

Cache Clouds: Cooperative Caching of Dynamic Documents in Edge Networks

Lakshmith Ramaswamy[◇],

Ling Liu[◇]

Arun Iyengar[♠]

[◇]College of Computing, Georgia Tech
Atlanta GA 30332
{laks, lingliu}@cc.gatech.edu

[♠]IBM T. J. Watson Research Center
Yorktown Heights NY 10598
aruni@us.ibm.com

Abstract

Caching on the edge of the Internet is becoming a popular technique to improve the scalability and efficiency of delivering dynamic web content. In this paper we study the challenges in designing a large scale cooperative edge cache network, focusing on mechanisms and methodologies for efficient cooperation among caches to improve the overall performance of the edge cache network. This paper makes three original contributions. First, we introduce the concept of cache clouds, which forms the fundamental framework for cooperation among caches in the edge network. Second, we present dynamic hashing-based protocols for document lookups and updates within each cache cloud, which are not only efficient, but also effective in dynamically balancing lookup and update loads among the caches in the cloud. Third, we outline a utility-based mechanism for placing dynamic documents within a cache cloud. Our experiments indicate that these techniques can significantly improve the performance of the edge cache networks.

1. Introduction

The enormous increase of the dynamic web content in the past decade has posed a serious challenge to the performance and scalability of the World Wide Web. Caching on the edge of the network has received considerable attention from the research community as a promising solution to ameliorate this problem. The underlying philosophy of edge caching is to move data, and possibly some parts of the application, closer to the user.

Designing an efficient cooperative edge caching scheme is very attractive considering the potential benefits it can provide. First, when an edge cache receives a request for a document that is not locally available, it can try to retrieve the document from nearby caches rather than contacting the remote server immediately. Retrieving a document from a nearby cache can significantly reduce the latency of a local miss. It also reduces the number of requests reaching

the remote servers, thereby reducing their load. The second benefit of cooperation among edge caches is the reduction in the load induced by the document consistency maintenance on the origin servers. When the caches are organized as cooperative groups, the server can communicate the update message to a single cache in a cache group, which distributes it to the other edge caches within its group.

Building a large scale cooperative edge cache network poses several research challenges. First, an effective mechanism is needed to decide the appropriate number of edge caches needed for the edge network, and the geographical locations where they have to be placed. Second, these caches need to be organized into cooperative groups such that the cooperation among the caches within a group is effective and beneficial. Third, a dynamic and adaptive architecture is required for efficient cooperation within each cache group to deal with the constantly evolving nature of dynamic content delivery like continually changing document update and user request patterns. Concretely, there is a need for methodologies and techniques for flexible and low-overhead cooperation among the edge caches in terms of document lookups, document updates, as well as document placements and replacements.

The ultimate goal of our research is to design and develop techniques and system-level facilities for efficient delivery of dynamic web content in a large scale edge cache network, utilizing the power of flexible and low cost cooperation. Towards this end, we introduce the concept of *cache clouds* as a fundamental framework of cooperation among the edge caches. A cache cloud contains caches of an edge network that are located in close network proximity. The caches belonging to a cache cloud cooperate both for serving misses and for maintaining freshness of the cached document copies.

The main research contributions in this paper are three fold: (1) We present the architecture of the cache clouds, which are designed to support efficient and effective cooperation among their caches. (2) We propose a *dynamic hashing*-based cooperation scheme for efficient document lookups and document updates within each cache cloud.

This scheme not only improves the efficiency of document lookups and updates, but also balances the document lookup and update loads dynamically among all caches in each edge cache cloud in anticipation of sudden changes in the request and update patterns. (3) We develop a utility-based document placement scheme for strategically placing documents among caches of a cache cloud, so that available resources are optimally utilized. This document placement scheme estimates the costs and benefits of storing a document in a particular edge cache, and stores the document at that cache only if the benefit to cost ratio is favorable. We evaluate these techniques through trace-based experiments with a real trace from a highly accessed web site, and with a synthetic dataset. The results indicate that these schemes can considerably improve performance of edge cache networks.

2. Cache Clouds

A cache cloud is a group of edge caches from an edge network that cooperate among themselves to efficiently deliver dynamic web content. The caches in a cache cloud cooperate in several ways to improve the performance of edge cache networks. First, when a cache experiences a miss, it tries to retrieve the document from another cache within the cache cloud, instead of immediately contacting the server. Second, the caches in a cache cloud collaboratively share the cost of document updates in the sense that the server needs to send a document update message to only one cache in a cache cloud, which is then distributed to other caches that are currently holding the document. Third, the edge caches in a cache cloud collaborate with each other to optimally utilize their collective resources by adopting smart strategies for document lookups, updates, placements and replacements.

An important factor that determines the effectiveness of cooperation in cache clouds is the manner in which the cache clouds are constructed. In order for the cooperation to be efficient the caches belonging to a cache cloud should be located in close vicinity within the Internet. We have developed an Internet landmarks-based technique to create cache clouds by accurately clustering the caches of an edge network [12]. Due to the space limitation we omit any further discussion on this technique in this paper, and assume that the cache clouds are formed using this scheme. In the rest of this paper, we concentrate on the design issues within a cache cloud.

2.1. Architecture and Design Ideas

We have discussed three forms of cooperation within a cache cloud, namely collaboratively serving misses, cooperatively handling document updates, and optimally utiliz-

ing the collective resources within the cache cloud. A cache that needs to retrieve a document needs to locate the copies of the document existing within the cache cloud. We refer to the mechanism of locating the copies of a document within a cache cloud for the purpose of retrieving it as the *document lookup protocol*. The mechanism used by the cache cloud to communicate the document update to all its caches that are currently holding the document is the *document update protocol*.

