially resulting in a high data access rate. Being an interactive feedback-driven paradigm, it is critical that the user receives responses to her navigation requests with little or no time lag.

To address the opposing needs of scalability and real-time response, the XmdvTool team took on the challenge of integrating a visualization tool with a database management system. XmdvTool [26] is a public-domain tool for multivariate data visual exploration we have been developing at WPI. In [20], we first applied semantic caching techniques [5] for maintaining the client-side cache. Caching provides the advantage of allowing most frequently requested data to be kept in the cache to speed up future responses. In [7], we augmented semantic caching techniques with the prefetching of data into the cache during idle times. We investigated the use of different static prefetching strategies and showed that some form of prefetching is better than no prefetching at all. This paper now describes a continuation of our work into prefetching, this time proposing a strategy selection framework to achieve adaptive prefetching.

1.2. General Characteristics of Visual Exploration

Visual exploration packages usually exhibit the following characteristics which can be exploited for caching and prefetching purposes [7]:

- *Locality of exploration:* In general, users doing data exploration usually explore one area of a display at a time before moving on to another area. It is not common to randomly jump from one area to a remote area. This locality of exploration increases the predictability of the user's next request. Prefetching strategies can be designed to exploit this.

- *Contiguity of user movements:* User navigation operations are translated into queries to the database. Exploration using visual navigation tools such as sliders and knobs translates to consecutive queries that are contiguous. Such contiguity also increases the predictability of the user's next request.

*This work is supported under NSF grant IIS-0119276.

- *Presence of idle time*: Users usually pause to understand the display and look for patterns in the data, so there is idle time between queries to the database. Such idle times can then be exploited for background tasks such as prefetching.

These are general characteristics usually found in packages that contain display navigation tools such as sliders and knobs. The prefetching concepts we discuss in this paper exploit these general characteristics.

1.3. From Static to Adaptive Prefetching

Static prefetching strategies [7] are not tailored to changes in the user navigation patterns. Drawbacks include:

1. Static prefetchers lack a feedback mechanism. They typically generate predictions independent of their past performance on previous predictions, even if those were detrimental or non-effective.

2. Different users have varied navigation patterns. Such variations are likely due to the user's inherent navigation preference, user's familiarity with the data set, user's familiarity with the visualization tool, and the patterns present in the data. This implies that no single static prefetching strategy will work best for all types of users (verified in Section 5).

3. Even navigation patterns of a user within a single session may change as the user gains more knowledge about the data, becomes more familiar with the data visualization tool, or changes her goal of exploration. This implies that a single strategy may not be sufficient even within one user session (verified in Section 5).

To address these shortcomings, a logical next step is to make prefetching adaptive. An adaptive prefetcher changes its prediction behavior in response to a changing environment with the goal of improving performance. Specifically, we propose a strategy selection framework for prefetching wherein we can adaptively shift between prefetching strategies within a user session and find the most appropriate combination that best supports the user's changing patterns.

1.4. Contributions

The main contributions of this paper include:

- the first to study adaptive prefetching in the context of visual data exploration;

- a proposed framework for adaptive prefetching via strategy selection, as opposed to the common approach of strategy refinement; and

- empirical results showing benefits of strategy selection over a wide range of user navigation traces.

1.5. Organization

The remainder of this paper is organized as follows: Section 2 describes caching and static prefetching in XmdvTool. Section 3 outlines our adaptive prefetching approach. Section 4 describes our implementation. Section 5 describes our experiments on real user traces. Section 6 lists related work, while Section 7 concludes the paper.

2. Visualization Tool Case Study

While the concepts of adaptive prefetching we propose are general, we incorporate our proposed techniques within an actual system, XmdvTool, in order to evaluate them. The major hurdles XmdvTool overcomes are the problems of display clutter and intuitive navigation. Given that interactive exploration must support large multivariate data sets, XmdvTool has a database backend which supports caching and prefetching [20].

2.1. Structure-Based Brush in XmdvTool

We now illustrate how the characteristics listed in Section 1.2 can be exploited for caching and prefetching in XmdvTool. For this, consider Parallel Coordinates (Figure 1) as one of the displays used for depicting multivariate data sets. Structure-Based Brush (Figure 2) is one of the navigation tools we use for exploring hierarchical data sets.

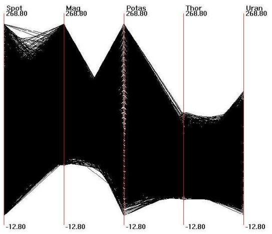

Figure 1: Cluttered Parallel Coordinates.

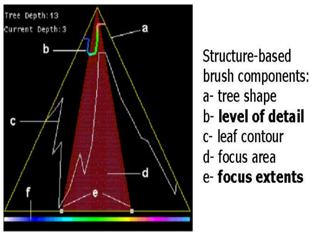

Figure 2: Structure-based brush in XmdvTool.

Figure 1 shows the parallel coordinates display of a five dimensional data set having 16,384 records. In this display, each of the N dimensions is represented as a vertical axis. A data point in an N-dimensional space is mapped to a polyline that traverses across all N axes crossing each axis at a position proportional to its value for that dimension. As seen from Figure 1, displaying all the data to the user at the same time results in display clutter. Hence, we need to provide the user with operations such as drilling down and rolling up the level of detail of the data. Towards this end, XmdvTool first clusters similar data points together to form a hierarchy (a cluster tree), and then associates aggregate information with each cluster node [24].

