PREDICTIVE WEB PREFETCHING USING MOUSE MOVEMENT

BY

YIWEI YANG

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Professor Matthew Caesar
ABSTRACT

The delay of web page loading time becomes one important factor of user experience. Lots of users are impatient. Therefore reducing the delay is important for both individuals and companies. This paper will use the prefetching techniques to predict and fetch the next clicked links objects before the user clicks on that link to reduce user-perceived latency. Though lots of prefetching techniques are already studied, none of them use user mouse movement trace to do prediction. This paper will deploy the trace as source and will examine three simple heuristics for the prediction. Each heuristic will be evaluated through simulation and implemented. The results show that they would work well on average under certain parameter values and there are still limitations to be improved.

Keywords: web prefetching, mouse movement trace, latency
ACKNOWLEDGMENTS

This project is supervised by Professor Matthew Caesar. This project won’t be possible without his support. Many thanks to Professor Brighten Godfrey and Alejandro Gutierrez who provide me lots of useful idea on solving the problem. Also thanks to Wenxuan Zhou who suggests me how to improve the older version of the thesis. The original prefetching idea is from Ankit Singla and Efe Karakus provides me some way to start.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER 1</th>
<th>INTRODUCTION</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER 2</td>
<td>RELATED WORKS AND BACKGROUND</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Reducing Web Page Loading Time</td>
<td>3</td>
</tr>
<tr>
<td>2.2</td>
<td>Prefetching Techniques</td>
<td>4</td>
</tr>
<tr>
<td>2.3</td>
<td>Mouse Trajectory Prediction</td>
<td>6</td>
</tr>
<tr>
<td>CHAPTER 3</td>
<td>DESIGN</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>Overview</td>
<td>7</td>
</tr>
<tr>
<td>3.2</td>
<td>Heuristics</td>
<td>7</td>
</tr>
<tr>
<td>CHAPTER 4</td>
<td>SIMULATION DESIGN</td>
<td>12</td>
</tr>
<tr>
<td>4.1</td>
<td>Data Collection</td>
<td>12</td>
</tr>
<tr>
<td>4.2</td>
<td>Data Parsing</td>
<td>14</td>
</tr>
<tr>
<td>4.3</td>
<td>Simulation for User Behavior</td>
<td>15</td>
</tr>
<tr>
<td>4.4</td>
<td>Fetching Web Pages</td>
<td>15</td>
</tr>
<tr>
<td>CHAPTER 5</td>
<td>EVALUATION RESULT</td>
<td>17</td>
</tr>
<tr>
<td>5.1</td>
<td>Measuring Benefits vs. Costs</td>
<td>18</td>
</tr>
<tr>
<td>5.2</td>
<td>Measuring Prediction Accuracy</td>
<td>19</td>
</tr>
<tr>
<td>CHAPTER 6</td>
<td>DISCUSSION</td>
<td>23</td>
</tr>
<tr>
<td>CHAPTER 7</td>
<td>IMPLEMENTATION</td>
<td>25</td>
</tr>
<tr>
<td>CHAPTER 8</td>
<td>FUTURE WORK AND IMPROVEMENTS</td>
<td>26</td>
</tr>
<tr>
<td>CHAPTER 9</td>
<td>CONCLUSION</td>
<td>27</td>
</tr>
<tr>
<td>REFERENCES</td>
<td></td>
<td>28</td>
</tr>
</tbody>
</table>
Nowadays as an “information explosion” era, people become more and more reliable on Internet on various aspects of life such as education, entertainment, marketing and so on. Most of the time people interact with Internet through browser fetching web pages from remote server using HTTP. Therefore, the delay to load a web page becomes an important factor for user experience. Users are not patient. Long delay will definitely make users unhappy and companies will also lose customers and revenues. For example, Microsoft’s Bing found out that by a 2 seconds delay, they had “a 4.3% loss in revenue per visitor”[1]. Therefore, reducing the web page loading time is necessary.

Lots of researches have been done to reduce the page-load time itself. However, no matter how well the time is reduced, there are still latencies that could not be improved. For example, the propagation delay won’t be improved no matter how large your bandwidth is and how fast your CPU operates since it depends on the physical distance and the speed of light[2]. Thus we would reduce the delay by overlapping the latency with the time that user spends viewing information on current web page. This is achieved by predicting the next link on current web page to be clicked and fetching the web page corresponding to that link in background before requesting so that when the user actually clicks on the link, it appears to the user that the delay is decreased, that is, the user-perceived latency is reduced. This belongs to an already existing technique called “Prefetching”. By predicting, it reduces the number of web pages that need to be prefetched, which makes it more scalable and allows it to require less bandwidth. However, most of the existing prediction algorithms for prefetching depend on some user-specific data such as user’s browsing or page requests history, user’s preference, access patterns and so on (see Chapter 2).