Among the several issues that influence the design of the cache cloud architecture, some of the important ones are: (1) providing an efficient document lookup protocol (2) designing a low-overhead document update protocol, and (3) developing a utility-based scheme for document placement decisions.

We adopt a distributed approach to designing the cache cloud architecture, wherein all the caches in the cloud share the functionalities of lookups and updates. Each cache is responsible for handling the lookup and update operations for a set of documents assigned to it. In a cache cloud, if the cache Pc_i^j is responsible for the lookup and update operations of a document Dc , then we call the cache Pc_i^j as the *beacon point* of Dc . The beacon point of a document maintains the up-to-date lookup information, which includes a list of caches in the cloud that currently hold the document. A cache that requires document Dc , contacts Dc 's beacon point, and obtains its lookup information. Then it retrieves the document from one of the caches currently holding the document. Similarly, if the server wants to update document Dc , it sends an update message to its beacon point, which then distributes this message to all the holders of the document.

An immediate question that needs to be addressed is how to decide which of the caches in the cloud should act as the beacon point of a given document. In designing the cache cloud architecture, our goal is to assign beacon points to documents such that the following properties are satisfied:

- The caches within the cache cloud and the origin server can discover the beacon point of a document efficiently.
- The load due to document lookups and updates is well distributed among all beacon points in the cache cloud.
- Load balancing is preserved when the lookup and update patterns change over time.
- The beacon point assignment should be resilient to failures of individual caches in the cloud

A straightforward solution for the beacon point assignment problem would be to use a random hash function. These hash functions uniquely hash the document's URL to one of the edge caches (beacon points) in the cache cloud, which acts as the document's beacon point. We refer to this scheme as the *static hashing* scheme. The static hash-

ing scheme has a significant drawback: Lookup and update loads often follow the highly skewed Zipf distribution, and under such circumstances random hashing cannot provide good load balancing among the caches belonging to the cloud. Consistent hashing [5] has been popular as a technique for providing good load balancing among a network of nodes. In this technique, the document URLs and the edge cache identifiers are both mapped on to a unit circle. Each document is assigned to the cache node (its beacon point) that is nearest to its identifier on this circle. While consistent hashing can distribute the URLs uniformly across the caches, it significantly increases the lookup and the update costs, especially for those documents that are hot or that are updated frequently. If a cloud contains N edge caches (beacon points), with consistent hashing, the beacon point discovery process might take up to $O(\log N)$ time-steps. Besides, uniform distribution of URLs across beacon points does not yield good load balancing when the lookup and update loads follow a skewed distribution. These shortcomings make the consistent hashing scheme less attractive to the scenarios where the performance of the lookups and updates is very crucial. Further, both static and consistent hashing schemes cannot preserve load balancing when the update and lookup load patterns change over time.

Considering the drawbacks of the above schemes, we propose a dynamic hashing-based mechanism for assigning the beacon point of a document. This mechanism supports efficient lookup and update protocols, provides good load balancing properties, and adapts to changing load patterns effectively.

2.2. Design of Dynamic Hashing Scheme

Consider a cache cloud with N edge caches. Each of these caches maintains lookup information about a set of documents. In the dynamic hashing scheme, the assignment of documents to beacon points can vary over time so that the load balance is maintained even when the load patterns change.

In our scheme the edge caches of a cache cloud are organized into substructures called *beacon rings*. A cache cloud contains one or more beacon rings, and each beacon ring has two or more beacon points. Figure 1 shows a cache cloud with 4 beacon rings, where each beacon ring has 2 beacon points. All the beacon points in a particular beacon ring are collectively responsible for maintaining the lookup information of a unique set of documents. Suppose a cache cloud has K beacon rings numbered from 0 to $K - 1$, and a document Dc is mapped to the beacon ring j . Any of the beacon points in the beacon ring j may be assigned to serve as the beacon point of Dc . The assignment of the beacon point within a beacon ring is done dynamically and may change over time for maintaining good balance of loads among the

beacon points within a beacon ring.

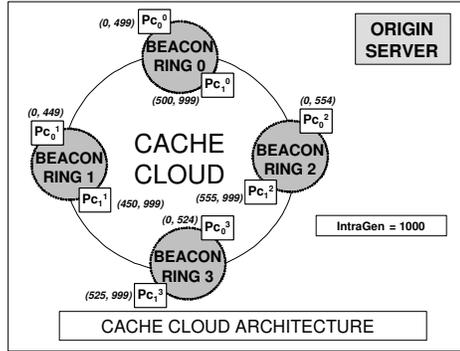


Figure 1. Architecture of Edge Cache Cloud

We now explain the technique for assigning beacon points to documents within a beacon ring. For simplicity, let us suppose that the cache cloud contains a single beacon ring, which is numbered 0. Let the N caches be represented as $\{Pc_0^0, Pc_1^0, \dots, Pc_{N-1}^0\}$. A dynamic hashing technique called the *intra-ring hash* is used for distributing the documents to the N beacon points, which is designed as follows. An integer that is relatively large compared to the number of beacon points in the beacon ring is chosen and designated as the intra-ring hash generator (denoted as *IntraGen*). The range of intra-ring hash values $(0, IntraGen - 1)$ is divided into N consecutive non-overlapping sub-ranges represented as $\{(0, MaxIrH_0^0), (MinIrH_1^0, MaxIrH_1^0), \dots, (MinIrH_{N-1}^0, IntraGen - 1)\}$. Each edge cache within the beacon ring is allocated one such sub-range. For example, the beacon point Pc_i^0 is assigned the sub-range $(MinIrH_i^0, MaxIrH_i^0)$. The scheme also hashes each document's URL to an integer value between 0 and $(IntraGen - 1)$. This value is called the document's intra-ring hash value or IrH value for short. For example, for a document Dc , the IrH value would be $IrH(Dc) = MD5(URL(Dc)) \text{ Mod } IntraGen$, where $MD5$ represents the MD5 hash function, $URL(Dc)$ represents the URL of the document Dc and Mod represents the modulo function. Each edge cache would serve as the beacon point for all the documents whose *IrH* value lies within the sub-range allocated to it, i.e. Pc_i^0 will serve as the beacon point of a document Dc , if $MinIrH_i^0 \leq IrH(Dc) \leq MaxIrH_i^0$.

