**Self-Organizing Wide-Area Network Caches**

*Samrat Bhattacharjee (Corresponding Author)*
*Kenneth L. Calvert*
*Ellen W. Zegura**
Networking and Telecommunications Group
College of Computing
Georgia Tech
Atlanta, GA 30332-0280
{bobby,calvert,ewz}@cc.gatech.edu

**GIT–CC–97/31**

# Abstract

A substantial fraction of all network traffic today comes from applications in which clients retrieve objects from servers. The *caching* of objects in locations "close" to clients is an important technique for reducing both network traffic and response time for such applications. In this paper we consider the benefits of associating caches with switching nodes *throughout* the network, rather than in a few hand-chosen locations. We also consider the use of various *self-organizing* or *active* cache management strategies for organizing cache content. We evaluate caching techniques using both simulation and a general analytic model for network caching. Our results indicate that in-network caching can make effective use of cache space. In particular, self-organizing caching schemes yield better average round-trip latencies than more traditional approaches, while requiring much smaller caches per node.

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332–0280

*This work was supported in part by DARPA and NSF Careers Award MIP-9502669.

# 1 Introduction

A substantial fraction of all network traffic today comes from applications in which clients retrieve objects from servers (e.g. the World-Wide Web). The *caching* of objects in locations "close" to clients is an important technique for reducing both network traffic and response time for such applications. Studies have shown [5, 6] that caching can substantially improve performance.

In this paper we consider the benefits of associating caches with switching nodes *throughout* the network, rather than in a few hand-chosen locations. We also consider the use of various *self-organizing* or *active* cache management strategies. Our goal is to maximize the benefit from each cached object by placing it where it will do the most good. The idea is that each node caches those objects that most frequently "pass through" it, and objects end up cached at "hot spots" where streams of requests come together.

We develop the self-organizing algorithms in light of an "active" network [9]. In an active network, the routing nodes can execute code on behalf of users. The processing required by the self-organizing caching algorithms form a natural application for active networks (as this processing is confined to within the network, and uses knowledge that is available only within the network). In the rest of the paper, we use the terms "self-organizing" and "active" interchangeably.

We define performance in terms of the average number of packet-hops required to satisfy a client request. This measure accounts for both aspects of the benefits of caching: reduction in network traffic, and reduction of the request-response latency. Our studies involve both analysis and simulation, and use object request distributions that are consistent with studies of actual Web traffic [2]. Our results indicate that in-network caching can make effective use of cache space. In particular, active caching yields better average round-trip latencies than more traditional approaches, while requiring much smaller caches per node. Since cache cost grows (and performance shrinks) at least linearly with size, this means that performance is improved overall without increasing cost.

Placing caches at network nodes almost certainly would degrade overall performance today, due to the overhead of checking the cache at nodes en route. Note, however, that while transmission and processing speeds will continue to increase, propagation and queuing overhead is likely to remain constant. At some point in the future it seems likely that node processing will be sufficient, and it will be reasonable to trade node processing for latency.

The remainder of this paper is organized as follows. In Section 2 we provide background on wide-area caching and outline our assumptions about the application and network. We also describe related prior work in caching; we will refer collectively to these techniques as "traditional". In Section 3 we develop a set of new mechanisms for caching that take advantage of the enhanced processing capabilities available in the network due to active networking. The remainder of the paper contains an evaluation of the new and traditional caching mechanisms. In Section 4, we describe a simple analytical model for network caching. After validating the model, we demonstrate its use in tuning one of the new caching mechanisms. In Section 5, we describe our simulation model, experiment methodology and setup. Section 6 presents the results of our experiments. We conclude in Section 7.

# 2 Background

## 2.1 Wide-area caching

We assume an application in which clients request objects from servers located throughout the network. Each object is assumed to have a globally unique identifier (e.g. a hash function of a URL), and to fit in a single "message". That is, each transaction consists of a request message sent from a client toward a server, containing the ID of one requested object. The request message travels through the network until it reaches a node where the requested object is stored, which may be a cache or the server itself. A response message containing the object then travels from the server (or cache) back to the originating client. This model ignores the significant problem of reliable transfer of large objects in a datagram network, by assuming that each object fits in a single message.

