Proxy Caching with Hash Functions

Florin Baboescu

Abstract— **Internet traffic doubles every three months, with web traffic accounting for more than 75% of the total traffic between major ISP providers. Web caching at proxies can reduce this traffic as well as web user latencies. One way to increase the hit rate of caches is to have caches cooperate using the Internet Cache Protocol or using hash functions. This paper provides an experimental and theoretical analysis of two well known hash based caching schemes. Our paper also proposes a new allocation scheme which improves the performance of one of the analyzed techniques. It reduces the standard deviation of the URLs distribution on caches by a factor greater than two.**

I. Introduction

With web traffic accounting for more than 75% of the total traffic between major ISP providers [3], proxy caching to reduce web traffic and latency is of great interest. Unfortunately, the hit rate of individual proxy caches is small; studies have shown that cache hit rate grows logarithmically with the number of users [6] , [14]. One way to improve the hit rate is to have caches cooperate. For example, [14] shows 40% of documents were accessed by more than one user in different departments but only by one user in each department. This number is three times more than the number of documents that are accessed by more than one user from the same department. This implies that cache hit rates are low without cooperation between caches maintained by different administrative units.

The Harvest [1] system implements cooperative caching using a hierarchy of cooperative caches. Each cache is configured a list of neighboring peer caches (e.g., department caches for a campus) and a parent cache (e.g., cache for the region). When a cache gets a request it cannot serve, it first sends a query (using a lightweight protocol called ICP [12]) to all its neighbor caches; if all neighbors reply with a miss, the request is sent to the parent. If the server is local, the cache queries the server directly.

F. Baboescu is with the Department of Computer Science and Engineering, Univ. of California, San Diego. Email: baboescu@cs.ucsd.edu

Despite its simplicity, cooperative caching using ICP has some major drawbacks:

- An ICP cache must query all its neighbors to determine the location of cached information; the process is inefficient and generates extra traffic due to the ICP messages which are exchanged. For the NLANR cache system, we found using a whole week of traces, that the ratio between the ICP requests and HTTP requests has values between 3 and 9.
- The number of queries is directly proportional with the number of cache siblings.
- Popular objects end up being located in all the caches. An increase of up to 30% in the number of objects which are replicated on the caches might be due to the popular objects.

In order to address this problem a new cooperative cache architecture scheme was introduced [7], [4], [9]. It uses one or more hash functions applied both to the URL and to the set of caches in order to identify the *single* cache which should hold the object from a specific URL. This reduces traffic (as only a single neighbor cache is queried) and prevents replication of popular documents. An important goal for this type of architecture is to have a uniform distribution of URLs to caches.

Our paper compares two hash based cooperative caching schemes (which we will call the MIT scheme [7] and the Microsoft scheme [4]. The Microsoft scheme called CARP is implemented in the Microsoft Proxy Server; the MIT scheme is proposed in a set of papers from MIT and may have provided some motivation for the (secret) scheme used by AKAMAI. We show both by simulation and theoretical proof that the algorithm proposed by [7] can at most reach the performance in terms of URLs distribution at caches of [4]. We also show how a small modification on the assignment policy of URLs to caches described in [7], can reduce the standard deviation of the URL distribution by more than a factor of two.

This paper is organized as follows. Section III introduces hash based caching and shows its implementation in both the MIT and Microsoft proposals. Section V describes our simulation methodology, while sectionVI examines the performance of the two hash based schemes in relation with the URLs distribution to the proxies. It shows the high impact in the URLs distribution caused by a small modification of the allocation policies of URLs to caches. It also shows that the algorithm implemented in [7] can at most reach the performance in terms of URLs distribution at caches of [4]. Our results are summarized in VII.

II. Related Work

A first proxy cache implementation is the one from CERN [8]. Several recent studies of proxy cache traffic show a direct dependence between the hit ratio for a web proxy and both the client population and the number of requests at the proxy ([2], [6], [14]). This observation strongly motivates an extensive work on cooperative Web caching as a way to improve the latency perceived by the end users and to reduce the bandwidth utilization.

A very first scheme for cooperative Web caching is based on ICP and it is implemented in Squid [13]. When a cache gets a request it can not resolve, it first sends an ICP query to all its neighbor caches; if all neighbors reply with a miss or they do not reply during an interval of timeout the request is sent to the a parent cache or directly to the server.

