URL Forwarding and Compression in Adaptive Web Caching

B. Scott Michel, Konstantinos Nikoloudakis, Peter Reiher, Lixia Zhang

Abstract— **Web caching is generally acknowledged as an important service for alleviating focused overloads when certain web servers' contents suddenly become popular. Cooperative caching systems are more effective than independent caches due to the larger collective backing store that cooperation creates. One such system currently being developed at UCLA, *Adaptive Web Caching* (AWC), uses an application-level forwarding table to locate the nearest copy of a requested URL's contents. This paper describes one specific design in AWC, a simple URL table compression algorithm allowing efficient content information sharing among neighboring caches. The compression algorithm is based on a hierarchical URL decomposition to aggregate URLs sharing common prefixes and an incremental hashing function to minimize collisions between prefixes. The algorithm's collision rate is derived analytically and verified by five sets of Web trace data. The results demonstrate that the collision rate is bounded and has little impact on page fetching latency. Finally, this compression method is compared to the *Summary Cache* method.**

Keywords— **Active networks, application-level forwarding and routing, web caching, hashing, cyclical redundancy checks, URL compression**

I. INTRODUCTION

WEB caching is generally recognized as an important service for alleviating focused overloads when certain web servers' contents become popular. A cooperative Web caching system has many design parameters, ranging from the infrastructure architecture to individual cache's backing store management algorithms. Above all, it must be effective in the face of user browsing behavior. Measurement data shows that user browsing behavior contains characteristics such as diurnal behavior [1] and "90-10" behavior, where 90% of the total requests account for 10% of the cached content [2]. Hot spots develop from time to time where user browsing behavior creates network congestion in the topological vicinity of and sustained workload at a particular Web server. The NASA/JPL Mars Pathfinder landing, the well-publicized Starr Report, and downloads of updated Netscape Communicator and Microsoft Internet Explorer browsers are several examples of activity that generated Internet-wide hot spot events. A more recent phenomenon analogous to geologic geyser activity is the traffic generated by portal sites, news and sports services, where the Web server's content is periodically updated during the course of the day causing users to periodically refresh their copy of the content.

The *Adaptive Web Caching* project (AWC) [3] starts from the premise that hot spot and geyser events will become the norm, not the exception. Rapid content dissemination through a scalable caching infrastructure is the key to absorb and dissipate these events. The AWC design builds such a caching infrastructure in two steps. First, caches form localized groups (neighborhoods) with overlap points between adjacent groups. Viewing cache groups as "subnets", and overlapping points as "routers", AWC creates an application-level topology where the caching infrastructure connects the Web servers at one end to the clients at the other. AWC then builds data dissemination trees on demand over this topology, thus enabling efficient delivery of popular contents to large numbers of clients.

Rapid content dissemination in the AWC system is accomplished by the *Content Routing Protocol* (CRP) which populates a URL forwarding table at each cache. When a Web server experiences heavy demand for some subset of its content, that subset will be stored in a set of caches' backing stores. Other caches forward requests received for the same content subset toward caches storing it. These other caches require a protocol that locates where this content is most likely to be stored, which is either along the Web server's data dissemination tree or within the confines of the searching cache's neighborhood.

CRP fulfills this role by propagating content reachability information throughout the cache topology, similar to the way popular network routing protocols such as *RIP* and *OSPF* propagate network reachability information. An IP router maintains a packet forwarding table by collecting and managing associations between network addresses and next-hop routers. Analogously, individual caches maintain a URL forwarding table by collecting content reachability information, represented by associations between full URLs or URL prefixes and a next-hop cache. Two CRP protocols propagate these associations:

- The *source information* protocol propagates a Web server's URL namespace prefix throughout the cache topology. This namespace prefix is used to construct the Web server's data dissemination tree.

• The *local content state* protocol propagates the URLs currently stored in a particular cache's backing store to its neighborhood. This maximizes cache sharing within the neighborhood.

Upon receiving a Web page request, a cache first checks to see if a forwarding table entry for the request's URL points to a cache in the local neighborhood, when the requested content is not available from a cache's own backing store. If no neighborhood entry exists, it checks to see if a forwarding entry exists that leads toward the Web server's data dissemination tree. If the forwarding entry exists, the request is relayed to the specified next-hop cache.

Conceptually, the above request handling algorithm describes a two step search process. A longest matching prefix algorithm is applied to the URL forwarding table lookup process, unifying the local and remote content search. Full URLs are preferred over URL prefixes because they represent locally cached content whereas URL prefixes point to the direction of the content's Web server. The longest prefix matching algorithm blurs the distinction between the two content searches, combining both into a single step.

The difference between IP-style routing and URL forwarding is a matter of address space size: IP's address space is finite whereas the cardinality of URLs is infinite. An IP router's forwarding table reflects the current state of the underlying network's topology. An AWC cache does not need to populate its URL forwarding table with all URLs to locate all content in the caching system. It only needs to maintain forwarding state with respect to URLs corresponding to hot spot and geyser events.

