

Automatic Detection of Fragments in Dynamically Generated Web Pages

Lakshmith Ramaswamy^{1*}

Arun Iyengar²

Ling Liu¹

Fred Douglass²

¹ College of Computing, Georgia Tech
801 Atlantic Drive
Atlanta GA 30332
{laks, lingliu}@cc.gatech.edu

² IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
{aruni, fdouglass}@us.ibm.com

ABSTRACT

Dividing web pages into fragments has been shown to provide significant benefits for both content generation and caching. In order for a web site to use fragment-based content generation, however, good methods are needed for dividing web pages into fragments. Manual fragmentation of web pages is expensive, error prone, and unscalable. This paper proposes a novel scheme to automatically detect and flag fragments that are cost-effective cache units in web sites serving dynamic content. We consider the fragments to be interesting if they are shared among multiple documents or they have different lifetime or personalization characteristics. Our approach has three unique features. First, we propose a hierarchical and fragment-aware model of the dynamic web pages and a data structure that is compact and effective for fragment detection. Second, we present an efficient algorithm to detect maximal fragments that are shared among multiple documents. Third, we develop a practical algorithm that effectively detects fragments based on their lifetime and personalization characteristics. We evaluate the proposed scheme through a series of experiments, showing the benefits and costs of the algorithms. We also study the impact of adopting the fragments detected by our system on disk space utilization and network bandwidth consumption.

Categories and Subject Descriptors

H.3.0 [Information Systems]: Information Storage and Retrieval—General

General Terms

Algorithms, Design, Performance

Keywords

Dynamic content caching, Fragment-based caching, Fragment detection, Shared fragments, L-P fragments

1. INTRODUCTION

The amount of information on the World Wide Web continues to grow at an astonishing speed. The number of dynamic web pages

*Most of this work was done while Lakshmith was an intern at IBM Research in the summers of 2002 and 2003.

Copyright is held by the author/owner(s).
WWW2004, May 17–22, 2004, New York, New York, USA.
ACM 1-58113-844-X/04/0005.

that are typically generated by programs executing at request time is also increasing at a rapid pace. Web caching technologies to date have been successful for efficient delivery of static web pages but they have not been so effective for delivering dynamic web content due to their frequent changing nature and their diversified freshness requirements.

Several efforts have been made to address the problem of efficient serving of dynamic pages, among which *Fragment*-based publishing and caching of web pages [2, 10, 11, 13] stands out; it has been successfully commercialized in recent years. Conceptually, a fragment is a portion of a web page which has a distinct theme or functionality and is distinguishable from the other parts of the page. A web page has references to these fragments, which are stored independently on the server and in caches. In the fragment-based publishing scheme, the cacheability and the lifetime are specified at a fragment level rather than at the page level.

The advantages of the fragment-based schemes are apparent and have been conclusively demonstrated [11, 13]. By separating the non-personalized content from the personalized content and marking them as such, it increases the cacheable content of the web sites. Furthermore, with the fragment-based solution, a whole web page need not be invalidated when only a part of that page expires. Hence the amount of data that gets invalidated at the caches is reduced. In addition, the information that is shared across web pages needs to be stored only once, which improves disk space utilization at the caches.

Fragment-based caching solutions typically rely on the web administrator or the web page designer to manually fragment the pages on the web site. Manual markup of fragments in dynamic web pages is both labor-intensive and error-prone. More importantly, identification of fragments by hand does not scale as it requires manual revision of the fragment markups in order to incorporate any new or enhanced features of dynamic content into an operational fragment-based solution framework. Furthermore, the manual approach to fragment detection becomes unmanageable and unrealistic for edge caches that deal with multiple content providers. Thus there is a growing demand for techniques and systems that can automatically detect “interesting” fragments in dynamic web pages, and that are scalable and robust for efficiently delivering dynamic web content. By interesting we mean that the fragments detected are cost-effective for fragment-based caching.

Automatic detection of fragments presents two unique challenges. First, compared with static web pages, dynamically generated web pages have three distinct characteristics. On the one hand, dynamic web pages seldom have a single theme or functionality and they

Fragments

Football Sport Today Page

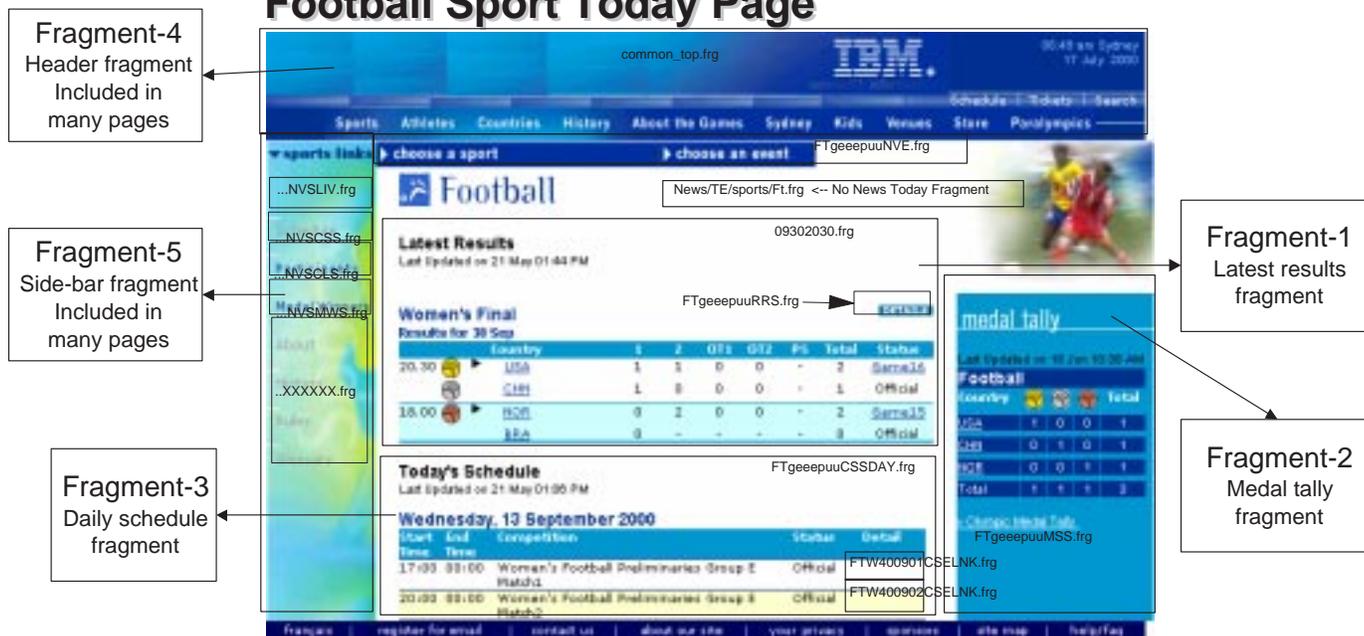


Figure 1: Fragments in a Web Page

typically contain several pieces of information with varying freshness or sharability requirements. On the other hand, most of the dynamic and personalized web pages are not completely dynamic or personalized. Often the dynamic and personalized content are embedded in relatively static web page templates [5]. Furthermore, dynamic web pages from the same web site tend to share information among themselves.

Figure 1 shows a dynamic web page generated through a fragment-based publishing system. This Football Sport Today Page was one of the web pages hosted by IBM for a sporting event. It contains five interesting fragments that are cost-effective candidates for fragment-based caching: (1) the latest football results on the women’s final, (2) the latest medal tally, (3) a daily schedule for women’s football, (4) the navigation menu with the IBM logo for the sport site on the top of the page and (5) the sport links menu on the left side of the page. These fragments differ from each other in terms of their themes, functionalities, and invalidation patterns. The latest results fragment changes at a different rate than the latest medal tally fragment, which in turn changes more frequently than the fragment containing the daily schedule. In contrast, the navigational menu on the top of the page and the sport links menu on the left side of the page are relatively static and are likely to be shared by many dynamic pages generated in response to queries on sport events hosted from the web site.

