Dynamic Organization Schemes for Cooperative Proxy Caching

Spiridon Bakiras
Dept. of Electrical & Electronic Engineering
The University of Hong Kong
Pokfulam Road, Hong Kong
sbakiras@eee.hku.hk

Thanasis Loukopoulos
Dept. of Computer Science
Hong Kong University of Science and Technology
Clearwater Bay, Hong Kong
luke@cs.ust.hk

Ishfaq Ahmad
Dept. of Computer Science & Engineering
University of Texas at Arlington
Texas, USA
iahmad@cs.ust.hk

Abstract

In a generic cooperative caching architecture, web proxies form a mesh network. When a proxy cannot satisfy a request, it forwards the request to the other nodes of the mesh. Since a local cache cannot fulfill the majority of the arriving requests (typical values of the local hit ratio are about 30-50%), the volume of queries diverted to neighboring nodes can substantially grow and may consume considerable amount of system resources. A proxy does not need to cooperate with every node of the mesh due to the following reasons: (i) the traffic characteristics may be highly diverse; (ii) the contents of some nodes may extensively overlap; (iii) the inter-node distance might be too large. Furthermore, organizing N proxies in a mesh topology introduces scalability problems, since the number of queries is of the order of N^2. Therefore, restricting the number of neighbors for each proxy to $k < N - 1$ will likely lead to a balanced trade-off between query overhead and hit ratio, provided cooperation is done among useful neighbors. For a number of reasons static the selection of useful neighbors is not efficient. An obvious reason is that web access patterns change dynamically. Furthermore, availability of proxies is not always globally known. This paper proposes a set of algorithms that enable proxies to independently explore the network and choose the k most beneficial (according to local criteria) neighbors in a dynamic fashion. The simulation experiments illustrate that the proposed dynamic neighbor reconfiguration schemes significantly reduce the overhead incurred by the mesh topology while yielding higher hit ratios compared to the static approach.

1. Introduction

A caching hierarchy is defined through parent-child and sibling relations among the participating proxies. In the basic scheme introduced by the *Harvest* system [3], client requests arrive at the lowest level and cache misses are propagated to the upper levels until the root node is reached. If the root is unable to satisfy a request, the web server is contacted. Although hierarchies usually yield high hit ratios for the intermediate and topmost nodes, they possess two main drawbacks: (i) the benefit (in terms of response time) for end-users is not always possible (especially if the topmost cache lies behind a slow link), (ii) the upper level nodes may become overloaded. For these reasons, the number of levels is commonly restricted to three, i.e., institutional, regional, and national.

Cooperative caching (also referred to as distributed caching) can be viewed as a step forward in an attempt to overcome the deficiencies of hierarchies. In a pure distributed scheme, institutional proxies cooperatively satisfy user requests without the presence of regional and national caches being necessary. Hybrids between distributed and hierarchical caching have also been proposed (e.g., in [13] where the hierarchy is only used for propagating metadata information concerning content locations). *Squid* [16], the successor of Harvest, provides enough versatility in the cache configuration to account for hybrid architectures, and includes a dedicated protocol for inter-proxy querying (*internet cache protocol* ICP [17]). In the basic scheme, when a miss occurs, the cache broadcasts the query to its neighbors and retrieves the page from the first one that replies positively. If none of the neighbors has a cached copy, the request is forwarded either to the parent cache or to the web server. An alternative is provided by *cache digests* [12], where proxies exchange periodically their directory information in the form of compressed hash arrays. In this way a proxy checks the digests of its neighbors (stored locally) and forwards the request only to the neighbors that have cached the page. If possible, a proxy should only make neighbors other nearby (in terms of network latency) proxies with similar access patterns.

An optimal grouping of proxies to neighborhoods is a difficult task for several reasons: (i) global information on cache contents is not always available, (ii) access patterns may change and as a

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)

result the neighborhood graphs may need to be updated continuously, and (iii) each proxy should independently take decisions about its neighbors in order to maximize its hit ratio. In this paper we propose a set of distributed algorithms that dynamically group proxies into neighborhoods. The algorithms estimate the potential for content sharing, based solely on the information available at each proxy. Since it is not feasible to have the knowledge about all the participating proxies, an *exploration* step discovers caches with similar access patterns.

The problem can be thought of as *second level caching*, where the cached objects are the best neighbors of each proxy v and the capacity k of the cache represents the maximum allowable number of neighbors. When v determines that a proxy v_i (not currently in v's neighborhood) could provide a large number of hits, it adds v_i to the list of its best neighbors by evicting the least beneficial existing neighbor. We can distinguish two cases depending on whether a proxy satisfies queries originating from non-neighbors or not. In this paper we assume that although a proxy sends queries only to neighboring nodes, it satisfies requests originating from every node. This results in asymmetric neighbor relations whereby proxy v can have v_i as a neighbor without the opposite being true.