This paper will use users’ mouse movement traces as source for the pre-
prediction and use various heuristics to decide what links to be prefetched from and when to preload the web pages. This method has three advantages over other prefetching methods: 1. It is client side only prefetching and doesn’t need other infrastructures to support so it is easy to be installed, configured and deployed. 2. It only uses mouse movement information for the prediction and doesn’t rely on user-specific or private information such as the identity of the user, the way that the user uses the mouse, the user’s browsing history, access patterns and so on. Therefore it can predict the next link on web pages that have never been visited before and it is more secure since it doesn’t need any private information from users. 3. Because the source is only mouse movement trace, it can always make prediction while others can’t under some situations. For example, prefetching based on access pattern doesn’t work well when a user surfs the websites from one page to another without purpose. Prefetching based on history might make inaccurate prediction if it is the first time for the user to visit a web page.

Besides, there are lots of studies on mouse movement tracing and mouse trajectory prediction but none of them are used for link prediction. Therefore this paper will present 3 heuristics and evaluate each based on just the mouse movement to do the prediction for prefetching with different parameters setting. Limitations will also be examined and improvement will be discussed. The goal is to increase the prediction accuracy, to reduce user-perceived latency and to reduce the bandwidth usage waste.

The rest of the paper is organized as follows. In Chapter 2 I will review other techniques on reducing web page loading time and a brief introduction on prefetching and mouse movement trace prediction. Chapter 3 details the design of 3 heuristics used for prefetching. Chapter 4 would describe the simulation process and evaluation results would be presents in Chapter 5. Chapter 6 would have discussion on the limitation. Chapter 7 would show how to implement the prefetching scheme while Chapter 8 would discuss the future works. Conclusion would be drawn in Chapter 9.
2.1 Reducing Web Page Loading Time

Based on Wang et al.’s study, web page loading is mainly made up of object loading and computation where computation includes html tag parsing, CSS and JavaScript evaluation and object rendering[3]. These two components become important metrics to be considered.

Previous studies provide lots of techniques to reduce the page loading time by reducing the web objects downloading time. Some of them change the protocol (instead of using HTTP). For example, SPDY saves time by multiplexing the web page objects transfer with compressed header and using server push[4]. Besides protocol change, we can reduce the number of HTTP requests to build the page, like using Silo[5]. Caching is another technique to solve this problem since a caching hit could serve the client’s request locally instead of fetching web objects from original server[6]. But caching without prefetching technique would only serve requests that are made before in the past by the same user on the same session and it works only if the cached objects are not expired or the corresponding web’s content has not been changed.

Another way to reduce the page-load time is to shrink the computational time. This way is quite effective since Wang et al.’s study describes that the bottleneck for web page loading is on the computation which counts for 35% of the total loading time[3]. Zhang et al. develops the smart caching techniques to eliminate time for redundant computation on style formatting and layout calculation[7] (cited in [3]). JavaScript can also be cached to save time[5].
2.2 Prefetching Techniques

Prefetching is a wide area technique that was first developed for operating system. For example, prefetching was used to get the next accessed file into memory to speed up file system procedure[8](cited in [9]) in order to increase system performance. Later on, it was studied in networking system.

Prefetching reduces neither the page retrieval time nor the computation time but rather intersects the page retrieval time with user’s web reading time. It is usually made up of a prediction engine and a fetching engine. For prediction engine, the problem to be solved is what to be prefetched. The naive way of prefetching from every links would not be applicable in terms of bandwidth consumption and the total time it needs to finish all the prefetching. Thus prediction engine needs to run prediction algorithms to generate hints on which link or links the user is going to request from next. These hints will be given to fetching engine which will decide whether to prefetch them or not depending on some conditions like whether there is extra bandwidth or not and whether the system is idle or not. Besides deciding when to fetch, it is also in charge of retrieving web pages from the corresponding links.

Depending on where the prediction engine and fetching engine located, there are usually two kinds of architecture for prefetching[10]. One is client-server where prediction engine is located on server side and fetching engine is on client side. The other is client-only where both engines are put on the client side. In web-based system, prefetching with various prediction sources and algorithms for both architectures are developed to load web page objects ahead of user request time to decrease user-perceived latency.

