Computer Networks and ISDN Systems 30 (1998) 457–467

COMPUTER NETWORKS and ISDN SYSTEMS

Using path profiles to predict HTTP requests

Stuart Schechter [*,a,1], Murali Krishnan [b,2], Michael D. Smith [c,1]

[a] *Harvard University and Microsoft, 29 Oxford St. #110, Cambridge, MA 02138, USA*
[b] *Microsoft, One Microsoft Way, Redmond, WA 98052, USA*
[c] *Harvard University, 29 Oxford St. #130, Cambridge, MA 02138, USA*

Abstract

Webmasters often use the following rule of thumb to ensure that HTTP server performance does not degrade when traffic is its heaviest — provide twice the server capacity required to handle your site's average load. As a result the server will spend half of its CPU cycles idle during normal operation. These cycles could be used to reduce the latency of a significant subset of HTTP transactions handled by the server.

In this paper we introduce the use of path profiles for describing HTTP request behavior and describe an algorithm for efficiently creating these profiles. We then show that we can predict request behavior using path profiles with high enough probability to justify generating dynamic content before the client requests it. If requests are correctly predicted and pre-generated by the server, the end user will witness significantly lower latencies for these requests. © 1998 Published by Elsevier Science B.V. All rights reserved.

Keywords: Prediction; Path profiles; HTTP performance; Dynamic content

1. Introduction

When the first World Wide Web sites came online the entirety of the content handled by Hypertext Transfer Protocol (HTTP) servers was static. Caching at the server and client reduced the latencies of user requests. The arrival of HTTP proxies put an additional layer of cache between the client and server. Dynamic content, first introduced to the Web in the form of CGI scripts, has increased the usefulness of the Web dramatically. Unfortunately, when content is generated differently for each request, caching can no longer be used to reduce latency.

Currently, the industry is concentrating its efforts to reduce the latency of HTTP requests for dynamically generated content by increasing the speed of dynamic content generators. However, there is a limit to the amount by which this technique can decrease the user's perceived response time, as the current metaphor requires the server to wait for a user's request before it starts the dynamic generation process.

An idle server waiting for a new request is similar to a microprocessor waiting for its branch unit to tell it which section of code should be executed next. All modern high-performance microprocessors use branch prediction to speculate on the location

* Corresponding author.
[1] E-mail: {stuart,smith}@eecs.harvard.edu
[2] E-mail: muralik@microsoft.com

0169-7552/98/$19.00 © 1998 Published by Elsevier Science B.V. All rights reserved.
PII S0169-7552(98)00106-8

of future instructions. If the machine predicts correctly then execution continues and the latency of the branch execution is completely hidden. If the branch prediction is incorrect there is little harm done — the machine efficiently discards the results of the speculative instructions and starts executing the instructions that the branch instructs it to execute.

If a Web server could predict the next URL to be requested from a user, in a manner similar to that in which a microprocessor predicts the location of future instructions, then the server could then generate dynamic content before a user requested it. This would reduce the latency of requests that were predicted correctly.

The technique we use to predict URLs is based on the concepts of point and path profiles — tools that have been well received in the area of compiler optimization. A page's *successor* is the page requested immediately after that page in a URL sequence. Intuitively, a *point profile* contains, for any given page, the set of that page's successors (in all URL sequences) and the frequency with which that successor occurred. More concisely, for any two pages X, Y, a point profile contains the frequency with which Y was accessed immediately after X. Point profile statistics are readily available from a number of existing tools that use them to determine information such as the success of advertising links. Path profiles are harder to collect and are not supported by existing tools.

A *path* is a sequence of URLs accessed by a single user, ordered by the time of access. A path may contain the same URL more than once. The length of a path P, expressed as $|P|$, is equal to the number of URL instances in P. Note that this differs from the definition of a path length in the compiler literature on path profiles [5]. The path that describes the full set of requests made by a user within a given time frame is called a user session, or simply a *session*.

A *path profile* is a set of pairs, each of which contains a path and the number of times that path occurs over the period of the profile. The profile is recorded over the set of all user sessions. Recording this information efficiently is non-trivial, as the number of paths in a user session S grows as a function of $|S|^3$. We want to construct path profiles so that we store the majority of the important paths without recording so many of the unimportant paths that we run out of memory.