2.3. Determining the Beacon Point Sub-Ranges

In this section we explain the mechanism of dividing the intra-ring hash range into sub-ranges such that the load due to document lookups and updates is balanced among the beacon points of a beacon ring. This process is executed periodically (in cycles) within each beacon ring, and

it takes into account factors such as the beacon point capabilities, and the current loads upon them. Any beacon point within the beacon ring may execute this process. This beacon point collects the following information from all other beacon points in the beacon ring.

- **Capability:** Denoted by Cp_i^0 , it represents the power of machine hosting the beacon point Pc_i^0 . Various parameters such as CPU capacity or network bandwidth may be used as measures of capability. We assume a more generic approach wherein each beacon point is assigned a positive real value to indicate its capability.
- **Current Sub-Range Assignment:** Denoted by $(MinIrH_i^0, MaxIrH_i^0)$, it represents the sub-range assigned to the beacon point Pc_i^0 in the current cycle.
- **Current Load Information:** Represented by $CAvgLoad_i^0$, it indicates the cumulative load due to document lookup and update propagation averaged over the duration of the current period. The scheme can be made more accurate if the beacon points also collect load information at the granularity of individual IrH values, which we denote by $CIrHLD_i^0(h)$. $CIrHLD_i^0(h)$ is the sum of the loads induced by all documents whose IrH value is h . However, the $CIrHLD$ information is not mandatory for the scheme to work effectively.

After obtaining this information the process of determining the sub-ranges for the next cycle begins. The aim is to update the sub-ranges such that the load a beacon point is likely to encounter in the next cycle is proportional to its capability. For each beacon point we verify whether the fraction of the total load on the beacon ring that it is currently supporting is commensurate with its capability. If the fraction of load currently being handled by a beacon point exceeds its share then its sub-range shrinks for the next cycle, thus shedding some of its load. In case a beacon point is handling a smaller fraction of load, its sub-range expands increasing its load for the next cycle.

Specifically, the scheme proceeds as follows. First, we calculate the total load being experienced by the entire beacon ring (represented as $BRingLd^0$), and the sum of the capabilities of all the beacon points belonging to the ring (represented as $TotCp^0$). Then for each beacon point we calculate its appropriate share of the total load on the beacon ring as $AptLd_i^0 = \frac{Cp_i^0}{TotCp^0} \times BRingLd^0$. Now, we examine all the beacon points in the beacon ring starting from Pc_0^0 , and compare their $CAvgLd$ with their $AptLd$. If $CAvgLd_i^0 > AptLd_i^0$, then Pc_i^0 is currently supporting more load than its appropriate share. The value $(CAvgLd_i^0 - AptLd_i^0)$ is called the load-surplus at beacon point Pc_i^0 , and is represented as $LdSpls(Pc_i^0)$. In this case, the scheme shrinks the sub-range of the beacon point for the next cycle by decreasing its $MaxIrH_i^0$ value. The amount (say t) by which the $MaxIrH_i^0$ is decreased is cal-

culated using the $CIrHLD$ information. $MaxIrH_i^0$ is decreased by t IrH values, such that the sum of the loads generated by these IrH values is equal to the load-surplus at Pc_i^0 . In other words, the scheme shifts t IrH values from the end of Pc_i^0 's sub-range to the beacon point Pc_{i+1}^0 , such that $\sum_{h=MaxIrH_i^0-t}^{MaxIrH_i^0} CIrHLD_i^0(h) = LdSpls(Pc_i^0)$. The new $MaxIrH$ value of Pc_i^0 represented as $NewMaxIrH_i^0$ would be equal to $(MaxIrH_i^0 - t)$. When the sub range of a beacon point Pc_i^0 shrinks, some of its load would be pushed to the beacon point Pc_{i+1}^0 . The scheme takes into account this additional load on the beacon point Pc_{i+1}^0 when deciding about its new sub-range.

On the other hand if $CAvgLd_i^0 < AptLd_i^0$ then the scheme expands the sub-range of the beacon point Pc_i^0 by increasing its $MaxIrH$ value. The amount by which $MaxIrH$ is increased is determined in a very similar manner as the shrinking case discussed above. In this case, Pc_i^0 acquires additional load from the beacon point Pc_{i+1}^0 . In this manner the sub-range assignments of all the beacon points in the beacon ring are updated. Some beacon points might find it costly to maintain the $CIrHLD$ information for each of the hash values within its sub-range, in which case the $CIrHLD_i^0(p)$ for all hash values in the sub-range of the beacon point Pc_i^0 are approximated by averaging $CAvgLd_i^0$ over its sub-range of IrH values.