For client-server architecture, the prediction engine is on server (or on a proxy). Thus it can use server information to do prediction. According to Bouras et al., most predictions use historical information on server like “web access logs” as sources[9]. Some can be client-based (where data depends on each user or a group of users), and some can be server-based (which aggregates data from all users sending request to this server)[11]. For example, based on the browsing or request history of current page, the next most frequently visited one or several pages based on previous requests\(^1\) or one or

\(^1\)One technique is prediction by “n most popular” approach[9], also called popularity-based prefetching.
several pages with highest probabilities to be visited based on users access patterns or sequence of requested pages\(^2\) would be prefetched. Data mining techniques can also be used to extract user access pattern from log files\([12]\). Besides log files, some researches use web page content as source. For instance, \([13]\) (cited in \([9]\)) use recently requested web pages’ content to build links ranking based on text-similarity between words around links, that is, to guess user’s interest to make prediction. Therefore not like the history-based method, it can also make prediction on non-visited objects considering current web page content and the ranking. Some would take both access logs and web page content as sources to build dependency graph on web page objects where weights on arcs relates to the number of accesses and build hint list under certain user-defined threshold\([2]\). \([14]\) proposes a data mining algorithm doing similar things. \([10]\) even improves this to build a double dependency graph which distinguishes the HTML objects and embedded objects by using two different arcs for objects on the same page and objects on different pages respectively.

Usually the prediction engine lies on server side since it has data from all users accessing it. However, \([15, 16]\) implements everything on client side, that is, using the client-only architecture. \([16]\) modifies algorithm of FasterFox\([17]\) which is a Firefox plug-in. \([15]\) uses “folder level prefetching” which means if a file is downloaded, then it will download all files from the folder where that file is placed. Usually client-only prefetching is less accurate than client-server prefetching since the latter one has more correlated data from people who access pages from that server but the latter one is harder to be implemented and needs more infrastructures to support.

One advantage about prefetching is that it predicts and fetches the web objects while user is viewing the information, which means the prefetching technique deploys the system’s idle resources to finish its task. Usually caching can be combined with prefetching since prefetching needs caching to store the content. And Palpanas also shows that prefetching can help increasing cache hit rate\([11]\) where hit rate is the number of requests served by the cache to the number of total requests\([18]\).

While many researches related to web prefetching focus on algorithm, some deploy these algorithms to really implement the prefetching mechanisms. For

\(^2\)Example can be prediction by Partial Matching\([11]\), also called PPM which uses Markov model to store previous context.
example, Firefox [19, 20] and Chrome[21] both support prefetching. But the current web page html needs to include the <link> tag to explicitly specify what needs to be prefetched. However, [22] improves this by including the “deciding what to be prefetched” part to the implementation on Mozilla with both the history-based predictor and the content-based predictor, using the latter one as complementary to the former one. [23] augments the requested html with JavaScript to download hint list (generated using PPM) from servers and prefetches the objects.

2.3 Mouse Trajectory Prediction

These researches focus on predicting user’s final target to which will be pointed based on mouse trajectory in order to shorten the pointing time, which usually falls into the human-computer interaction area. [24] samples mouse location n times with fixed amount of time interval and calculates the mouse movement vector and the cumulative angles between the movement vector and the vector build from current mouse location and targets’ center location. The one with the smallest cumulative angels would be the predicted target. [25] uses “electrocorticographic signals” while [26, 27] tries to analyze the characteristics of cursor trajectory itself which might help the prediction.
3.1 Overview

My prefetching method deploys the client-only architecture. That is, both the prediction and fetching mechanism are implemented on the client side. Three heuristics are designed. They need to solve two problems: 1. Which links will be fetched? This is to determine the candidates set. This set is dynamic during the browsing and links can be include or exclude from the set. 2. When to start fetching from a particular link from the current candidates set? In order to limit the total bandwidth waste, a variable called window size is introduced in each heuristic. This variable provides an upper bound on maximum total number of links that could be prefetched. The larger the window size is, the easier the prediction hit will be, while resulting in more wrong web pages to be fetched, which leads to unnecessary bandwidth waste and increase in traffic. If this number is reached, no more links will be considered for the prefetching. In real implementation, user could adjust this window size based on how much bandwidth he or she could allow to be wasted.

3.2 Heuristics

3.2.1 Heuristic 1

Heuristic 1 is based on the simple fact that if the user wants to click on the link, the link has to be within certain radius of the mouse. Therefore if the link is within certain radius of current mouse coordinate, it will be put into the candidates set. How to decide which link will be chosen to be fetched and
when to start the fetching? If the user is really interested in a link, he might have the mouse moving around the link for some time. Therefore there is a threshold representing the total time that the link is within the radius of the mouse. So this time is recorded for each link. Whenever the link falls out of the radius, this time will be set to 0 again. If this time exceeds the threshold then it begins fetching objects related to this link.