Cache Digests [10] and Summary Cache [5] are two cooperative web caching scheme which are trying to efficiently locate objects in neighboring caches. A cache digest is a lossy compression of all cache keys with a lookup capability. Digests are made available using HTTP, and a cache downloads its neighbor's digests at startup. A cache is able to know if a neighbor holds a particular object by looking into its digest. Summary Cache implements a similar ideea.

A third category of cooperative Web caching scheme is represented by the hash based schemes. The Cache Array Routing Protocol (CARP) [11] has been designed by the University of Pennsylvania and Microsoft and it is implemented in the Microsoft Proxy Server [4]. A client may requests documents from a farm of proxy servers. The requests are deterministically forwarded to the proxy server in charge with handling the request. The selection of the proxy server is done using a hash function which takes as arguments both the URL from the request and the name of proxy server. A similar scheme is proposed in a set of papers from MIT [7].

This paper expands previous research done in cooperative Web caching scheme based on hash function. It compares both schemes nominated above and show how the scheme proposed by MIT [7] may be improved to gain a better load distribution without introducing additional computation.

III. Hash Based Cooperative Caching

A solution to the ICP based cache scheme's drawbacks is to use a "queryless" cooperative caching, based on hash functions. A hash based distributed cache hierarchy avoids the drawbacks of the Harvest caching model by:

• eliminating multiple queries. Only the cache which should have the copy is interrogated by a children;

• avoiding popular copy duplication in all the siblings; each URL is stored at exactly one cache.

• automatically adjusting to addition/deletion of a cache sibling without causing high disruption in URL location assignment.

In a hash routing model an URL is assigned only to one cache. All users can determine the URL-to-Cache mapping in a unique way using hash functions.

Let's consider a naive way to allocate URLs at two caches in which one would first map an infinite space of URLs seen as ASCII strings to a hash space, say 1..100 (figure 1). The hash space is partitioned in two intervals: 1-50, 51-100. The URLs which are mapped to values in 1-50 are assigned to the first cache while the ones mapped in 51-100 are assigned to the 2nd cache. If the hash function which is used to do the mapping from the URL space to the 1-100 interval has a uniform distribution then URLs will be uniformly distributed between the two caches.

Despite the simplicity, this naive implementation has a major drawback: its lack of robustness. In [9] the author shows that when a new cache is introduced in the system about half of the objects are now located in a wrong sibling. The size of a cache system which provides good cache performance is

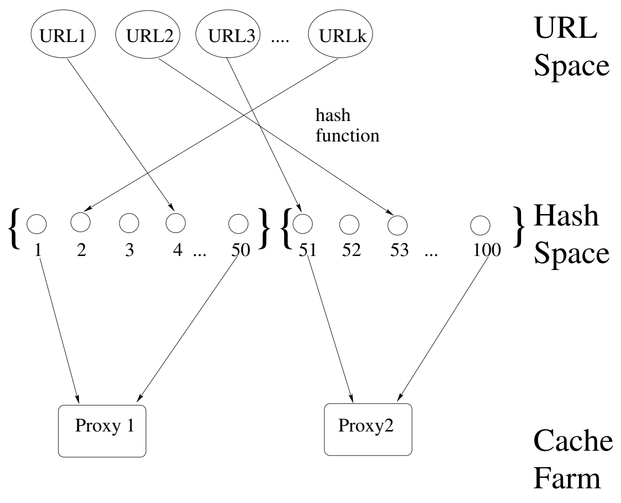

Fig. 1. Naive Allocation of URLs to Caches using Hash Routing

usually about 10-100 Gigabyte; it takes some weeks following its startup until it stabilizes. Therefore a high disruption caused by a new cache introduced in the system should be avoided as much as possible.

There are two proposed techniques for implementing robust hash functions. The first is the Cache Array Routing Protocol (CARP), a protocol which is implemented in the Microsoft Proxy Server [11]; the second was developed by a group from MIT [7], some variant of which may be deployed in the Akamai global cache hierarchy.

Before we describe the CARP and MIT schemes, we note that (despite the benefits of hash based schemes) there are situations when the Harvest/ICP approach is better. Hash based schemes should not be used for large systems in which the round trip times seen by users to different caches are largely changing. The reason is that the latency perceived by the users is linearly dependent on the round trip time between the user and the cache. TableI shows some sample measurements we did tabulating the round trip times perceived by users from different points in the US accessing the caches in the NLANR system. The names in the first column are cache names in the NLANR system.