Three tightly coupled issues must be resolved to make the longest prefix matching algorithm work efficiently. The first is URL decomposition such that the URL name space is amenable to longest prefix matching. The second is scaling the URL forwarding table to accommodate a large number of forwarding entries while maintaining desirable longest prefix match algorithm performance. The third issue is compression of large sets of full URLs down to a reasonable size with minimal collision rate to enable efficient local content state updates.

II. Approach

A uniform resource locator (URL) defines a hierarchical namespace that identifies a resource by its primary access mechanism [4]. A simple decomposition parses URLs into a scheme, a network location, and individual path components to the resource. For example, *http://www.fribbles.org/interesting/index.html* decomposes to *http*, *www.fribbles.org*, *interesting*, and *index.html*. While this simple approach supports fine grained path component aggregation, it does not immediately support network location prefix aggregation. Network location prefix aggregation is desirable when HTML documents contain URL references located on servers in the same DNS domain: consider an HTML page from the Web server *www.fribbles.org* containing references to *fribbles.org* domain Web servers such as *images.fribbles.org* and *search.fribbles.org*. A single aggregated prefix for all hosts in the *fribbles.org* domain consumes fewer forwarding table entries and permits the source information protocol to build domain-based data dissemination trees, when administratively useful or convenient. The network location is decomposed in reverse order and produces the fine-grained hierarchy supporting the desired prefix aggregation. Thus, the previous example URL decomposes to *http*, *org*, *fribbles*, *www*, *interesting*, and *index.html*. URL prefixes are distinguished from complete URLs by a single "*" wildcard which is decomposed as a special component. The URL prefix *http://*.fribbles.org* represents the entire URL namespace for the *fribbles.org* domain, *http://www.fribbles.org/** represents a URL prefix for the *www.fribbles.org* Web server, whereas *http://www.fribbles.org/interesting/index.html* represents a complete URL. In the present scheme, *http://www.fribbles.org/interesting/f** is not recognized as a URL prefix.

It might be argued that this particular URL decomposition is too fine grained. A misbehaved or insecure CRP source information routing protocol variant could potentially hijack a large part of the cache system by advertising the URL prefix *http://*.org*, or worse, *http://**, thereby creating a different kind of focused overload. This problem can be solved by filtering the URL prefixes, by adding authentication and validation to the routing protocol, or by requiring that all URL prefixes contain at least three components (scheme, top-level and second-level DNS names.) There are circumstances where these rogue prefixes are useful. The *http://** prefix is analogous to an IP default route and could be used as the basis for a client-side AWC cache discovery protocol, assuming clients maintain their own URL forwarding tables. The *http://*.org* prefix could be used to engineer part of an AWC cache infrastructure for load-balancing based on top-level domain name. Nevertheless, the URL parser could merge the scheme and the DNS top level domain name components together as a single component without loss of generality.

A collection of decomposed URLs naturally forms a tree structure, as illustrated in figure 1. A large fan-out from the DNS top level domain nodes to the DNS second level domain nodes and from the second level domain nodes to the third level domain nodes is observed as more URLs

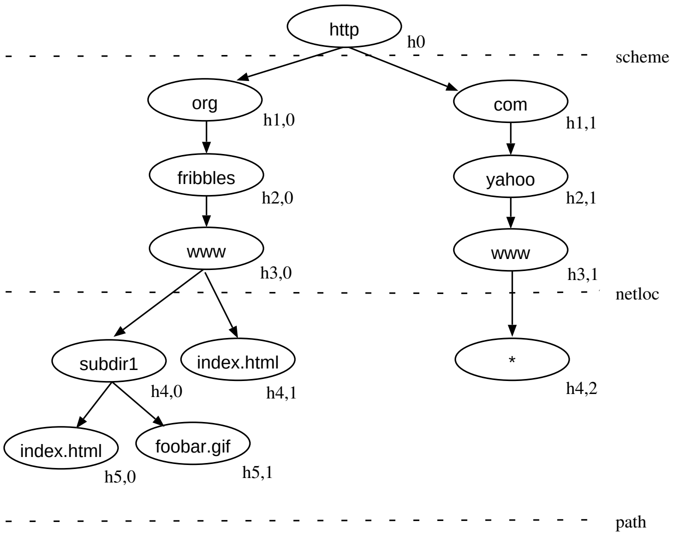

Fig. 1. An example URL decomposition tree

are stored in this structure. The longest matching prefix algorithm's performance becomes $O(nm)$ in the average case, where n is the number of decomposed URL components and m is the average number of children per node. The approach adopted in the forwarding table's current implementation is a hash table where each node in a given bucket maintains a pointer to its parent component, preserving URL component relationships and tree semantics. This reduces the longest prefix matching algorithm's performance to $O(n\alpha)$, where n is the number of components in the decomposed URL and α is the hash table's loading factor[1]. Forwarding data is stored in the tree's "leaves" which are always a decomposed URL's last component. The last component will either be the final element of the URL's path or a wildcard. The forwarding data minimally includes the next-hop cache and an expiration timer that detects when a forwarding entry is no longer valid and should be removed. Forwarding entries are assumed to be periodically refreshed, which causes an entry's expiration timer to be reset.