Second, it is apparent from the above example that humans can easily identify fragments with different themes or functionality based on their prior knowledge in the domain of the content (such as sports in this example). However, in order for machines and programs to automate the fragment detection process, we need mechanisms that on the one hand can correctly identify fragments with different themes or functionality without human involvement, and on the other hand are efficient and effective for detecting and flagging such fragments through a cross-comparison of multiple pages from a web site.

In this paper, we propose a novel scheme to automatically detect

and flag fragments in dynamic web pages which are cost-effective for fragment-based caching. We analyze web pages with respect to their information sharing behavior, personalization characteristics, and the change frequencies over time. Based on this analysis, our system detects and flags the “interesting” fragments in a web site. We consider a fragment interesting if it has good sharability with other pages served from the same web site or it has distinct lifetime characteristics. This paper contains three original contributions:

- First, we propose an efficient fragment-aware data structure to model dynamic web pages, including an augmented fragment tree with shingles encoding and a fast algorithm for computing shingles incrementally. This data structure forms the first step towards the efficient detection of fragments.
- Second, we present an efficient algorithm for detecting fragments that are shared among M documents, which we call the *Shared Fragment Detection Algorithm*. This algorithm has two distinctive features: (1) it uses node buckets to speed up the comparison and the detection of exactly or approximately shared fragments across multiple pages. (2) it introduces sharing factor, minimum fragment size, and minimum matching factor as the three performance parameters to measure and tune the performance and the quality of the algorithm in terms of the fragments detected.
- Third, we present an effective algorithm for detecting fragments that have different lifetime characteristics, which we call the *Lifetime-Personalization based (L-P) Fragment Detection Algorithm*. A unique characteristic of the L-P algorithm is that it detects fragments which are most beneficial to caching based on the nature and the pattern of the changes occurring in dynamic web pages.

We discuss several performance enhancements to these basic algorithms, and report our experiments for evaluation of the proposed

fragment detection scheme, showing the effectiveness and the cost of our approach.

2. CANDIDATE FRAGMENTS

Our goal for automatic fragment detection is to find interesting fragments in dynamic web pages, which exhibit potential benefits and thus are cost-effective as cache units. We refer to these interesting fragments as *candidate fragments* in the rest of the paper.

The web documents considered here are *well-formed* HTML documents [8] although the approach can be applied to XML documents as well. Documents that are not well formed can be converted to well-formed documents through document normalization, for example using HTML Tidy [3].

Concretely, we introduce the notion of candidate fragments as follows:

- Each Web page of a web site is a candidate fragment.
- A part of a candidate fragment is itself a candidate fragment if any one of the two conditions is satisfied:
 - The part is shared among “M” already existing candidate fragments, where $M > 1$.
 - The part has different personalization and lifetime characteristics than those of its encompassing (parent or ancestor) candidate fragment.

A formal definition of candidate fragments for web pages of a web site is given below:

DEFINITION 1. (Candidate Fragment)

Let NW denote the set of web pages available on a web site S and $CF(x)$ denote the set of all the fragments contained in fragment x . A fragment y is referred to as an ancestor fragment of another fragment x iff y directly or transitively contains fragment x . Let $AF(x)$ denote all the ancestor fragments of the fragment x and FS denotes the set of fragments corresponding to the set of documents D_i in NW , $FS = \cup_{i=1}^{|NW|} CF(D_i)$. For any document D from web site S , a fragment x in $FS(D)$ is called a candidate fragment if one of the following two conditions is satisfied:

1. x is a maximal Shared fragment, namely:
 - x is shared among M distinct fragments F_1, \dots, F_M , where $M > 1$, $F_i \in FS$, and if $i \neq j$ then $F_i \neq F_j$; and
 - there exists no fragment y such that $y \in AF(x)$, and y is also shared among the M distinct fragments F_1, \dots, F_M .
2. x is a fragment that has distinct personalization and lifetime characteristics. Namely, $\forall z \in AF(x)$, x has different personalization and lifetime characteristics than z .

We observe that this is a recursive definition with the base condition being that each web page is a fragment. It is also evident from the definition that the two conditions are independent. These conditions define fragments that benefit caching from two different and independent perspectives. We call the fragments satisfying Condition 1 **Shared fragments**, and the fragments satisfying Condition 2 **L-P fragments** (denoting Lifetime-Personalization based fragments). Lifetime characteristics of a fragment govern the time duration for which the fragment, if cached, would stay fresh (in tune with the value at the server). The personalization characteristics of a fragment correspond to the variations of the fragment in relation to cookies or parameters of the URL.

It can be observed that the two independent conditions in the candidate fragment definition correspond well to the two aims of fragment caching. By identifying and creating fragments out of the parts that are shared across more than one fragment, we aim to avoid unnecessary duplication of information at the caches. By creating fragments that have different lifetime and personalization properties we not only improve the cacheable content but also minimize the amount and frequency of the information that needs to be invalidated.

3. FRAGMENT DETECTION: THE BASICS

In this section we discuss the basic design of our automated fragment detection system, including the system architecture, the efficient fragment-aware data structure for automating fragment detection, and the important configurable parameters in our system.

3.1 System Overview

The primary goal of our system is to detect and flag candidate fragments from dynamic pages of a given web site. The fragment detection process is divided into three steps. First, the system is conceived to construct an *Augmented Fragment Tree* (AF tree) for the dynamic pages fed into the fragment detection system. Second, the system applies the fragment detection algorithms on the augmented fragment trees to detect the candidate fragments in the given web pages. In the third step, the system collects statistics about the fragments such as the size, how many pages share the fragment, access rates, etc. These statistics aid the administrator in deciding whether to enable fragmentation. Figure 2 gives a sketch of the system architecture.

We provide two independent fragment detection algorithms: one for detecting Shared fragments and the other for detecting Lifetime Personalization based (L-P) fragments. Both algorithms can be collocated with a server-side cache or an edge cache, and work on the dynamic web page dumps from the web site.

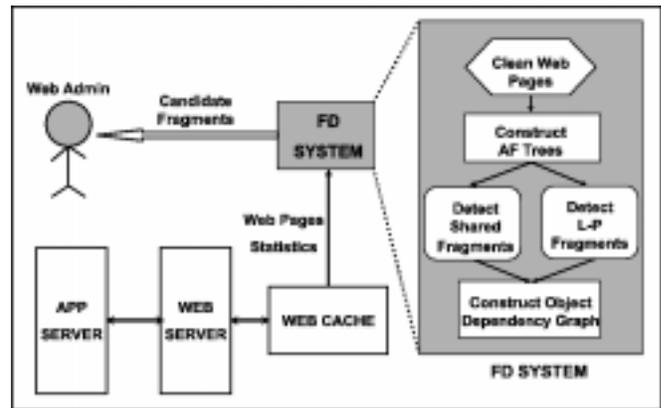


Figure 2: Fragment Detection System Architecture

The algorithm for detecting Shared fragments works on a collection of different dynamic pages generated from the same web site, whereas the L-P fragment detection algorithm works on different versions of each web page, which can be obtained from a single query being repeatedly submitted to the given web site. For example, in order to detect L-P fragments, we need to locate parts of a fragment that have different lifetime and personalization characteristics. This can be done by comparing different versions of the dynamic web page and detecting the parts that have changed over time and the parts that have remained constant. While the in-

put to the L-P fragment detection algorithm differs from the shared fragment detection algorithm, both algorithms work directly on the augmented fragment tree representation of its input web pages. The output of our fragment detection algorithms is a set of fragments that are shared among a given number of documents or that have different lifetime or personalization characteristics. This fragmentation information will then be served as recommendations to the fragment caching policy manager or the respective web administrator (see Figure 2).

3.2 Augmented Fragment Trees with Shingles Encoding

Detecting interesting fragments in web pages requires efficient traversal of web pages. Thus a compact data structure for representing the dynamic web pages is critical to efficient and accurate fragment detection. Of the several document models that have been proposed, the most popular model is the Document Object Model (DOM) [1], which models web pages using a hierarchical graph. However, the DOM tree structure is less efficient for fragment detection for a number of reasons. First, our fragment detection algorithms compare pages to detect those fragments whose contents are shared among multiple pages or whose contents have distinctive expiration times. The DOM tree of a reasonably sized HTML page has a few thousand nodes. Many of the nodes in such a tree correspond to text formatting tags that do not contribute to the content-based fragment detection algorithms. Second and more importantly, the nodes of the DOM do not contain sufficient information needed for fast and efficient comparison of documents and their parts. These motivate us to introduce the concept of an augmented fragment tree (AF tree), which removes the text formatting tag nodes in the fragment tree and adds annotation information necessary for fragment detection.