The rest of this paper is organized as follows. Section 2 presents the related work on distributed caching. Section 3 describes the caching algorithms. Section 4 illustrates the experimental results, and Section 5 concludes the paper with some final observation and a discussion about the future research directions.

2. Related Work

Several papers have focused on quantifying the potential gains of cooperative caching. In [8] the authors analyze traces from Bell Labs reporting that the performance improvement for an ideal distributed scheme is significant but varies a lot depending on the day of the traces. They also observe that the cooperation of only a small set of proxies has a significant impact on performance. The study in [19] analyzes traces from a large number (tens of thousands) of end-clients, and provides an analytical model to predict the system's behavior. The authors conclude that the largest benefit from cooperative caching is expected when the number of clients is relatively small.

In [4], the authors estimate the average response time in hierarchical and distributed caching architectures. The authors conclude that the speedup from distributed caching is higher than that of hierarchical caching. In [11] the authors break the response time into connection and transmission time. They suggest that cooperative caching accounts for larger connection times, but smaller transmission delays, since lower level links are usually not congested. They also argue that a hybrid architecture comprising multiple caching meshes organized hierarchically, achieves the best performance. A hybrid architecture is also proposed by [20] with proxies participating in a hierarchy of overlapping multicast groups.

Another important issue in distributed caching is how to locate a specific page. [13] proposes the use of hints that are simple records of the form <object_id, closest_neighbor>, cacheable at each proxy. A static hierarchy is used for hint propagation. The approach followed in [9] is based on keeping centralized information about the contents of all proxies. *Cachemesh* [15] employs URL routing tables, maintained in a way similar to the IP routing tables, for redirecting requests to appropriate caches. The *cache array routing protocol* (CARP) [14] splits the URL space using hash functions and allocates different portions to each proxy, taking into account their processing capacity.

Currently, the most popular system for distributed caching is *Squid* [16] according to which proxies are manually configured in fixed neighborhoods. The current version of Squid implements *cache digests* [12, 6], which are compact representations of cache contents based on Bloom filters [2]. Figure 1 illustrates a simple example, when the cache digest is an array of m bits which are initially set to 0. When a new page W is cached, its MD5 signature [10] is hashed using n hashing functions $h_1, \ldots, h_n$ (each with range $\{1,\ldots,m\}$), and the bits at positions $h_1(W), \ldots, h_n(W)$ are set to 1. In addition, there exists a second array of counters. The insertion of W will increase the counters at positions $h_1(W), \ldots, h_n(W)$. When W is evicted from the cache, the value of each counter in these positions is decreased by one. If some counter becomes 0, the corresponding bit in the cache digest is also set to 0.

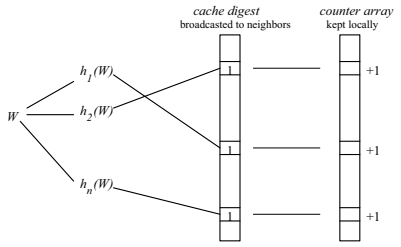

Figure 1. Example of cache digest.

The counter array is only kept at the corresponding proxy, whereas the cache digest (bit array) is sent to all the neighbors. When a local miss occurs at proxy v, v redirects the request to a neighbor proxy v_i that has the required page (according to the digest of v_i stored locally at v), thus avoiding the extra latency introduced by ICP. In order for v to locate a page W in v_i's digest, it only needs to check the bits at positions $h_1(W), \ldots, h_n(W)$: (i) If all bits are 1, v conjectures that W is in v_i's cache, although there is a probability of a *false positive*. False positives occur when multiple pages set the same bits. The trade-off between space overhead and percentage of false positives is tuned by choosing appropriate values for parameters n and m. (ii) If any of the bits at positions $h_1(W), \ldots, h_n(W)$ is 0, v concludes that W is not in v_i's cache.

Since v_i's digest is not necessarily up-to-date, a *false miss* may occur if v_i has cached W after it sent its digest to v. Similarly, an outdated digest may also cause *false hits*, if a page that appears in the remote digest has meanwhile been evicted from the local cache. As shown in the simulation results of [6], cache digests achieve significant bandwidth savings compared to ICP querying. Furthermore, the authors observe that even with infrequent summary updates the performance loss (due to false misses or false hits) is marginal. In particular, [6] conclude that updated digests should be propagated to neighbors only after 1%-10% of the cache contents change. [4] suggest that updates should happen on a daily

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)

basis when the network traffic is low (e.g., during the night). Due to the apparent benefits of summaries, we include them in our design.

In general, existing work seems to agree on the benefits from cooperative caching. Furthermore, a number of papers [11, 20] motivate (implicitly) the need for algorithms to perform dynamic neighbor selection.

3. Second Level Caching

In this section we describe the proposed algorithms for neighbor selection. The goal is to allow each node to dynamically update its neighborhood list in order to maximize the number of hits from other proxies. Neighboring nodes exchange their cache digests so that when a miss occurs, the missing page may be obtained directly. Since the most useful neighbors vary continuously with the access patterns, the system should include a mechanism to replace the least beneficial neighbors with new ones that may provide more hits.