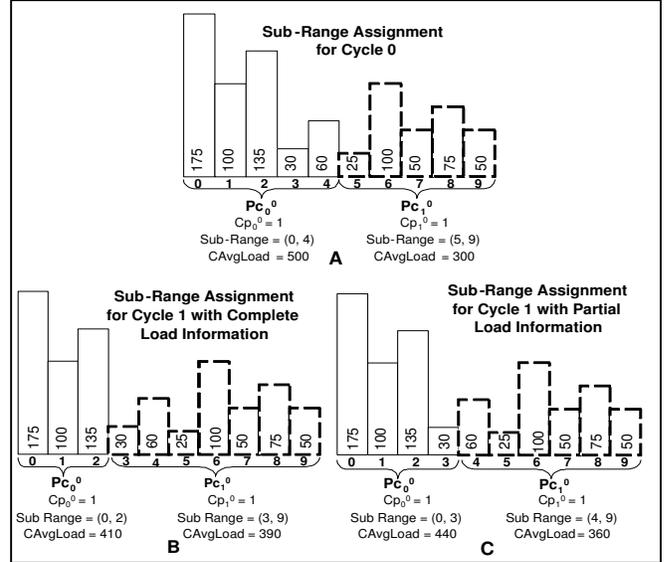


Figure 2. Illustration of Sub-Range Determination

After determining the sub-range assignments for the next cycle, all the caches in the cache ring and the origin server are informed about the new sub-range assignments. Beacon points that have been assigned new IrH values obtain lookup records of the documents belonging to the new IrH values

from their current beacon points.

We now illustrate the sub-range determination scheme with an example. Consider the beacon ring with two beacon points Pc_0^0 and Pc_1^0 . Let both the beacon points have equal capabilities, and let $IntraGen$ be 10. Initially the range $(0, 9)$ is divided equally between the two beacon points. Figure 2-A illustrates this scenario. The vertical bars represent the update and object lookup loads corresponding to each hash value. As we see, equal division of the intra-ring hash range does not ensure load balancing between the two beacon points due to the skewness in the load. The total load experienced by the two beacon points in cycle 0 are 500 and 300 respectively. At the end of cycle 0, the sub-ranges are updated taking into account the current load patterns. Now we consider 2 scenarios. Figure 2-B represents the first scenario, wherein the beacon points maintain $C IrHLd$ for each hash value. In this case, two hash values are moved from Pc_0^0 to Pc_1^0 . The loads on the two beacon points would now be 410 and 390 respectively. In the second scenario, which is represented in Figure 2-C, the beacon points do not maintain the $C IrHLd$ information, and hence they have to use $C AvgLd_i^0$ to approximate the $C IrHLd$ value, which would be 100 for all hash values belonging to Pc_0^0 . We shift only one hash value between beacon points. The loads on the two beacon points would be 440 and 360 thus showing that the scheme is more accurate when the load information is available at the granularity of IrH values.

In the discussion, up to now we have assumed that the cache cloud contains a single beacon ring. However, if the cache cloud contains several caches, having a single beacon ring is not practical, since the cost and complexity of the sub-range determination process increases as the beacon rings become larger. In this case it is advantageous to have multiple beacon rings. Suppose a cache cloud has K beacon rings, and each beacon ring has M beacon points. Then the beacon point of a document Dc is determined in a two-step process. In the first step, the beacon ring of the document is determined by a random hash function. For example, if $j = MD5(URL(Dc)) \text{ Mod } K$, then the ring- j is the beacon ring of Dc . In the second step, out of the M beacon points within the j^{th} beacon ring, the beacon point of Dc is determined through the intra-ring hash function as we discussed before. The beacon point whose current sub-range contains $IrH(Dc)$ would be the beacon point of Dc . In our scheme, the document lookup and update protocols work as follows. When a cache needs to locate a document Dc , it determines the document's beacon point using the two-step process described above. Then it contacts the document's beacon point and obtains the list of caches that hold the cached copies of the document within the given cache cloud. When the server needs to communicate an update to the document Dc , it uses the two-step process and deter-

mines the document's beacon point for each cache cloud. It sends a document update message to these beacon points (one for each cloud), which in turn communicate it to the caches in their cache clouds, which are currently holding the document.

An important question that needs to be addressed is: *What should be size of the beacon rings in a cache cloud for optimal performance of the scheme?* While larger beacon rings provide better load balancing, they also increase the cost and complexity of the sub-range determination process. However, the other extreme would be to have beacon rings with single beacon points in them. In this case the dynamic hashing scheme reduces to the static hashing scheme, and hence cannot provide good load balancing. It can be theoretically shown that by having two beacon points in each beacon ring we can obtain significantly better load balancing when compared with static hashing, and further increasing the size of beacon rings improves the load balancing incrementally, but at a higher load balancing cost [11]. Considering these issues we conclude that beacon rings should have at least 2 beacon points, but their sizes should be small enough for the sub-range determination process to be simple and efficient. Our experimental results validate this observation.

The dynamic hashing mechanism can be extended to provide resilience to failures of individual beacon points by lazily replicating the lookup information. Due to space constraints we do not discuss the failure resilience property in this paper.

3. Document Placement in Cache Clouds

A good document placement scheme is very important for a cache cloud to optimally utilize the resources available. In this section we briefly discuss a utility-based scheme for placing dynamic documents in cache clouds.

A simple document placement scheme would be to place a document at each cache that has received a request for that document. We refer to this scheme as the *ad hoc* document placement scheme. Although the ad hoc placement scheme seems natural, it leads to uncontrolled replication of documents, which not only increases the document freshness maintenance costs, but also causes higher disk-space contention at the caches, thereby reducing the aggregate hit rate of the cloud [10]. These performance limitations are the manifestations of the shortcomings of the ad hoc placement scheme, which regards the caches in the cloud as completely independent entities, and makes document placement decisions without the knowledge about the other caches in the cache cloud.

An alternative approach for document placement, called the *beacon point caching*, would be to store each document only at its beacon point. This policy results in the beacon

points of hot documents encountering heavy loads. Further, this policy causes the edge caches to contact each other very frequently for retrieving documents. This not only causes heavy network traffic within the cache cloud, but also leads to clients experiencing high latencies for their requests.