In our caching study, we model the network as a collection of *domains*, each of which is represented as a graph of switching nodes connected by links. Domains are of two types, *transit*, which (as their name implies) carry transit traffic, and *stub*, through which only packets addressed to or from some node in the domain are carried. The graph models used in our simulations are constructed using the GT-ITM internet topology modeling package [11]. The paths along which packets travel in the simulations have the following characteristics:

- The path connecting two nodes in the same domain stays entirely within that domain.
- The shortest path connecting node $u$ in stub domain $U$ to node $v$ in another stub domain $V$ goes from $U$ through one or more transit domains to $V$, and does not pass through any other stub domains.
- In case two stub domains are connected directly via a stub-stub edge, the path between two nodes on the two domains *may* (but need not) go along that edge and avoid any transit domains.

## 2.2 Related work

The Harvest Cache [4] project at the University of Colorado is the largest wide area cache implementation in the Internet. Harvest caches are usually arranged in a hierarchy, and Web clients are manually configured to access a particular cache in the hierarchy that is designated as the client's proxy. If the client request can be satisfied at the initial proxy cache, it is served by the proxy. In case of a miss, the parent and sibling caches in the hierarchy are contacted using the connection-less mode of the Internet Cache Protocol (ICP [10] discussed below). If the requested object is not available at the sibling caches, or at the parent, then the client's proxy cache generates another HTTP query with its parent cache as the target using the HTTP proxy protocol or a connection-oriented version of ICP. This process is recursively repeated till the request is either served, or the object is retrieved from its origin (by the root of the hierarchy). Once the item is retrieved, it is cached on its way down to the leaf. This scheme is effective in reducing the wide-area bandwidth requirements, and in accessing "hot-spots" only once per hierarchy (as the "hot" item is then cached within the hierarchy, and subsequent requests to the same object can be satisfied by a Harvest cache hit).

The geographical push caching scheme [7] of Seltzer and Gwertzman at Harvard uses a friends-of-friends algorithm in which servers selectively push their content to *friend* caches that reside in active client networks. An similar push caching approach in which servers disseminate popular pages has been proposed by Oritz and German at the University of Waterloo [8]. In both these schemes, the server initiates the caching of an object (that it deems popular) at a remote site. In the simulation results given in the Harvard work, the "friendly" caches that act as proxys for a given server are chosen to be the nodes closest to networks that generate the most requests. Further, the server sends a redirect message to clients that contact the server but have a push cache close to them. In the Waterloo work, clients identify themselves to servers as willing push caches, and servers can choose to replicate popular objects at certain clients. The servers keep track of currently active push clients, and can redirect subsequent requests to its known proxies.

The Internet Cache Protocol (ICP) [10] defined by the Network Working Group of the IETF is a message format used for communicating among Web caches. Harvest (and its successor—Squid [1]) both use ICP to exchange information about objects cached at neighboring caches.

## 3 Self-Organizing Network Caching

A prudent amount of caching inside the network can lead to large benefits, both in terms of access latencies and bandwidth usage [4, 7]. Traditional approaches towards network caching have been to place large caches at specific points in the network, with little or no co-ordination between the caches. In contrast, we consider networks in which small caches are placed at every node. As a response message moves through the network, each cache makes a decision whether or not to store the object. Effective use of a large number of small caches is a non-trivial problem. Unless effectively organized, only a very small number of items will be cached (as the caches are much smaller), but these cached items will be replicated at many locations throughout the network. Thus, accesses to the small number of cached objects will exhibit very low latencies, but in general, the latency will not be decreased appreciably due to caching. However, if objects are cached too sparsely, then the latency does not tend to decrease, and cache performance is again sub-par. In this section, we will develop "self-organizing" caching schemes that address the problems outlined above by autonomously organizing a large number of very small caches. We will show that these schemes obviate the need to decide *where* to place caches — which can be a critical decision for traditional caching mechanisms. Of course, the self-organizing caching mechanisms work well with larger caches as well, but unlike traditional mechanisms, large caches are not required by our mechanisms to provide good cache performance.