Decomposed complete and prefix URLs are inserted by computing incremental hash codes. Given a list of components, U,

$$U = (scheme, n_1, n_2, \ldots, n_i, p_1, p_2, \ldots, p_j)$$

where $n_1, n_2, \ldots, n_i$ are network location components, $p_1, p_2, \ldots, p_j$ are path components, and a hash function, $H(h, s)$, the hash codes for the URL are computed as follows:

$$\begin{aligned} h_0 &= H(0, scheme) \\ h_1 &= H(h_0, n_1) \\ h_2 &= H(h_1, n_2) \\ &\vdots \\ h_i &= H(h_{i-1}, n_i) \\ h_{i+1} &= H(h_i, p_1) \\ &\vdots \\ h_{i+j} &= H(h_{i+j-1}, p_j) \end{aligned} \tag{1}$$

Incremental hashing minimizes the collision probability between URL prefixes. This property is exploited by the local content update protocol when a cache transmits the current set of URLs stored in its backing store to its neighborhood. Assume that the cache's backing store's metadata is organized as a URL decomposition tree. The sending cache transmits a list of $\langle depth, hash\ code\rangle$ tuples (a *hash chain sequence*) produced by a depth first, left-to-right traversal of the tree. The tree shown in figure 1 produces the hash chain sequence:

$$\begin{array}{cccc} \langle 0, h_0\rangle, & \langle 1, h_{1,0}\rangle, & \langle 2, h_{2,0}\rangle, & \langle 3, h_{3,0}\rangle, \\ \langle 4, h_{4,0}\rangle, & \langle 5, h_{5,0}\rangle, & \langle 5, h_{5,1}\rangle, & \langle 4, h_{4,1}\rangle, \\ \langle 1, h_{1,1}\rangle, & \langle 2, h_{2,1}\rangle, & \langle 3, h_{3,1}\rangle, & \langle 4, h_{4,2}\rangle \end{array}$$

The receiving caches insert the tuple sequence directly into their respective forwarding tables using place holding nodes. Relationships between nodes are derived from the hash chain sequence and preserved by the receiver. A receiver may reconstruct an individual hash chain such as

$$\langle 0, h_0\rangle, \langle 1, h_{1,0}\rangle, \langle 2, h_{2,0}\rangle, \langle 3, h_{3,0}\rangle, \langle 4, h_{4,1}\rangle$$

[1] A hash table's loading factor is defined as $\frac{N}{B}$, where N is the total number of elements stored in the hash table and B is the number of buckets.

from more than one sender. In general, the routing protocol is the final arbiter with respect to which sender is chosen as the forwarding table's next-hop cache and whether alternative next-hops are kept or discarded. The current implementation of the local content state update protocol chooses the first sender as the next-hop cache. Subsequent senders are stored as alternative next-hops and used when the first sender's forwarding entry expires.

Hash chain sequences doubles the longest prefix matching algorithm's worst-case performance because comparisons include both exact matches against strings and inexact matches against the place holding nodes. The principal benefit of this encoding scheme is that it achieves a simple form of compression which imposes little processing overhead at the receiver. The primary source of compression in this forwarding table derives from its tree structure. The tree consolidates common URL prefixes, based on the observation that URLs from a single Web server often share a common path or small set of common paths.

A hash chain's imprecision can create forwarding loops. Consider the case where both caches A and B store URL $\mathcal{U}_1$ in their respective backing stores. $\mathcal{U}_1$ produces the same hash chain, $\mathcal{H}$, as $\mathcal{U}_2$. Cache C receives the hash chain $\mathcal{H}$ from both caches A and B during the next local content state update and chooses to insert a forwarding entry that points to cache A. Cache A will insert a forwarding entry for $\mathcal{H}$ that points to cache B. Cache B will insert a forwarding entry for $\mathcal{H}$ which points to cache A. Cache C receives a request for $\mathcal{U}_2$, does not find it in its backing store, and sends the request to cache A. Cache A does not find $\mathcal{U}_2$ in its backing store and sends the request to cache B. Cache B examines its backing store, sends $\mathcal{U}_2$ back to cache A, and creates a forwarding loop. This particular loop is easy to detect and break by restricting caches from forwarding a request back to the previous sender. When a one hop forwarding loop is detected, the cache fetches the requested URL's contents directly from the appropriate Web server. Two or more hop forwarding loops are more difficult to detect and break without adding more information to the local content state updates.

An alternative to detecting a one-hop loop is to prevent it from happening in the first place. The receiving cache modifies the hash chain insertion process by verifying that its backing store won't produce the same hash chain. A forwarding entry is not inserted if the same hash chain would be produced by the receiver's backing store. In the previous example, caches A and B will never point at each other and the forwarding loop will never be created. This modification requires additional CPU overhead to traverse the backing store's URL decomposition tree and meta-data for each reconstructed hash chain and may be expensive for large local content state update messages. Moreover, this modification may be overkill if one-hop loops are the only kind of loops observed in practice.