This is a very naive heuristic. Therefore it will fail for many reasons especially on pages that have lots of links clustered. The first problem is using this heuristic is easy to make wrong prediction. If the user just leaves the mouse close to some other links that he doesn’t care while he is looking at his interested link without using mouse to point to it or the mouse is still far from the link when he is reading, then it will prefetch wrong links which might reach the window size so that it even couldn’t download from the right link at all. The second problem is that it might begin the prefetching too late. If the user knows which link to be clicked on next then he might suddenly move mouse towards the link and click on it without reading it. Just reducing the radius helps first problem but makes the second problem worse since the small radius makes the link to be added to the candidates set even later. Just reducing the threshold would help the second problem but makes the first problem worse since the smaller threshold will allow more links to be fetched but there is only one link to be clicked by user finally. Thus the best way is to both reduce the radius and the threshold. This is reasonable since from simple analysis, we know if the radius is big, then the threshold has to be big or it will fetch many unnecessary links. If the radius is small, then the threshold has to be small since the link is already added to the candidates set late so that it has to start the fetch earlier. However, if the radius is too small, no matter how small the threshold is (let’s say the threshold = 0 which means a link will be prefetched right after it is included in the candidates set), the prefetching might not be finished or even this link might not be included in the candidates set if the user moves the mouse too fast. Therefore we do need to choose a reasonably large radius and a corresponding large threshold. Finding the pair of value is not easy, not to mention that different user moves mouse at different speed with different habits. Thus I could only try to find some value that work relatively well on average.
3.2.2 Heuristic 2

From heuristic 1, we can see one problem is that both the radius and threshold need to be reasonably large. But if the radius is too large, then it might put more links into the candidates set and might download more from more wrong links. If the threshold for how long the link stays in the radius is too large, then it might start the downloading too late. Therefore I want to both reduce the candidates set and start the prefetching earlier. Consider shortest distance would help achieving this goal. For each mouse coordinate, I calculate all the links distances from this mouse. Only links with shortest distance and second shortest distance will be included in the candidates set and all the others will be excluded from the candidates set if they are there before. And other parts remain the same\footnote{I use radius and the threshold for total time that the link stays in the radius.} as heuristic 1 except now it can use a slightly larger radius and start the downloading earlier. The reason why two links are considered instead of just one is because when there are two links, say A and B, that are vertically aligned and B is close to A and below A. Then when you want to click on the link A instead of link B, the distance from mouse to A might be larger than distance from mouse to B since most part of the mouse is on B while only the mouse pointer is close to A.

This heuristic is better than heuristic 1 for most cases but still doesn’t solve all the problems. For example, I still couldn’t decide what threshold would be good for a link to start downloading. If the threshold is small, and even it only considers two links per mouse coordinate to be included in candidates set, it would still end up downloading from many unnecessary links. And this heuristic 2 may work worse than heuristic 1 in page that contains lots of links clustered together since the more links there are, the more closer the mouse has to be to the link in order to let it be the shortest or second shortest distance link. So even small threshold is used, it might be too late to recognize a link as a candidate and start the prefetching. And using large radius would make this situation worse. So for this case heuristic 2 would work better with smaller threshold and correspondingly a smaller radius.
3.2.3 Heuristic 3

When users move mouse towards the same target on the same web page, they move the mouse with different speed and different traces to that target. However, one common phenomenon is that when user moves mouse in a very slow speed, he or she can move the mouse in any direction with any curvature, which would make trajectory hard to be predicted. However, if user moves the mouse very fast, it is very likely the trajectory will be an almost strict line and if the user wants to change the direction, he or she will first slow down the mouse and then change the direction. Based on this phenomenon, heuristic 3 will do the following: it will calculate the speed for each mouse movement based on two consecutive set of coordinates, that is \((x_{i-1}, y_{i-1})\) and \((x_i, y_i)\). Since each mouse coordinate is recorded within an almost fixed interval of time\(^2\), therefore I only need to calculate the distance between \((x_{i-1}, y_{i-1})\) and \((x_i, y_i)\) as speed. I define high speed threshold \(T_{\text{high}}\) and low speed threshold \(T_{\text{low}}\). And there are two states in this algorithm (See Figure 3.1).

The initial state is S1. If current state is S2, it will interpolate a straight line using:

\[
\begin{align*}
  y &= a_0 + a_1 x \quad \text{if } x_i \neq x_{i-1} \\
  x &= x_i \quad \text{if } x_i = x_{i-1}
\end{align*}
\]

\(^2\)The interval depends on how fast the browser could generate the mousemove events. See section 4.1.
where $a_0 = \frac{x_{i-1}y_i - x_i y_{i-1}}{x_i - x_{i-1}}$ and $a_1 = \frac{y_{i-1} - y_i}{x_i - x_{i-1}}$. And it will only consider links that intersect with this line since it is unlikely for user to change mouse movement direction with high speed. And it applies the same methods in heuristic 2. If current state is S1, the speed is slow. Then I just use heuristic 2.