**Caching Radius and Modulo Caching** As mentioned before, effective use of a large number of small caches is a difficult problem. In traditional (processor) cache situations, the cache replacement policy (along with cache size) determines the effectiveness of the cache. However, in order to minimize thrashing, and to stop excessive replication, a small cache must first decide whether or not to cache a particular item. This problem is effectively solved by introducing a cache *radius*: an item is cached only once within a given cache radius (in hops). Thus, the cache radius mechanism distributes an item throughout the network. By using the cache radius, we avoid making ad-hoc decisions about whether to cache particular items closer to servers or clients. In general, each of these ad-hoc mechanisms lead to poor performance. If items are cached closer to servers, the cache hit rate is high, the the overall latency is not reduced appreciably. On the other extreme, caching items closer to clients leads to very low cache hit rates (as a lot of popular items need to be cached close to busy clients to get good cache performance). We will see in our performance evaluation section that the radius caching mechanism is extremely robust across a wide range of access patterns and client–server distributions.

For a given radius, we use a modulus function to obtain an even distribution of objects (within the path). The modulus function is applied to the unique item identifier and is used to identify *where* a specific item should be cached within a segment (of length equal to the radius) of a client–server path. Thus, any given cache has a specific set of objects that can be cached there, and those sets do not overlap. In effect, nodes on the path between a client and server are partitioned into equivalence classes, so that adjacent nodes are in different equivalence classes. Objects are similarly partitioned, and objects are only cached at nodes belonging to their equivalence class. An object will be cached at every node on the path between server and client where it is allowed to be cached. We refer to the application of modulo functions within a cache radii as **modulo caching**.

**Lookaround** Network caches store *relatively* large items compared to the amount of space required to store the location of such an item within the network. For example, an object in a network cache can be several thousand bytes, while its location (node address) could be an IP address (four bytes). If an object is cached, then it has multiple "home" locations — one at the original server, and one at each of the caches. A self-organizing cache can dedicate some of its object cache space to store *locations* of *nearby* items. During a cache search, the object is searched for in local memory, and in a list containing items in "nearby" caches. If a hit is detected in a nearby cache, the request can be re-routed to the nearby cache. We can generalize this *lookaround* cache check algorithm to checking nodes within any given lookaround level. However, in our simulations, we constrain the lookaround to within a given domain, i.e. neighboring caches are checked if and only if the neighbor is a member of the same stub or transit domain. In this way, even very small caches can look like "virtual" large caches. We refer to the lookaround extensions of the modulo caching scheme as **modulo caching with lookaround**.

The lookaround scheme can easily be implemented in an active network. Within an active network, active nodes keep a periodically updated list of items cached at neighbors. The cache check at any node includes checking the in-node cache, and the list of caches of neighbors. If a hit is detected at a neighboring cache, the datagram is forwarded to the neighbor (with the same source and destination addresses). If the information about the neighbor's cache is incorrect, the datagram is forwarded to the destination (through the neighbor's shortest path). Otherwise a cache hit is recorded at the neighbor. Note that even with the lookaround scheme, the caching algorithm is stateless and backward compatible; a mix of active and non-active nodes may exist in the network, and the active cache functions may fail at any time without disrupting the routing functions.

We compare our self-organizing caching scheme to the following (location dependent) caching schemes:

- **Cache at Transit Nodes ("Transit-Only").** Transit nodes have to be traversed for every non local stub domain access; a large fraction of (shortest) paths in the network have to go through transit routers. This ubiquity of transit nodes in network paths make them prime candidates for caches.
- **Cache at Stub Nodes Connected to Transit Nodes ("SCT").** Stub nodes connected to transit nodes have to be traversed in order to access the transit network. Thus, these stub nodes form good locations for network caches. This scheme is similar to the Harvest [4] scheme for placing caches.

In addition, we consider an approach in which caches are located in every node (like self-organizing caches), but without any self-organizing mechanisms enabled. This case corresponds to a "null" active function in an active network. This is referred to as **"No AN"**.

**Comparison of Self Organizing Caches to Traditional Caches** Many traditional caching schemes do *not* explicitly minimize latency, but, rather minimize bandwidth consumption on transit links. The classic example is the Harvest cache in which several different queries may be initiated within caches in a domain before either the request is satisfied, or the query is sent to the original server. In many cases, this results in higher latency, but no bandwidth consumption in the transit network. In contrast, our schemes are designed to reduce latency — the bandwidth savings are secondary to reducing pure latency. Thus, both traditional and the self-organizing schemes can be implemented in conjunction in a network. Within a stub network, caches could be set up in a hierarchy aimed towards saving bandwidth, while in the wide area, the self-organizing schemes could be used to reduce latency.