Path profiles can be collected either by an HTTP client recording its user's paths through the entire World Wide Web or by an HTTP server recording the paths of all users that access its site. Client-side profiling has the advantage of focusing on a single user and revealing data about that user's intersite behavior. A smart HTTP client might also be able to predict the information its user will enter into fields with common labels such as "address". However, no standard currently exists for logging HTTP behavior from the clients so customized client software would be required to perform profiling.

Server-side profiles can be generated from standard HTTP server logs and thus require no changes to existing software. Using server logs, a profile containing records of a large number of accesses can be generated quickly. However, server side profiles cannot track activity that takes place off the server and can rarely be used to predict the data that users enter into form fields.

In this paper we describe an algorithm for efficiently generating path profiles from information contained in standard HTTP server logs. Along the way we discuss design decisions that must be made when customizing the algorithm for use with different Web applications.

We then show that these profiles predict HTTP request behavior with high enough probability to justify their use in page pre-generation. A common rule of thumb for Webmasters is to provide server capacity of at least twice that required to handle average load. If a Webmaster follows this rule, then the server will spend half of its CPU cycles idling during normal operation. These cycles could be used to reduce the latency of a significant subset of HTTP transactions handled by the server.

In Section 2 of this paper we describe how we use server logs to obtain the information needed to generate path profiles. Section 3 describes the algorithm for efficiently finding common paths and the frequency with which they are accessed. Section 4 contains the experimental setup for using path and point profiles to predict URL requests, with the results and analysis discussed in Section 5. Section 6 describes future work and other uses of path based prediction. Section 7 concludes this paper.

2. Server-side profiling using server logs

All HTTP logs we have encountered contain the following five fields describing each request:

- Date,
- Time,
- Source (Client) IP Address,
- Name of file or script requested (derived from URL),
- Parameter field (derived from URL).

Before we can generate path profiles from the user session paths as described in Section 1, we must first isolate these user paths from the HTTP log entry information. In order to do this we need to decide whether the URLs used to form paths should contain the parameter field or only the name of the file or script. We need to match the requests in the log to their users so that we may string these requests together by user. We must also understand the limitations of the information that can be derived from server logs.

The initial versions of HTTP provided no means for a client to pass any information to the server beyond the name of the file being requested. The parameter field was added to the URL request string and is separated from the file name portion of the URL by the "?" character. As the Web developed, the parameter field was used for everything from passing information from user entered forms to passing the identity of the user. These parameters are often an essential part of the URL — containing search strings, record identifiers, or even the name of another URL to be accessed. Unfortunately, these parameters may also contain information that serves no purpose other than to track the user's session. Session specific information, if used to determine whether a prediction is correct or not, can falsely lower predictability rates. Finding an automatic method for determining which parameters to ignore and which should be considered part of a URL remains an open problem. For now, we will generate profiles that either ignore all parameters or include all parameters as part of the URL.

Once we have determined what form of URL to use, we need to compress the URLs down to a form that is more compactly stored and more quickly compared. A unique integer is mapped to each unique URL string that occurs in the logs. If two strings represent the same URL, they are assigned the same number. For the rest of the paper, we assume that all URLs have been mapped to these numbers.

Our prediction algorithm will guess the current user's next request by looking at what past users who behaved similarly have done. While the concept of a user's session is essential to path profiling, few HTTP log entries identify the user making each request. Because the concept of a user is essential to path profiling, we are left with two options to differentiate between users: supplement the logs with unique user IDs or distinguish by their IP address. IP addresses are less desirable because many such addresses actually represent proxy servers that multiplex requests from many different users. Microsoft's Site Server Usage Analyst, which generates information similar to point profiles, provides a runtime filter for Microsoft's server that distinguishes users using unique identifiers stored in HTTP cookies [4]. However, since this tool has only recently been made available and only works with Microsoft's server, we were forced to rely on IP addresses except where otherwise noted.

It is impossible for a single HTTP server to trace the path of a user through other sites. As a result, it is also impossible to determine if a user has passed through another site on the way between two pages on a single server site. In order to determine whether two hits from the same user were made during a single visit to a site, we adopt the heuristic that any two HTTP requests separated by more than thirty minutes are not part of the same user session.

Finally, we must realize that the information available in the server log is limited. In this paper, some of the documents we profile are static and as a result they may be cached at proxies. If a request is short-circuited by a proxy, the server will never see the request and no log entry will be created.