An augmented fragment (AF) tree with shingles encoding is a hierarchical representation of a web (HTML or XML) document with the following three characteristics: First, it is a compact DOM tree with all the text-formatting tags (e.g., <Big>, <Bold>, <I>) removed. Second, the content of each node is fingerprinted with Shingles encoding [6, 7, 18]. Shingles are fingerprints with the property that if a document changes by a small amount, its Shingles encoding also changes by a small amount. Third, each node is augmented with additional information for efficient comparison of different documents and different fragments of documents. Concretely each node in the AF tree is annotated with the following fields:

- Node Identifier (NodeID): A vector indicating the location of the node in the tree.
- NodeValue: A string indicating the value of the node. The value of a leaf node is the text itself, and the value of an internal node is NULL (empty string).
- SubtreeValue: A string that is defined recursively. For a leaf node, the SubtreeValue is equal to its NodeValue. For all internal nodes, the SubtreeValue is a concatenation of the SubtreeValues of all its children nodes and its own NodeValue. The SubtreeValue of a node can be perceived as the fragment (content region) of a web document anchored at this subtree node.
- SubtreeSize: An integer whose value is the length of SubtreeValue in bytes. This represents the size of the structure in the document being represented by this node.
- SubtreeShingles: An encoding of the SubtreeValue for fast comparison. SubtreeShingles is a vector of integers representing the shingles of the SubtreeValue.

We use Shingles because they have the property that if a document changes by a small amount, its Shingles also change by a small amount. Other fingerprinting techniques such as MD5 do not behave similarly. Figure 3 illustrates the high sensitivity of Shingles by comparing it with the MD5 hash through an example of two strings. The first and the second strings in Figure 3 are essentially the same strings with small perturbations (the portions that differ in the two strings have been highlighted). The MD5 hashes of the two strings are totally different, whereas the shingles of the two strings vary just by a single value out of the 8 values in the shingles set (shingle values that are present in one set but are absent in the other have been underlined in the diagram). This property of shingles has made it popular in estimating the resemblance and containment of documents [6].

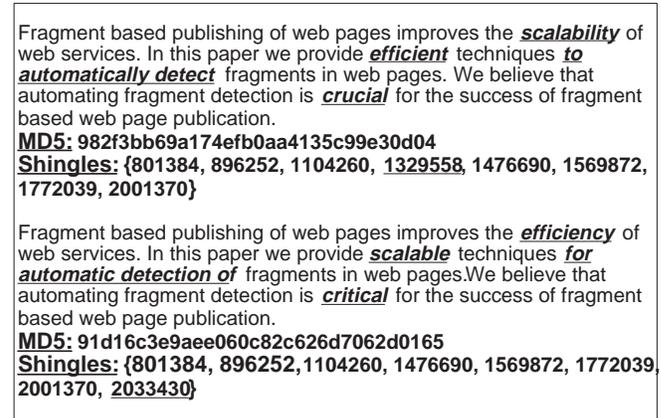


Figure 3: Example of Shingles versus MD5

AF Tree Construction

The first step of our fragment detection process is to convert web pages to their corresponding AF trees. The AF tree can be constructed in two steps. The first step is to transform a web document to its DOM tree and prune the fragment tree by eliminating the text formatting nodes. The result of the first step is a specialized DOM tree that contains only the content structure tags (e.g., like <TABLE>, <TR>, <P>). The second step is to annotate the fragment tree obtained in the first step with NodeID, NodeValue, SubtreeValue, SubtreeSize and SubtreeShingles. Once the SubtreeValue is known, we can use a shingles encoding algorithm to compute its SubtreeShingles. We briefly discuss the basic algorithm [6] to compute the shingles for a given string.

The Basic Shingling Algorithm

Any string can be considered as a sequence of tokens. The tokens might be words or characters. Let $Str = T_1T_2T_3...T_N$, where T_i is a token and N is the total number of tokens in Str . Then a shingles set of window length W and sample size S is constructed as follows. The set of all subsequences of length W of the string Str is computed. $SubSq = \{T_1T_2...T_W, T_2T_3...T_{W+1}, ..., T_{N-W+1}T_{N-W+2}...T_N\}$. Each of these subsequences is hashed to a number between $(0, 2^K)$ to obtain a token-ID. A hash function similar to Rabin's function [23] could be employed for this purpose. The parameter K governs the size of the hash value set to which the subsequences are mapped. If the parameter K is set to a small value many subsequences might be mapped to the same token-ID, leading to collisions. Larger values of K are likely to avoid these collisions of subsequence, but increase the size of the hash value set. We now have $(N - W + 1)$ token-IDs, each corresponding to

one subsequence. Of these $(N - W + 1)$ token-IDs, the minimum S are selected as the (W, S) shingles of string Str . The parameters W, S , and K can be used to tune the performance and quality of the shingles encoding.

The basic shingles computation algorithm is suitable for computing shingles for two independent documents. However, computing the shingles on the SubtreeValues independently at each node would entail unnecessary computations and is inefficient. This is simply because the content of every node in an AF tree is also a part of the content of its parent node. Therefore computing the SubtreeShingles of each node independently leads to a much higher cost due to duplicated shingles computation than computing the SubtreeShingles of a parent node incrementally. We propose an incremental shingles computation method and call it the **Hierarchical Shingles Computing** scheme (the **HiSh** scheme for short).

3.3 Efficient Shingles Encoding - The HiSh Algorithm

In this section we describe a novel method to compute shingles incrementally for strings with hierarchical structures such as trees. By incremental we mean the HiSh algorithm reuses the previously computed shingles in the subsequent computation of shingles.

Consider a string $A = A_1A_2A_3...A_nA_{n+1}...A_m$ with m tokens, $m \leq 1$. Let B and C be two non-overlapping substrings of A such that A is a concatenation of B and C . Let $B = A_1A_2...A_n$ and $C = A_{n+1}A_{n+2}...A_m$. Now we describe how to incrementally compute the (W, S) shingles of A , if (W, S) shingles of B and C are available. Let $Shng(A, W, S)$, $Shng(B, W, S)$ and $Shng(C, W, S)$ denote the (W, S) shingles of the strings A, B and C respectively. We define the *Overlapping Sequences* to be those subsequences which begin in B and end in C . These are the subsequences that are not completely present in either shingles of B or shingles of C . Let the hashes of these subsequences be represented by the set $OvlpHsh = \{Hsh(A_{(n-W+2, n+1)}), Hsh(A_{(n-W+3, n+2)}), \dots, Hsh(A_{(n, n+W-1)})\}$. Then we can obtain the (W, S) shingles of A as follows:

$$Shng(A, W, S) = Min_S \{ Shng(B, W, S) \cup Shng(C, W, S) \cup OvlpHsh \}$$

Here $Min_S(Z)$ represents the operation of selecting the S minimum values from values in set Z .

As the shingles of B and C are available, the only extra computations needed are to compute the hashes of overlapping sequences. This is the central idea of the HiSh algorithm. Figure 4 illustrates the working of the HiSh scheme on an example string. In this example, $(8, 4)$ shingles of the string B and string C are pre-computed and available, and we want to compute the $(8, 4)$ shingles of the concatenation of the two strings. The HiSh algorithm computes the overlapping subsequences between the two strings (which is shown as *Overlap* in the figure) and computes the shingles on this overlapping string. Finally, the algorithm selects the minimum 4 values from all the three strings to yield the shingles of the entire string.

Our experiments (see Section 6.4) indicate that the HiSh optimization can reduce the number of hashes computed in constructing the AF tree by as much as 9 times and improve the shingles computation time by about 6 times for 20-Kbyte documents, when compared to the basic algorithm. The performance gain will be greater for larger documents.