We now develop an analytic model to evaluate in-network caching, and the self-organizing algorithms.

# 4 Analysis of Cache Performance

## 4.1 Network Caching Performance Model

We present an analytic model for the expected one-way latency in accessing an item in a network that has intra-network caching. We introduce variations on the basic model to encompass a wide range of caching locations and policies. The basic model makes the following assumptions:

1. The set of items is partitioned into "popular" items and "unpopular" items. Initially, this division is independent of the client requesting the item; in a refinement of the basic model, we consider local client preferences.
2. Only popular items are cached.
3. The items cached at a particular node are unique; i.e., no node caches the same item in more than one of its cache slots at a given point in time.
4. Every cache is full.

5. The cached items encountered from one client access to the next are independent. This independence assumption keeps the model tractable, though it clearly departs from reality.

Consider an access from a client to an item. If the item is unpopular, the one-way latency is simply the average client-server path length. If the item is popular, it may be encountered at a cache in the network. Let $L$ denote the average one-way latency, measured in hop count, when a client accesses an item. Let $q$ denote the probability that the item is popular; thus the item is unpopular with probability $(1-q)$. Let $L_u$ denote the average length of a client-server path and $L_p$ denote the average length of a path from a client to a popular item (possibly encountered at a cache). Then:

$$L = (1-q)L_u + qL_p$$

The expression for $L_u$ is straightforward. Let $N_s$ denote the average path length from a stub node to the transit node for a stub domain. Let $N_t$ denote the average path length across transit domains. Then $L_u = 2N_s + N_t$, since a client-server path traverses two stub domains (client and server domains) and a path through the transit domains.

We focus the rest of the discussion on deriving an expression for $L_p$. Let $H_i$ denote the event "a hit occurred at distance $i$ from the client"; let $M_i$ denote the event "a miss occurred at *every* distance up to and including $i$ from the client". Let $d$ denote the distance from the client to the server. Then

$$\begin{aligned} L_p &= \sum_{i=1}^{d} i Pr\{H_i \cap M_{i-1}\} \\ &= \sum_{i=1}^{d} i Pr\{H_i \mid M_{i-1}\} Pr\{M_{i-1}\} \end{aligned}$$

The probabilities in this expression are affected by details of the locations of the caches, the cache policy and the client access pattern. We develop variations on the basic model to handle different instances of these details.

**Model A.** In this model, we assume that the items stored in different caches are *independent*. This is reasonable when the access pattern is uniform and the caching policy does not attempt to eliminate duplicates along a particular path. We further assume that each cache is equally likely to store any popular item. This is reasonable for caching at locations in the middle of the network, at nodes that will "see" all popular items with equal probability.

Since the caches are independent, $Pr\{H_i \mid M_{i-1}\} = Pr\{H_i\}$. We are focusing on the latency when accessing popular items; the probability of a hit when accessing a popular item and examining a cache of size $S$ is $S/P$, where $P$ denotes the number of popular items. Obviously, the probability of a hit is 1.0 if the request reaches the server. Thus, if $S_i$ denotes the size of the cache at distance $i$, then:

$$\begin{aligned} L_p &= \sum_{i=1}^{d} i Pr\{H_i\} Pr\{M_{i-1}\} \\ &= d\left(\prod_{j=1}^{d-1}(1-\frac{S_j}{P})\right) + \frac{1}{P}\sum_{i=1}^{d-1} i S_i \left(\prod_{j=1}^{i-1}(1-\frac{S_j}{P})\right) \end{aligned}$$

Note that the first term accounts for the boundary condition where the request reaches the server. A caching policy that increases $S_i$, for example by checking the contents of the caches at one's neighbors, can significantly improve the one-way latency.

**Model B.** In this model, we assume the best possible conditions regarding duplication in items stored in caches along the path. Namely, we assume that the items stored in different caches are *different*. Though this is difficult to achieve in a distributed environment, it provides an upper bound on cache performance (which is a lower bound on average round-trip length). We keep the assumption from Model A that each cache is equally likely to store any popular item.