3. HTTP request prediction using paths

Recall that a path P is any sequence of URLs, and that a user's session is the path that contains the ordered list of URLs accessed by that user within a specified time constraint. In Section 2 we discussed the inferences that allow us to generate session paths from HTTP server logs. We now describe how these sessions are used to generate a tree of important

paths. In the second subsection we describe the prediction algorithm and how it uses this tree.

3.1. An efficient path discovery algorithm

Recall that the path length $|P|$ is the number of URLs in the sequence. By inspection one can see that, in a given session S, there are $|S|$ paths of length one, $|S| - 1$ paths of length two, $|S| - 2$ paths of length three, and so on. There is only one path of length $|S|$, the session path. As a result, we see that a session contains $|S|^2/2$ paths P where $|P| \leq |S|$. Further analysis shows that the average length of the paths is $|S|/3$, and so the total number of URLs that would be needed to store every path, using a naïve algorithm, is $|S|^3/6$.

To address the worst-case space constraints our algorithm stores all paths in the form of a tree in which nodes may have a variable number of children. The tree is built around the root node such that a walk down the tree is equivalent to a walk through a path of URLs. While related work on paths uses multiple trees [5], there is no need to keep track of multiple trees when a root node may have a variable number of children, and thus can hold together all of the path trees.

When recording a path in the tree, the first URL in a path will be stored as a child of the root node of the tree. The second URL in the path will be stored in a node that is a child of the first URL's node. This may continue until the end of the path. To determine if a path is stored in the tree, one may simply walk down the tree, checking along the way that at step N there exists a child of the current node with the label that corresponds to the Nth URL in the sequence and stepping down to that child. However, not all paths will be stored on the tree, as resource constraints make this impossible.

In order to illustrate how paths can be stored in this tree efficiently, we must first define the *maximal prefix* of a path. A path Q is a *prefix* of path P iff

- $|Q| < |P|$, and
- the elements of Q are the first $|Q|$ elements of P, in the same sequence.

Q is a *maximal prefix* of path P iff Q is a prefix of P and $|Q| = |P| - 1$. In other words, a maximal prefix is the sequence containing, in order, all the URLs of the original path except for the last one. We also define a suffix R to the elements of R sequentially match the *last* $|R|$ elements of path P.

We can reduce the number of potential paths in the tree at the time of storage by only adding paths of length greater than one to the tree if the path's maximal prefix has occurred at least T times. T is a threshold that can be configured based on available memory resources.

Recall that in Section 2 we reduced URLs from strings to unique integers. Assume that for each user session there is an array, *URLSequence*, of these integers. The URL numbers in *URLSequence* occur in the same order as the URLs occurred in the user's session.

We start the algorithm with an empty tree of paths, *PathTree*, which contains only a root node. Upon creation, each node in the tree will be initialized with an *OccurrenceCount* value of 0, except for the root, which is initialized with an OccurrenceCount of T. Each tree node, except the root, is also labeled with a URL number.

The tree is then constructed by applying the following algorithm to each sequence of URLs that represent a user's session:

- ○ **FOR** each URL in the sequence (stepping through using a SequenceIndex)
  - ■ CurrentNode ← root node of PathTree
  - ■ Index ← SequenceIndex
  - ■ **DO**
    - ■ Increment the OccurrenceCount of the CurrentNode (CurrentNode → OccurrenceCount)
    - ■ URL_Number ← URLSequence[Index]
    - ■ **IF** there does not exist a child of CurrentNode labeled with URL_Number **AND** CurrentNode → OccurrenceCount ≥ T
      - ■ Create a child node of CurrentNode labeled with URL_Number
    - ■ **IF** there exists a child of CurrentNode labeled with URL_Number
      - ■ CurrentNode ← Child of CurrentNode labeled with URL_Number

- ■ **ELSE**
  - ■ EXIT Do/While Loop
- ■ **WHILE** (++Index < length of URLSequence)
- ○ **END** - FOR

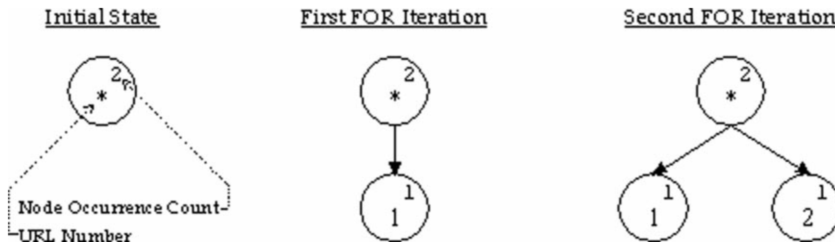
By zeroing the OccurrenceCount variable of all tree nodes except the root, we can re-run the algorithm over the set of all user sessions and refine the shape of the PathTree. We iterate this process until the shape of the PathTree stabilizes. Typically, this requires no more than 15 iterative steps.