3.1. Utility-based Document Placement

In this section we discuss the design of a utility-based document placement scheme, in which the caching decisions rely upon the utility of a document-copy to the cache storing it and to the entire cache cloud. This *utility value* of the document copy is represented as $Utility(Dc)$ for document copy Dc . The utility of document copy Dc estimates the benefit-to-cost ratio of storing and maintaining the new copy. A higher value of utility indicates that benefits outweigh the costs, and vice-versa. When a cache retrieves a document it calculates its utility value and decides whether or not to store the document based on this utility value.

Our formulation of the utility function has four components. Each of these components quantifies one aspect of the interplay between benefits and the costs. We now give a brief overview of each of these components. A detailed description of the utility function components including their mathematical formulations is available in the technical report version of this paper [11]. Throughout this discussion we assume that a cache cloud CC has N edge caches represented as $\{Pc_0, Pc_1, \dots, Pc_{N-1}\}$, and the edge cache Pc_l has retrieved the document copy Dc , and is calculating its utility value to decide whether to store it locally.

Document Availability Improvement Component

Represented by $DAIC(Dc, Pc_l)$, this component quantifies the improvement in the availability of the document in the cache cloud achieved by storing the document copy at Pc_l . Improving the availability of a document increases the probability that a future request for the document would be served within the cache cloud, thereby yielding higher cumulative hit rates.

Disk-Space Contention Component

This component (represented as $DsCC(Dc, Pc_l)$) captures the storage costs of caching the document copy at Pc_l in terms of the disk-space contention at Pc_l . The disk-space contention at the cache Pc_l determines the time duration for which the document can be expected to reside in the cache Pc_l before it is replaced. A higher value of $DsCC(Dc, Pc_l)$ implies that the new document copy of Dc at the cache Pc_l is likely to remain longer in the cache cloud than the existing copies (if any) of Dc within the cloud, and hence, it is beneficial to store this copy.

Consistency Maintenance Component

Denoted by $CMC(Dc, Pc_l)$, this component accounts for the costs incurred for maintaining the consistency of the new document copy at Pc_l , and the advantages that are obtained by storing Dc at Pc_l by avoiding the cost of retrieving the document from other caches on each local access. A high value of $CMC(Dc, Pc_l)$ indicates that the document Dc is accessed more frequently than it is updated, and vice-versa.

Access Frequency Component

This component of our utility function (represented as $AIC(Dc, Pc_l)$) quantifies how frequently the document Dc is accessed in comparison to other documents stored in the cache. If the access frequency of Dc at the cache Pc_l is high when compared to other documents in the cache, it is advantageous to store the document Dc at Pc_l .

The Utility Function

The above-mentioned four components form the building blocks of the utility function. We observe that for each component, a higher value implies that the benefits of storing Dc are higher than the overheads, and vice-versa. We define the utility of storing the document Dc at cache Pc_l , denoted as $Utility(Dc, Pc_l)$, to be a weighted linear sum of the above four components.

$$Utility(Dc, Pc_l) = W_{DAIC} \times DAIC(Dc, Pc_l) + W_{DsCC} \times DsCC(Dc, Pc_l) + W_{CMC} \times CMC(Dc, Pc_l) + W_{AFC} \times AFC(Dc, Pc_l)$$

In the above equation W_{DAIC} , W_{DsCC} , W_{CMC} , and W_{AFC} are positive real constants such that $W_{DAIC} + W_{DsCC} + W_{CMC} + W_{AFC} = 1$. These constants are assigned values reflecting the relative importance of the corresponding component of the utility function to the performance of the system.

Concretely, the utility-based document placement scheme works as follows: Suppose the cache Pc_l encounters a local miss for document Dc , and retrieves it from another cache in the cache cloud, or from the origin server (in the event of a group miss). The cache Pc_l now evaluates the utility function $Utility(Dc, Pc_l)$ locally using the request and update patterns of the document collected through continued monitoring in the recent time duration. Dc is stored at Pc_l only if the value of the utility function exceeds a threshold, represented as $UtilThreshold(Pc_l)$.

4. Experiments and Results

We have evaluated the proposed schemes through trace-based simulations of an edge cache network. The simulator can be configured to simulate different caching architectures such as edge network without cooperation, cooperative caching with static hashing, and cooperative cache clouds with dynamic hashing. It can also simulate ad

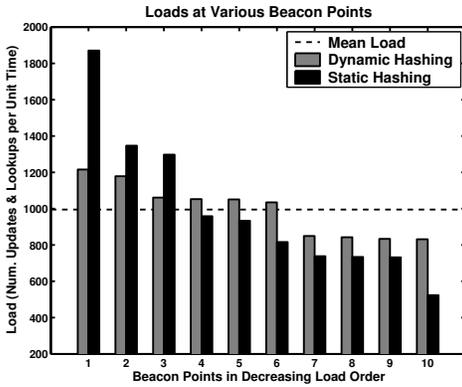


Figure 3. Load Distribution for Zipf-0.9 Dataset

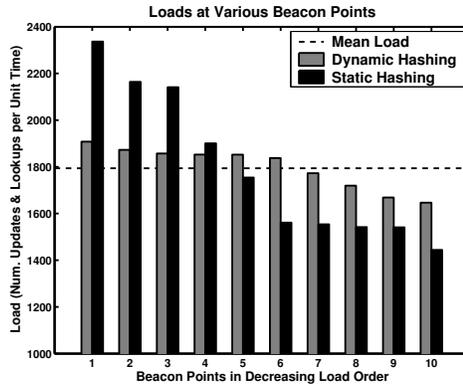


Figure 4. Load Distribution for Sydney Dataset

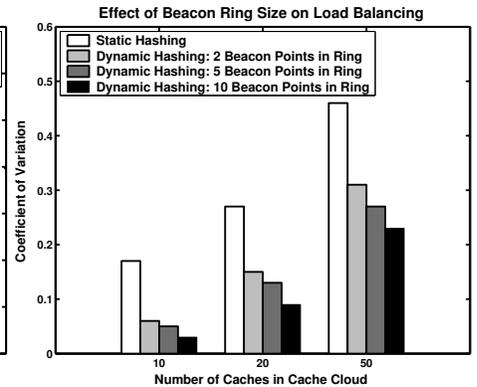


Figure 5. Impact of Beacon Ring Size on Load Balancing

hoc, beacon point and utility-based document placement schemes. Each cache in the cache cloud receives requests continuously according to a request-trace file, and the server continuously reads from an update trace file. Upon reading an update entry for a document D_c , the server sends the updated version of D_c to its beacon points within each cache cloud, which is then distributed by the beacon points to all the caches in their cache clouds which are currently holding the document. The document request rates, the document update rate, the number of caches and the number of beacon rings in the cache cloud are system parameters and can be varied.