III. RESULTS

Hash chains are a compressed and imprecise representation of URLs present in one cache's backing store that are easy for another cache to directly insert into its forwarding table. Imprecision makes it necessary to calculate the hash chain collision probability, when two distinct URLs with the same number of decomposed components generate the same hash chain. This probability, also known as the false positive rate, determines how often a URL will be incorrectly forwarded to a cache where its contents are not stored. The false positive rate is shown to have a negligible impact on request latency.

A. Current Implementation Properties

A 19-bit CRC function is used in the current CRP forwarding table's implementation. This particular size for the hash function is a trade-off between the hash collision probability and the memory footprint of the hash table. A 19-bit CRC requires a minimum of 2MB of memory to hold the the hash table's bucket pointers, assuming a 32-bit pointer model. Additional bits in the CRC code increase the size of the bucket pointer array by corresponding factors of 2, increasing the table's capacity and reducing the collision probability and the average search time. This memory footprint should be balanced with the cache application's overall memory needs. It is common practice to dedicate a pool of memory to transient URLs in order to eliminate disk accesses. Increasing the hash table's footprint and stealing from the transient object pool when real, not virtual, memory is a premium may not be desirable.

B. Methodology

Five URL collections were used in this analysis to simulate two common cache configurations. Two URL collections represent edge caches, the first cache that a user's request encounters at the edge of the AWC caching system. The remaining three collections represent caches in the middle of the caching infrastructure. The characteristics of these collections are as follows:

1. *uclacs_fall*: These URLs were snooped from a single UCLA Computer Science Department's subnetwork between September 12, 1998 and October 11, 1998. A total of 634,702 URLs were captured in this trace.
2. *uclacs_spr*: These URLs were snooped from the UCLA Computer Science Department's egress router's link to the campus backbone network between April 1, 1999 and May

1, 1999. A total of 1,485,317 unique URLs were captured in this trace.
3. *canetii*: This is an aggregation of the log files from the CA*netII *Squid* root cache between May 21, 1999 and May 31, 1999. A total of 1,073,302 unique URLs were extracted from these log files. Other sanitized log files for a 21 day window are available from *http://ardnoc41.canet2.net/cache/squid/rawlogs/*.
4. *ircache_bo*: This is an aggregation of the log files from the NCAR (Boulder, CO) Squid root cache between June 3, 1999 and June 9, 1999. It contains 2,495,449 unique URLs.
5. *ircache_sj*: This is an aggregation of the log files from the MAE West (San Jose, CA) exchange point between June 4, 1999 and June 9, 1999. It contains 556,214 unique URLs.
ircache_bo and *ircache_sj* traces were created from the sanitized log files provided at *ftp://ftp.ircache.net/Traces/*. This particular repository only keeps a week's worth of sanitized logs.

C. Hash Chain Collision Probability

The hash code generated for each decomposed URL component is related to its predecessor's hash code via the recurrence relation:

$$\begin{aligned} h_0 &= H(0, U_0) \\ h_n &= H(h_{n-1}, U_n) \end{aligned} \tag{2}$$

where U_0 is the URL's scheme and U_n is the n-th URL component following the scheme. This is particularly well suited for cyclical redundancy check codes where the remainder of polynomial division performed in the GF(2) field is computed. CRCs can be computed by a shift-register with feedback [5]. Thus, each h_n corresponds to the value found in the shift register at the end of the n-th component. The polynomial, in this case, is the bit string that results from concatenating successive URL components together. The probability, P_{coll}, that a polynomial f of degree m will generate the same remainder x as polynomials $f_i{}'$ of degree $m \geq i \geq l$, where l is the degree of the generator polynomial, is given by:

$$\begin{aligned} P_{coll} &= Pr\left[f_i{}' \bmod g = x \mid f \bmod g = x\right] \\ &= \frac{1 + \sum_{i=l}^{m} 2^{i-l}}{2^l + \sum_{i=l}^{m} 2^i} \\ &= \frac{1 + \sum_{i=l}^{m} 2^{i-l}}{2^l \left(1 + \sum_{i=l}^{m} 2^{i-l}\right)} \\ &= 2^{-l} \end{aligned} \tag{3}$$

P_{coll} defines the probability that two distinct URLs will generate the same hash code for the n-th component, independent of the number of characters by which they differ in that component.

A hash chain collision occurs when two distinct URLs, $\mathcal{U}_x$ and $\mathcal{U}_y$, with the same number of components generate the same tuple sequence for all components. The probability that this event occurs, P_{hc} is calculated in the general case from the probability that all n components produce the same hash codes, i.e.

$$\begin{aligned} P_{hc} &= \prod_{i=1}^{n} P_{coll} \\ &= P_{coll}^{n} \end{aligned} \tag{4}$$

This is an optimistic lower bound. For example, if the URL namespace is restricted to the same scheme, then all URL decompositions will share the same first hash code, h_0. Thus, if two URLs share a common prefix of $j < n$ components, P_{hc} becomes:

$$\begin{aligned} P_{hc} &= \left(\prod_{i=1}^{j} 1\right)\left(\prod_{i=j+1}^{n} P_{coll}\right) \\ &= P_{coll}^{n-j} \end{aligned} \tag{5}$$

Thus, the hash chain collision probability is bounded by P_{coll} on the left and P_{coll}^{n-1} on the right. The current implementation's 19 bit CRC function yields $P_{coll} = 2^{-19}$ for the upper bound for P_{hc}.