The following example shows the simplicity of this algorithm. We demonstrate using a server log that contains only four pages:

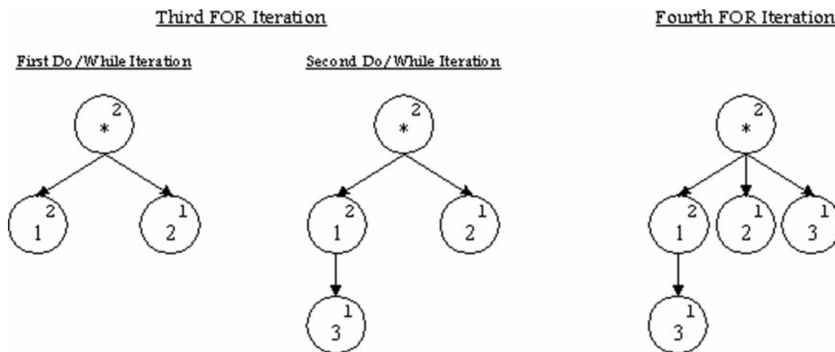
(1) http://stuart.student.harvard.edu/index.html
(2) http://stuart.student.harvard.edu/a.html
(3) http://stuart.student.harvard.edu/b.html
(4) http://stuart.student.harvard.edu/c.html

Without loss of generality, we assume that there is only one user session recorded in the logs and that the value $T = 2$. The sequence of URLs requested by the user during the session is represented by the URLSequence array [1,2,1,3,4].

At the beginning of the algorithm the tree is initialized to contain only the root, labeled with a "*" character. In the first iteration of the FOR loop, the path of length one containing URL #1, represented [1], is added to the path tree. The second iteration of the FOR loop stores a representation for path [2].

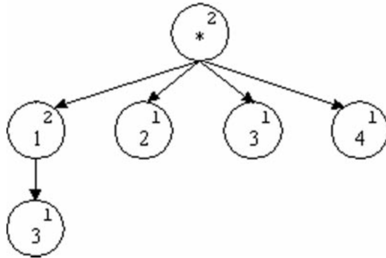


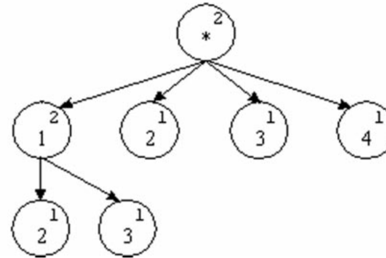
In the third iteration, the second (Tth) occurrence of the path [1] is recorded in the tree. Thus all paths that have the path [1] as an immediate predecessor may now be added to the tree. The DO/WHILE loop, now at a CurrentNode with OccurrenceCount $\geq T$, iterates a second time and records the path consisting of URL #1 followed by URL #3, represented [1,3].



After the first iteration of the algorithm the tree is still missing the path [1,2]. We must iterate the algorithm if we want to accurately count all paths with immediate predecessors that occur at least T times. Before each supplemental iteration, we must clear the OccurrenceCount values of all leaf nodes so that the new-leaf threshold is not reached prematurely. In order to make the final counts accurately reflect path frequencies, we run a final iteration of the algorithm with the tree structure in place but with

all OccurrenceCount values reset to zero:





3.2. An algorithm for predicting HTTP requests using paths

It is not immediately clear what prediction technique provides the best prediction accuracy. An intelligent algorithm might combine the tradeoffs between the depth of a path and the quality of data (such as number of samples available) at each depth. Our intent is not to explore the many techniques for making path-based predictions but to show the merit in investigating profiling at path depths beyond the trivial case corresponding to point profiles.

Before describing the algorithm for predicting HTTP requests using paths, it is first useful to understand a few special properties of the tree we have constructed. If a path is found in the tree, than its maximal suffix also exists in the tree. By induction, if a path occurs in the tree, then all of its suffixes occur in the tree. Secondly, note that a path can never occur more often than its suffix. This is intuitive because a path cannot occur in a session unless its suffix also occurs.