Since all items are different, the probability of a hit when accessing a popular item and examining $S$ cache locations (potentially distributed over multiple nodes) is $S/P$. If we know that a miss has occurred at locations at every distance up to and including $i-1$, then we know that the item we are looking for is in a reduced set of popular items, namely reduced by the number of cache locations we have already checked. Let $T_{i-1}$ denote the total number of items checked at distances up to and including $i-1$ [1]. Then

$$
\begin{aligned}
T_{i-1} &= \sum_{j=1}^{i-1} S_j \\
L_p &= \sum_{i=1}^{d} i Pr\{H_i \mid M_{i-1}\} Pr\{M_{i-1}\} \\
&= d\left(1 - \frac{T_{d-1}}{P}\right) + \sum_{i=1}^{d-1} i \frac{S_i}{P - T_{i-1}} \left(1 - \frac{T_{i-1}}{P}\right) \\
&= d\left(1 - \frac{T_{d-1}}{P}\right) + \sum_{i} i \frac{S_i}{P}
\end{aligned}
$$

Again, the first term is the boundary condition for the request reaching the server. For clarity, we omit this term in the remaining model development.

An effective cache policy should eliminate duplicates on the path, to the extent possible. We can modify this model to reflect a caching policy that provides some (though not complete) elimination of duplicates. Let $U_{i-1}$ denote the total number of *unique* items in caches at distances up to and including $i-1$. Then

$$
\begin{aligned}
L_p &= \sum_i i \left(\frac{U_i - U_{i-1}}{P - U_{i-1}}\right)\left(1 - \frac{U_{i-1}}{P}\right) \\
&= \sum_i i \left(\frac{U_i - U_{i-1}}{P}\right)
\end{aligned}
$$

**Model C.** We now relax the assumption that each cache is equally likely to store any popular item. While this is reasonably accurate for caches in the middle of the network, it is not accurate for caches located, for example, in the same domain as servers. These caches will be more likely to store popular items from the nearby servers, rather than popular items from other servers.

[1] We assume that $T_i < P$ for all $i$. If this assumption does not hold, then at some point in the path, all popular items will have been encountered, and the hit probability will go to 1.0.

We restrict this model to the case where the caches on the path up to distance $k$ are all equally likely to store any popular item. After distance $k$, the path crosses into the stub domain containing the desired server. At this point, we assume that all caches (until we reach the server) will store from a set of $R$ popular items, and will not store any of the other $P - R$ popular items. $R$ might be determined, for example, by the number of popular items at servers in this domain. We further assume that each cache in the domain selects independently from the $R$ locally popular items.

Until we reach distance $k$, the equations derived earlier for $Pr\{H_i \mid M_{i-1}\}$ and $Pr\{M_{i-1}\}$ are accurate. After distance $k$, we must consider that the caches contain only items from the $R$ locally popular, and further that this request is for one of the locally popular items, otherwise the path would not be entering this domain. Since each cache in the stub domain selects independently from the $R$ locally popular items, $Pr\{H_i \mid M_{i-1}\} = S_i/R$. Further, $Pr\{M_{i-1}\} = Pr\{M_{i-2}\}(1 - S_i/R)$. Thus,

$$\begin{aligned} L_p &= \sum_{i=1}^{k} i Pr\{H_i \mid M_{i-1}\} Pr\{M_{i-1}\} \\ &+ \sum_{i=k+1} i \frac{S_i}{R} \left( \prod_{j=k+1}^{i-1} (1 - \frac{S_j}{R}) \right) Pr\{M_k\} \end{aligned}$$

Caches within stub domains containing servers are likely to contain mostly items from the local servers. While the hit probability will be high at these caches, the savings in latency is fairly small: the query has nearly reached the server. It may be reasonable to impose a cache policy that explicitly does not cache these local items.

**Model D.** In the final variation, we extend the idea of locally popular items to include the caches at nodes in the same domain as a client. This local popularity will result, for example, if the clients in the domain have a skewed access pattern.

Again, let $R$ denote the number of locally popular items. Let $k$ denote the number of nodes encountered within the stub domain of the client. Assume each cache within the stub domain independently caches items from the $R$ locally popular. Further, assume that the client is requesting one of these locally popular items. Then,

$$\begin{aligned} L_p &= \sum_{i=1}^{k} i \frac{S_i}{R} \left( \prod_{j=1}^{i-1} (1 - \frac{S_j}{R}) \right) \\ &+ \sum_{i=k+1} i Pr\{H_i \mid M_{i-1}\} Pr\{M_{i-1}\} \end{aligned}$$

Caches within stub domains containing clients are likely to contain mostly items of local interest to the clients. We expect a significant performance gain with caches in these locations, especially when client access patterns are skewed.