The large variability in round trip times implies that such caches should not be used with a hash based scheme. The hash may result in a user picking a cache that is "far" away because it has no concept of locality. Hash based schemes thus seems best confined to cooperation between caches that are in a single domain or area. Of course, the two schemes can be combined with hash based caching used at the sibling level, and the ICP scheme used to contact parents in the cache hierarchy (if the assigned sibling does not have the page).

A. *Hash Routing in CARP*

CARP uses a hash function that takes two parameters, a URL and a cache name, and hashes some combination of its two parameters to a single integer. Thus if the system has N caches, how does a client know which cache to query for the data in URL x? The client does so by computing N different hash values for x with each of the N cache names, and picking the cache whose hash value is numerically the highest.

It can easily be shown by symmetry (if the hash values are uniformly distributed, it is equally likely that any of the values will be the highest) that URLs are uniformly distributed among caches. [9] shows that for a cache farm made up of N caches, if one cache fails then all URLs earlier assigned to caches that are still alive still stay assigned to the same cache after failure. However, roughly $\frac{1}{N}$ URLs (which were assigned to the cache that failed) will be reassigned to other caches. Similarly, if a new cache is added only about $\frac{1}{N+1}$ (the disruption coefficient) of the total number of URLs need to be reassigned, which is again the best possible. Naive hashing schemes (based on simply hashing the URLs and then assigning caches to ranges of hash values) have, by contrast, much larger disruption coefficients.

B. *MIT Hash Routing*

The MIT method uses two hash functions. One maps an URL address from the URL space to the interval $[0, 1)$ while the other maps a Proxy name into the same interval. The interval $[0, 1)$ is closed, all the operations being done modulo the size of the interval. The cache in charge of serving a request for an URL is the one from which the hash value of the URL is closest, clockwise, to the hash value of the cache name.

TABLE I
ROUND TRIP TIMES TO PROXY CACHES IN THE NLANR CACHE HIERARCHY

Proxy	U. of Chicago	UCSD	MIT	Cornell	Wash. U.
bo1	26.8	41	57	53	95
bo2	26.2	43	55	54	110
pa	59.2	154	91	103	76
pb	12.4	84	27	20	42
rtp	75.7	205	74	29	110
sd	66.7	1	92	92	109
sv	52.1	109	78	73	96
uc	5.8	77	33	32	-

IV. THEORETICAL COMPARISON OF MIT SCHEME VS. MICROSOFT SCHEME

[7] states the major requirements a cache system should satisfy:

• $balance$: documents should be evenly distributed to the caches;
• $monotonicity$: when a new cache is added, and the range of the hash function is changed, the mapping of documents to caches should not be completely reshuffeled. Normally when a new cache is introduced in the system an object should move from an old cache to a new cache but it should never move from an old cache to another old cache;
• $spread$: the number of different caches that are assigned responsibility for a document when there exist multiple views about the number of caches in the system, should be low,
• $load$: the number of URLs a cache is responsible for in the presence of multiple views.

We also suggest that another parameter worth considering is the resistance of a scheme to an adversary attack. [7] contains a detailed theoretical description of the properties of the MIT hash function. However, no such theoretical characterization has been done for the Microsoft scheme, making it appear that the Microsoft scheme does not satisfy some of the pleasing theoretical properties enjoyed by the MIT scheme. To remedy this, we have proved the following theorems that show that the Microsoft hashing scheme has exactly the same theoretical properties as the MIT scheme. The reader should compare our theorems with the corresponding theorems in [7] for the MIT scheme.

We consider the two schemes discussed above and we use the following notation:
• $MIT(h_1, h_2)$ = the algorithm implemented by MIT in which h_1 and h_2 are the two hash functions which are used to distributed the cache names and URLs to the domain $[0, 1)$;
• $MS(h)$ = the algorithm implemented by Microsoft in which h is a hash function having as arguments both the URL and the cache name.

Let consider the following definitions:
• Let B denote all the caches in the system;
• Let $V =$ denote a view of the system ($V \in B$);
• Let $f_V(i) =$ the cache on which the item i is allocated under the view $V \in B$;

Theorem 1: For any view V of the cache system and any item u, the probability that the item u to be allocated to an element $b \in V$ is $\frac{1}{|V|}$ in the Microsoft scheme.