We want our algorithm to select the best paths with which to predict the next URL. To this algorithm we pass a user's current session in the form of a path S. The most recent URL accessed is the most important predictor because the page returned by that URL contains the hypertext links from which the user is likely to choose his next destination. We thus work backwards from most recent to least recent, applying S to the path tree in order to predict the next URL in the session. In order for a path P in the path tree to be used for prediction, S must be a maximal prefix of P. The tail URL of the most commonly occurring P is the URL we choose as our prediction.

To select the path to be used for prediction, we used an easily coded backwards-stepping algorithm that is equivalent to the more efficient, but also more code-intensive, technique we describe later. We start with a path Q that is the shortest suffix of S, and then find a matching path M with Q as its maximal prefix. We then repeat with increasing suffix size until we no longer find a matching path or until $Q = S$. We use the longest matching path M as the predicted path P.

For example, to find the predicted path from a history path $[A, B, C]$, we would first find all paths with maximal prefix $[C]$. One such path might be $[C, D]$. We would then find all paths with maximal prefix $[B, C]$, such as $[B, C, E]$. If there is no path with maximal prefix $[A, B, C]$, and $[B, C, E]$ was the most frequently occurring maximal prefix of $[B, C]$, then we would predict that the client would request page E next.

If we were to implement a system where the computational efficiency and memory footprint of the prediction process were of greater importance, we would likely want to rid ourselves of the path tree in its current form. Instead, we would use the path tree to construct a list of all paths stored in reverse order, with each entry representing a reverse-ordered path and the number of times that the path occurred. Longer paths that make the same prediction as their shorter counterparts could then be filtered out. We can do this because there is no reason to predict using longer paths when they yield the same answers as paths that take less history into account.

The final step in the efficient system is to sort the list of the reverse-order paths or turn them into another tree. Finding the set of paths is now as

simple as walking backwards through the current user's session while efficiently searching through the list or tree of paths. The longest path for a session history is found and a prediction is made in time bounded by the log of the number of paths.

4. Experimental setup

4.1. Methodology

The predictability of HTTP requests was measured using training and testing data consistent with the rules of cross-validation. We used separate data sets to construct the path tree and to test its utility in prediction. To fully simulate a practical application of these logs, testing logs contain only those requests that occur after all of the training log requests were collected. In the event that only one log was available from a given site, the log was split into two pieces, and the first section of which was used as training data for building the path tree.

Many HTTP requests are images and background sound files. While such files are usually static and often quite predictable, some sites contain dynamically customized images, such as stock graphs. Because this paper analyzes sites without a significant amount of dynamic graphic content, the inline images files are neither predicted nor taken into account in the prediction process.

As mentioned in Section 2, the parameter field of the URL may be essential to the nature of the request or it may contain garbage that causes false prediction misses. In order to ensure the generality of the algorithm, tests have been performed twice to ensure that parameters will not significantly affect our results. In one case, the parameter field is used for prediction and matching parameters are required for a prediction to be deemed correct. Tests were also run with parameters ignored in both the prediction process and in prediction correctness evaluation.

We choose a threshold value of $T = 3$ for the path construction algorithm in Section 3.2. This value ensured a reasonable amount of path expansion to allow us to fully differentiate between path and point predictions.

For each test we tracked how many HTTP requests we encountered in our test data and how often the prediction of the next request was correct. Predictability rates can be measured from the perspective of either the server or the client. The rate seen by the server counts as incorrect those predictions that are made after the user no longer requests any further documents. The client-viewed predictability rate takes misprediction into account only if it will result in a longer response time for a future request. Hence client-viewed predictability rates throw out the case where the server makes a prediction but the user never requests another page because the client never sees the consequences of the misprediction. Since we are most concerned with how the client perceives latency, we use client-viewed rates. It is important to note that prediction begins after the first client request and that we do not include this request in the calculation of our client-based prediction rate. We predict the sequence of requests within a user's session; we do not predict the start of a session.

4.2. Data set

Logs were obtained from a number of sites. The logs referred to as *PlanetAll* were recorded at the members-only area of www.planetall.com, a site composed almost entirely of dynamic content specific to a user's profile. This data set is particularly interesting because it represents a small but extremely useful Web application where good performance is as important as it is difficult to achieve.

The Lotus Corporation provided us with a log file from a Domino-based server. This data set, referred to as *Domino*, was used to ensure that our results were not server dependent.