4. DETECTING SHARED FRAGMENTS

This section discusses our algorithm to detect shared fragments. Given a collection of N dynamic web pages generated in response

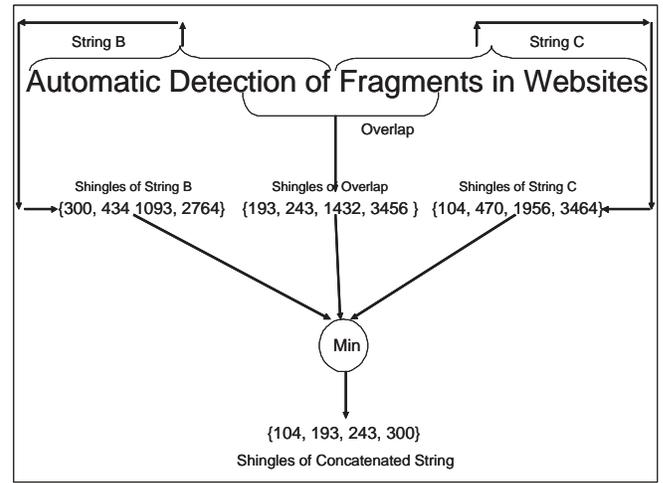


Figure 4: HiSh Algorithm

to distinct queries over a web site, let AF_i ($1 \leq i \leq N$) denote the AF tree of the i^{th} page. We call a fragment $F \in AF_i$ a *maximal shared fragment* if it is shared among M ($M < N$) distinct fragments (pages) and there is no ancestor fragment of F which is shared by the same M fragments (pages). Here M is a system-defined parameter. With this definition in mind, the immediate question is how to efficiently detect such shared fragments, ensuring that the fragments detected are cost-effective cache units and beneficial for fragment-based caching. Our experiences with fragment-based solutions show that any shared fragment detection algorithm should address the following two fundamental challenges. First, one needs to define the measurement metrics of sharability. In a dynamic web site it is common to find web pages sharing portions of content that are similar but not exactly the same. In many instances the differences among these portions of content are superficial (e.g., they have only formatting differences). Thus a good automatic fragment detection system should be able to detect these approximately shared candidate fragments. Different quantifications of what is meant by “shared” can lead to different quality and performance of the fragment detection algorithms. The second challenge is the need for an efficient and yet scalable implementation strategy to compare the fragments (and the pages) and identify the maximal shared fragments.

Approximate Sharability Measures

The Shared fragment detection algorithm operates on various web pages from the same web site and detects candidate fragments that are “approximately” shared. We introduce three measurement parameters to define the appropriateness of such approximately shared fragments. These parameters can be configured based on the needs of a specific application. The accuracy and the performance of the algorithm are dependent on the values of these parameters.

- **Minimum Fragment Size** ($MinFragSize$): This parameter specifies the minimum size of the detected fragment.
- **Sharing Factor** ($ShareFactor$): This indicates the minimum number of pages that should share a segment in order for it to be declared a fragment.
- **Minimum Matching Factor** ($MinMatchFactor$): This parameter specifies the minimum overlap between the SubtreeShingles to be considered as a shared fragment.

The parameter *MinFragSize* is used to exclude very small segments of web pages from being detected as candidate fragments. This threshold on the size of the documents is necessary because the overhead of storing the fragments and composing the page would be high if the fragments are too small. The parameter *ShareFactor* defines the threshold on the number of documents that have shared each candidate fragment. Finally, we use the parameter *MinMatchFactor* to model the significance of the difference between two fragments being compared. Two fragments being compared are considered as sharing significant content if the overlap between their SubtreeShingles is greater than or equal to *MinMatchFactor*.

Detecting Shared Fragments with Node Buckets

The shared fragment detection algorithm detects the shared fragments in two steps as shown in Figure 5. First, the algorithm creates a sorted pool of the nodes in the AF trees of all the web pages examined using node buckets. Then, the algorithm groups those nodes that are similar to each other together and runs the condition test for maximal shared fragments. If the number of nodes in the group exceeds the minimum number of pages specified by the *ShareFactor* parameter, and the corresponding fragment is indeed a maximal shared fragment, the algorithm declares the node group as a shared fragment and assigns it a fragment identifier.

Step 1: Putting Nodes into a sorted pool of node buckets

More concretely, our algorithm uses the *bucket* structures to create a sorted pool of nodes. The algorithm creates N_B buckets. Each bucket Bkt_i is initialized with bucket size B_{s_i} , and is associated with a pre-assigned range of the SubtreeSizes, denoted as $(MinSize(Bkt_i), MaxSize(Bkt_i))$. The AF trees are processed starting from the root of each tree, and a node is placed into an appropriate bucket based on its SubtreeSize, such that the SubtreeSizes of all nodes in bucket Bkt_i are between $MinSize(Bkt_i)$ and $MaxSize(Bkt_i)$. If in the process of putting nodes into buckets, a bucket grows out of its current size B_{s_i} , it will be split into two or more buckets. Similarly, if the first step results in a pool of buckets with uneven distribution of nodes per bucket, a merge operation will be used to merge two or more buckets into one.

After all the AF trees have been processed and the nodes entered into their corresponding buckets, each bucket is sorted based on the SubtreeSize of the nodes in the bucket. At the end of the process we have a set of buckets containing nodes, each of which is sorted based on the SubtreeSize of the node. The STEP 1 in Figure 5 shows the working of this step on two AF trees A and B. The nodes of the two trees are put into 5 buckets based on their SubtreeSizes. The buckets are sorted, and the buckets BT_3 , BT_4 and BT_5 are merged to obtain a set of sorted buckets.

There are three system-supplied parameters: (1) the number of buckets (N_B) employed for this purpose, (2) the size B_i of each bucket, and (3) the range of each bucket $(MinSize(Bkt_i), MaxSize(Bkt_i))$. Various factors may affect the decision on how to set these parameters, including the number of AF trees examined, the average number of nodes in each AF tree and the range of the SubtreeSizes of all the nodes. The performance of this step would be better if the nodes are evenly distributed in all the available buckets. One way to achieve such balanced distribution of nodes across all buckets is to set the ranges of the buckets at the lower end of the size spectrum to be smaller, and let the range of the buckets progressively increase for the buckets at the higher end of the size spectrum. This strategy is motivated by the following observations. First, it is expected that the number of nodes at a lower

level of the AF trees would be larger than the number of nodes at a higher level. Second, the SubtreeSizes of the nodes at the lower level is expected to be smaller than the SubtreeSizes of the nodes in the higher levels of the AF tree. Therefore, we encourage having smaller ranges for the buckets at the lower end of the size spectrum, and gradually increase the range of the buckets at the higher end of the size spectrum.

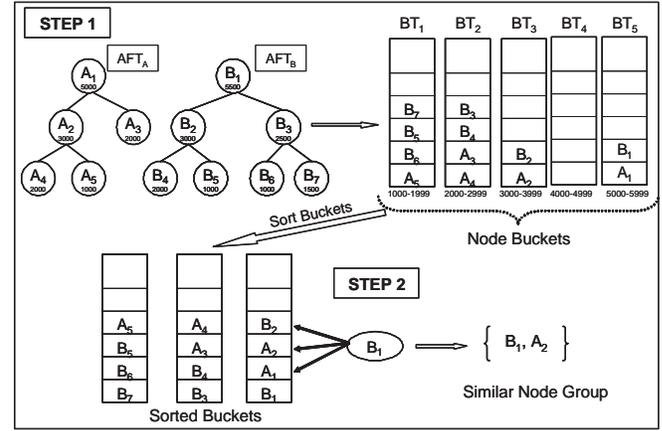


Figure 5: Shared Fragment Detection Algorithm

Step 2: Identifying maximal shared fragments through grouping of similar nodes

The output of the first step is a sorted pool of buckets in descending order of the ranges of buckets, and each bucket contains a list of nodes sorted by their SubtreeSizes. The task of the second step is to compare nodes and group nodes that are similar to each other together and then identify those groups of nodes that satisfy the definition of maximally shared fragments. This step processes the nodes in the buckets in decreasing order of their sizes. It starts with the node having the largest SubtreeSize, which is contained in the bucket with the highest *MaxSize* value. For each node being processed, the algorithm compares the node against a subset of the other nodes. This subset is constructed as follows. If we are processing a node A_i , then the subset of nodes that A_i is compared against should include all nodes whose sizes are larger than $P\%$ of the SubtreeSize of A_i , where P ranges from 0% to 100%. Let $CSet(A_i)$ denote the subset of nodes with respect to node A_i . We can use the following formula to compute $CSet(A_i)$.

$$CSet(A_i) = \{A_j | SubtreeSize(A_j) \geq \frac{P \times SubtreeSize(A_i)}{100}\}$$

It is important to note that the value setting of the parameter P has implications on both the performance and the accuracy of the algorithm. If P is too low, it increases the number of comparisons performed by the algorithm. If P is very close to 100, then the number of comparisons decrease; however, it might lead the comparison process to miss some nodes that are similar. In practice we have found a value of 90% to be appropriate for most web sites.