Proof: The symmetry of the Microsoft hash function guarantees that for any view V of the cache system and any item u, the probability that the item u to be allocated to an element $b \in V$ is $\frac{1}{|V|}$ ■

Theorem 2: The Microsoft hash function is monotone: $\forall views V_1 \subseteq V_2 \subseteq B, f_{V_1}(i) \in V_1 \Rightarrow f_{V_1}(i) = f_{V_2}(i)$.

Proof: Let $(V_1 \subseteq V_2 \subseteq B) \wedge (f_{V_2}(i) \in V_1) \Rightarrow \exists v \in V_1$ such that $f_{V_2} = v \Rightarrow \forall p \in V_2, p \neq v, hash_{MS}(p, i) < hash_{MS}(v, i)$ Also, $V_1 \subseteq V_2 \Rightarrow \forall p \in V_1, p \neq v, hash_{MS}(p, i) < hash_{MS}(v, i) \Rightarrow f_{V_1}(i) = v$ which completes the proof that $f_{V_2}(i) = f_{V_1}(i)$. ■

Theorem 3: $(\forall h_1, h_2)(\exists h) MIT(h_1, h_2) = MS(h)$

Proof: For any hash functions $h_1(url)$ and $h_2(proxy)$ used in the MIT algorithm choose a hash function h for the Microsoft's algorithm such that $h(url, proxy) = 1- \parallel h_1(url) - h_2(proxy) \parallel$ ■

Our theorem shows that for any hash function that might be chosen to be used in the MIT scheme the scheme implemented by Microsoft can at least choose another hash function to get at least the same performances in terms of URLs distribution at proxies. [7] shows a detailed analysis of their algorithm but we believe that its limitation resides in the fact that it uses two hash functions which maps the URL and cache name domain to a finite one dimensional interval after which it takes the decision of which cache to be chosen. Since the hash functions of the caches can be precomputed, all we have to do to locate a cache for a given URL, is to hash the URL and to search for the hash with the closest clockwise value. This can be done using a simple modification of a priority queue in $\log N$ time, where N is the number of caches. The Microsoft algorithm has to determine N values for each request and select the one which is maximum which theoretically takes $O(N)$ time. However the values of N are small (typically less than 20), thus this does not seem to be an important factor.

However, the MIT scheme is susceptible to the single hash of the individual cache names; if this is not uniform, then the distribution of URLs is poor. The Microsoft scheme is less susceptible to this lack of uniformity because (in some sense), for each URL considered, the cache names are hashed independently. Thus when averaged over all the URLs considered, the lack of uniformity of any one hash is not a big factor. We will see this in the experimental results below. One intuitive way to make the MIT scheme less susceptible to uniformity is to use multiple hash points for each cache (using different numbers of points for each cache even allows a form of prioritization of caches). Intuitively, again, by hashing a cache name multiple times we can reduce the possibility of uneven distributions of URLs to caches.

The theorems above do not say anything about the behavior over changing views. Different views may appear when a cache changes its status. The clients in the system will have an inconsistent view about the new version for a while. The natural question is how much is this going to affect the performance of the system? There are two cases. If an active cache, let's say C goes down and a user A makes a request for a page u which should be served by C. For this situation the user in both systems (MIT and Microsoft) will not be served by the cache A. It will make a second request to the next valid cache selected through the algorithm (which represents the right behavior in case of failure). The second case represents the one in which a new cache is introduced into the system and it takes a while until all the clients get to know about it. The MIT scheme suggests using a modified DNS implementation to solve the mappings of requests to servers.

Caching at the DNS level may contribute to inconsistent views. A client with an old view may make requests to old caches even if in the new view it should ask for at the new introduced one. In this case it takes about one day to propagate the data about the new cache through the DNS system. In the Microsoft Proxy Server implementation a client during the startup of a session loads an html file containing the description of the cache system. Inconsistencies may occur because of sessions which may be opened for a long time without noticing the new modification in the cache system description file.

A different case is represented by a possible adversary attack on the cache system. An adversary may create unnecessary redundancies in the cache system by directly requesting a document from a cache which should not be in charge with the respective document. Both systems are not protected against this kind of attack at this moment. A simple solution to avoid it will be that for each request a cache should check if it is the one which should serve the request. If yes than it serves the request; otherwise, it can reply with a status code $Moved....$ This new behavior has other benefits, too. A cache may notice that it has an invalid view about the system if there are a number of clients (above a setup limit) which asks for documents which should not be served by it. In this case it can do an update of its view. A second benefit is that by getting a reply “Server Moved...” a client may initiate an update of its own view of the system.