## 4.2 Model Validation

The analytic models contain assumptions that we know are simplifications of reality. We have developed a wide-area caching simulator to study more complex and realistic scenarios [2]. We

[2] The simulator is described in detail in Section 5.

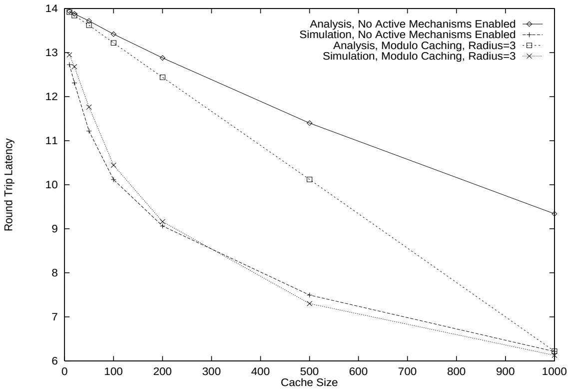


Figure 1: Round Trip Latency with varying cache size, Low Access Correlation

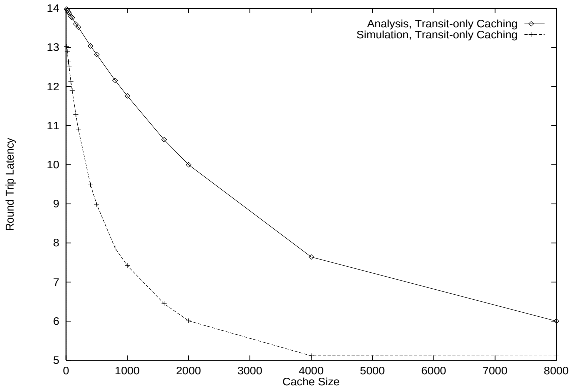


Figure 2: Round Trip Latency with varying cache size

can use this simulator to determine how well the analytic model can predict latency for the various caching schemes of interest. We use Model A with equal size caches at all nodes to model "No-AN" caching. We use Model A with equal size caches at transit nodes (and no caches at stub nodes) to model Transit-only caching. Since the Modulo caching scheme makes an attempt to reduce duplicates on a path, we use Model B to represent Modulo caching, and increase the effective cache size per node to represent the effect of lookaround.

In Figures 1 and 2 we compare the round trip latency as predicted by the model to the latency measured in a simulation. The simulated graph had 1500 nodes, an average degree of 3.71, and an average server-client round trip latency of 13.1. There were 7200 popular items. In the analytic model, we assumed that the path from client to server traversed two nodes in the client domain, three transit nodes, and two nodes in the server domain. Thus the round trip latency in the analytic model was 14 hops.

The model is not particularly accurate for the No-AN and Modulo caching schemes, generally predicting much larger round trip times than are reported by the simulation. The model is somewhat more accurate for the Transit-only caching scheme. There are fewer caches encountered in the Transit-only scheme; a hit either occurs at one of the three transit nodes, or the access goes all the way to the server. Some inaccuracy is introduced by the discrepancy in the round trip latencies, but the simplifications inherent in the model are also relevant.

Thus, the model in this basic form is useful only for rough approximations of the latency. As we will see in the next section, more careful analysis can be used to refine the model and achieve much more accurate predictions.

## 4.3 Optimal cache partitioning

As discussed in Section 3, unlike in traditional caches, the identifier for a location (typically a network address of length between four and ten bytes) in a network cache is much smaller than an actual cached object (which can be thousands of bytes). It may, therefore, be beneficial to use some of the network cache's memory to store locations of *nearby* objects. We investigate

| | | |
|---|---|---|
| S | – | Size of each cache |
| $C$ | – | Maximum number of objects in cache of size S |
| $\alpha$ | – | Number of object locations that can be stored instead of one object |
| $\delta$ | – | Average degree of the graph |
| $d$ | – | Average one-way length of server–client path |
| $\eta$ | – | Fraction of cache memory used for storing objects |