Subsequently we propose two techniques: the first one is an adaptation of the *least recently used* (LRU) strategy, and the second one extends the *least frequently used* (LFU) paradigm by taking into account the special characteristics of the problem.

3.1. LRU

Assume that a node v has k neighbors $v_1,\ldots,v_k$. When a miss occurs, v sends the request to one of $v_1,\ldots,v_k$ that has cached the page (according to the cache digests). If multiple neighbors can serve the request, the one (denoted by v_i) that is closer (in terms of latency) is chosen. When v_i returns the page it becomes the most recent neighbor. In case of a false hit/positive the process is repeated for all neighbors whose digest contains the page.

If none of the neighbors can provide the page, v sends the request to the server and at the same time initiates an *exploration* process. The goal of the process is to identify other nearby proxies that have the page since such proxies may be beneficial for subsequent requests. This premise is intuitive since users usually navigate by following the links contained in a web page and therefore it is likely that once a neighbor can provide a page it can also satisfy subsequent requests originating from browsing its links. The pseudo-code for the algorithm is illustrated in Figure 2. The pseudo-code distinguishes two cases: *serve_request,* which corresponds to the situation that a page W is requested by a client, and *process_query* where a query is received from another proxy. This query can be (i) a request for a page, or (ii) an exploration query.

The exploration process needs further elaboration. When node v receives a client request for a page W not cached locally or in its neighbors, it sends an exploration query to all the neighbors v_i with probability $a \leq 1$. A node v_i that receives an exploration query first checks its own cache and if it contains W, it replies directly to v. Otherwise, v_i forwards the query to all neighbors whose digest contains W, and to the remaining ones with probability a.

The forwarding process continues until the maximum number of hops h is reached. Like parameter a, the value of h adjusts the

```
Algorithm Serve_Request (Page: W)
1. if W is cached locally then serve request and return
2. l=list of neighbors whose digests contain W sorted by network latency
3. while l is not empty
4.    remove the closest neighbor v_i from l
5.    query (get_page, W, v, v_i)
6.    if hit then send W to client, update recency of v_i and return
7. end //while
8. for each neighbor v_i query(explore, W, v, v_i) with prob. a
9. l_2=collect-exploration-result(W,time-out) // l_2 is a list of proxies that have cached W
10.let v_new be the closest node in l_2
11.if latency(v_new)<latency(server) then
12.   get cache digest of v_new
13.   v_new becomes the most recent neighbor (possibly by evicting the least recent one)
end Serve_Request

Algorithm Process_Query (String: op_code, Page: W, Node: v (originator), Node: v_i (current node))
1. CASE (op_code)
2. get_page:
3.    if W in cache then send W to v and signal hit = true
4.    else (W not in cache) then signal hit = false
5. explore:
6. if W in cache then notify v and return
7. if limit of hops has been reached then return
8. for each neighbor v_j of v_i
9.    if v_i's digest contains W then query(explore, W, v, v_j)
10.   else // v_i's digest does not contain W
11.         query (explore, W, v, v_k) with prob. a
end Process_Query
```

Figure 2. Pseudo-code for LRU.

trade-off between the extent of exploration and traffic overhead. The original node v accumulates responses from proxies caching W, until a time-out period is exceeded. Then it makes as its most recent neighbor, the proxy v_{new} that contains W and has the lowest network latency. If the list of neighbors is full, the least recent neighbor is evicted. Note that due to false misses, v_{new} may already be a neighbor of v, in which case v_{new}'s digest is updated (no neighbor is evicted). If no response arrives before the time-out interval the recency status remains unchanged. There is no forwarding of actual page requests since they are satisfied either at: (i) the proxy v where they arrived, (ii) a first degree neighbor of v, (iii) the web server. If v determines that no neighbor proxy contains W, it sends directly the request to the appropriate server without waiting for the results of the exploration process.

A final remark concerns with some implementation issues. For querying, the *http get* method provides the functionality of the *get_page* op_code. We can implement *explore* in ICP, by adding a new op_code in one of the unused slots. The payload of the new op_code should include apart from the URL, the number of hops that the query has traveled so far and the id of the proxy where it originated, so that the receiver can answer directly to it. The algorithm also requires the proximity (latency) computation between proxies, in order to choose the fastest neighbor (in case of multiple available choices). For existing neighbors, the proximity can be calculated by taking a weighted average of the past experienced latency [7]. For proxies accessed (by exploration) for the first time, the latency can be estimated by measuring the RTT using the *ping* utility. In our simulations we assume that knowledge of the closest neighbor is always available.

3.2. LFU

A potential problem with LRU is that it may impose network overhead due to the frequent reconfiguration of the neighbors and exchanges of digests. In addition, LRU may quickly replace some "good" neighbors that do not provide hits for a short period of time, although they are beneficial in the long run. The second strat-

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)

egy, LFU, aims at overcoming these problems by collecting statistics and performing reconfiguration if certain conditions hold.