Since heuristic 3 considers different cases between high speed and slow speed, it works slightly better than heuristic 2. However, this one adds two more variables $T_{\text{high}}$ and $T_{\text{low}}$ which makes it even harder to find a best combination of variables value in order to achieve the best performance on average. Besides, it is hard to find a common value of $T_{\text{high}}$ for different people since everyone moves mouse with different speed. If $T_{\text{high}}$ is too small, heuristic 3 will mistakenly only consider small set of links so it might work even worse than heuristic 2. So heuristic 3 might be more suitable for customized usage (like different people have different setting for $T_{\text{high}}$ and $T_{\text{low}}$) rather than being generally used for every one which heuristic 1 and heuristic 2 is designed for.
This section will explain the simulation process. After user’s mouse movement traces and their corresponding web page information is collected, the data will be fed, on a per user per web page base, into some C programs which are written to simulate the user behavior of browsing a web page and the prefetching process with each heuristic.

4.1 Data Collection

A JavaScript is used to collect the following data from users on each web page they browse: 1. The current page’s URL (for prediction algorithm debugging) and all links’ URLs on this page. 2. Each link’s location which is represented by the left, right, top, bottom value relative to the document object. 3. The URL of the link the user clicks on\(^1\) and the time. 4. The user’s mouse movement trace and the corresponding time when each coordinate is recorded. This is done through recording the mouse coordinate every 8ms\(^2\) if it changes. This recording is not only triggered by mouse movement, but also by scrolling. Since in JavaScript I can’t directly get the mouse’s coordinates if scrolling happens, I use scrolling bar’s relative position to calculate the coordinates, which means if a user’s first time scrolling happens before first time mouse moving, then I have to discard all the information this time since I don’t get the correct mouse coordinates value.

Since one page’s information can be as large as 30KB, I use a client-server model to store these information in server’s database system instead of storing the information locally on client side. The data is sent back to server using AJAX. The final click information is stored in local variables and is

\(^1\)This includes left click, middle click or right click.

\(^2\)This interval is determined by the frequency that the browser generates the mouse-movement event with. In my case it is 8 ms.
sent back when user clicks on any link on this web page since it is small. However, the limited cache size can’t store the mouse movement trace and all the links information locally. Instead, it begins sending the mouse movement whenever user moves or scrolls the mouse and sending the link information one by one when current web page finished loading. Along with these data, a unique identifier is also sent. Similarly, the same identifier will be sent along with the click information. This identifier is used to match the mouse movement information and links information to the corresponding click information. And also since the links information would only be sent once, it also keeps track of the previous identifier and sends it along with the click information so that if user clicks on multiple links on the same web page, the server knows which set of links information to be copied for this click information if it is not the first time click on this web page.

There are several questions solved when recording the URLs and locations of links on web pages.

The first question is link modification avoidance. Some URLs of links will be changed when users click on them. For example, Google search page uses onmousedown event to change the original link URL to a longer Google version URL with original URL encoded and embedded for client events traction, which results a mismatch between user-clicked link’s URL and its real URL when recorded after page is loaded. This won’t be a problem for real implementation phase, but it is a problem for simulation phase since URL match would be used to decide whether a prediction is correct or not. Therefore on Google search pages these search entries’ onmousedown event is replaced to empty. Another example is on Yahoo search page, the href attribute of a link would be overwritten by the value in dirtyhref attribute when user clicks on that link. Therefore dirtyhref attribute of each search entry link is removed on Yahoo search page. There are also link translation happens on Facebook, Twitters which can be solved by using [28].

The second question is locating hidden links. offset() is used to get links’ location relative to document object. However, according to [29], this function won’t get the location of link with display:none CSS property. Therefore I first change its display property to “block” and its visibility property to “hidden” before I get the location. However, I don’t consider hidden links

\footnote{Also the location needs to be considered for the link match since there can be multiple links with the same URL appearing on the same web page.}
that are not hidden using either display:none or visibility:hidden.

The way used to collect the data\textsuperscript{4} is by recruiting 20 participants with
different ages and backgrounds and locally installing the JavaScript on par-
ticipants’ Firefox browsers using GreaseMonkey and collecting data when
participants turn on the script and click on a link on a website page.

4.2 Data Parsing

The program will first read and parse all the data related to one user’s
behavior on one webpage including all the mouse movement coordinates from
the time when user loads the page in the browser to the time when the user
clicks on any link on this webpage. It will also read information related to
this web page such as all the links’ location and URLs and final clicked link’s
URL. For each link, the total fetching time is simulated using wget with -p
option to get everything that needs to display the webpage which includes
all the images, JavaScripts, CSSs, htmls as a rough measurement for the
total objects retrieval time. After wget fetching all the objects, the objects
size is recorded as a rough measurement for the total fetching size for this
web page. One special case is that some links are extremely long so that
wget will get “file name too long” problem so that it can’t store the fetched
objects. And -O option (which can rename downloaded files instead of using
the original URL) can’t be used since -p option downloads more than one
objects from the same URL, which means using -O will result in every object
being named the same and getting “object already exists” error. Instead I
use wget –spider which will not really download the object but read from
the server response to get size. But this way is restricted since dynamic web
page’s size will be undefined unless you really download it. I neglect those
links which can’t get size from both ways. Finally the data is passed on to
each heuristic to do prefetching.