When comparing the node being processed with the nodes in its $CSet$, the algorithm compares the SubtreeShingles of the nodes. All such nodes whose shingles overlap more than the minimal matching factor specified by *MinMatchFactor* with the shingles of the node being processed are grouped together. Step 2 of Figure 5 demonstrates the comparison and grouping of the nodes in the sorted buckets.

If this group has at least *ShareFactor* nodes then we have the possibility of detecting it as a fragment. However before we declare the group as a candidate fragment, we need to ensure that the fragment corresponding to this group of nodes is indeed a maximally shared fragment and not a trivial fragment. To ease the decision on whether a group of nodes with similar shingles is a maximally shared fragment, we mark the descendent of each declared fragment with the fragment-ID assigned to the fragment. When similar nodes are detected, we check whether the ancestors of all of the nodes belong to the same fragment. If so, we reject the node group as a trivial fragment. Otherwise we declare the node group as a candidate fragment, assign it a fragment-ID and mark all of the descendant nodes with the fragment-ID. Once we declare a node-group as a candidate fragment, we remove all the nodes belonging to that group from the buckets. The algorithm proceeds by processing the next largest node in the node group in the same manner.

5. DETECTING L-P FRAGMENTS

The L-P fragments are lifetime personalization based fragments. Typically, the L-P fragments have different lifetime and personalization characteristics than their encompassing (parent) fragment. One way to detect the L-P fragments is to compare various versions of the same web page and track the changes occurring over different versions of the web page. The nature and the pattern of the changes may provide useful lifetime and personalization information that is helpful for detecting the L-P fragments.

The first challenge in developing an efficient L-P fragment detection algorithm is to identify the logical units in a given web page that may change over different versions, and to discover the nature of the change. Web pages can undergo a variety of changes between versions. Parts of a web page might be deleted or moved around in the web page, and new parts may be added. Therefore a simple algorithm that only compares the parts appearing at the same relative position in different versions of the web page is unlikely to yield accurate fragments.

The second challenge is to detect candidate fragments that are most beneficial to caching. Suppose we have a structure such as a table in the web page being examined. Suppose the properties of the structure remain constant over different versions of the web page, but the contents of the structure have changed over different versions. Now there are two possible ways to detect fragments: Either the whole table (structure) can be made a fragment or the substructures in the table (structure) can be made fragments. Which of these would be most beneficial to caching depends upon what percentages of the substructures are changing and how they are changing (frequency and amount of changes).

In the design of our L-P fragment detection algorithm, we take a number of steps to address these two challenges. First, we augment the nodes of each AF tree with an additional field *NodeStatus*, which takes one value from the set of three choices $\{UnChanged, ValueChanged, PositionChanged\}$. Second, we provide a shingles-based similarity function to compare different versions of a web page, and determine the portions of a web page that have distinct lifetime and personalization characteristics. Third, we construct the Object Dependency Graph (ODG) [11] for each web document examined on top of all candidate fragments detected. An Object Dependency Graph is a graphical representation of the containment relationship between the fragments of a web site. The nodes of the ODG correspond to the fragments of the web site and the edges denote the containment relationship among them. Finally, we propose to use the following configurable parameters to measure the quality of the L-P fragments in terms of cache benefit and to tune the performance of the algorithm:

- **Minimum Fragment Size**(*MinFragSize*): This parameter indicates the minimum size of the detected fragment.
- **Child Change Threshold**(*ChildChangeThreshold*): This parameter indicates the minimum fraction of children of a node that should change in value before the parent node itself can be declared as *ValueChanged*. This parameter can take a value between 0.0 and 1.0.

The L-P fragment detection algorithm works on the AF trees of different versions of web pages. It installs the first version (in chronological order) available as the *base version*. The algorithm compares each subsequent version to the base version and identifies candidate fragments. A new base version is installed whenever the web page undergoes a drastic change when compared with the current base version. In each step, the algorithm executes in two phases. In the first phase the algorithm marks the nodes that have changed in value or in position between the two versions of the AF tree. In the second phase the algorithm outputs the L-P fragments which are then merged to obtain the object dependency graph.

Phase 1: Comparing the AF trees and detecting the changes

Concretely, if we have two AF trees *A* and *B* corresponding to two versions of a web page, our algorithm compares each node of the tree *B*, to a node from *A* which is most similar to it. We employ the *SubtreeShingles* of the nodes for similarity comparison. Let $ShingleSim(A_i, B_j)$ denote the similarity function based on similarity of shingles of A_i and B_j . We can compute $ShingleSim(A_i, B_j)$ using the following formula:

$$ShingleSim(A_i, B_j) = \frac{SubtreeShingles(A_i) \cap SubtreeShingles(B_j)}{SubtreeShingles(A_i) \cup SubtreeShingles(B_j)}$$

If we are processing node B_j from AF tree *B*, we obtain a node A_i from tree *A* such that $ShingleSim(A_i, B_j) \geq OvlpThrshld$, and there exists no A_h such that

$$ShingleSim(A_h, B_j) > ShingleSim(A_i, B_j)$$

where *OvlpThrshld* denotes a user-specified threshold for the quantity *ShingleSim*, which can take a value between 0 and 1.0. If no such node is found in tree *A*, then it means that there is no node in *A* that is similar to the node B_j . This means that this node corresponds to a part that has been added in this version. Hence, the node B_j is marked as *ValueChanged*.

If a node A_i is found similar to node B_j , the algorithm begins comparing node B_j with node A_i . The algorithm compares the *SubtreeValues* and the *NodeIDs* of the two nodes. If both *SubtreeValue* and *NodeID* of the two nodes exactly match then the node is marked *UnChanged*. If the *NodeIDs* of the two nodes differ, then it means that the node has changed its position in the tree and hence it is marked as *PositionChanged*.

If the *SubtreeValues* of the nodes A_i and B_j do not exactly match then the algorithm checks whether they are leaf nodes. If so, they are marked as *ValueChanged*. Otherwise, the algorithm recursively processes each child node of B_j in the same manner described above marking them as *ValueChanged*, *PositionChanged* or *UnChanged*.

The algorithm addresses the second issue of discovering the fragments based on the extent of changes it is undergoing by calculating the fraction of B_j 's children that are marked as *ValueChanged*. If this fraction exceeds a preset threshold, which we call the *ChildChangeThreshold*, then B_j itself is marked as *ValueChanged*. The algorithm recursively marks all the nodes in the tree in the first phase.

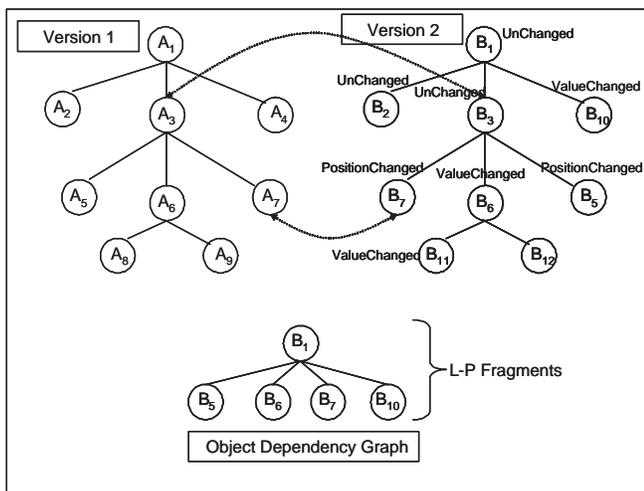


Figure 6: L-P Fragment Detection Algorithm

Phase 2: Detecting and labeling the L-P fragments

In the second phase, the algorithm scans the tree again from the root and outputs the nodes that are marked as *ValueChanged* or *PositionChanged*. In this pass the algorithm descends into a node's children if the node is marked as *PositionChanged* or *UnChanged*. If the node is marked as *ValueChanged*, the algorithm outputs it as a L-P fragment, but does not descend into its children. This ensures that we detect maximum-sized fragments that change between versions.