The pseudo-code for LFU is very similar to that of Figure 2 and thus is not included here. The algorithm keeps a hit counter at each proxy, which maintains the *positive responses* (page retrievals or exploration hits) from other proxies. Each page retrieval or exploration hit received by node v increases at most one hit counter (of the fastest node) in v, even though multiple proxies may respond. Thus, closer neighbors are favored and the network latency is implicitly included in the number of hits. Moreover, since a proxy needs to obtain only one copy of a page (ideally the closest available) the existence of other copies yields no local benefit.

When reconfiguration is performed, the new neighborhood of a proxy v is defined as the set of k nodes that provided the largest number of positive responses to v. Some of these nodes may be already in the neighborhood of v, and no special action is required. For the rest of the nodes, v requests and maintains locally their cache digests by evicting the digests of aborted neighbors. The exchange of digests implies that digests are usually up to date. In some situations, however, it is possible that a proxy v_i will remain a neighbor of v for a long period of time. Special care must be taken in order to update its digest because the methods available for Squid are now inapplicable. Recall that in Squid all the neighbors of v_i obtain (simultaneously) the same version of v_i's digest. Thus, v_i can decide locally when the digest is outdated and broadcast the new version to all its neighbors. On the other hand, in our methods this decision is taken asynchronously at each receiving node (since nodes have different versions of the digest depending on when they configured v_i as a neighbor); a proxy will ask for a new digest from a neighbor, if the percentage of false misses from this neighbor (discovered through exploration) exceeds a threshold.

An interesting issue regards the appropriate conditions for reconfiguration. At one extreme, if reconfiguration occurs after every positive response, the strategy will transform to LRU. At the other extreme, if reconfiguration is very infrequent, LFU will behave like a static scheme. In order for the statistics to be meaningful, reconfiguration is initiated after a number l of positive responses has been collected. Local hits, or queries that do not yield any exploration results, do not provide any information about the contents of other proxies; only page hits from neighbors or exploration hits are useful for the computation of neighbors. When l (good values of l are determined experimentally) is exceeded, a non-neighbor v_j, will replace a neighbor proxy v_i, if the value of the counter for v_j is above $r\%$ of the corresponding value for v_i. If, for instance, $r = 100$, v_j will replace v_i, if it provides more positive answers. In practice, since neighbors are favored because they are requested first, the value of r should be lower.

Notice that since we aim at maximizing the hit ratio from other proxies, we only take into account the number of hits and not the page sizes. Intentionally we tried to keep the neighbor replacement policy as simple as possible by adopting the well-known LRU and LFU paradigms, because our goal is to demonstrate the advantages of the neighbor reconfiguration concept in general, and not of the individual policies. We also experimented with alternative caching strategies that consider additional parameters such as detailed benefit and latency measures, but found that the additional gains (if any) are negligible.

4. Experiments

We evaluate the proposed algorithms as follows. Section 4.1 describes the traces used in all experiments. Section 4.2 compares LRU with static Squid variants, in order to confirm the viability of our concept. Section 4.3 measures the performance of LFU against LRU. Section 4.4 provides some insight on the behavior of different strategies. Finally, Section 4.5 summarizes the results.

Although the traces used originated from real proxies, we did not have any information about the network topology. Therefore, we assume a fully connected network where the inter-proxy (one-way) latency follows a Gaussian distribution with mean 70ms and standard deviation 20ms. Values below 10ms and greater than 130ms were cut off. The (one-way) latency between proxies and web servers is fixed to 1 second in order to simulate the situation where fetching pages from proxies is much faster than doing so from the servers. This assumption is valid, since (i) requests for 'local' servers do not yield any ICP queries, and (ii) the ICP_OP_SECHO opcode may be used to identify whether the server is closer than the neighbors. As a measure of performance we employ the number of neighbor hits, i.e., local misses served by the (1^{st} degree) neighbors, because it is less sensitive than other measures (e.g., average response time) to the (artificial) network latencies.

For the implementation of Squid simulations we followed the guidelines of [12]. Each proxy broadcasts a new digest version to his neighbors whenever the cached contents change by 1%. In all simulations, the cache for each proxy is equal to 10% of the total size of the locally requested objects. The (local) page replacement strategy for all proxies is LRU.

4.1. Datasets

Real Data: We collected traces from the 10 available proxies of the *National Laboratory for Applied Network Research* (NLANR [1]). These proxies are based on the Squid software and are located throughout the United States. Their aim is to provide hierarchical caching services to organizations and individuals. The traces depict all requests between 15/11/01 and 18/11/01. We decided to exclude the "sj" proxy from our experiments since it accounts for very light and dissimilar workload compared to the rest. Moreover, only HTTP requests with the GET method are considered, since only this type of requests may trigger an ICP query. URLs containing "cgi-bin", ".asp" and "?" substrings are excluded as un-cacheable objects. The same is true for requests with a result code TCP_CLIENT_REFRESH_MISS, since they account for a *no-cache pragma*, control command. Finally, we deleted requests for partial content (status 206) and requests that resulted in 0 byte transfers. This methodology has been suggested in previous related work [5]. The statistics for the remaining pages are summarized in Table 1.