Table I shows the results from the five URL collections, where the URLs were decomposed and inserted into a forwarding table. Each bucket in the forwarding table was scanned for pairs of last URL components (leaf nodes) having the same tree depth. If both URL's parental relationships (parent, grandparent, great-grandparent, etc.) encountered the same sequence of hash buckets, a hash chain collision was noted. The collision rates are one and two orders of magnitude higher than the expected 2^{-19} (1.91×10^{-6}) in all five collections. It is suspected that the reason for this is similar to that pointed out in [6]: the CRC function's input is not random but highly regular text.

Two distinct types of hash chain collisions were observed: "Last Component" and "Whole URL". Last component collisions occurred when the two URLs differed only in the last component whereas whole URL collisions occurred when the pair of URLs were disjoint in every component except the scheme. No definite conclusion can be drawn beyond the fact that hash chains are not suitable for advertising a Web server's URL prefix, e.g. *http://www.fribbles.org/**, in the CRP source information protocol. This would result in a cache forwarding requests up the incorrect source rooted tree, adding an indeterminate

TABLE I
OBSERVED HASH CHAIN COLLISIONS

Collection	Total URLs	Total Collisions	Collision Rate	Last Component	Whole URL
uclacs_fall	634702	37	0.00583%	19	18
uclacs_spr	1485317	174	0.0117%	132	42
canetii	1073302	111	0.0103%	62	49
ircache_bo	2495499	336	0.0135%	166	170
ircache_sj	556214	21	0.00378%	18	3

amount of delay to the request before the error is discovered. On the other hand, a hash chain collision has a negligible impact on per-group request forwarding performance in the absence of forwarding loops and the assumption that no URLs are removed the cache's backing store. The average hash chain collision rate for all five collections is 0.00902% or 1 incorrectly forwarded URL every 11,087 requests. Putting this into perspective, assuming it takes a cache 100ms to probe its backing store, forward, and transmit a request to a next-hop cache, the collision rate adds 9.02μs additional average request delay per cache group. Moreover, the incorrectly forwarded request is delayed by a single cache, which would subsequently forward the request up the appropriate Web server's tree, if the Web server's URL prefix exists in its forwarding table, or request the URL directly from the Web server.

The hash chain collision rates observed in this experiment are worst case collision rates. Each collection is an aggregation, e.g. the *uclacs_spr* collection represents a month's aggregation of user activity. No URLs were removed or expired from the cache's backing store during the experiment. It is expected that a deployed AWC caching system would have a backing store manager which periodically reaps unused URL's contents. All 1,485,317 URLs would not be present in the *uclacs_spr* cache's backing store, only a working set of the most frequently accessed URLs would be present. It would be reasonable to conclude that a smaller URL working set would produce fewer hash chain collisions, as the *uclacs_fall* and *ircache_sj* results bear out.

D. Compression

Table II shows the forwarding table's characteristics for each of the five URL collections and illustrates the compression achieved by the hash chain method. "Total URL Components" designates the total number of nodes stored in the forwarding table and the total number of nodes in the equivalent URL decomposition tree. Assuming that each generated $\langle depth, hash\ code\rangle$ pair is encoded in a 32-bit quantity, the hash chain's size in bytes is computed by multiplying the total URL components by 4. The hash chain's size is compared to the number of bytes required to transmit all of the strings generated by a left-to-right, depth first traversal of the equivalent URL decomposition tree. Each node in the decomposition tree is encoded as $\langle depth, length, string\rangle$, where $depth$ and $length$ are single byte quantities. Thus, the decomposition tree's size is computed as $2 \times Total\ URL\ Components + Total\ String\ Length$. *Total String Length* is the sum of all node's string lengths. The compression achieved is computed as $1 - Hash\ Chain/\,Tree\ size$.

The relatively uniform compression indicates that many URLs in a given collection share common network location and path prefixes. The low ratio between total URL components and total URLs per collection shown in table III indicates that the URL decomposition tree is relatively flat, with most of its nodes at the leaves. Adding URLs from an existing Web server with common path components increases the number of nodes in the URL decomposition tree by one to three nodes on the average. This is based on the assumption that five out of a possible eight nodes (the average depth from the *uclacs_spr* collection) already exist in the tree. They correspond to the URL's scheme, three components in the Web server's name (i.e. *www.fribbles.org*), and one common path component. The number of nodes in the tree grows faster and results in a larger ratio when URLs from disjoint Web servers are inserted into the tree, resulting in six or more nodes per addition (the URL's scheme and the DNS top level domains are omnipresent.) The fact that the ratio is low is not an unexpected result: an HTML page often contains references to images that are kept in the same file system directory as the HTML page itself.