The logs referred to as *ASP* represent another site rich in dynamic content — the ASP sample pages that formerly resided at iisa.microsoft.com. The ASP sample pages were available over the Internet to beta users of Microsoft's Internet Information Server.

We also analyzed log files from www.microsoft.com to test the suitability of our approach on an extremely large site that resembles a data store more than an application. We refer to these logs as *Microsoft.com*. The extended format of these logs included unique identifiers to differentiate each user. Microsoft.com had by far the most traffic and the largest set working set of any of our logs. The

test data contains over 8000 independent URLs accessed by over 50,000 users in a small portion of a day.

Unfortunately, the microsoft.com logs were collected on a single server operating in a multi-server environment. Train and test data for this site were obtained using the same server log, within the constraints described above, to guarantee that both data sets would correspond to the same server and the same site structure.

5. Results and analysis

During each test, we tallied results for three types of prediction: *point*, *path*, and *agreement*. Point predictions guess the next request by looking only at the last URL requested by the user. Path prediction uses the algorithm described in Section 3.2. Both of these techniques make a prediction, be it correct or incorrect, about the next request only when information exists to make a prediction. Figure 1 reports the percentage of requests that were predicted by our point and path techniques. Recall that a point-based prediction is equivalent to a path-based prediction that stops after the first iteration, using only the shortest suffix for P. For both point and path profiles, the minimal amount of information required to make a prediction is the same — the tree must contain a path of length two must which has as its maximal prefix the most recent URL requested (the shortest suffix of the user session). For this reason we do not differentiate between the two types of predictions when determining whether these algorithms have enough information to make a prediction.

Our final technique, which we call agreement, first determines what prediction our point and path techniques would make, and if these predictions match, only then does the agreement technique make a prediction. As our later results show, agreement-based prediction results in higher correct prediction rates. However, this technique also makes fewer predictions than the point-based and path-based techniques. If server resources are limited, we may achieve a net performance win by restricting the cases in which we make predictions (i.e. pre-generate some dynamic content). Checking if the point and path predictions agree is one method for assigning a confidence to the prediction. A similar idea has been proposed in the field of branch prediction to reduce the wasting of processor resources on branches that are extremely hard to predict correctly [3]. Figure 2 shows that our heuristic often restricts us to predicting only half of future URLs requested. The Microsoft.com logs showed the more agreement than other logs because the large number of URLs in the logs resulted in a broad tree where path predictions were often made from the same path of length two as the point prediction.

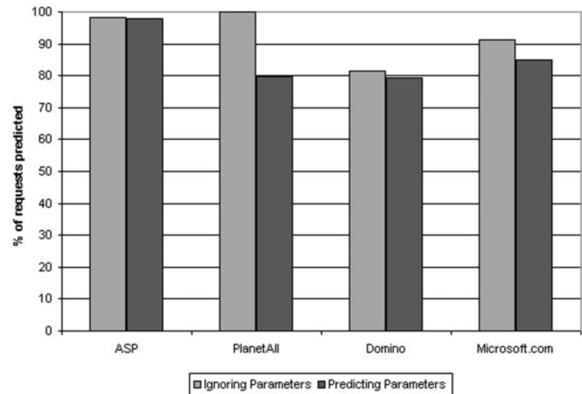


Fig. 1. Point/path prediction. This graph plots the percent of requests in our test log for which the path-profile and point-profile based prediction algorithms had enough data to attempt to predict the next page to be requested.

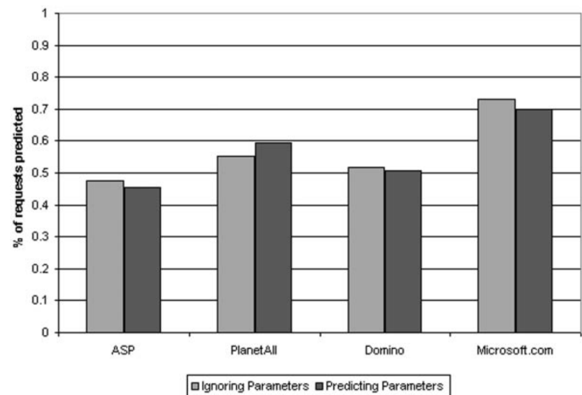


Fig. 2. Agreement prediction. This graph plots the percent of requests in our test log for which the agreement based prediction algorithm attempted to predict the next page to be requested.