We use two types of traces for our experiments. The first trace is a synthetic dataset, called as the Zipf-0.9 dataset. In this dataset there are 25,000 unique documents. Both accesses and invalidations follow the Zipf distribution with the Zipf parameter value set to 0.9. The second dataset is a real trace from a major IBM sporting and event web site¹. This trace was obtained by capturing the accesses and updates in a 24-hour time period. The number of unique documents in this data set is 52,574. We refer to this dataset as the Sydney dataset.

4.1. Evaluating the Effectiveness of Beacon Rings

In the first set of experiments we study the load balancing properties of the dynamic hashing mechanism. All the beacon points within the cache cloud are assumed to be of equal capabilities, which implies that perfect load balancing is achieved when all the beacon points encounter the same amount of load. In all three experiments in this set the intra-ring hash generators (*IntraGens*) are set to 1000 for all beacon rings in the cache cloud and the cycle length of sub-range determination is set to 1 hour. We use the coefficient of variation of the loads on the beacon points to quantify load balancing. Coefficient of variation is defined

¹The 2000 Sydney Olympic Games web site

as the ratio of the standard deviation of the load distribution to the mean load. The lower the coefficient of variation is, the better is the load balancing.

First, we compare the load balancing accomplished by the static and the dynamic hashing schemes in a cache cloud with 10 caches. For the dynamic hashing scheme the cache cloud is configured to contain 5 beacon rings, each with 2 beacon points. The bar-graphs in Figure 3 and Figure 4 show the load distribution among the beacon points for the static and the dynamic hashing schemes on the Zipf-0.9 data set and the Sydney dataset respectively. On the X-axes are the beacon points in decreasing order of their loads, and on the Y-axes are the loads in terms of the number of document updates and document lookups being handled by the beacon points per unit time. The dashed-lines in the two graphs indicate the mean value of the loads on the beacon points. The Zipf-0.9 dataset induces a high degree of load imbalance in the cache cloud with static hashing. In this case, the load on the most heavily loaded beacon point is 1.9 times the mean load of the cache cloud. In the dynamic hashing scheme this ratio decreases to 1.2, thus providing 37% improvement over static hashing. The dynamic hashing scheme also provides a 63% improvement on the coefficient of variation when compared with static hashing. On the Sydney data set, the dynamic hashing scheme improves the ratio of the heaviest load to the mean load by around 20%, and the coefficient of variation by 63%. For this dataset, the ratio of heaviest load to mean load for the dynamic hashing scheme is just 1.06, thus showing that this scheme achieves very good load balancing. The better load balancing achieved by the dynamic hashing scheme is a result of the dynamic sub-range determination process, which takes into account the current load on the beacon points, while allocating the sub-ranges for the next cycle.

The second experiment (Figure 5) studies the effect of the size of the beacon rings on the load balancing using the Sydney dataset. We evaluate the dynamic hashing scheme

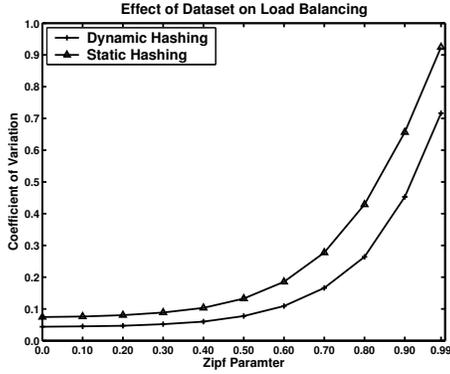


Figure 6. Impact of Zipf parameter on Load Balancing

on cache clouds consisting of 10, 20 and 50 caches. For each cache cloud we consider three configurations in which each beacon ring contains 2, 5 and 10 beacon points. The dynamic hashing scheme with 2 beacon points per ring provides significantly better load balancing in comparison to static hashing. When the size of the beacon rings is further increased we observe an incremental improvement in the load balancing achieved by the dynamic hashing scheme. This observation, namely bigger beacon rings yielding better load balancing, can be explained as follows: The beacon point sub-range determination process tries to balance the load only among the beacon points within each beacon ring. Larger beacon rings result in the load being balanced among larger numbers of beacon points, and hence provide better load balancing.

In the third experiment (Figure 6), we study the impact of the dataset characteristics on the static and the dynamic hashing schemes. For this experiment we consider several datasets all of which follow the Zipf distribution with parameters ranging from 0.0 to 0.99. The skewness of the load increases with increasing value of the Zipf parameter. At low Zipf values both schemes yield low coefficient of variation values. As the load-skewness increases the coefficient of variation values also increase for both schemes. However, the increase is more rapid for the static hashing scheme. At a Zipf parameter value of 0.9, the coefficient of variation for the static hashing scheme is around 48% more than that of the dynamic hashing scheme.