The hash chain's transmission size presents some concern when used outside of a simulation. For example, one cache containing the *ircache_bo* collection's URLS in its backing store would transmit 13.93MB of data to the other members of its cache group. Multicasting the update packets would reduce the network resources consumed versus replicated unicast to individual cache group

TABLE II
COMPRESSION FROM HASH CHAINS

Collection	Total URL Components	Total String Length	Hash Chain (bytes)	Tree size (bytes)	Compression
uclacs_fall	839,196	9,248,351	3,356,676	10,926,689	69.28%
uclacs_spr	2,008,462	22,569,995	8,033,848	26,586,919	69.78%
canetii	1,421,787	15,660,725	5,687,148	18,504,299	69.27%
ircache_bo	3,652,747	38,512,997	14,610,988	45,818,491	68.11%
ircache_sj	767,685	8,286,134	3,070,740	9,821,504	68.73%

TABLE III
RATIO OF COMPONENTS TO URLS FOR EACH COLLECTION

Collection	Component to URL Ratio	Average Tree Depth	Maximum Tree Depth	Standard Deviation
uclacs_fall	1.32	8.40	54	2.29
uclacs_spr	1.35	8.94	58	2.57
canetii	1.32	8.48	177	3.21
ircache_bo	1.46	9.63	321	4.25
ircache_sj	1.38	9.15	356	8.15

members. However, the receiving caches could stall user requests while processing approximately 9,926 UDP packets created by the hash chain generation. Update packets from the generated hash chain could made independent of each other by repeating the path from the root to the current leaf at the start of a new update packet, thus eliminating user request stalling. Additionally, missing update packets only impact the snapshot accuracy from the perspective of an individual cache.

Minimizing user request latency should be balanced with the snapshot accuracy of the group members' backing stores. This suggests that a local content state update protocol's implementation should adopt incremental transmission or only transmit the most frequently accessed URLs in its backing store. Results presented in [2], [7] indicate that many URLs in a cache's backing store are "read-once" and would not be useful in a local content state update. A more refined but CPU intensive approach might utilize the data mining technique presented in [8] that would reduce the number of URLs transmitted during a local content state update to those most likely to be traversed. The transmission sizes presented in this paper are also likely to be unrealistic because no backing store management policy is assumed, as previously noted.

IV. COMPARISON TO *Summary Cache*

Summary Cache is a proposed mechanism which allows one cache to implicitly probe other caches' backing stores for a requested URL's content. The *Squid* cache system is an example system which motivated the *Summary Cache* research. *Squid* is a hierarchical caching infrastructure, using the *Internet Cache Protocol* (ICP) for inter-cache communication. Without a *Summary Cache*-like mechanism, each *Squid* cache first sends an ICP query to a set of local sibling caches when a request is not satisfied from its backing store. If the local ICP query fails as the result of a time out waiting for a positive acknowledgment, the cache sends the request to its parent who repeats the process. The principal problem with this approach is the implied negative acknowledgment. The timeout value is adaptive with an initial value of 2 seconds. Explicit probing of the siblings' backing store consumes a non-negligible amount of network bandwidth and increases client request latency; a detailed description of the problem can be found in [9].

Hash chains and *Summary Cache* digests perform the same fundamental function: propagating a snapshot of the URLs currently present in one cache's backing store to members of a local cache group. *Summary Cache* utilizes a Bloom filter to create a backing store digest where the filter's hash codes are extracted from each URL's MD5 signature. The principal difference between the two methods is how the decision to send the request to the next-hop cache is made. A *Squid* cache enhanced by the *Summary Cache* mechanism examines the digests received from its siblings and determines which ones are likely to have the request's contents stored. When the request is sent to a sibling incorrectly due to a false positive, the cache must still time out before sending the request to its parent. The

AWC application-level forwarding approach does not require an implied negative acknowledgment: an incorrectly forwarded request is simply reforwarded. The longest matching prefix algorithm combines the two step "local versus dissemination tree" decision process into a single step.

Both mechanisms share a common trade-off in the number of bits used in the hash codes and the memory required. Summary cache's memory requirements scale in terms of the size of the filter, a function of the average number of URLs the backing store can manage times a constant load factor. Hash chains scale in terms of the number of bits in the CRC function and the number of decomposed URL components. Using the *ircache_bo* URL collection for comparison purposes, a *Summary Cache* using load factor of 32 transmits approximately 9,981,996 bytes versus 14,610,988 bytes for hash chains or 32% less data. The *Summary Cache* false positive rate, based on figure 5 in [9], appears to be in the range 0.005 to 0.01, comparable to what the hash chain approach achieves. *Summary Cache* can reduce the false positive rate either by increasing the load factor and increasing the hash function's range or by using more hash functions. Hash chains reduces its false positive rate by increasing the number of bits in the CRC function, a single parameter.