In Fig. 3, the results from PlanetAll show client-viewed prediction rates above 40% for all algorithms

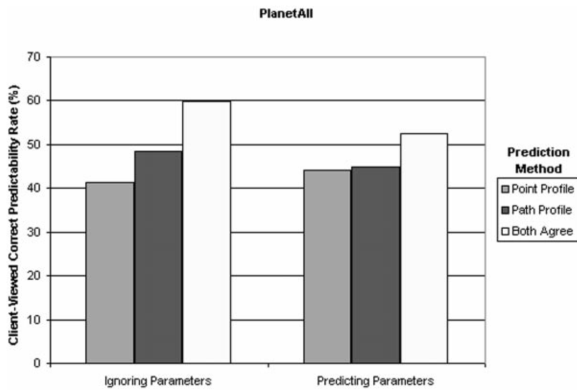


Fig. 3.

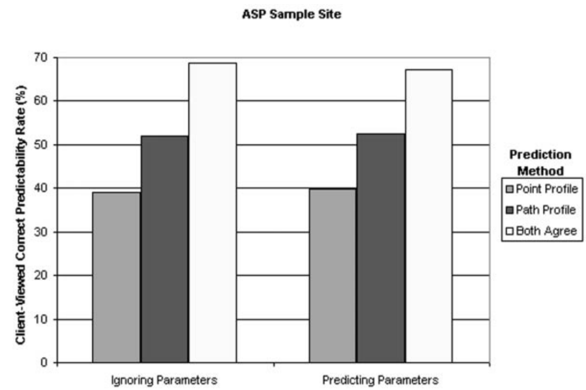


Fig. 5.

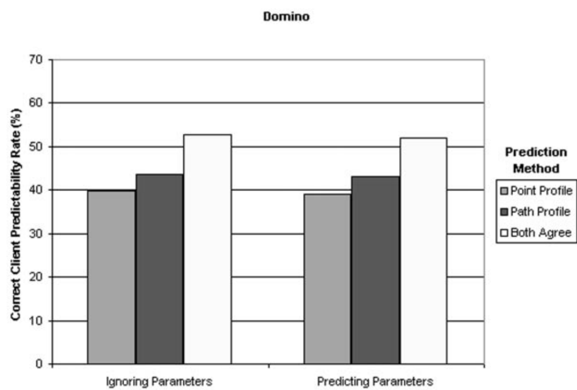


Fig. 4.

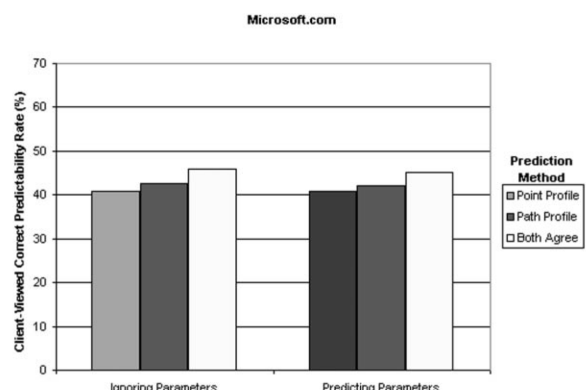


Fig. 6.

and both choices of parameter inclusion. Regardless of the handling of the parameter field, the path-based prediction rate was slightly superior to point-based prediction. Predictions made with agreement between point-based and path-based techniques were correct more than half the time.

The predictability rates from the Domino site, shown in Fig. 4, were only slightly lower than those for PlanetAll.

The results from Microsoft's ASP sample site were even more impressive. We see in Fig. 5 that path-based guesses predicted the correct next page more than half the time.

As expected the largest site, Microsoft.com, had the lowest levels of predictability (see Fig. 6). It is still impressive to note that a site with over 8000 URLs accessed over a short time period yielded predictability rates of over 40%.

While the predictability rates above do not approach the levels attainable for simpler problems such as predicting one of two directions for a conditional branch, these rates are surprisingly high when one considers the number of options a user has when she approaches a given page. These prediction rates are certainly high enough to justify further research into the area of page pre-generation.

6. Future and related work

Past attempts at predicting HTTP requests have focused on prefetching images that are referenced in HTML documents. The Wcol proxy, developed at the Nara Institute of Science and Technology, starts prefetching images from the server as it sends the HTML documents back to the client [2].

Predicting a user's next step is just one benefit of obtaining a having such a profile. There are many other uses of path profiles of HTTP server logs. For example, upon learning that the majority of users go to page *c* after going from page *a* to page *b*, an intelligent Web designer may find it beneficial, for both the user and the server, to place a direct link from page *a* to page *c*.