\textsuperscript{4}IRB approval from UIUC has been obtained to guarantee the ethical use of data from
human subjects.
4.3 Simulation for User Behavior

In program for each heuristic, since each mouse coordinate and the link click have the corresponding time recorded, the computation can follow these times to simulate the real behavior. I use the first mouse coordinate time as real start time (realstart), the link click time as real stop time (realstop). And in the beginning of the program I record the system time as simulation start time (simustart). Whenever I read in the next mouse coordinate with recorded time $t_1$, I will not proceed to do computation based on this mouse coordinate until $t_1 - \text{realstart} \leq \text{current system time - simustart}$. I don’t use the while loop to block for the same amount of time since the user might just leave the mouse unmoved for a very long time. Instead I save the extra time $t_1 - \text{realstart} - (\text{current system time - simustart})$ to another variable (passedtime) and proceed immediately. So now I use current system time + passedtime instead of just current system time to simulate the current time. And if current system time + passedtime - simustart $\geq$ realstop - realstart, I immediately stop the computation or stop reading mouse coordinate since user already clicks on a link. Since the recorded time is in milliseconds and the system time is measured in microseconds, this simulation is accurate to mimic the real user behavior.

4.4 Fetching Web Pages

Clients can have more than one connection with the server. Therefore I use parallel downloading, meaning that if I have more than one link to be prefetched, then I download them in parallel using threads instead of using a queue to store these requests and downloading them sequentially. According to [30], each browser has a restriction on maximum number of parallel connection. And based on these numbers, I use 6 as the upper bound for number of concurrent downloading. Once the download is started, it won’t stop until it is finished or the user clicks on a link.
4.4.1 Unlimited Bandwidth Parallel Downloading

In this case, I assume that I have big enough bandwidth for simplicity, that is, every downloading thread can utilize full bandwidth and not interfere with each other. When one of the candidates starts to be fetched, a new downloading thread will be generated to download the content. Instead of actually generating the thread to download, I just record the time it starts downloading which is current system time + \textit{passedtime} (if there are less than 6 links under downloading) or the finish time of the first finished link + \textit{passedtime} (if there are already 6 links downloading so this link needs to wait until at least one of the link finished downloading) and I get the download finish time by adding wget time (which is gotten from Section 4.1) for this link to the start time. If this finish time - \textit{simustart} \leq \textit{realstop} - \textit{realstart}, then it means I finish prefetching objects from this link before the user clicks on any link.

4.4.2 Limited Bandwidth Parallel Downloading

In this case, each thread will interfere with each other in bandwidth if they are downloading at the same time. And at most 6 threads could proceed downloading at the same time. I use conditional wait to restrict the concurrent thread number. In this case I can’t directly get the downloading time from Section 4.1. Instead, after a thread is generated, I call wget on the link to re-measure the downloading time. This case is more accurate than the above case. However, this case is not scalable for the evaluation since I recalculate the downloading time for each prefetched link for each combination of variables value for each heuristic, while the above case can be much faster.
To understand how well a heuristic performs, there are three definitions for a success:

**Full success**: I finish prefetching objects from the correct link before or when the user clicks on this link.

**Half success**: I already start the downloading from the correct link but haven’t finished the downloading when user clicks on it.

**Small success**: I do include the correct link into candidates set but haven’t started the downloading when user clicks on the link.

I only consider the full success and half success as a real success that user can benefits from to experience the reduced delay. There are three metrics I would measure to evaluate each heuristic with each set of variables value:

**Accuracy**: Total number of tested pages that gets either full success or half success divided by total number of pages tested.

**Reduced User-perceived Latency** (**RUL**): Latency from the tested web page with prefetching scheme deployed divided by the one without prefetching. This indirectly shows how much portion of time the heuristic could save.

**Increased Network Traffic** (**INT**): Total downloading size from the web page with prefetching scheme divided by the one without prefetching. This also represents the bandwidth waste.

We definitely want accuracy to be high while RUL and INT to be low. But there is a trade-off. For example, in order to let the heuristic be more accurate, I might download from more links. But then INT will increase. Finding a balance would be important.

The unlimited bandwidth parallel downloading scheme (in Section 4.4.1) is used to figure out good combinations of variables value since it is fast. The following results are from the 2486 clicks\(^1\) collected from those 20 participants.