To support the visual navigation of cluster trees for large data sets, we designed a navigation tool called *structure-based-brush* (see Figure 2) [8]. This tool has two major "sliders". The *level-of-detail* 'b' slider allows us to navigate the tree vertically and view clusters at different levels of detail. The *focus extents* 'e' slider allows us to move horizontally and focus on a subset of clusters within the same level. The left and right extents of the 'e' slider can also be adjusted individually to modify the width of the focus area. Figure 3 displays the same data set as Figure 1 but focused on a specific cluster of data points; this is after the user narrows the width of the focus area using 'e' and performs a drill-down operation using 'b' as reflected in Figure 4. Figure 5 displays the same data set as Figure 3 but showing the mean and range of the data points in that cluster. This is after the user performs a roll-up operation using 'b' as reflected in Figure 6.

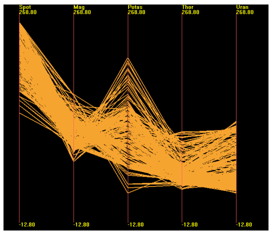

Figure 3: Parallel Coordinates After Focused Area Drilled-Down.

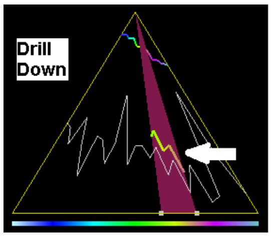

Figure 4: Structure-based brush Showing Drill-Down.

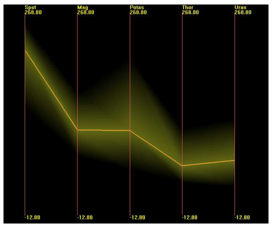

Figure 5: Parallel Coordinates After Focused Area Rolled-Up.

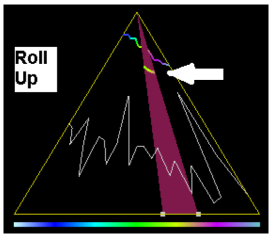

Figure 6: Structure-based brush Showing Roll-Up.

The structure-based brush exhibits the three characteristics listed in Section 1.2. First, data exploration using this tool can only be accomplished by the two sliders. Hence, locality of exploration is guaranteed. Second, queries to the database generated by this brushing tool are inherently contiguous and directional. A single brush movement (e.g., sliding the brush to the left) is actually divided into a series of smaller incremental queries and display updates. This ensures that the image displayed changes smoothly as the brush is moved. This inherent directionality of the queries adds to the predictability of the user movements. Lastly, users exploring the data clusters in a large data set using this brushing tool will likely pause to look for patterns in the data. Data can be prefetched during this idle time.

2.2. Semantic Caching in XmdvTool

Memory organization is critical in interactive applications since it influences the performance of subsequent operations. When a request for new objects is issued by the front-end, the difference between the set of objects just selected and the current content of the cache has to be quickly computed. Thus, we need to be able to know what data resides in the memory without fully traversing the cache.

For this purpose, XmdvTool uses a semantic caching architecture [19]. Semantic caching is a high level type of cache in which the cache contents at time i are maintained as a set of queries $q^i_{content}$, rather than a set of data points. In semantic caching, the front-end does not ask for specific data points. Instead, it passes a query $q_{requested}$ to the back-end: "are the data points within within this brush available?". The difference between $q_{requested}$ and $q^i_{content}$ represents the data points that are not in the cache and need to be loaded next.

2.3. Static Prefetching in XmdvTool

Here, we describe two simple prefetching strategies available in XmdvTool [7] which we used in our adaptive prefetcher.

Random Strategy: The random strategy is based on randomly choosing the direction in which to prefetch next. The random strategy applicable to the structure-based brush only considers four directions: left/right at the same level in the hierarchy and up/down the different levels of detail. This strategy is appropriate when the predictor either cannot extract prefetching hints or provides hints with a low confidence measure.

Direction Strategy: Direction strategy is analogous to the sequential prefetching scheme discussed in other prefetching papers [4, 12]. This strategy assumes that the most likely direction of the next next brush movement can be determined. In general, it is intuitive, for instance, that the user will continue to use the same navigation tool (e.g., slider or knob) for a while before changing to another one. In the structure-based brush, each slider happens to precisely control one direction only. Based on a user's past explorations, the predictor assigns probabilities to the four directions. The strategy then is to "prefetch data in the direction" currently with the highest probability.

3. Adaptive Prefetching via Strategy Selection

In general, an adaptive system is a system that changes its behavior in response to a changing environment with the goal of improving performance. It monitors the effects of its behavior on its environment through a feedback mechanism, with the aim of exploiting previously beneficial behavior and exploring alternative behavior [14].

Specifically, an adaptive prefetcher is a prefetcher that changes its prediction behavior in response to changing data access patterns. There are at least two approaches for making a prefetcher adaptive: strategy refinement and strategy selection. In *strategy refinement*, the parameters of a single prefetching strategy are adjusted within a user session. In *strategy selection*, there are several strategies to choose from and the choice can change within a session, with the goal of adapting to changing user navigation patterns. In this research, we target the three drawbacks listed in Section 1.3. Thus, we focus this research on strategy selection.

3.1. Strategy Selection Framework

When using strategy selection for prefetching, every time prefetching needs to be done, a strategy is selected from the set of prefetching strategies based on that strategy's performance on past prefetching requests within the current session. In strategy selection, we extract feedback by measuring the performance of each prefetching strategy every time it is selected.