Although traces from institutional proxies could be more appropriate for our study, we were unable to collect a sufficient number of them. Nevertheless, we believe that recreating the behavior of the topmost proxies in the NLANR hierarchy is still sufficient for illustrating the main merits of the proposed strategies and providing useful insight. It is reasonable to expect similar or higher

	startap	bo2	bo1	pa	sv	sd	uc	pb	rtp
Tot.Size (GB)	1.63	2.16	3.03	3.06	6.93	6.97	7.76	20.38	33.53
Reqs(Millions)	0.46	0.35	0.42	0.76	2.35	1.29	0.79	3.09	6.29
Dist. Pages (Mil.)	0.22	0.24	0.28	0.35	0.80	0.73	0.51	1.58	2.98
Avg. Pg. Size (KB)	7	9	10	9	9	10	15	13	11

Table 1. Statistics of NLANR traces.

performance gains for institutional proxies where the sharing potential is higher.

Synthetic Data: In order to test how the parameters of the algorithms and the network size affect performance, we created two synthetic datasets representing requests for 45 proxies. In the first set (SYNTH I), each of the 9 initial NLANR traces was split into 5 equal parts/proxies. Every request was sequentially assigned to one of the 5 proxies in a round-robin way (a similar method was followed in [18]). Thus, the proportional size differences of the initial NLANR traces were also preserved in SYNTH I. The second dataset (SYNTH II) was created again using the round-robin method, but the larger proxies were split in more parts in order to minimize the size differences of the resulting 45 proxies. Experiments with SYNTH I aim at evaluating performance and scalability in an "expanded" NLANR hierarchy. SYNTH II approximates better the case of institutional level proxies where cache size is not expected to vary significantly. Whenever the results are similar, we only present SYNTH I. We were unable to follow the most intuitive approach of splitting the requests of the initial trace according to the origin IP since the anonymizer used by Squid (i.e., the process that modifies the IP before updating the log file) does not produce consistent IPs across multiple days.

4.2. LRU vs. static methods

In this section we compare LRU against static alternatives. We start with NLANR (9 proxies) and continue with the synthetic datasets. The parameters of LRU are set as follows: probability to send an exploration query to a neighbor $a = 0.5$, maximum number of hops for exploration $h = 2$, number of (outgoing) neighbors $k = 3$. We measure performance in terms of neighbor hits against two Squid configurations obtained as follows: (i) we executed 30 experiments using random static configurations where each proxy has 3 outgoing neighbors and an arbitrary number of incoming ones; (ii) for each execution we counted the total number of neighbor hits; (iii) the configuration that provided the mean of the total hits (i.e., the 15^{th} best configuration) is *Squid_average*; (iv) the best (of 30) configuration is *Squid_best*. We also include the maximum number of neighbor hits that can be obtained if all proxies are connected (*All_to_all*). Figure 3 shows the sum of hits of all proxies per hour (traces of 4 days – 96 hours).

LRU achieves a significant increase in the neighbor hit ratio compared to both static schemes with the same number of neighbors. This is expected since it dynamically modifies the initial configuration according to the access patterns. Its difference from the optimal hit ratio (*All_to_all*) is not large considering the limited numbers of neighbors (3) and exploration hops (2). Figure 4 illustrates the number of digests exchanged per hour. We only include *Squid_best*, because all static configurations result in almost the same frequency of exchanges. Since in the first few hours there exists a lot of exchanges until the caches get full, we only show the results after the 4^{th} hour.

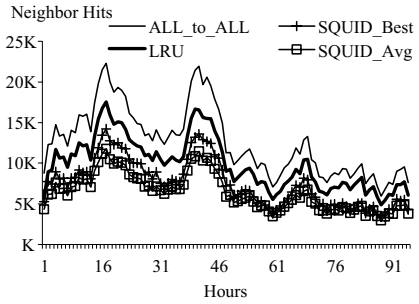

Figure 3. The number of pages obtained from neighbors.

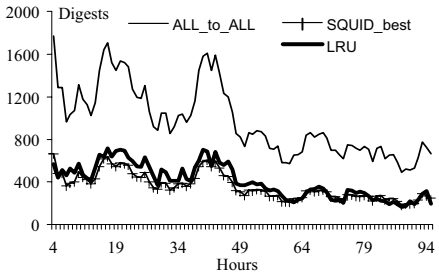

Figure 4. The number of digests exchanged.