\(^1\)Clicks with loading time less than 150ms are discarded since 150ms is negligible.
mentioned in Section 4.1. The RUL and INT are measured in average.

Figure 5.1: Performance of heuristic 1 and heuristic 3 under window size 10

5.1 Measuring Benefits vs. Costs

There is a trade-off between these two metrics. Intuitively, if INT increases, RUL would reduce since the more links we fetched from, it would be more likely that we fetch the correct link or start the prefetching earlier. However, if we really make bad prediction using bad combination of radius and threshold, where lots of wrong links are prefetched and the right link is not prefetched at all or prefetching it is started late, then both the INT and RUL are large. We could see this pattern in Figure 5.1 which are scatter plots with different radius and threshold combination. The outer bound approximates the best RUL that each heuristic could achieve for a certain INT\(^2\). Comparing 5.1(a) and 5.1(b), heuristic 1’s points are more concentrated on larger INT while heuristic 3’s points are on smaller INT since heuristic 1 considers all links within its radius when selecting candidates while heuristic 3 only considers the smallest distance and second shortest distance link which intersect with the line interpolated from consecutive mouse coordinates if user moves mouse fast so potentially heuristic 3 is unlikely to prefetch a link than heuristic 1.

Window size would limit INT. For a particular small INT (for example, \(2\) m, it is approximation since I regard two INTs equal if their first two digits match and I choose the point with the smallest RUL from each set of equal INTs.)
INT = 3), we can see in Figure 5.2\textsuperscript{3} that smaller window size would help achieving better RUL. This is good since if a user wants to limit his INT and he sets his window size to be small, then this small window size would actually help him get better RUL than it with large window size. Similarly for large INT (for example, INT = 7), larger window size would help us get better RUL which is also good for users that don’t care too much about INT and are willing to set large window size.

Heuristic 2 and heuristic 3 also works better than heuristic 1 (Figure 5.3). We can see heuristic 2 and heuristic 3’s performance don’t differ too much (but heuristic 3 still has a slight improvement in performance) since when users are moving the mouse with a slow speed, heuristic 3 acts exactly the same as heuristic 2.

![Figure 5.2: Performance of heuristic 1 and heuristic 3 under three window sizes](image)

(a) Heuristic 1 with radius from 100 to 1500 pixels and threshold from 0 ms to 1500 ms
(b) Heuristic 3 with radius from 100 to 1500 pixels and threshold from 0 ms to 1500 ms

5.2 Measuring Prediction Accuracy

There is a correlation between radius and threshold on how they affect the accuracy. Intuitively, for one fixed radius, if the threshold is too small, we prefetch from lots of wrong links which might occupy the window size and prevent us to download from correct link. If the threshold is too big, we might start the prefetching too late so it will not finish before the user clicks the

\textsuperscript{3}The lines are derived from the outer bound for each heuristic under that window size.
Figure 5.3: Performance of three heuristics under same window sizes

link. So we should first see increase and then see decrease and this pattern can be seen in Figure 5.4\(^4\). When the window size is larger, smaller threshold will work better since we can start prefetching earlier so the peak value is shifted to the left which can be seen by comparing \(w = 5\) line with the \(w = 15\) line in Figure 5.4(a). And for certain window size (i.e. window size is 5), we can see larger radius would achieve highest accuracy on larger threshold (the \(w = 5\) line on Figure 5.4(c)) while small radius on smaller threshold (the \(w = 5\) line on Figure 5.4(b)) which matches the analysis in Section 3.2.1.

Comparing three heuristics, we can see with the same radius and same window size, heuristic 3 works slightly better than heuristic 2 which works better than heuristic 1 on average (Figure 5.5) on smaller threshold due to the reason that heuristic 2 and heuristic 3 are less likely to fill up the window size than heuristic 1. However, heuristic 1 works better than these two on larger threshold since it considers all links within its radius when selecting candidates.

For heuristic 1, predicting correct or not on each pages follows a binomial distribution, and \(\frac{X-\mu}{\sigma}\) approximates a standard normal distribution if \(X\) follows binomial distribution. Thus the 95% confidence interval (where error = 5%) is derived using formula \((p - z_{1 - \frac{1}{2} \alpha} \sqrt{\frac{1}{2} p(1-p)}, p + z_{1 - \frac{1}{2} \alpha} \sqrt{\frac{1}{2} p(1-p)})\) where \(p\) is the sample success rate with a particular radius and threshold and \(z_{1 - \frac{1}{2} \alpha}\) is \(1 - \frac{1}{2} \alpha\) quantile of standard normal distribution which in our case is \(z_{1 - \frac{1}{2} \alpha \times 5\%} = 1.96\). In Figure 5.4(a) we can see three vertical bars which separately represent the confidence interval on best combination point of ra-

\(^4\)I tested on radius ranging from 100 to 1500 and radius 100 happens to be the one where the three heuristics all achieved the highest accuracy value.
dius and threshold which achieves the maximum accuracy on the 2486 pages under each window size. These intervals show how well the accuracy would be when applying heuristic 1 on other webpages. And for heuristic 2 and heuristic 3, the same logic applies.