Basic building blocks of strategy selection include:

- **Set of Strategies** - a set of individual prefetching strategies to choose from.
- **Performance Measures** - statistics to measure how well each prefetching strategy has performed.
- **Fitness Function** - a function of one or more performance measures that is used by the strategy selection policy to decide which strategy to select next.
- **Strategy Selection Policy** - a rule used to determine which among competing strategies to select.

The basic flow of our adaptive prefetching algorithm that use these building blocks is as follows:

```
select individual strategies for set STRAT
initialize performance measures and fitness of each
    strategy Si in STRAT
loop {
  set Si to null
  if idle time {
    select one strategy Si based on fitness
    start executing Si in separate thread
  }
  if user request comes and Si not finished,
    pre-empt prefetching thread
  service the user request
  if Si not null {
    calculate performance measures of Si
    update fitness of Si
  }
}
```

3.2. Set of Individual Prefetching Strategies

For strategy selection, we have selected the following individual prefetching strategies: no prefetch, random, and direction (see Section 2.3). In [7], the Xmdv team tested the basic random and direction strategies alongside more complex prefetching strategies, namely focus and vector strategies. Our experiments revealed that, if we were to choose one strategy among these strategies that works well with different types of user navigation patterns, the direction strategy is the strategy to choose for the structure-based brush. This is likely due to the inherent directionality of the queries associated with the structure-based brush as explained in Section 2.1. However, in situations when the user's movements are short and very random, [7] showed that the random strategy is better than direction strategy. Because of these observations, we use random and direction strategies as the base strategies for this research.

We also included the *no prefetching* strategy in our list for cases where it is better not to prefetch. For example, when the time difference between queries is too short to allow for prefetching, or when the remaining cache space is too small to accommodate prefetched data, it is better not to prefetch.

3.3. Performance Measures

To measure the quality of a prefetcher in predicting which objects are needed by the next user query, we utilize the following measures from [9] (see Figure 7):

		Required by user	
		Yes	No
Predicted by prefetcher	Yes	Correctly predicted	Mis-predicted
	No	Not predicted	

Figure 7: Prediction Measurements

Correctly predicted objects (CP) represent the part of the cache content that was prefetched and then requested by the next user query. The larger the values of CP, the better.

Not predicted objects (NP) represent the objects requested by a user query but were not predicted by the last strategy. These objects either had to be fetched on demand, or were in the cache because of previous queries. The sets of objects correctly predicted and not-predicted together make up the set of required objects.

Mis-predicted objects (MP) represent the objects that were incorrectly predicted and resulted in wasted bandwidth. A smaller MP means the prefetcher fetched fewer unnecessary objects.

Our definition of CP, NP and MP requires that any prefetched object be accessed immediately in the next user query. An alternative is to consider a prefetch a success as long as a user request comes at some later time while the object is still in the cache. To do this requires each prefetched object to be tagged with the strategy that predicted it and how long ago it was predicted. This raises several issues: if successive strategies predicted it, which strategies should take credit, and how much credit should be given? Hence, we elected to stay with the above simpler model.

Other performance measures include response time and network traffic. *Response time* measures how quickly can the data corresponding to the next user request be displayed to the user. We define it as $time(dataDisplayed) - time(userRequest)$. Response time can be reduced by correctly predicting and then prefetching objects before the next user request comes in, thus implying lower number of not-predicted objects. However, any performance measure involving time is likely affected by external factors such as the database server workload, the location of the network (remote or local), the size of the cache, the size of the data set, the implementation of the database (e.g., presence of indexing), the presence of pre-empted prefetching operations, and the size of the user query. Hence, care must be taken in interpreting results based on response time. Also, response time does not measure the accuracy of the current prefetcher. The response time may be fast simply by retrieving objects that were previously fetched by some other prefetcher.

Network traffic measures how much data access is required between the tool on the client side and the database on the server side. Network traffic can be determined by the number of objects retrieved from the database during fetching and prefetching. The lowest possible value would occur when (i) we do not prefetch at all (implying that when the user request arrives, we will load only data that is required) or (ii) we do not prefetch any data that will not be required in the next user request. To reduce network traffic, we thus want to minimize the number of mis-predicted and not-predicted objects. Since network traffic is a simple function of the number of mis-predicted and not-predicted objects, we do not show it in Section 5.

3.4. Fitness Function

To take into account all the performance measurements above, we need to design a function that summarizes the overall performance of a prefetcher with a single number. This function, which we call a fitness function (a term taken from Genetic Algorithms literature [13]), is a function of one or more performance measures. We use it for the strategy selection policy to decide which strategy to select.

The fitness function should take into account the performance of the prefetchers on several queries, and not just on one single query. Here, we chose misclassification cost as our fitness function. Misclassification cost, defined as cost associated with making a wrong classification (of required data) [25], is given by:

$$Cost = \frac{(C_{NP} \times \#_{NP}) + (C_{MP} \times \#_{MP})}{\#_{CP} + \#_{NP} + \#_{MP}}$$

where C_{NP} is the penalty assigned for not predicting required objects, and C_{MP} is the penalty assigned for mis-predicting unnecessary objects. We have set the value of $C_{NP} = 0.5$ and $C_{MP} = 0.5$. The lower the value for misclassification cost, the better the prefetcher. We chose misclassification cost for the following reasons:

- Prefetching is like a statistical binary outcome prediction problem. One way to measure the accuracy of such a predictor is with the use of misclassification cost. This cost represents a trade-off between the two types of errors a predictor can make - not predicted and mis-predicted. The cost factors (C_{NP} and C_{MP}) represent the weight given to each type of error.
- Since response time is affected by a number of external factors (as mentioned in Section 3.3), we decided not to directly include it in our fitness function. However, since response time is correlated with the percentage of not-predicted objects, it is therefore indirectly incorporated in the misclassification cost.
- Unlike response time, misclassification cost is not affected by external factors beyond our control, making its interpretation more straightforward.