V. Evaluation Methods

In next coming section, we show results from trace-driven simulations of both the MIT and Microsoft schemes. We consider as inputs a set of

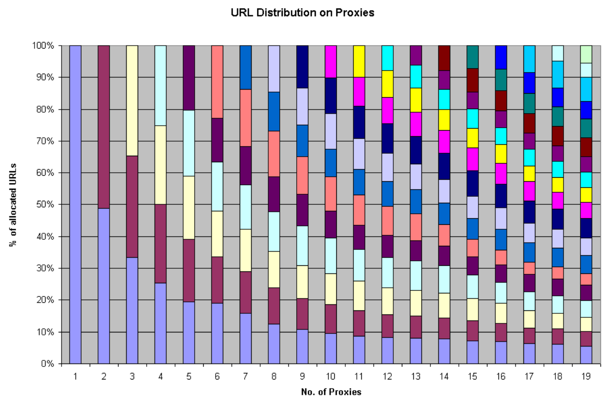

Fig. 2. Access distribution to caches in Microsoft scheme

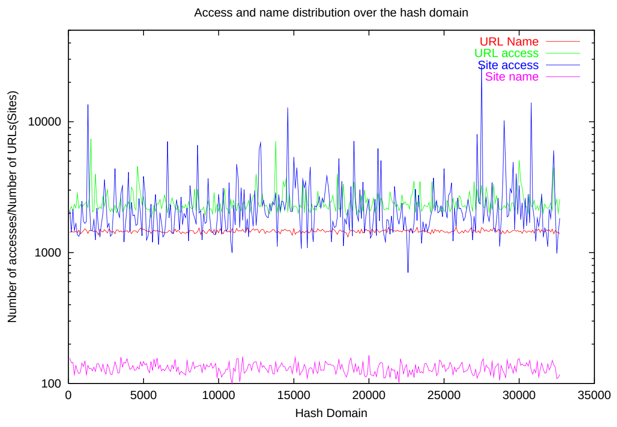

Fig. 3. Access and name distribution over the hash domain

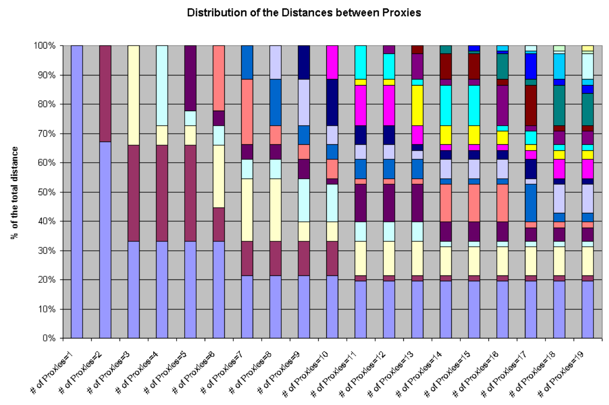

Fig. 4. URL distribution to caches in MIT scheme

traces provided by NLANR for the NLANR cooperative cache systems. The traces contain user accesses to several caches. An entry contains the access time, the URL which was accessed, the size and type of access, etc.

Our simulations study the URL distribution on the caches for both algorithms using the traces described above. We would like to answer several questions.

- Which of the algorithms offers a better distribution of the accesses to the proxies?
- How many number of points should be associated with a cache on the hash domain in order to guarantee a relatively even distribution for the MIT algorithm?
- What can be done in order to achieve a more even distribution of the accesses to caches in the MIT scheme?

VI. Results

Our first comparison is between the distribution of URLs in the trace as hashed by the MIT and Microsoft scheme to caches, for different numbers For the MIT system we assume that each cache has only one point associated on the hash domain.

From Figure 2, we can see that the URL distribution to caches in the Microsoft implementation is extremely even, with a very small deviation. This follows from the theorem in Section III. For example, when the number of caches is equal to two we can see an almost equal distribution of URLs to caches. Increasing the number to 3 we get about 33% of the total number of URLs distributed to each cache. It continues with about 25% of URLs if the total number of caches is 4 and so on.