Figure 5.4: Accuracy of three heuristics under fixed radius implemented with three window sizes
Figure 5.5: Accuracy of three heuristics with radius 100 pixels and window size 5
Using a client-based architecture and only mouse movement as sources, there are many limitations.

One problem is that since my prefetching way is not history-based or content-based, I can’t implement object-based prefetching but rather downloading every object from the links. This means that even if there is any non-cachable object, I still prefetch it and waste the bandwidth and time. I would also fetch objects that might expire soon since I don’t know the life time of each object. So in one word, I don’t have enough information about the objects on the webpage ([11] knows frequency of web page content change and dependency rate which could optimize the prefetching). So I can’t optimize on what set of objects I could prefetch but rather prefetch all of them.

Another problem is that since I use mouse movement as source for prediction, this way could never achieve the optimum. There are two goals for prefetching: 1. Save user-perceived latency (save time). 2. Save bandwidth that is wasted for prefetching unnecessary links (save resource). And there is also a requirement: prefetching shouldn’t interfere with the regular request. Based on the goals and requirement, I define the optimum for prefetching as the following: After the current page finishing downloading (or more precisely, the browser and the server are both idle now), it will start prefetching from one and only one link which is exactly the link that client will click on. This optimum can be realized by client-server model technique. For example, prefetching based on PPM[11] which is Markov model recording access pattern and based on user access logs can achieve this optimum. For mine which depends on mouse movement, it definitely couldn’t achieve this since in order to make some mouse movement, it needs some time after the page finished downloading, so I can’t start the prefetching right away after the web page finished downloading. Only after waiting for some time for user to
make some mouse movement and waiting for some threshold to avoid false positive, I can start prefetching from some links.
CHAPTER 7

IMPLEMENTATION

Since the prefetching scheme is only client-side so it is easy to be implemented. I translate the heuristics simulation c program into JavaScript and combine it with part of the data collection JavaScript and use Greasemonkey to install it on Firefox browser. Once the heuristic decides to begin downloading from a link, it will create an iframe element and load the prefetched page into iframe. Using iframe has the following advantages: 1. If the prefetched web page is already cached, the cachable objects won’t be fetched again but will be directly read from cache. 2. I don’t need to care about what objects need to be fetched but just load everything as if we are visiting that page. 3. Cross-domain fetching problem caused by same origin policy is easily solved since it uses a different separate frame to load page. The window size parameter could also be changed by users to limit how much bandwidth they are willing to waste.
CHAPTER 8

FUTURE WORK AND IMPROVEMENTS

Due to time limitation, there are things that can be improved in future.
1. Add-on for Chrome which is used by lots of users should also be implemented.
2. Instead of using wget, some more scalable ways need to be find out so that the method in Section 4.4.2 could be used to make more accurate evaluation.
3. So far all my three heuristics don’t work well when many links are clustered. I need to design better heuristics. For example, I can use some machine learning techniques such as cluster algorithm to find out two points that best represent the user behavior and interpolate a line with certain curve and the links that intersect with this curve might be the candidates. And there are other researches done on mouse trajectory prediction using other infrastructures. I might deploy those to predict the next link.
4. Being more precise on what needs to be downloaded rather than downloading everything that supports the web page might help saving bandwidth. For example, I need to download images, JavaScript and CSS but may not necessarily download the html since that part might not contribute much to the web page load time.
5. The prefetching should also support all or part of dynamic links where links locations could change. This requires better crawler JavaScript and more complicated heuristics to do the prediction.
6. More participants need to be recruited to make the data less biased.
7. Instead of using fixed parameter value for the prefetching algorithm, it might be better if the algorithm could dynamically change the parameters value to adapt to different browsing situation.
CHAPTER 9

CONCLUSION

In this paper I develop a prefetching scheme that uses user mouse movement trace as source to do the prediction for prefetching in order to reduce web page loading time for improving user experience. I analyze three heuristics and evaluate on each through simulation. The first heuristic is based on the radius and the time the link stays inside this radius. The second heuristic adds the shortest distance as another factor on top of heuristic 1. The third one differentiates between high speed and low speed mouse movement on top of heuristic 2. These heuristics can work well on average under certain parameter values. Because of the simplicity, there are many limitations such as predicting wrong links, starting the prefetching too late. Therefore I still need to look for better heuristics and deploy more complicated algorithms combined with machine learning techniques to make the prediction more precise on what I want to prefetch and when to start the prefetching.
REFERENCES