One misclassification cost value can be calculated for each *prefetch* query. But we do not want to base the strategy selection on the cost of just the most recent query. Instead, we want to average the cost for a series of queries, giving more weight to the most recent queries. One way to do this is with the use of exponential smoothing. For misclassification cost, the exponentially smoothed average value (which we call *localAvg*) is given by $localAvg[1] = Cost[1]$ and $localAvg[t] = \alpha \times Cost[t] + (1 - \alpha) \times localAvg[t-1]$ where α = smoothing parameter between $[0, 1]$. The choice of the smoothing parameter α dictates the aggressiveness of decaying older values. For our system, we used α =0.75 to give more weight to the performance of the most recent queries.

CP, MP and NP are the same component statistics used in calculating precision and recall, popular performance evaluation measures used in Information Retrieval. Precision is $\#_{CP}/(\#_{CP} + \#_{MP})$ while recall is $\#_{CP}/(\#_{CP}+\#_{NP})$. Precision measures the usefulness of the prefetched set, while recall measures the completeness of the prefetched set. An alternative fitness function might be to combine these two measures (to produce a single number) such that the trade-off between them is reflected.

3.5. Strategy Selection Policy

A strategy selection policy is a rule used to determine which among competing strategies to select based on the fitness values. At the start of a user session, the strategies should be put on equal footing. For our implementation, we initialized the localAvg to 0 (all are good). This ensures that all strategies are selected during the first few runs of the policy.

Several policies have been proposed in the context of Genetic Algorithms and Operating Systems (see Section 6). For our problem, we experimented with two policies:

SelectBest: This policy chooses the strategy with the best fitness function value. If ties occur, we randomly choose among the tied strategies. One potential disadvantage of this policy is that a single strategy might dominate the selection early in the process and no other strategies might get selected from then onwards.

SelectProp: This policy chooses a strategy with a probability proportional to its fitness function value (assuming that a higher fitness value means better performance). In Genetic Algorithms [13], such a strategy is called "fitness proportionate selection". The general algorithm proceeds as follows: Let f_i be the fitness value of strategy i and N be the number of prefetching strategies. The probability of a strategy being selected is given by $p_i = \frac{f_i}{(\sum_1^N f_i)}$. SelectProp policy allows for some degree of exploration [13], i.e., allows currently lesser performing strategies to be selected and executed to prevent a single strategy from dominating the selection process.

4. Implementing Prefetching in XmdvTool

The adaptive prefetching framework described above has been implemented in XmdvTool 5.0 [26] (see Figure 8). XmdvTool is coded in C++ with Tcl/Tk and OpenGL primitives. The caching and prefetching modules are written in C with Pro*C (embedded SQL) primitives for Oracle8i. We focus our discussion here on the modules that interact with the prefetching module. For a complete description of the architecture and caching, see [19].

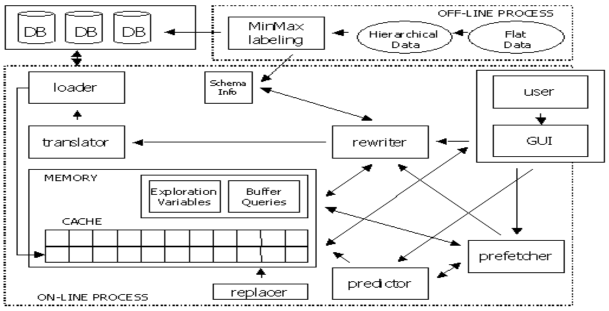

Figure 8: XmdvTool system architecture. Rectangles represent modules. Ovals represent data. Arrows show control flow.

When the system is idle, the *prefetcher* thread is created which coordinates the prefetching process and communicates with the *predictor* to make decisions about the next most probable data to be prefetched depending on the prefetching strategy and the current cache content. The prefetching query is then passed to a *rewriter*. The *rewriter* consults the cache contents and generates sub-requests to adjust the data in the cache. Each sub-request is transformed into an SQL query by the *translator*. The queries are passed to the *loader*, which fetches the necessary objects from the database and places them in the cache. Whenever the cache is full, the *replacer* removes the objects from the cache by examining probability values that depend on their semantic distance from the active region. On explicit user request, the fetching process is started by the *main* thread. If the *prefetcher* thread is still running, it is preempted and the contents of the cache are adjusted for consistency. The *predictor* module implements the framework explained in Section 3.1.

5. Experimental Evaluation

5.1. Experimental Setup

To verify drawbacks #2 and #3 listed in Section 1.3 and to evaluate the performance of our adaptive prefetcher, we investigate real user traces. In a user study, we collected the navigation traces of 14 users of XmdvTool. Each user used the structure-based brush to navigate through a hierchical parallel coordinates display of a data set containing 8 dimensions and 16,384 data points. The cache size was limited to 1/4 the number of data points. Each user used the tool for 30 minutes and made around 300-3000 brush movements (i.e., requests for data) using the two sliders described in Section 2.1. We now use these traces as input to our tool and we varied settings such as prefetching strategies.