The optimal (*All_to_all*) method is very expensive since each proxy sends its updated digest to all the other eight proxies. Rather surprisingly, the overhead of LRU is similar to that of static Squid. In LRU, a digest is sent from v_i to v_j when (i) v_i becomes a neighbor of v_j or (ii) v_j discovers a false miss in an existing neighbor v_i. In practice, the second case may be ignored since it is very infrequent. Therefore, essentially Figure 4 implies that the number of exchanged digests due to neighbor changes (in LRU) is more or less the same as the number of broadcasts in Squid (when the update threshold is 1%). We will explore this point further and study the effect of the network size in subsequent experiments with synthetic datasets.

In addition to digest exchanges, LRU (and all our methods) impose the overhead of exploration messages. Given that the size of each message is 3-4 orders of magnitude smaller than that of a digest (i.e., a few hundred of bytes as opposed to several hundred of Kbytes), the bandwidth overhead of digest transfers dominates

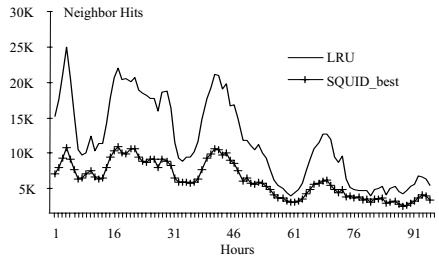

Figure 5. The number of pages obtained from neighbors (SYNTH I).

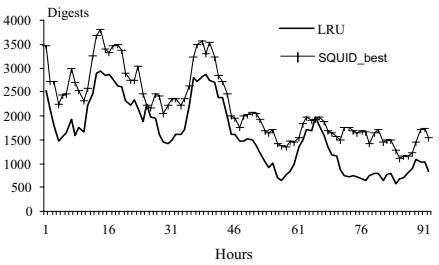

Figure 6. number of digests exchanged (SYNTH I).

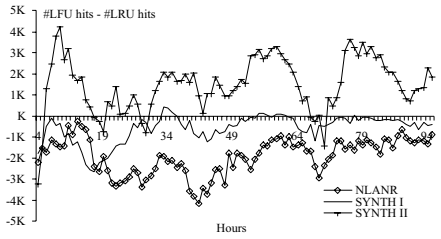

Figure 7. The benefit of LFU in terms of neighbor hits.

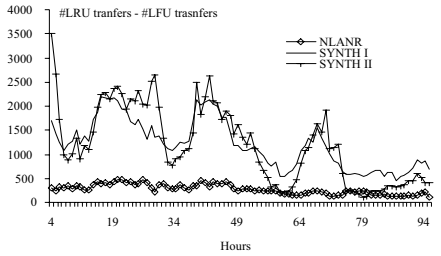

Figure 8. The benefit of LFU in terms of digest exchanges.

that of exploration messages.

Next, we use the dataset SYNTH I (45 proxies) to test the generality of the first observations. Notice that by splitting the contents of a proxy in smaller parts (i.e., the process that we followed to create the synthetic datasets) the total number of neighbor hits will increase since some local hits (i.e., at the same proxy) will now become neighbor hits. However, it is practically impossible to determine the actual number of neighbor hits since the size of the network is prohibitive for applying the *All_to_all* method. Instead, we compare LRU (using the same values for parameters a, k and h) with the best Squid alternative obtained after executing 30 random configurations. Figure 5 shows the results. The improvement of LRU in this case is impressive. The small number of neighbors with respect to the total number of proxies restricts the benefit of static schemes, which can only search in their proximity. On the other hand LRU, even with a limited number of exploration hops (2), can gradually relate nodes that are several hops apart through the intermediate proxies in their path.

Similar to Figure 4, Figure 6 compares the overhead of LRU and *Squid_best* in terms of the number of digest transfers. Since the network now contains 45 proxies (as opposed to 9 in the first experiment), the overhead of *Squid_best* is about 5 times higher. LRU is less sensitive to network size since the frequency of digest exchanges also depends on the quality of the neighbors. Notice that Squid can reduce the number of exchanges by increasing the update threshold from 1% to a higher percentage. This, however, would have a negative effect on the number of neighbor hits. Another subtle point refers to the utilization of digests. According to Squid a proxy will broadcast the new version of its digest to all its neighbors even if it is not useful to them. On the other hand, all our policies update digests on-demand; that is, new versions are only requested by neighbors that actually use them.

Finally, we tested the effect of the various parameters (a, k and h) on the performance of LRU. The results were as expected and thus are not included here. In particular, the neighbor hits and the rate of digest transfers increase with the number of neighbors (k) and the exploration probability (a). On the other hand, although the exploration messages increase exponentially with the maximum number of hops (h), the page hits and digest exchanges are not influenced considerably. This implies that if a page can be found in the network, it probably lies in the neighborhood of the requesting proxy and extensive exploration is not usually beneficial. We also replaced SYNTH I with SYNTH II and observed almost identical results to the ones in Figures 5 and 6, suggesting that the performance of LRU depends on the total number of potential neighbor hits rather than the structure or configuration of individual proxies. In summary, LRU increases significantly the number of neighbor hits, especially for large networks. An obvious improvement over LRU concerns the reduction of digest transfer. Towards this direction, we evaluate the performance of LFU.