4.2. Evaluating the Utility-based Scheme

We report a set of experiments to show the evaluation of our utility-based document placement scheme. The first experiment considers a cache cloud comprised of 10 caches. On this cache cloud we simulate ad hoc, beacon point and utility-based placement policies. The caches in this exper-

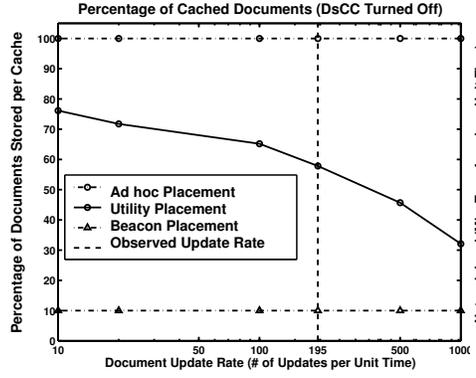


Figure 7. Percentage of Documents Stored (DsCC Turned Off)

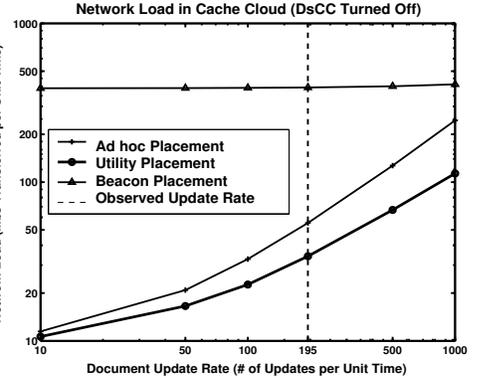


Figure 8. Network Load Under Different Placement Schemes (DsCC Turned Off)

iment are assumed to have an unlimited amount of disk-space. Therefore the disk-space component of the utility function is turned off by setting W_{DsCC} to 0. The weights of availability, consistency maintenance, and access frequency components are all set to 0.33. The $UtilThreshold$ values for all the caches are set to 0.5. In this experiment the access rates at caches are fixed, whereas we vary the document update rate to study the effect of the three document placement policies. We have experimented both with the Sydney dataset and the Zipf-0.9 synthetic dataset. Due to space constraints we restrict our discussion only to the results obtained on the Sydney dataset.

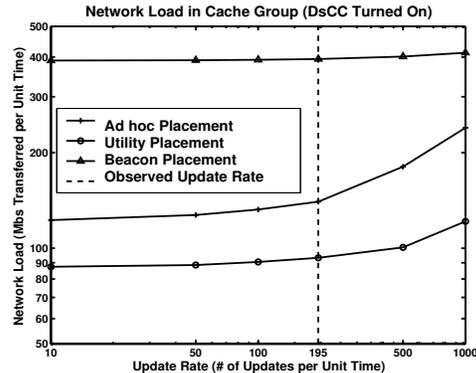


Figure 9. Network Load Under Different Placement Schemes (DsCC Turned On)

The graph in Figure 7 shows the percentage of the total documents in the trace that are stored at each cache in the cache cloud at various document update rates. The X-axis represents the document update rate in number of updates per minute on the log scale, and the Y-axis represents the percentage of documents cached. The vertical broken line indicates the observed document update rate. As the ad hoc policy places each document at every cache which receives

a request, almost all documents are stored at all caches. In contrast the beacon point placement stores each document only at its beacon point. Hence, each cache stores around 10% of the total documents. The percentage of documents stored per cache in the utility-based scheme varies with the update rate. When the update rates are low, a large percentage of documents are stored at each cache, owing to the small consistency maintenance cost. As the update rate increases, the *CMC* (consistency maintenance component) values of all the documents decrease, leading to a decrease in the percentage of documents stored at each cache. This shows the sensitivity of the utility-based document placement to the costs of handling document updates within the cache cloud.

One may ask why is it important for the placement scheme to be sensitive to the costs of handling document updates. To answer this question we plot the total network traffic in the clouds generated by the three document placement policies in the Figure 8. The results indicate that the utility-based document placement creates the least network traffic at all update-rates. The improvement provided by the utility-based placement scheme over the ad hoc placement scheme increases with increasing update rate. This is because while the number of replicas present in the cache cloud essentially remains a constant in the ad hoc placement scheme, the utility-based scheme creates fewer replicas at higher update rates, thereby reducing the consistency maintenance costs significantly. With the beacon point policy, the network traffic is very high at all update rates, as in this scheme only one copy of each document is stored per cache cloud irrespective of its request rate.

In the next experiment (Figure 9), we study the performance of the three document placement policies when the disk-spaces available at the edge caches are limited. In these experiments the disk-space at each cache is set to 50% of the sum of sizes of all documents in the trace. We use the least recently used (LRU) policy for document replacement. As the disk-space is a limiting factor in this series of experiments, we turn on the disk-space component of the utility function. The weights of all the utility function components are set to 0.25. Figure 9 indicates the total network traffic generated by the three document placement policies at various update rates. As in the previous experiment, the utility-based document placement places the least load on the network. However, the results in this experiment differ from the previous experiment considerably. The percentage improvement in the network load provided by the utility scheme over the ad hoc scheme is higher in the limited disk-space case at low document update rates. For example, the improvement is 28% when the document update rate is 10 updates per unit time for the limited disk-space experiment, whereas it is around 8% for the unlimited disk-space case. However, the percentage improvement in the

unlimited disk-space case grows much faster in the limited disk-space scenario. These observations are the manifestations of the different roles the utility placement scheme is playing at different update rates in the limited disk-space scenario. At low document update rates the utility scheme assumes the predominant role of reducing disk-space contention at individual caches. Whereas at higher update rates its predominant effect is to reduce consistency maintenance cost.