Additional measures were used in this section. *% directionality per minute* measures the user's tendency to move the brush in the same direction as his previous movement. *Number of requests per minute* measures the frequency of brush movements. *Width of brush* measures approximately the size of each request. *Global average misclassification cost* is the average of the misclassification costs from time 0 to the current time.

5.2. User Trace Analysis

To characterize the navigation patterns of each user, we measured % directionality per minute, number of requests per minute, and width of the brush. We then performed cluster analysis based on these measurements to group together similar navigation patterns. For the cluster analysis, we used SAS' Average Linkage Method in PROC CLUSTER [16]. Figure 9 summarizes the results of this analysis. The vertical axis shows the normalized values of the 3 measurements (normized such that the smallest value is mapped to 0 while the largest value is mapped to 100), while the horizontal axis lists the users and their corresponding clusters. Cluster 1 consists of users who like to change directions often but they pause longer; we labelled them *random-starers*. Cluster 3 consists of users who like to explore in the same direction often and move often; we labelled them *directional-movers*. Cluster 2 consists of everyone else; we labelled them *indeterminates*. By identifying distinct groups of navigation patterns, we can now systematically investigate each group in more detail.

5.3. Detailed Analysis of A Directional User

User 13 Characteristics: We look closer at the navigation patterns of user 13, a directional-mover user, and use this insight to explain how adaptive prefetching worked for

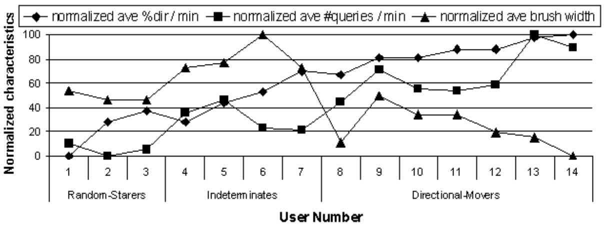

Figure 9: Analysis of User Traces

him. Figure 10a shows the % directionality per minute for user 13. Recall that for the structure-based brush described in Section 2.1, the user can only move in 4 directions (up, down, left, right). Based on Figure 10 this user's directionality changes over time, from being 100% directional at times to being 40% directional at other times. On average, he stays 73% directional.

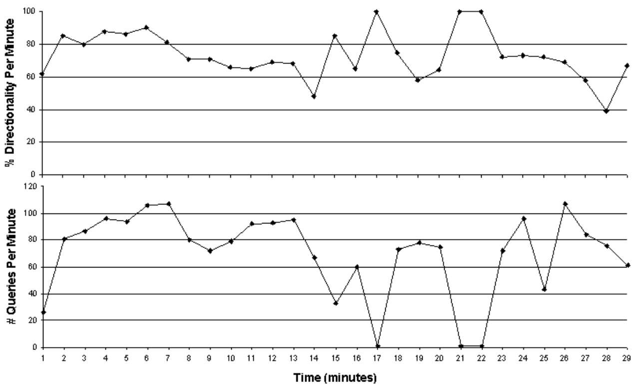

Figure 10: (a). % Directionality/Minute and (b) Number of Requests/Minute for User 13

Figure 10b shows the frequency in which the user moves the brushing tool, which translates to the number of times data is requested from the cache. Frequent moves mean lesser time to prefetch. The frequency for this user varies from 105 movements per minute to no movements for 2 minutes. On average, he moves 70 times per minute.

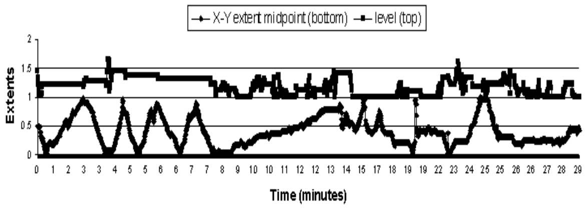

Figure 11: Brush movements vs. Time for User 13

Figure 11 illustrates the movements by the user in both horizontal (left/right) and vertical (up/down) directions. The lower line indicates horizontal movements by the user. Value 0 means the user is at the leftmost extent and value 1 means he's at the rightmost extent. The upper line indicates the vertical movements by the user. Value 1 indicates the user is at the bottom of the cluster tree (drill-down operation). Value 2 indicates the user is at the top of the cluster tree (roll-up operation). As seen in Figure 11, the user analyzes data horizontally most of the time with hardly any vertical slider movements initially. Then he starts analyzing the same data at a different level of detail, and so on.

Performance for User 13: Figure 12 shows the global average misclassification cost (*globalAvg*) over time. The lower the globalAvg, the better. Among the static prefetching strategies, the *direction* prefetching strategy gives the best globalAvg. This result was expected as this user appears to be directional (average 73% as noted in Figure 10a). Also note that for adaptive prefetching, the globalAvg for "Best" (SelectBest) is even better as it selects the best strategies, namely, *direction* and *no prefetching* strategies, for improving the performance further.