On the other hand, the MIT algorithm computes for each request the hash value associated with the requested URL. The cache which is responsible for answering the request is the one having the hash value of its name closest clockwise to the hash value of the hash value of the URL.

In figure 3 we show the distribution of both accesses and names in the hash domain for our set of traces. Several conclusions can be drawn from this figure. Both the names of the sites which are accessed and the URLs (in the trace) are uniformly distributed over the hash domain. The number of accesses are also uniformly distributed over the hash domain with some spikes related with the popularity of the documents or sites which are accessed. This suggests that the cache hash function for an URL should be computed only based on the site name.

There is no gain in terms of a better load distribution or access distribution by computing the hash value based on the full URL.

The results in figure 3 show also that an uniform distribution of the URLs/accesses to caches can be gained only by having an uniform distribution of the hash values for the cache names over the hash domain.

Figure 4 shows an uneven distribution of the distances between the hash values computed for the caches using the trace on the MIT scheme. This uneven distribution of the distances between caches is the cause of an unbalanced distribution of URLs/accesses to the caches in the system. In a real system, this would lead to poor load balancing among caches as some caches get assigned more URLs than others. Figure 6 shows that a first approach to making the hash domain distribution to proxies more even is by increasing the number of points associated with each proxy. Our results show that the number of pints to be used should be about log(No. of Proxies). We now suggest a modification of the MIT scheme to make the distance distribution more uniform. We call it ModMIT.

A. A New URL to Cache Allocation Scheme (ModMIT)

We propose a simple modification of the MIT scheme in which a URL is allocated to the cache for which the hash value of the cache is the closest one to the hash value of the URL, *regardless* of the direction (unlike the MIT scheme which considers only clockwise nearness). Intuitively, for the case $N = 2$ this will provide perfect fairness even if the two hash values happen to hash to nearby points; on the other hand, the MIT scheme will do very badly for the case of $N = 2$ (and a single hash point per cache) if the two hash values do not evenly break up the number line(figure 5).

We simulate our new implementation versus the MIT implementation. If we consider the variance of the hash domain distribution to the caches for both allocation schemes as it is shown in Figure 6, one can easily notice that our allocation scheme outperforms the MIT scheme in all the cases which are considered. The figure also reveals how many points should be allocated for each cache inside of the hash domain. A number of 3 points associated for each

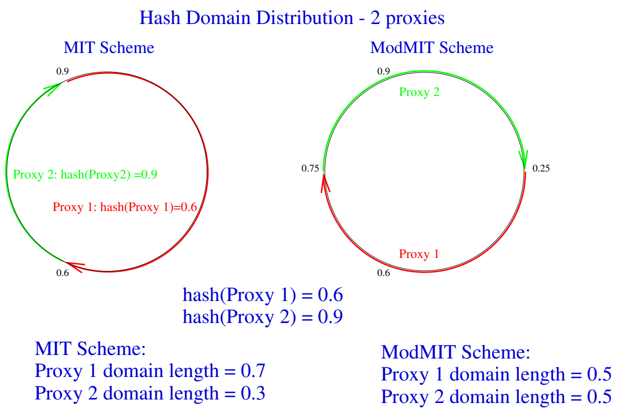
Hash Domain Distribution - 2 proxies
MIT Scheme
ModMIT Scheme
0.9
Proxy 2: hash(Proxy2) =0.9
Proxy 1: hash(Proxy 1)=0.6
0.6
0.9
Proxy 2
0.75
0.25
Proxy 1
0.6
hash(Proxy 1) = 0.6
hash(Proxy 2) = 0.9
MIT Scheme:
Proxy 1 domain length = 0.7
Proxy 2 domain length = 0.3
ModMIT Scheme:
Proxy 1 domain length = 0.5
Proxy 2 domain length = 0.5

Fig. 5. Hash Domain Distribution to Proxies in MIT and ModMIT Scheme

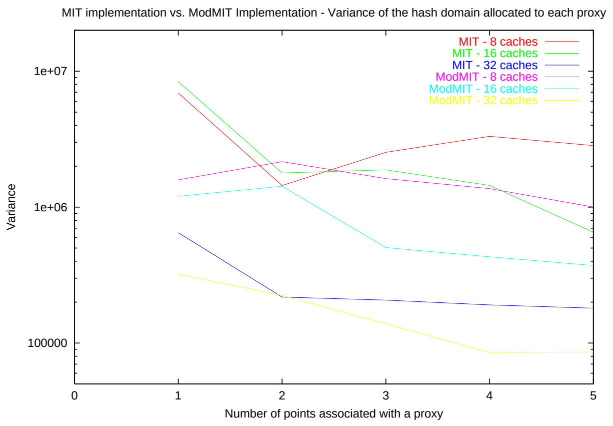

Fig. 6. The variance of the distribution of the hash domains which are allocated to caches