Figure 6 demonstrates the execution of one step in the L-P fragment detection algorithm. In the figure we compare the nodes of the AF tree of version 2 with the appropriate nodes of the AF tree of version 1. For example the node B_7 is compared with A_7 although these nodes appear at different positions in the two AF trees. We also indicate the *NodeStatus* of each node in version 2. In this example we set the *ChildChangeThreshold* to be 0.5. The node A_6 is marked as *ValueChanged* as both of its children have changed in value. The figure also indicates the fragments discovered in the second pass of the algorithm.

In summary, our L-P fragment detection algorithm detects the parts of a web page that change in value and parts of web pages changing their position between versions. Only the nodes that have changed in value are counted when deciding about the status of the parent node. The nodes that have changed only in position are as good as being unchanged for this purpose. This is because when a node just changes its relative position within its parent node, the value of the parent node would not change to a considerable extent.

6. EXPERIMENTAL EVALUATION

We have performed a range of experiments to evaluate our automatic fragment detection scheme. In this section we report four sets of experiments. The first and second sets of experiments test the two fragment detection algorithms, showing the benefits and effectiveness of the algorithms. The third set studies the impact of the fragments detected by our system on improving the caching efficiency, and the fourth set of experiments evaluates the Hierarchical Shingles computation scheme.

The input to the schemes is a collection of web pages including different versions of each page. Therefore we periodically fetched web pages from the web sites of BBC (<http://news.bbc.co.uk>), IBM's portal for marketing (<http://www.ibm.com/us>), Internetnews

(<http://www.internetnews.com>) and Slashdot (<http://www.slashdot.org>) and created a web 'dump' for each web site. While most of these sites share information across their web pages and hence are good candidates for Shared fragment detection, the Slashdot web page forms a good candidate for L-P fragment detection for reasons explained in Section 6.2.

6.1 Detecting Shared Fragments

In our first set of experiments, we study the behavior of our Shared fragment detection algorithm. The data sets used in this experimental study were web page dumps from BBC, Internet news and IBM. Due to space limitations, we primarily report the results obtained from our experiments on the BBC web site.

BBC is a well-known news portal. Primarily, the web pages on the BBC web site can be classified into two categories: web pages reporting complete news and editorial articles (henceforth referred to as the 'article' pages) and the 'lead' pages listing the top news of the hour under different categories such as 'World', 'Americas' 'UK' etc. We observed that there is considerable information sharing among the lead pages. Therefore, the BBC web site is a good case study for detecting shared fragments.

Our data set for the BBC web site was a web dump of 75 distinct web pages from the web site collected on 14th July 2002. The web dump included 31 'lead' pages and 44 'article' pages.

Figure 7 illustrates the number of Shared fragments detected at two different values of *MinFragSize* and *MinMatchFactor* (recall that *MinFragSize* is the minimum size of the detected fragment and *MinMatchFactor* is the minimum percentage of shingles overlap). When *MinFragSize* was set to 30 bytes and *MinMatchFactor* was set to 70%, the number of fragments detected was 350. The number of fragments increased to 358 when the *MinMatchFactor* was set to 90% and to 359 when the *MinMatchFactor* was set to 100%. In all of our experiments we observed an increase in the number of detected fragments with increasing *MinMatchFactor*.

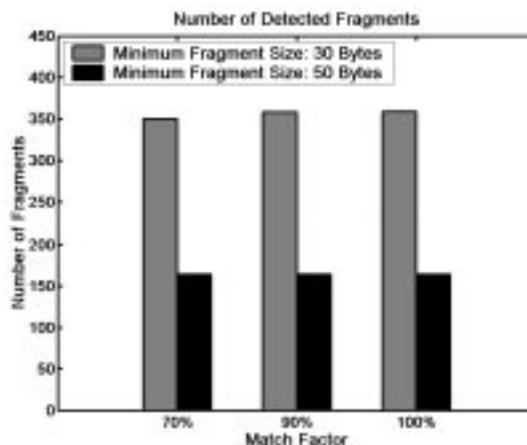


Figure 7: Number of Fragments Detected for BBC Data set

This phenomenon can be explained as follows. When *MinMatchFactor* is set to a high value, the algorithm looks for (almost) perfect matches. Suppose we had a node A in the AF tree of one document, with children B, C and D. Suppose the same node is present in the AF tree of another document, but in this case it has children B, C and E. Obviously, the nodes in the two trees don't match perfectly. If *MinMatchFactor* of the algorithm were set to 90%, then the nodes B and C would be detected as fragments.

If on the other hand *MinMatchFactor* were set to 70%, then the parent node *A* would be detected as one single fragment. So when the *MinMatchFactor* is set to higher values, the number of fragments detected increases. However, the size of the detected fragments falls with increasing *MinMatchFactor*.

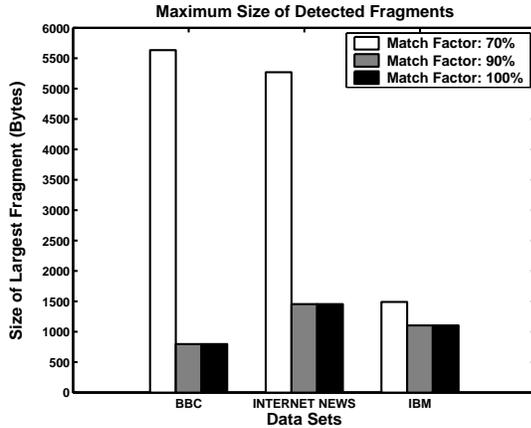


Figure 8: Maximum Size of the Detected Fragments

Figure 8 illustrates this effect. The graph indicates the maximum size of the detected fragments for various data sets when *MinMatchFactor* was set to 70% and 90%. For the BBC web site, the change in the size of the largest detected fragment is rather drastic. The size falls from 5633 bytes to 797 bytes when *MinMatchFactor* increases from 70% to 90%.

The pie chart in Figure 9 indicates the percentage of fragments according to the number of pages sharing the fragments for the BBC data set. We see a large number of fragments (a little over 50%) are being shared by exactly two pages. 13% of the fragments were shared among exactly 3 pages, and 11% of the pages were shared by 10 pages or more. All 75 pages shared one fragment, and 3 fragments were shared by 69 pages. The mean of the number of pages sharing each of the detected fragments was 13.8.

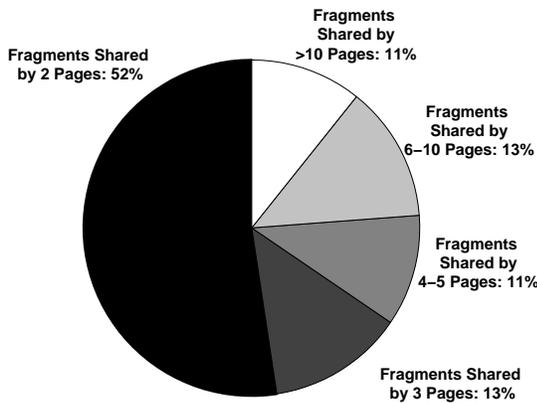


Figure 9: Distribution of Fragment Sharing for the BBC Data set

A similar type of behavior was observed in all three data sets. A large percentage of the detected fragments were shared by a small number of pages, but a few fragments were shared by almost all the web pages of the site.

6.2 Detecting L-P Fragments

We now present the experimental evaluation of the L-P fragment detection algorithm. Though we experimented with a number of web sites, due to space limitations, we restrict our discussion to the web site from Slashdot (<http://www.slashdot.org>).