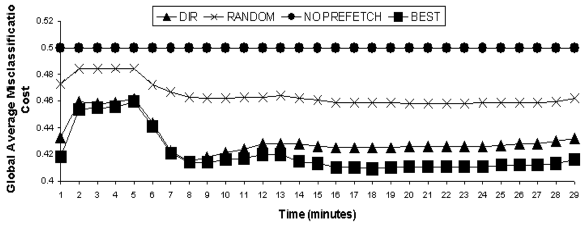

Figure 12: Misclassification Cost vs. Time for User 13

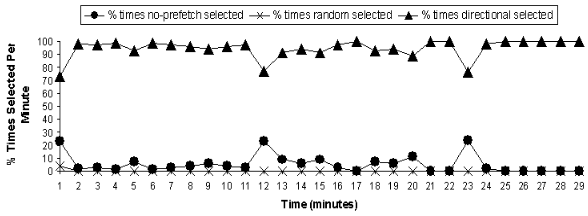

Figure 13: Strategy selection vs. Time for User 13

Strategy selection over time for user 13 is exemplified in Figure 13. Since the *direction* strategy is best for user 13, we note that it is the one that gets selected most of the time. Also, sometimes the *no prefetching* strategy is selected as the user appears a bit random at those time instances (from Figure 10a) and the movements are more frequent (from Figure 10b), thus indicating not to prefetch in order to reduce the number of objects mispredicted.

From Figure 14, response time is better for SelectBest for user 13, even though it was not directly used as the objective function for strategy selection. But as we will see below, this is not always the case for other users.

5.4. Detailed Analysis of An Indeterminate User

User 5 Characteristics: Let us look at how the adaptive prefetcher works for user 5, an indeterminate user, and compare it with user 13. Figure 15a gives the directionality for

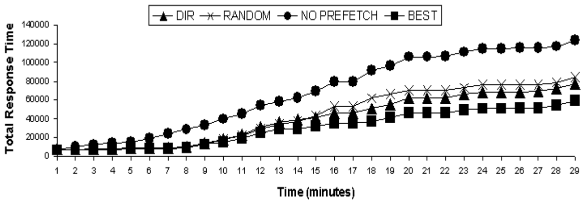

Figure 14: Response Time vs. Time for User 13

user 5 at any given time during the session. This chart shows that the user's directionality on average stays around 40%. Also, the change in directionality for this user is steady throughout. In Figure 15b, the user speed again changes with respect to time. Compared to user 13, user 5 is slow and steady with an average of 40 movements per minute.

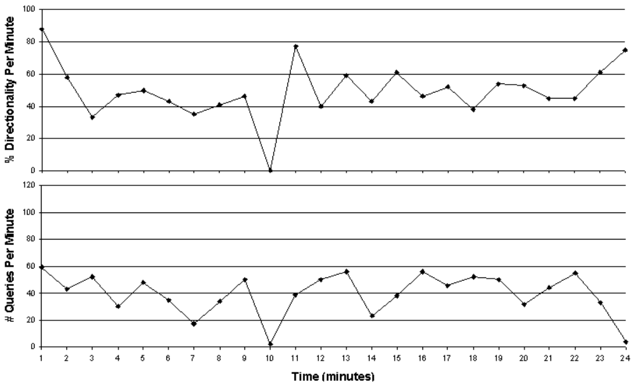

Figure 15: (a). % Directionality/Minute and (b) Number of Requests/Minute for User 5

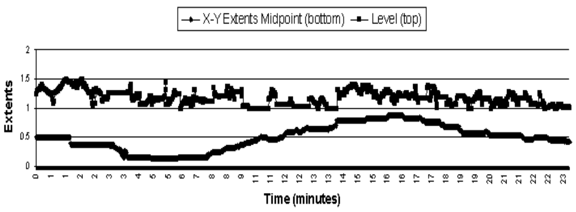

Figure 16: Brush movements vs. Time for User 5

Figure 16 illustrates the movements by user 5 in horizontal and vertical directions. This figure shows that user 5 mostly analyzes data vertically, systematically selects different subsets and visualizes them at different level of details.

Performance for User 5: As shown in Figure 17, among the static prefetching techniques, the *random* strategy gives the best misclassification cost compared to *no prefetching* as well as *direction* strategy. This is different from what we observed for user 13 where *direction* strategy is superior. This confirms our claim that no single prefetching strategy wins for all users. Also for adaptive prefetching, the misclassification cost is even better as it selects the best strategies for improving the performance further.

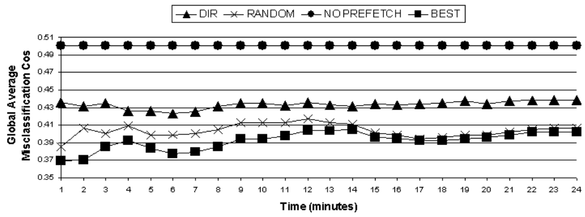

Figure 17: Misclassification Cost vs. Time for User 5

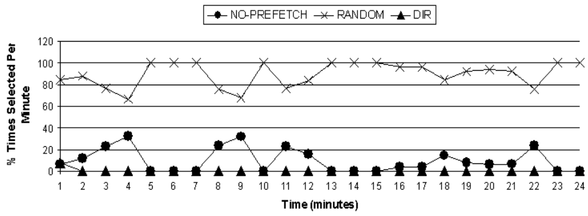

Figure 18: Strategy selection vs. Time for User 5

Strategy selection over time for user 5 is shown in Figure 18. Since *random* strategy is best for user 5, it is the one that gets selected most of the time. Also, sometimes *no prefetching* strategy is selected when the movements are quick, thus indicating not to prefetch in order to reduce the number of objects mispredicted. The *direction* strategy is hardly ever selected as its misclassification cost is high.