4.3. LFU vs. LRU

Here, we compare LRU and LFU. The same parameter values are used for both methods ($a = 0.5$, $k = 3$ and $h = 2$). Furthermore, l (number of positive responses required for reorganiza-

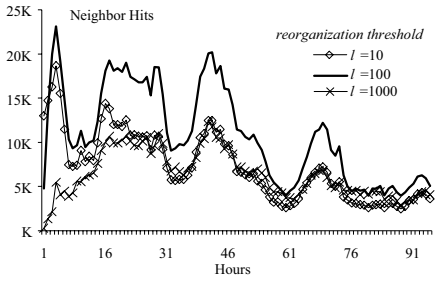

Figure 9. The number of neighbor hits for various reorganization thresholds.

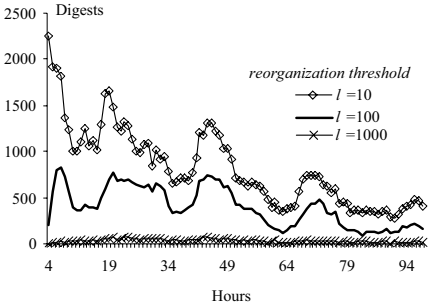

Figure 10. The number of digests exchanged for various reorganization thresholds.

tion) for LFU is set to 100, whereas r (weight factor for neighbor responses) is set to 0.5. The first experiment in Figure 7 shows the relative benefit of LFU for NLANR, SYNTH I and SYNTH II. The benefit is measured as *#LFU hits - #LRU hits* and can be positive or negative, depending on whether the number of hits increases or decreases. LRU is better for NLANR (and to a lesser extent for SYNTH I), while LFU is better for SYNTH II. Next we measure the benefit of LFU in terms of digest transfers. Figure 8 illustrates *#LRU transfers - #LFU transfers* for NLANR, SYNTH I and SYNTH II. The advantage of LFU is clear since it reduces considerably the network overhead in all cases. The difference is higher for the larger networks, indicating better scalability. A comparison with the absolute values of LRU for NLANR (Figure 4) and SYNTH I (Figure 6) suggests savings up to 70%-80%.

The effects of the common parameters (a, k and h) are similar to LRU and not included. We only investigate the impact of the reorganization threshold (for l = 10, 100 and 1000) on SYNTH I. The number of hits (Figure 9) is optimized for l =100. If l =10, LFU does not have enough statistics to select "good" neighbors, whereas if l =1000, LFU cannot follow closely the changing request patterns. The network overhead caused by digest exchanges (Figure 10) is inversely proportional to the value of l. In general, the proper tuning of l is crucial for achieving good performance, while maintaining low overhead. An optimal value of l is difficult to compute, since in addition to the traces, it depends on the proxy configuration and the values of the other LFU parameters.

In summary, LFU with appropriate parameter tuning is superior to LRU since it achieves a similar number of neighbor hits with a significantly lower overhead.

4.4. Sharing behavior

In this section, we explore the content sharing patterns imposed by the various alternatives. In particular we choose one of the proxies (*bo2*) and illustrate in Figure 11 the number of pages sent to or received from other proxies depending on the caching policy. Notice that the proxies on the x-axis are sorted according to their cache size (which is set to 10% of the total size of the locally requested objects). *Bo2* is the second smallest proxy after *startap*.

With LRU (left diagram) *bo2* only receives pages without servicing any requests. Moreover, most of these pages come from large proxies. This actually is a common pattern for all small proxies: they attach themselves to some large cache and remain there most of the time. In this case, the neighbors of *bo2* are: *bo1* (which as will see has very similar contents with *bo2*) and the four largest proxies in the network. This situation is not desirable since it may lead to over-congestion of the popular nodes.

LFU (lower left diagram) on the other hand, achieves some kind of load balancing since *bo2* exchanges pages with all proxies. The explanation is that when *bo2* joins the neighborhood of another proxy, it will remain there until the next reorganization phase. During this period it serves requests from the other proxy, thus the number of pages sent to other nodes is increased with respect to LRU.

4.5. Summary

The overall conclusion is that the dynamic neighbor reconfiguration caching strategies achieve better performance compared to static approaches in terms of both neighbor hits and traffic overhead. Specifically, LRU closely follows the changes in access patterns by frequently changing the neighborhood list. It achieves a near-optimal hit ratio with a significantly smaller number of neighbors compared to a full mesh topology (i.e., lower cost). LFU, on the other hand, changes the neighborhood list periodically, based on the collection of statistics during the reconfiguration period. The result is a considerably lower amount of overhead traffic, and savings up to 80% (compared to LRU) were observed. Furthermore, LFU performs slightly better in terms of neighbor hits, when the selection of 'good' neighbors is not very clear.