cache is a minimum to be used in both MIT algorithm and our modification. The variance drops by a factor of $2-3$ when we change from an implementation with only one hash value per cache to an implementation with 3 values per cache. Our implementation does not increase the complexity of the MIT algorithm because all the hash values for the caches are computed in advance. The complexity of the search process is $O(log(numberOfPointsPerCache * NoCaches))$ in both cases.

VII. Summary

In this paper we analyzed cooperative caching mechanism and compared two major proposals for hash-based cooperative caching. Hash based cache systems should be used as a way to implement cooperative caching mostly in local environments like an ISP domain, where the round trip time to access the caches in the systems are similar.

Our comparisons of the MIT and Microsoft schemes using real traces show that:

• The hash value associated with an URL may be computed only based on the site name part from the URL. In this way all the requests addressed for a particular site will be handled by the same cache system. Our traces indicate that this does not contribute to an unbalanced load of the caches in the system. In fact, it may be beneficial for improving the efficiency of the connection between the cache and the content server using persistent HTTP.

• The MIT scheme in the best case can at most reach the performances of the Microsoft scheme in terms of balancing URL assignments to caches.

• Microsoft scheme has an even distribution of both URL names and accesses to caches based on the symmetry characteristics of the algorithm that it uses.

• The MIT scheme has an uneven distribution of URLs to caches because of an uneven distribution of the hash domain to caches. An even distribution can be reached by increasing the number of points in the hash domain which are associated with each cache.

We also introduced a modification of the MIT scheme that reduces the variance of the distribution of URLs to caches. Our modification reduces the variance of the MIT scheme without increasing the complexity of the search process of determining which cache should handle a request.

VIII. Acknowledgements

We would particularly like to thank Duane Wessels from NLANR for the inside-full discussion about the NLANR cache system. We would also like to thank the people from NLANR who are making the traces we use available for study. We would also like to thank George Varghese for the fruitful discussion on this topic and Lili Qiu, Adriana Iamnitchi and Grigore Rosu for their suggestions.

References

[1] C. Bowman, P.B. Danzig, M. F. Scwartz, and al. Harvest: A scalable customizable discovery and access system. Technical Report CU-CS-732-94, University of Colorado at Boulder, CU-CS-732-94, 1994.

[2] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *the Proc. of IEEE INFOCOM'99*, pages 126–134, march 1999.

[3] K. Claffy. Internet measurement and data analysis:topology, workload, performance and routing statistics. In *NAE'99 workshop*, 1999.

[4] Microsoft Corp. Cache array routing protocol and microsoft proxy server 2.0. In *White Paper*, 1999.

[5] L. Fan, P. Cao, J. Almeida, and A.Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *the Proc. of ACM SIGCOMM'98*, august 1998.

[6] Steven D. Gribble and Eric A. Brewer. System design issues for internet middleware services: Deductions from a large client trace. In *Usenix Symposium on Internet Technologies and Systems*, august 1997.

[7] David Karger and al. Web caching with consistent hashing. In *WWW8 conference*.

[8] A. Lutonen, H. F. Nielsen, and T. Berners-Lee. Cern httpd. In *http://www.w3.org/pub/WWW/Daemon/Status.html*, july 1996.

[9] K.W. Ross. Hash-routing for collections of shared web caches. In *IEEE Network Magazine*, nov-dec 1997.

[10] Alex Rousskov and Duane Wessels. Cache digests. In *Proceedings of the Third International Web Caching Workshop*.

[11] V. Valloppillil and K. W. Ross. Cache array routing protocol v1.0. In *Internet Expired Draft*, february 1998.

[12] D. Wessels and K. Claffy. Application of internet cache protocol(icp) version 2. In *RFC 2186 - Informational RFC*, september 1997.

[13] D. Wessels and K. Claffy. Icp and the squid web cache. In *IEEE Journal on Selected Areas in Communication, Vol. 16,#3*, april 1998.

[14] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. In *the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 16–31, december 1999.