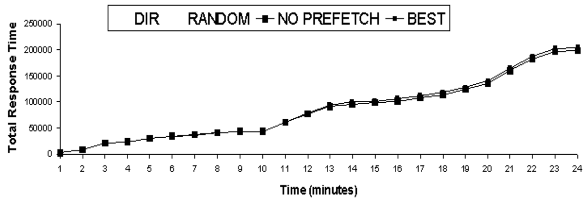

Figure 19: Response Time vs. Time for User 5

In Figure 19, the response time is similar for all the strategies. This indicates that adaptive prefetching might not help to improve the performance much if individual prefetching strategies do not have significant improvements.

The strategy selection done for user 13 resulted in the combination of direction strategy with no prefetching. On the other hand, the strategy selection done for user 5 resulted in the combination of random strategy with no prefetching. In both cases, no prefetching was chosen to compensate for the high mis-prediction done by direction and random strategies. This suggests that one could adjust the amount of data to be prefetched to achieve less % mis-predicted objects for both direction and random strategies.

5.5. Summary Charts

We experimented on all user traces listed in Section 5.2. To remove the possible effect of network traffic on the performance measures, each experiment was repeated 3 times and the measures were averaged. To get an overall picture of the performances, the results are summarized for each user cluster. Figures 20 and 21 show the global average misclassification costs and normalized response times for different prefetching strategies, summarized for each cluster.

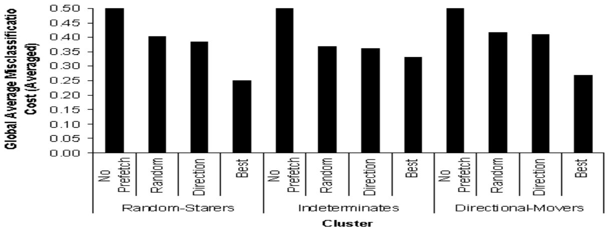

Figure 20: Global Average Misclassification Cost (Averaged) For Different User Clusters

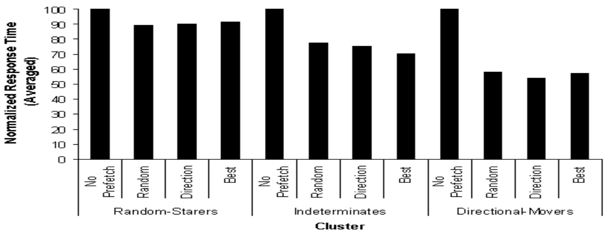

Figure 21: Normalized Response Time (Averaged) For Different User Clusters

Figure 20 shows that, on average, strategy selection improves the misclassification cost for the random-starers and directional-movers, and only improves slightly for the indeterminates. This could be due to the fact that the random and directional prefetchers do not have direct control over how much they prefetch and thus have large number of mispredictions. With strategy selection, there is the option to switch to 'no prefetching' when the movements become more random and more frequent, thus minimizing the number of mispredictions which in turn minimizes misclassification cost. This observation leads us to consider refining the random and directional prefetchers to allow the amount of data to be prefetched to change over time. For indeterminate users, strategy selection improved the misclassification cost only slightly compared to static prefetching.

Figure 21 shows that, on average, strategy selection does not improve the response time compared to the static prefetchers for the random-starers and directional-movers. Recall that response time is affected by several external factors (listed in Section 3.3). As such, the exact reason behind these summary patterns is hard to pinpoint.

We also tried the SelectProp strategy selection policy (described in Section 3.5) to investigate if a more exploratory approach (SelectProp) is better than a greedy approach (SelectBest). Our experiments showed that SelectBest yielded better results compared to SelectProp. One reason for this is that SelectProp allows even the worst performing strategy to be selected (though just a small probability). One could combine SelectBest and SelectProp to create a new selection policy as follows: use SelectBest most of the time, but when there is one strategy that dominates and its fitness is declining over time, execute SelectProp once (to give other strategies some chance).

6. Related Work

There has been much research performed on adaptive prefetching for different applications. Davidson et al. [6] proposed a solution for predicting the next user command in the Unix shell prompt by using simple Markov chain predictors. These statistics, collected for each user, are aged over time in order to emphasize the recent commands by the user. This solution utilizes the concepts of strategy refinement and information aging. Our use of exponential smoothing in our fitness function also utilizes the concept of information aging. Other prefetching work involving Markov chains include [3, 10, 15].

Tcheun et al. [22] proposed an adaptive sequential prefetching scheme for hardware, which is similar to our static *direction* strategy. It adapts to a user's step size to ensure that only the best data gets in the memory. This solution utilizes strategy refinement but not strategy selection.

Some research efforts utilize the concept of learning (instead of strategy selection nor strategy refinement) in deciding the next adaptive action to take. Srikant et al. [1] present a data mining approach to gathering sequential patterns about time-series data. This algorithm can also be used to predict the next user movement. [21] discusses a model for capturing user behavior that may be useful to adapt to the changes in the user patterns. Learning is one potential enhancement for adaptation that we currently do not apply in our system.

Several strategy selection policies already exist in various fields, including Genetic Algorithms (fitness-proportionate selection, tournament selection) [13] and Operating Systems (lottery scheduling [23]). For our research, we tried fitness-proportionate selection (in SelectProp). However, other selection methods could be considered in the future.

For performance evaluation, we extracted the idea of using mis-predicted/not predicted/correctly predicted statistics from [9]. The idea of a fitness function in strategy selection is inspired by fitness functions in Genetic Algorithms [13]. Furthermore, since prefetching boils down to a statistical classification problem, we looked into the Statistics field for ideas for fitness function and found misclassification cost [25].