5. Conclusions

In this paper we presented algorithms that dynamically organize proxies into neighborhoods. Our solution is based on treating the problem as second level caching. Simulation results indicate that LRU and LFU achieve higher hit ratios compared to their static counterparts in all experimental datasets. Even in small network instances, where an all-to-all neighbor configuration is feasible, our methods are still useful as they achieve comparable performance at only a fraction of the bandwidth overhead. Furthermore, the second level caching formulation provides a simple framework that permits the application of previous results to this

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)

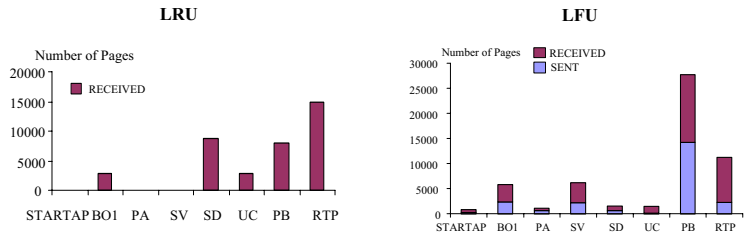

Figure 11. Sharing patterns for BO2.

problem. A straightforward extension of this work is to exploit other caching strategies that integrate latency, recency, frequency of requests, etc. Such techniques could be used to minimize measures like average response time or byte hit ratio.

It is obvious that the cooperation of two proxies often involves administrative issues. This allows us to apply our algorithms on top of the manually predefined list of legitimate neighbor candidates. Defining the optimal number of neighbors k, is not straightforward since it involves a trade-off between bandwidth consumption and hit ratio that only an administrative entity can decide. In this paper we assumed that k is given and remains fixed. In a parallel work to this one we investigate strategies where k varies depending on the neighbor quality. Finally, our ongoing research includes studying the case where a proxy accepts queries only from its neighbors. This leads to symmetric neighbor relations whereby a proxy v makes v_i a neighbor only if v_i reciprocates.

Acknowledgments

The authors would like to thank Dimitris Papadias for his contribution in various parts of this paper. Spiridon Bakiras is supported in part by the Areas of Excellence Scheme established under the University Grants Committee of the Hong Kong Special Administrative Region, China (Project No. AoE/E-01/99).

References

[1] National Lab of Applied Network Research, IRCache project. Sanitized access logs, available at: http://www.ircache.net/.

[2] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[3] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical internet object cache. In *Proceedings USENIX Annual Technical Conference*, pages 153–164, 1996.

[4] S. G. Dykes and K. A. Robbins. A viability analysis of cooperative proxy caching. In *Proceedings IEEE INFOCOM*, pages 1205–1214, April 2001.

[5] S. G. Dykes, K. A. Robbins, and C. L. Jeffery. Uncacheable documents and cold starts in web proxy cache simulations: How two wrongs appear right. Technical Report CS-2001-01, University of Texas at San Antonio, January 2001.

[6] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[7] M. L. Gullickson, C. E. Eiccholz, A. L. Chervenak, and E. W. Zegura. Using experience to guide web server selection. *Multimedia Computing and Networking*, January 1999.

[8] P. Krishnan and B. Sugla. Utility of co-operating Web proxy caches. *Computer Networks and ISDN Systems*, 30(1-7):195–203, 1998.

[9] M. Rabinovich, J. Chase, and S. Gadde. Not all hits are created equal: cooperative proxy caching over a wide-area network. *Computer Networks and ISDN Systems*, 30(22-23):2253–2259, 1998.

[10] R. Rivest. The MD5 message-digest algorithm. *Internet RFC 1321*, April 1992.

[11] P. Rodriguez, C. Spanner, and E. W. Biersack. Web caching architectures: Hierarchical and distributed caching. In *Proceedings International Web Caching Workshop*, April 1999.

[12] A. Rousskov and D. Wessels. Cache digests. *Computer Networks and ISDN Systems*, 30(22-23):2155–2168, 1998.

[13] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Web caching architectures: Hierarchical and distributed caching. In *Proceedings International Conference on Distributed Computing Systems*, June 1999.

[14] V. Valloppillil and K. W. Ross. Cache array routing protocol v1.0. *Internet Draft*, February 1998.

[15] Z. Wang. Cachemesh: a distributed cache system for world wide web. In *Proceedings International Web Caching Workshop*, June 1997.

[16] D. Wessels. Squid internet object cache. Available at: http://www.squid-cache.org/.

[17] D. Wessels and K. Claffy. Internet cache protocol (ICP) version 2. *Internet RFC 2186*, September 1997.

[18] C. Williamson. On filter effects in web caching hierarchies. *ACM Transactions on Internet Technology*, 2(1):47–77, February 2002.

[19] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. R. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. In *Proceedings ACM Symposium on Operating Systems Principles*, pages 16–31, 1999.

[20] L. Zhang, S. Michel, K. Nguyen, A. Rosenstein, S. Floyd, and V. Jacobson. Adaptive web caching: Towards a new global caching architecture. In *Proceedings International Web Caching Workshop*, 1998.