In the above experiments we have set the weights of the components by analyzing the scenario at hand. Each component of the utility function captures a different aspect of the benefit-to-cost ratio of placing a document at a particular cache. In our experiments we turn on a component if the benefit-to-cost aspect represented by it is likely to affect the performance of the cache cloud. Otherwise the component is turned off. In each of the scenarios, if w components are turned on, then we set the weight of each turned on component to $\frac{1}{w}$. We strongly believe that more sophisticated approaches to setting the weight values can further improve the performance of the utility scheme. One such approach would be to continuously monitor various system parameters and use a feedback mechanism to adjust the weight parameters as needed. Studying this and other approaches to setting weight values is a part of our ongoing work.

5. Related Work

Caching dynamic web content on the edge of the Internet has received considerable attention from the research community in recent years. Motivated by the idea of moving data and application closer to users, researchers have proposed several variants of edge caching depending upon how much of the application is offloaded to the edge caches [1, 2, 4]. Yuan et al. [17] present an in-depth evaluation of these variants studying the pros and cons of each approach. However, very few of the current edge caching techniques promote cooperation among the individual edge caches.

Cooperation among caches was first studied in the context of client-side proxy caches [5, 14, 15]. Most of these schemes were designed to cache static web pages, and assumed the Time-to-Live-based mechanism for maintaining their consistency. Our work provides enhanced support for caching dynamic data over these previous works such as stronger consistency mechanisms and consideration of object update costs.

Ninan et al. [8] describe *cooperative leases* - a lease-based mechanism for maintaining document consistency among a set of caches. They statically hash each document to a cache, which is assigned the responsibility of maintaining its consistency. Shah et al. [13] present a dynamic data disseminating system among cooperative repositories, in

which a dissemination tree is constructed for each data item based on the coherency requirements of the repositories. The server circulates the updates to the data item through this tree. The focus of both these works is on the problem of consistency maintenance of documents among a set of caches. In contrast, our cache cloud architecture systematically addresses various aspects of cooperation such as collaborative miss handling, cooperative consistency management, efficient document lookups, and cost-sensitive document placements, aiming at understanding and enhancing the power of cache cooperation on the performance of the cache clouds in a large-scale edge cache grid.

In addition to the above, researchers have studied various problems in the general area of caching dynamic web content including caching at different granularities for improving performance, minimizing the overheads of consistency maintenance, and performance analysis and comparison of various caching approaches [1, 6, 7, 16]. Researchers have also proposed various document replacement schemes for web caches [3, 9]. While a replacement algorithm decides which documents have to be evicted from a cache when its disk-space becomes full, the document placement scheme discussed in this paper aims at strategically placing documents in a cache cloud in order to improve its performance.

6. Conclusion

We have studied the challenges of designing a cooperative edge network for caching dynamic web content, and proposed the cache clouds as a framework for cooperation in large-scale edge cache networks. This paper presents the architectural design of a cache cloud, which includes dynamic hashing-based mechanisms for document lookups and updates. These mechanisms are efficient and involve minimal communication overhead. Further, the load due to document lookups and updates is dynamically distributed among all the caches in a cache cloud. Our dynamic hashing scheme is adaptive to varying document update and document request patterns. We have also presented a utility-based scheme for placing documents within each cache cloud so that the system resources are optimally utilized, and the client latency is minimized. Our experiments indicate that the techniques proposed in this paper are very effective in improving the performance of cooperative edge cache networks.

Acknowledgements

This work is partially supported by NSF CNS, NSF ITR, a DoE SciDAC grant, an IBM SUR grant, an IBM faculty award, and an HP equipment grant.

References

- [1] Edge Side Includes - Standard Specification. <http://www.esi.org>.
- [2] IBM WebSphere Edge Server. <http://www-3.ibm.com/software/webservers/edgeserver/>.
- [3] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *USENIX Internet Technologies Symposium*, 1997.
- [4] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar. Application Specific Data Replication for Edge Services. In *WWW-2003*.
- [5] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web Caching with Consistent Hashing. In *WWW-8*, 1997.
- [6] W.-S. Li, W.-P. Hsiung, D. V. Kalshnikov, R. Sion, O. Po, D. Agrawal, and K. S. Candan. Issues and Evaluations of Caching Solutions for Web Application Acceleration. In *VLDB-2002*.
- [7] D. A. Menasce. Scaling Web Sites Through Caching. *IEEE-Internet Computing*, July/August 2003.
- [8] A. Ninan, P. Kulkarni, P. Shenoy, K. Ramamritham, and R. Tewari. Scalable Consistency Maintenance in Content Distribution Networks Using Cooperative Leases. *IEEE-TKDE*, July 2003.
- [9] S. Podlipnig and L. Boszormenyi. A Survey of Web Cache Replacement Strategies. *ACM Computing Surveys*, December 2003.
- [10] L. Ramaswamy and L. Liu. An Expiration Age-Based Document Placement Scheme for Cooperative Web Caching. *IEEE-TKDE*, May 2004.
- [11] L. Ramaswamy, L. Liu, and A. Iyengar. Cooperative EC Grid: Caching Dynamic Documents Using Cache Clouds. Technical report, CERCS - Georgia Tech, 2005.
- [12] L. Ramaswamy, L. Liu, and J. Zhang. Constructing Cooperative Edge Cache Groups Using Selective Landmarks and Node Clustering. In preparation.
- [13] S. Shah, K. Ramamritham, and P. Shenoy. Resilient and Coherence Preserving Dissemination of Dynamic Data Using Cooperating Peers. *IEEE-TKDE*, July 2004.
- [14] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Beyond Hierarchies: Design Considerations for Distributed Caching on the Internet. In *ICDCS-1999*.
- [15] A. Wolman, G. M. Voelkar, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. In *SOSP 1999*.
- [16] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar. Engineering Web Cache Consistency. *ACM-TOIT*, August 2002.
- [17] C. Yuan, Y. Chen, and Z. Zhang. Evaluation of Edge Caching/Offloading for Dynamic Content Delivery. In *WWW-2003*.