Hash chain sequences are cheaper to compute than *Summary Cache* MD5 signatures. While MD5 signatures produce distinct 128-bit hash codes, their computation is difficult to optimize [10]. If lines of code can be used as a rough complexity metric, the FREEBSD kernel's MD5 implementation is 212 lines whereas the CRCcomputation as implemented is only 3 lines. Computing a component's hash code involves two shift operations, two XOR operations, and an AND operation per character. The hash code is not computed from the successive concatenations of URL components; the CRC code from the last component serves as the seed for the next component's code. Hash chains also assume that the backing store's meta-data is organized as a URL decomposition tree, which reduces hash chain sequence generation to traversing the tree. *Summary Cache*, as described, initially computes the Bloom filter from scratch for each URL and maintains slight additional overhead to keep the filter consistent as URLs are added and deleted from the backing store. Thus, it can incrementally update its neighbors with the filter's changed bit positions since the last update. An AWC cache must transmit the entire hash chain sequence in order to refresh its neighbor's forwarding entries.

Incremental hashing and the hash chain mechanism achieve an advantage over *Summary Cache* in request processing, despite its additional protocol and memory overhead. Assume, for the sake of simplicity, that the time taken to process a request by either a type of cache is characterized by three quantities:

- t_{probe}, the time it takes to probe the backing store,
- t_{decide}, the time it takes to decide where to send the request next, and
- $t_{network}$, the time it takes to send the request to the next recipient.

Further assume that t_{probe} and $t_{network}$ are equal for both caching systems and there are no false positives. The remaining difference between the two systems is t_{decide}. AWC's t_{decide} consists of computing the CRC code for the decomposed URL and inspecting at most $O(n\alpha)$ hash table nodes. Computing an MD5 signature, extracting hash codes from the signature, and inspecting a bit vector dominates *Summary Cache*'s t_{decide}. Again, the CRC's computational cheapness leads to the expectation that AWC per-request processing can handle higher request loads than *Summary Cache*, an assertion to be verified by deployment and measurement.

V. CONCLUSION

A common problem in cooperative Web caching systems is locating a proximate copy of a requested URL's contents. *Adaptive Web Caching* solves this problem by a per-cache URL forwarding table which maintains associations between full URLs and URL prefixes and next-hop caches. A full URL indicates that the requested URL's content is stored within a cache's neighborhood; a URL prefix indicates the parent cache on URL's Web server's data dissemination tree. Full URLs are preferred over URL prefixes to maximize local cache sharing. Deciding whether the requested URL's contents are located in the cache's neighborhood or along the web server's data dissemination tree is conceptually a two step decision process. A longest prefix matching algorithm is used whenever a cache looks up the forwarding entry corresponding to a requested URL because it combines the two searches into a single step.

URLs are hierarchically decomposed to support both network location and path aggregation, resulting in a list of components. Collections of hierarchically decomposed URLs naturally form a tree. The URL forwarding table is implemented as a hash table to achieve $O(n\alpha)$ search performance. URL components stored in the hash table keep their relationship with their parent component to preserve the tree-like semantics. URL components are inserted into the URL forwarding table using incremental hashing. Incremental hashing indirectly preserves relationships between successive URL components and is also used to encode the list of URLs stored in a cache's backing store. Assuming the cache's backing store's meta-data is organized as a URL decomposition tree, a left-to-right, depth-first traversal of the tree produces a sequence of hash chains that are subsequently transmitted by the cache to its neighborhood. The hash chains are reconstructed and inserted directly into the receiving caches' URL forwarding tables.

Incremental hashing in the current implementation of AWC's URL forwarding table is done via 19-bit CRC codes. The probability of a hash chain collision, where two disjoint URLs produce the same sequence of hash codes, is analytically shown to be 2^{-l}, where l is the degree of the CRC generator polynomial. Observations from five different data sets show that the collision rate can be two order of magnitudes higher because the input to the CRC function is highly regular text. Nonetheless, the average observed collision rate would cause 1 out of 11,087 requests to be forwarded incorrectly to a cache where the requested URL's contents were not stored in its backing store. Assuming that a cache takes 100ms to process a request, this collision rate adds 9.70μs average additional per-request latency.

The hash chain sequences provide a simple compression mechanism which disseminates an imprecise snapshot of a single cache's backing store. This is compared with the *Summary Cache* mechanism. *Summary Cache* and hash chain sequences provide caching systems with the ability to implicitly probe other caches' backing stores. While *Summary Cache* achieves smaller backing store snapshots with a comparable collision rate, the hash chain approach has the distinct advantage that the CRC codes are easier to compute than *Summary Cache*'s MD5 signatures and it is expected that AWC caches can forward requests faster when under load.

Application-level forwarding and routing is a successful mechanism for URL request processing in the context of *Adaptive Web Caching*. The approach can be extended to a more general application framework and applied to different name spaces other than URLs for application areas such as service discovery, naming services, file replication, and dynamic code dependency resolution in active networks. Forwarding facilitates constructing an integrated infrastructure connecting clients, application "routers" or "gateways", and servers. This approach also encourages experimentation with routing protocols which supply application-specific properties without disrupting the underlying Internet routing infrastructure at large. Further research in this topic will focus on the development and understand the benefits of a general application-level forwarding and routing framework.