Another application of path profiling includes the use of user paths as an indication of user behavior. Existing tools, which classify users strictly on the behavior of accessing a page [1] may benefit by classifying users by the paths they take through these pages. By using paths, such tools may benefit from the information inherent in the temporal ordering of user accesses to the site.

7. Conclusion

Path profiles can be efficiently created from HTTP server logs. Using these profiles, we find that the HTTP requests generated by today's dynamic Web applications have a surprisingly high level of predictability.

Given that most servers spend a significant portion of CPU cycles idling, we should take advantage of this predictability and use these free cycles to hide the latency of page generation from the user as often as possible. To best utilize a limited number of free cycles, we can use a heuristic of predicting pages only when both the point-based and the path-based predictions agree. On corporate Intranets that do not suffer from the bandwidth problems faced by the larger Internet, Web applications could push predicted pages to their clients before the page is even requested by the user.

Acknowledgements

David Treadwell and J. Allard of Microsoft gave us the freedom to research techniques outside the immediate needs of the Internet Information Server team. George Reilly, also of Microsoft, provided ideas and technical support. Brian Robertson of **PlanetAll** [3] was essential in providing us with data from an independent site. The Lotus Corporation provided logs from an independent server product.

At Harvard **Brad Chen** [4] and **Margo Seltzer** [5] provided ideas and pointers to related work. **Clifford Young** [6] provided a great deal of insight into path profiling.

Michael D. Smith is funded in part by a NSF Young Investigator award (grant no. CCR-9457779), a DARPA grant no. NDA904-97-C-0225, and research gifts from AMD, Digital Equipment, HP, and Intel.

References

[1] Breese, J., Fayyad, U., and Heckerman, D., Conversations of 8/97, Microsoft Research, Microsoft Corporation.

[2] Information Network Laboratory, Nara Institute of Science and Technology, Japan, Wcol: WWW Collector Home Page, http://shika.aist-nara.ac.jp/products/wcol/wcolD.html

[3] Jacobsen, E., Rotenberg, E., and Smith, J., Assigning confidence to conditional branch predictions, in: *Proc. of the 29th Annual ACM/IEEE International Symposium on Microarchitecture,* December 1996, pp. 142–152.

[4] Microsoft Corporation, USA, Microsoft Site Server usage analyst technical details, http://backoffice.microsoft.com/products/features/UsageAnalyst/SiteServer_TechDetails.asp

[5] Young, R., Path based compilation, Ph.D. Thesis, Division of Engineering and Applied Sciences, Harvard University, 1997.

Stuart E. Schechter is working on his Ph.D. in Computer Science at Harvard University. He is also a perpetual intern at Microsoft, where he has worked on the Internet Information Server, Java VM, and Windows 95 teams. His research interests include architecture, compilation, anonymity in computer security, and HTTP server performance. He received a B.S. in Computer and Information Science from The Ohio State University in 1996 and will receive his Masters in Computer Science from Harvard in June, 1998. Home page: http://www.eecs.harvard.edu/~stuart

[3] http://www.PlanetAll.com

[4] http://www.eecs.harvard.edu/~bchen

[5] http://www.eecs.harvard.edu/~margo

[6] http://www.eecs.harvard.edu/~cyoung

Murali Krishnan is the lead performance and application infrastructure developer for Microsoft's Internet Information Server. He has developed parts of the core Web server for Windows NT, as well as tools and analysis to improve the Web server performance. Currently, he is working on enhanced application infrastructure and support for future Web server applications. He received a B.E. in Computer Science and Engineering from Anna University, India in 1992 and an M.S. in Computer Science from University of Wisconsin-Madison in 1994.

Michael D. Smith is an Associate Professor of Electrical Engineering and Computer Science in the Division of Engineering and Applied Sciences at Harvard University. His research focuses on the experimental realization of innovative compilation techniques and novel computer architectures to improve the capability and performance of computer systems. He received a B.S. degree in Electrical Engineering and Computer Science from Princeton University in 1983, a M.S. degree in Electrical Engineering from Worcester Polytechnic Institute in 1985, and a Ph.D. in Electrical Engineering from Stanford University in 1993. He is a member of the IEEE and the ACM, and he is the recipient of a 1994 NSF Young Investigator Award. Home page: http://www.eecs.harvard.edu/smith