7. Conclusions

In this paper, we tackled the problem of single-strategy prefetching being unable to adjust to changing user navigation patterns. We showed empirically that there are benefits to having more than one prefetching strategy implemented and in allowing the choice of strategy to adaptively change over time, within and/or across user sessions, in response to changing user navigation patterns. The benefits of adaptive prefetching via strategy selection is reflected in the reduced number of prediction errors (not-predicted and mispredicted objects) and consequently, the reduced response time, as shown in our experiments over a wide range of real user navigation patterns. Although our experiments involved only the structure-based brush in XmdvTool, the concepts are applicable to any visual exploration package exhibiting the general characteristics listed in Section 1.3. This work is the first to study adaptive prefetching in the context of visual data exploration.

While existing adaptive prefetching research focuses on refining a single strategy, we instead put forth a framework that facilitates strategy selection. This framework involves several parameters. We have shown that even with simple parameter settings, strategy selection already improves system performance. A natural next step is to refine the direction and random strategies to make them adaptive (e.g., adapt the amount of data to prefetch), and then compare the performance of strategy selection against strategy refinement.

Acknowledgments: We gratefully acknowledge our colleagues in the XmdvTool group at WPI for their contributions to this research, as well as to NSF and NSA for the funding for this research.

References

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In P. S. Yu and A. L. P. Chen, editors, *Proc. 11th Int. Conf. Data Engineering, ICDE*, pages 3–14. IEEE Press, 6–10 1995.

[2] A. Aiken, J. Chen, M. Lin, M. Spalding, M. Stonebraker, and A. Woodruff. The tioga-2 database visualization environment. In *Workshop on Database Issues for Data Visualization*, pages 181–207, 1995.

[3] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *Proc of the 1993 ACM SIGMOD Intl Conf on Management of Data, Washington, D.C., May 26-28, 1993*, pages 257–266. ACM Press, 1993.

[4] F. Dahlgren, M. Dubois, and P. Stenström. Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors. In *1993 International Conference on Parallel Processing, August 1993*, volume 1, pages 56–63, 1993.

[5] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proc of 22th Intl Conf on Very Large Data Bases*, pages 330–341. Morgan Kaufmann, 1996.

[6] B. D. Davison and H. Hirsh. Predicting sequences of user actions. In *Proceedings of AAAI-98/ICML-98 Workshop*, pages 5–12. AAAI Press, 1998.

[7] P. R. Doshi, E. A. Rundensteiner, and M. O. Ward. Prefetching for visual data exploration. In *Database Systems for Advanced Applications (DASFAA)*, 2003, to appear.

[8] Y. Fua, M. Ward, and E. Rundensteiner. Structure-based brushes: A mechanism for navigating hierarchically organized data and information spaces. *IEEE Visualization and Computer Graphics, Vol. 6, No. 2, p. 150-159*, 2000.

[9] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *Proc of the 24th Annual Intl Symposium on Computer Architecture (ISCA-97)*, Computer Architecture News, pages 252–263. ACM Press, 1997.

[10] T. M. Kroeger and D. D. E. Long. Predicting file-system actions from prior events. In *USENIX 1996 Annual Technical Conference*, pages 319–328, 1996.

[11] M. Livny, R. Ramakrishnan, K. S. Beyer, G. Chen, D. Donjerkovic, S. Lawande, J. Myllymaki, and R. K. Wenger. DEVise: Integrated querying and visualization of large datasets. In *Proc ACM SIGMOD Intl Conf on Management of Data*, pages 301–312. ACM Press, 1997.

[12] S. Manoharan and C. R. Yavasani. Experiments with sequential prefetching. *Lecture Notes in Computer Science*, 2110:322–331, 2001.

[13] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1999.

[14] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[15] T. Palpanas and A. Mendelzon. Web prefetching using partial match prediction. In *Proceedings of the 4th International Web Caching Workshop*, 1999.

[16] SAS Institute Inc. SAS OnlineDoc Version 8 with PDF Files, 2000.

[17] P. G. Selfridge, D. Srivastava, and L. O. Wilson. Idea: Interactive data exploration and analysis. In *Proc of the 1996 ACM SIGMOD Intl Conf on Management of Data*, pages 24–34. ACM Press, 1996.

[18] B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley Publishing, third edition, 1997.

[19] I. D. Stroe. Scalable visual hierarchy exploration. Master's thesis, Worcester Polytechnic Institute, May 2000.

[20] I. D. Stroe, E. A. Rundensteiner, and M. O. Ward. Scalable visual hierarchy exploration. In *Database and Expert Systems Applications*, pages 784–793, 2000.

[21] N. Swaminathan and S. Raghavan. Intelligent prefetching in www using client behavior characterization. In *Proc of the 8th Int'l Symp on Modeling, Analysis and Simulation of Computer and Telecom Systems*, pages 13–19, 2000.

[22] M. K. Tcheun, H. Yoon, and S. R. Maeng. An adaptive sequential prefetching scheme in shared-memory multiprocessors. In *Proceedings of IEEE ICPP-97*. IEEE Press, 1997.

[23] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource mangement. In *Proc of the First Symp on Operating Systems Design and Impl*, pages 1–11, 1994.

[24] M. O. Ward, J. Yang, and E. A. Rundensteiner. Hierarchical exploration of large multivariate data sets. *Proceedings Dagstuhl '00: Scientific Visualization*, 2001.

[25] S. Weiss and C. Kulikowski. *Computer Systems That Learn. Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems*. Morgan Kaufmann Publishers, 1991.

[26] Xmdvtool home page. http://davis.wpi.edu/˜xmdv.