Table 1: Definitions and Notation

the effect of partitioning a node's cache memory into two parts. The *object cache* is a fraction of the cache the node uses to store objects (as before), and the remaining memory is used as a *location cache*, where the node stores locations of nearby objects. During a cache check at a node, if the local object cache misses, the node will look for the object in its location cache. Assume there is a hit in the location cache at location (say $l$). The location $l$ is presumably a nearby node where the object is cached. The request would be redirected towards the node $l$ (which may not be on the shortest path to the request's original destination). The request would then either be served at the node $l$, or if the node $l$ does not have the requested object (due to a recent cache flush), the request would be redirected towards the original destination. In this way, even small network caches can be used to *check* many items and locations at once, and object latency may be decreased[3]. As the amount of memory allocated to the location cache increases, the length of the detours increase, and the actual number of objects stored at the network nodes decrease. In the rest of this section, we analyze the optimal partition of the network caches into object and location caches.

We assume that in its location cache, each cache stores the names of objects stored at its neighbors. Let cache lookaround denote the checking of the location cache at a given node. We generalize the concept of cache lookaround by associating with it a *level* of lookaround — a lookaround of level $x$ means that the node stores cache information of nodes at a distance of $x$ from itself, and during a cache search, can determine whether the item being searched for is cached within a distance $x$ from itself (modulo recent cache updates). Thus, storing the names of items of direct neighbors correspond to a lookaround of level one, and normal caching with no location caches is equivalent to a lookaround of level zero.

Let the size of each cache be identical and equal to S, assume that each object is the same size. Let the maximum number of objects in each cache be C. Let the ratio of the size of an object and the size of a location be $\alpha$, i.e. $\alpha$ locations can be stored in the cache instead of one object. Let the average one-way length of server–client paths be d. We assume paths are symmetric, and as such, the average client–server round trip latency is $2d$. Starting from a given cache, as the level of lookaround is increased, the number of caches searched increases. For most graphs, this increase is exponential. As a simplification, we assume that the number of caches searched at a lookaround level of $x$ is exponential in $\delta - 1$, where $\delta$ is the average degree of the topology graph. [4]. In general, a fraction $\eta$ of the cache memory is used for caching objects and the rest is used for caching location information.

We now derive the value of fraction $\eta$ such that the round trip latency is minimized. The

[3]The protocols to update the location cache and the frequency of updates are considered orthogonal to and independent of the partitioning of a node's cache.

[4]The exponent is $\delta - 1$ because after the first level of lookaround, at least one node has already been searched at the previous lookaround depth

precise fraction that minimizes the latency depends on the size of the network, the size of the caches, and the number of locations that can be stored instead of each object. Using the basic caching model developed before, the expected latency is

$$L_p = \sum_i iPr\{H_i \mid M_{i-1}\}Pr\{M_{i-1}\}$$

Using our model of memory usage, the expected latency can be calculated as follows. For the local objects (in the object cache), the latency after traveling $i$ hops is $i$. If at node $i$, there is a hit in the location cache, then a detour has to be made. We can approximate the one-way length of the detour as follows:

We know that if $\eta$ fraction of the memory is used for caching objects, then $(1-\eta)S$ amount of memory is used for storing locations. This is equal to $(1-\eta)C\alpha$ number of locations. As each cache stores $\eta C$ number of objects, the $(1-\eta)C\alpha$ number of locations correspond to locations from $\frac{(1-\eta)\alpha}{\eta}$ number of nodes. Under the assumption that the number of nodes increase exponentially as the level of lookaround increases, $\frac{(1-\eta)\alpha}{\eta}$ nodes imply a maximum detour of length $\log_{\delta-1}\frac{(1-\eta)\alpha}{\eta}$. Let $f$ be the expected value of the length of this detour, and let $F = \log_{\delta-1}\frac{(1-\eta)\alpha}{\eta}$. At each level $x$ of lookaround, $(\delta-1)^x\eta C$ number of items are cached. Thus, the expected one way detour length is:

$$\begin{aligned}
f &= \sum_x^F x\frac{(\delta-1)^x\eta C}{(1-\eta)C\alpha} \\
&= \frac{\eta}{(1-\eta)\alpha}\sum_x^F x(\delta-1)^x \\
&= \frac{\eta}{(1-\eta)\alpha}F(\frac{(1+F)(\delta-1)^F}{(\delta-1)-1} - \frac{(\delta-1)^{F+1}-1}{((\delta-1)-1)^2})
\end{aligned}$$