VI. MOTIVATIONS, INSPIRATIONS, AND ACKNOWLEDGMENTS

The motivations and inspirations for application-level forwarding and routing in AWC came from numerous design discussions in addition to the literature. The authors would like to thank Adam Rosenstein and Khoi Nguyen, of the UCLA Computer Science Department, for their input and participation.

From the literature, it suffices to mention two papers which helped mold the AWC forwarding table into what it has become. The *Cache Array Routing Protocol* (CARP) [11] proposed client-side URL hashing as the basis for forwarding a request within an array of caches. Each cache in the array is responsible for a segment of the hash function's range; locating where content is likely to be cached becomes deterministic in this system. This algorithmic simplicity was desired in AWC's local neighborhood search. Incremental hashing was motivated by recursive n-gram hashing [12]. Given a sequence of symbols, $S = (s_1, s_2, s_3, \ldots, s_{N+(n-1)})$, an n-gram of the sequence is an n-long subsequence of consecutive symbols. [12, p. 291]. The attractive feature of n-gram hashing functions is that they produce distinct hash code sequences codes when applied to the first through ith n-gram. Polynomial division in the GF(2) field, also the basis of CRC codes, is one of the recursive hashing techniques discussed and has desirable computational characteristics, being composed of shift and XOR operations. The fast computation and the generally low collision rates of these functions were the main factors for adopting CRC functions in the AWC forwarding table's implementation.

REFERENCES

[1] S.D. Gribble and E. Brewer, "System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace," in *Proceedings of the 1997 USENIX Symposium on Internet Technolgoies and Systems (USITS)*, Monterey, CA, USA, Dec. 1997.

[2] P. Cao, L. Fan, G. Phillips, S. Shenker, and L. Breslau, "Web Caching and Zipf-like Distributions: Evidence and Implications," in *INFOCOM*, 1999.

[3] S. Floyd, L. Zhang, and V. Jacobson, "Adaptive Web Caching," http://irl.cs.ucla.edu/awc.html, May 1997, DARPA–funded Research Project.

[4] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax," Internet standards track RFC, August 1998, Updates RFC 1738.

[5] W. Wesley Peterson and E. J. Weldon, Jr., *Error-Correcting Codes*, The MIT Press, Second edition, 1972.

[6] Craig Partridge, Jim Hughes, and Jonathan Stone, "Performance of checksums and CRCs over real data," *SIGCOMM '95*, vol. 25, no. 4, pp. 68 – 76, October 1995, Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication.

[7] D.N. Serpanos and W.H. Wolf, "Caching Web objects using Zipf's law," in *SPIE - The International Society for Optical En-*

gineering. SPIE-Int. Soc. Opt. Eng, 1998, vol. 3527, pp. 320–6, (Multimedia Storage and Archiving Systems III, Boston, MA, USA, 2-4 Nov. 1998.).

[8] Ming-Syan Chen, Jon Soo Park, and Phillip S. Yu, "Efficient Data Mining for Path Traversal Patterns," *IEEE Transactions on Knowledge and Data Engineering*, vol. 10, no. 2, pp. 209–221, March/April 1998.

[9] Li Fan, Pei Cao, Jussara Alameida, and Andrei Broder, "Summary Cache: A Scalable Wide-Area Cache Sharing Protocol," in *ACM SIGCOMM'98*, 1998, pp. 245–265, Technical Report 1361, Computer Sciences Department, Univ. of Wisconsin-Madison, Feb 1998.

[10] J.D. Touch, "Performance analysis of MD5," *Computer Communication Review*, vol. 25, no. 4, pp. 77–86, Oct. 1995, (ACM SIGCOMM '95, Cambridge, MA, USA, 28 Aug.– 1 Sept. 1995.).

[11] K. W Ross, "Hash routing for collections of shared Web caches," *IEEE Network*, vol. 11, no. 6, pp. 37–44, Nov.-Dec. 1997.

[12] J.D. Cohen, "Recursive Hashing Functions for n-grams," *ACM Transactions on Information Systems*, vol. 15, no. 3, pp. 291–320, July 1997.

[13] A. Chankhunthod, P. Danzig, C. Neerdaels, M.F. Schwartz, and K.J. Worrell, "A Hierarchical Internet Object Cache," Technical Report 95-611, Computer Science Department, University of Southern California, Los Angeles, California, March 1995.

[14] National Laboratory for Applied Network Research, *Squid Internet Object Cache*, http://squid.nlanr.net/Squid/.

[15] N.G. Smith, "The UK national Web cache-the state of the art," *Computer Networks and ISDN Systems*, vol. 28, no. 7-11, pp. 1407–14, May 1996, (Fifth International World Wide Web Conference, Paris, France, 6-10 May 1996.).

[16] D. Neal, "The Harvest object cache in New Zealand.," *Computer Networks and ISDN Systems*, vol. 28, no. 7-11, pp. 1415–30, May 1996, (Fifth International World Wide Web Conference, Paris, France, 6-10 May 1996.).