Slashdot is a well known web site providing IT, electronics and business news. The front page of the Slashdot web site carries headlines and synopses of the articles on the site. The page indicates the number of comments posted by other users under each article. Thus, as new comments are added to existing articles and new articles are added to the web site, the page changes in small ways relative to the entire content of the page. It therefore forms a good case for L-P fragment detection, as well as other techniques that identify similarity across pages. The same Slashdot data set has been used in another study of similarity across pages at the level of unstructured bytes, finding that different versions of the Slashdot home page within a short time frame are extremely compressible relative to each other [17].

This web page provides a good case study to detect L-P fragments for a number of reasons. First, this web page is highly dynamic. Not only are there parts of the page that change every few minutes, the web page experiences major changes every couple of hours. Second, various portions of the web page have different lifetime characteristics. Third, the web page experiences many different kinds of changes like additions, deletions, value updates etc. Furthermore, there are parts of the web page that are personalized to each user.

Table 1 provides a synopsis of the results of the L-P fragment detection experiments. A total of 79 fragments were detected when the *ChildChangeThreshold* was set to 0.50, and 285 fragments were detected when *ChildChangeThreshold* was set to 0.70. We observe that higher numbers of fragments are detected when *ChildChangeThreshold* is set to higher values.

ChildChangeThreshold indicates the threshold for the percentage of the children to change in value before the parent itself is flagged as changed. If this threshold is set at higher values, it is more likely that nodes that are located deeper in the tree are flagged as fragments. As there are more nodes deeper in the tree, the number of fragments detected is higher. Equivalently, the average size of the fragment decreases as *ChildChangeThreshold* increases. Therefore we note that when *ChildChangeThreshold* is set at higher values, larger numbers of small fragments are detected. When it is set to lower values, fewer numbers of large fragments are detected.

In both cases, the depth of fragmentation was 3. When *ChildChangeThreshold* was set to 0.50, the number of fragments detected at depths 1, 2 and 3 were respectively 10, 7 and 62.

<i>ChildChangeThreshold</i>	0.50	0.70
Total Fragments	79	285
Average Fragment Size	822	219
Depth of Fragmentation	3	3

Table 1: Statistics for L-P Fragment Detection

6.3 Impact on Caching

Having discussed the experimental evaluation of our fragment detection system with regard to its accuracy and efficiency, we now study the impact of fragment caching on the performance of the cache, the server and the network when web sites incorporate fragments detected by our system into their respective web pages.

We start out by studying the savings in the disk space requirements of a fragment cache when the web pages incorporate the fragments discovered by our fragment detection system in comparison to a page cache that stores entire pages. Earlier we had explained the experimental evaluation of our shared fragment detection system on the BBC data set. We now compare the disk space needed to store the web pages in the data set when they are stored at the page granularity with disk space requirements for storing these web pages when they are fragmented as determined by our system.

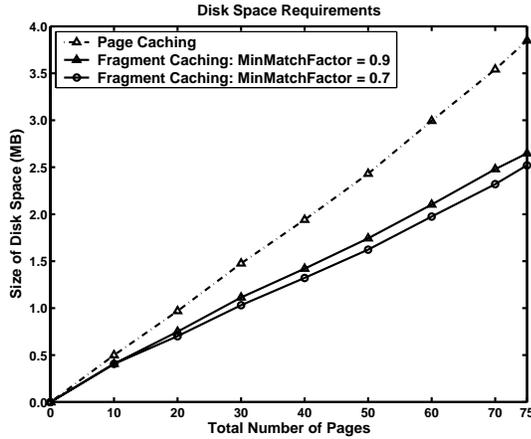


Figure 10: Total Storage Requirements for Page and Fragment Caches

Figure 10 indicates the total storage requirements as a function of the number of pages both for page caches and fragment caches. The graph shows that caching at the fragment level requires 22% to 31% less disk space than the conventional page level caching. The graph also shows that the improvements are higher when *MinMatchFactor* is set to low values. This is because when *MinMatchFactor* is set to low values, larger size fragments are discovered. When they are stored only once rather than being replicated, the savings obtained in terms of the disk space are higher.

Next we study the effects of L-P fragments detected by our system on the load on the network connecting the cache and the server. As we discussed in Section 2, incorporating L-P fragments into web pages reduces the amount of data invalidated at the caches, which in turn reduces the load on the origin servers and the backbone network. In order to study the impact of the L-P fragments on the server and network load, we use the L-P fragments detected by our algorithm on the Slashdot web site.

To study the load on the network we also need the access and the invalidation patterns of the web pages. As we did not have accurate traces indicating the patterns of access and invalidations, we make certain assumptions, which aid us to create a model for accesses and invalidations of these web pages.

First, we assume that the requests for web pages arrive according to a *Poisson process*, as supported by past analysis [19]. We assume the request arrival rate to be 100 requests per second. Second, we also model the invalidation processes of individual fragments as Poisson processes. We assume that the invalidation process of each fragment is independent of any other fragment in the web page. The invalidation rates in our experiments vary from 0.0001 invalidations per second to 5 invalidations per second.

Figure 11 indicates the total bytes transferred as a function of the number of requests arriving at the cache, at fragment invalidation rates of 0.001 and 0.1 invalidations per second. The X-axis

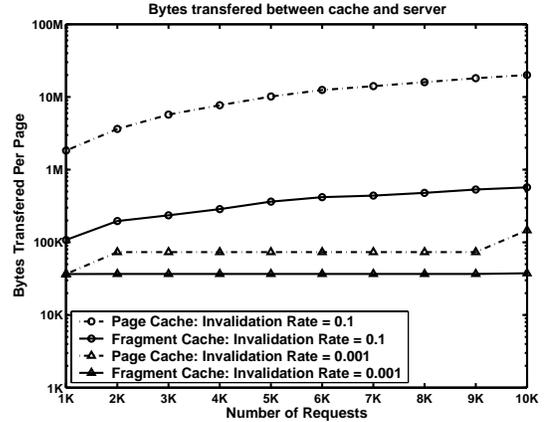


Figure 11: Bytes Transferred between Server and Cache

indicates the number of accesses and the Y-axis indicates the total number of bytes transferred, on log scale.

The number of bytes transferred for page-level caching is always higher than for fragment-level caching. When the invalidation rates are high, this effect is more pronounced. This experiment demonstrates the effectiveness of caching the fragments discovered by our fragment detection system in reducing the load on the network connecting the cache to the origin server.

6.4 Improving Fragment Detection Efficiency

In this section we evaluate the performance enhancements proposed by us. We have proposed a number of techniques to improve the performance of the fragment detection process including an incremental scheme to compute the *SubtreeShingles* of the nodes in the AF trees (HiSh algorithm). Due to space constraints, we restrict our discussion to the experimental evaluation of the HiSh algorithm.

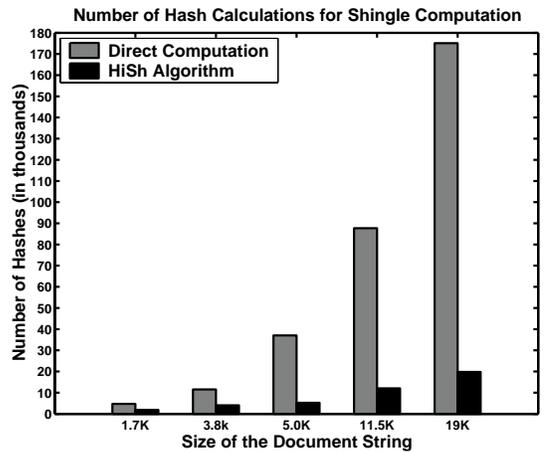


Figure 12: Number of Hashes Computed in Direct and HiSh schemes

Figure 12 shows the total number of hash computations involved in constructing the AF tree. For a document with 1.7K characters in its content string, the number of hash computations needed for the HiSh scheme is 2.6 times less than the number of hashes computed in the direct computation. For a document whose con-

tent string is 19K characters, the number of hashes computed in the HiSh scheme is almost 8.5 times less than the number of hashes computed in the direct computation. We note that the benefits of the HiSh algorithm are greater for larger documents.

7. RELATED WORK

Fragment-based publishing, delivery and caching of dynamic data have received considerable attention from the research community in recent years [11, 13]. Edge Side Includes [2] is a markup language to define web page components for page assembly at the edge caches. ESI provides mechanisms for specifying the cacheability properties at fragment level. Mohapatra et al. [21] discuss a fragment-based mechanism to manage quality of service for dynamic web content. Chan and Woo [12] use the structural similarity existing among various pages of a single site to efficiently delta-encode multiple web pages over time. Naaman et al. [22] present analytical and simulation based studies to compare ESI and delta-encoding, finding that ESI has potential performance advantages due to its ability to deliver only changing fragments. In addition to the above work, there is a considerable amount of literature in the more general area of the generation, delivery and caching of dynamic content [9, 10]. None of these previous papers addresses the problem of how to automatically detect fragments in web pages, however.

The work of Bar-Yossef and Rajagopalan [5] is related to our research on automated fragment detection, although the authors were addressing a different problem. They discuss the problem of template detection through discovery of *pagelets* in the web pages. However, our work differs from the work on template detection both in context and content. First, the work on template detection is aimed towards improving the precision of search algorithms. Our work is aimed at detecting fragments that are most beneficial to caching and content generation. Second, the syntactic definition of a pagelet in their paper is based on the number of hyperlinks in the HTML parse tree elements. They define a pagelet as an HTML element in the parse tree of a web page such that none of its children have at least K hyperlinks and none of its ancestors is a pagelet. This definition is very different from our working definition of a candidate fragment provided in Section 2. Further, their definition of pagelets forbids recursion. In contrast we permit embedded fragments. Third, our system has two algorithms: one to detect Shared fragments and another to detect L-P fragments. Both of these detect embedded fragments.

There has been significant work in identifying web objects that are identical, either at the granularity of entire pages or images [4, 16, 20] or pieces of pages [24], using MD5 or SHA-1 hashes to detect and eliminate redundant data storage and transfer. While the motivations of these researches are similar to that of the shared fragment detection algorithm, they are more restrictive in the sense that they work on full HTML pages and can only detect and eliminate pages (or byte-blocks) which are exact replicas. Pages that are similar at the level of entire web pages [25, 14] or pieces of web pages [17] can be identified using resemblance detection [6] and then delta-encoded. While these techniques have the potential to reduce transfer sizes, decomposing web pages into separately cached fragments accomplishes similar reductions in size without the need for explicit version management.

In addition to these, discovering and extracting objects from web pages has received considerable attention from the research community [8, 15]. While these projects aim at extracting objects based on the nature of the information they contain, our work concentrates on discovering fragments based on their lifetime, personalization and sharing characteristics.

8. CONCLUSIONS

There has been heavy demand for technologies to ensure timely delivery of fresh dynamic content to end-users [9, 10, 11]. Fragment-based generation and caching of dynamic web content is widely recognized as an effective technique to address this problem. However, past work in the area has not adequately addressed the problem of how to divide web pages into fragments. Manual fragmentation of web pages by a web administrator or web page designer is expensive and error-prone; it also does not scale well.

In this paper we have presented a novel scheme to automatically detect and flag “interesting” fragments in dynamically generated web pages that are cost-effective cache units. A fragment is considered to be interesting if it is shared among multiple pages or if it has distinct lifetime or personalization characteristics. This scheme is based on analysis of the web pages dynamically generated at given web sites with respect to their information sharing behavior, personalization properties and change patterns. Our approach has three unique features. First, we propose a hierarchical and fragment-aware model of the dynamic web pages and a data structure that is compact and effective for fragment detection. Second, we present an efficient algorithm to detect maximal fragments that are shared among multiple documents. Third, we develop an algorithm that effectively detects fragments based on their lifetime and personalization characteristics. We evaluate the proposed scheme through a series of experiments, showing the benefits and costs of the algorithms. We also report our study on the impact of adopting the fragments detected by our system on disk space utilization and network bandwidth consumption.

9. REFERENCES

- [1] Document Object Model - W3C Recommendation. <http://www.w3.org/DOM>.
- [2] Edge Side Includes - Standard Specification. <http://www.esi.org>.
- [3] HTML TIDY. <http://www.w3.org/People/Raggett/tidy/>.
- [4] H. Bahn, H. Lee, S. H. Noh, S. L. Min, and K. Koh. Replica-Aware Caching for Web Proxies. *Computer Communications*, 25(3), 2002.
- [5] Z. Bar-Yossef and S. Rajagopalan. Template Detection via Data Mining and its Applications. In *Proceedings of WWW-2002*, May 2002.
- [6] A. Broder. On resemblance and Containment of Documents. In *Proceedings of SEQUENCES-97*, 1997.
- [7] A. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic Clustering of the Web. In *Proceedings of WWW-6*, April 1997.
- [8] D. Buttler and L. Liu. A Fully Automated Object Extraction System for the World Wide Web. In *Proceedings of ICDCS-2001*, 2001.
- [9] K. S. Candan, D. Agrawal, W.-S. Li, O. Po, and W.-P. Hsiung. View Invalidation for Dynamic Content Caching in Multi tiered Architectures. In *Proceedings of VLDB-2002*, September 2002.
- [10] J. Challenger, A. Iyengar, and P. Dantzic. A Scalable System for Consistently Caching Dynamic Web Data. In *Proceedings of IEEE INFOCOM 1999*, March 1999.
- [11] J. Challenger, A. Iyengar, K. Witting, C. Ferstat, and P. Reed. Publishing System for Efficiently Creating Dynamic Web Content. In *Proceedings of IEEE INFOCOM 2000*, May 2000.
- [12] M. C. Chan and T. W. C. Woo. Cache-Based Compaction: A New Technique for Optimizing Web Transfer. In

Proceedings of INFOCOM-1999.

- [13] A. Datta, K. Dutta, H. Thomas, D. VanderMeer, Suresha, and K. Ramamritham. Proxy-Based Acceleration of Dynamically Generated Content on the World Wide Web: An Approach and Implementation. In *Proceedings of SIGMOD-2002*, June 2002.
- [14] F. Douglis and A. Iyengar. Application-Specific Delta Encoding Via Resemblance Detection. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.
- [15] X.-D. Gu, J. Chen, W.-Y. Ma, and G.-L. Chen. Visual Based Content Understanding towards Web Adaptation. In *Proceedings of AH-2002*, 2002.
- [16] T. Kelly and J. Mogul. Aliasing on the World Wide Web: Prevalence and Performance Implications. In *Proceedings of the 11th International World Wide Web Conference*, May 2002.
- [17] P. Kulkarni, F. Douglis, J. LaVoie, and J. Tracey. Redundancy Elimination Within Large Collections of Files. In *Proceedings of the USENIX Annual Technical Conference*, June 2004. To appear.
- [18] U. Manber. Finding Similar Files in a Large File System. In *Proceedings of USENIX-1994*, January 1994.
- [19] J. Mogul. Network Behavior of a Busy Web Server and its Clients. Technical report, DEC Western Research Laboratories, 1995.
- [20] J. Mogul, Y. Chan, and T. Kelly. Design, Implementation, and Evaluation of Duplicate Transfer Detection in HTTP. In *Proceedings of NSDI '04*, March 2004. To appear.
- [21] P. Mohapatra and H. Chen. A Framework for Managing QoS and Improving Performance of Dynamic Web Content. In *Proceedings of GLOBECOM-2001*, November 2001.
- [22] M. Naaman, H. Garcia-Molina, and A. Paepcke. Evaluation of ESI and Class-Based Delta Encoding. In *Proceedings of WCW - 2003*.
- [23] M. O. Rabin. Fingerprinting by Random Polynomials. Technical report, Center for Research in Computing Technology, Harvard University, 1981.
- [24] S. C. Rhea, K. Liang, and E. Brewer. Value-Based Web Caching. In *Proceedings of 12th WWW Conference*, 2003.
- [25] T. Suel, P. Noel, and D. Trendafilov. Improved File Synchronization Techniques for Maintaining Large Replicated Collections Over Slow Networks. In *Proceedings of ICDE 2004*, March 2004. To appear.