If we assume that the lookaround memory is arranged such that the content of nearest caches are searched first (i.e. caches at level 1 are searched before caches at level 2, and so on), then we can derive a simpler expression for the the expected value of the detour. Under this case, on average, half of the cache will be checked before a hit (if any) occurs in the lookaround cache. Thus, the expected length of a detour in this case is:

$$\begin{aligned}
f &= \log_{\delta-1}\frac{(1-\eta)C\alpha}{2\eta C} \\
&= \log_{\delta-1}\frac{(1-\eta)\alpha}{2\eta}
\end{aligned}$$

As previously defined, let $P$ be the total number of popular items, and let $T_{i-1}$ be the total number of items checked till distance $i-1$ from the source of the request. In our case, as the cache sizes are assumed to be identical and equal to $S$, $S_i = S$ (for all $i$), and $T_{d-1} = (d-1)S$.

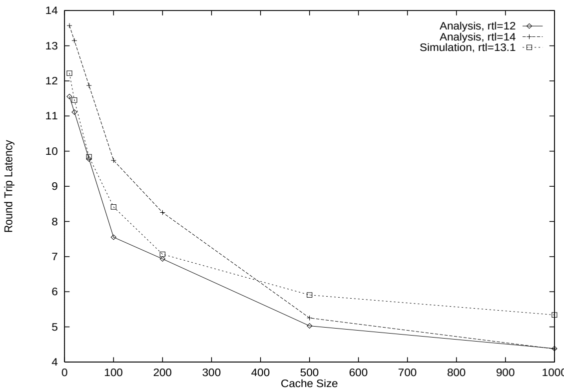


Figure 3: Comparison of analytic and simulation results — Lookaround Caches

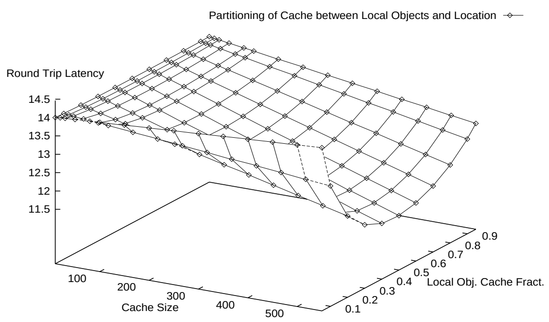


Figure 4: Optimal partitioning of cache space for local objects and location information

Thus, the expected value of the one way latency (using Model B developed above) is:

$$
\begin{aligned}
L_p &= \sum_i^d i Pr\{H_i \mid M_{i-1}\} Pr\{M_{i-1}\} \\
&= d(1-(d-1)\frac{S}{P}) + \sum_i^d i\frac{S_i}{P} \\
&= d(1-(d-1)\frac{\eta C + (1-\eta)C\alpha}{P}) + \sum_i^d \frac{\eta C}{P}(i) + \frac{(1-\eta)C\alpha}{P}(i+f) \\
&= d(1-(d-1)\frac{\eta C + (1-\eta)C\alpha}{P}) + (\frac{\eta C}{P} + \frac{(1-\eta)C\alpha}{P})\frac{d(d+1)}{2} + \frac{(1-\eta)C\alpha}{P}(f)
\end{aligned}
$$

As caches are updated independently, the location caches at each node may not point to all unique items across all the nodes. Thus, we can define a fraction of items, $u$, that are unique in each location cache. Thus, the number of items checked at each node in this case is $\eta C$ number of local items, and $u(1-\eta)C\alpha$ number of unique remote items.

The one way latency is this case is again:

$$
L_p = (\frac{\eta C}{P} + u\frac{(1-\eta)C\alpha}{P})\frac{d(d+1)}{2} + u\frac{(1-\eta)C\alpha}{P}(f) + d(1-(d-1)\frac{\eta C + u(1-\eta)C\alpha}{P})
$$

Minimizing the numerator with respect to $\eta$ gives the optimal amount of cache to be used for storing location information such that the expected latency is minimized.

In Figure 3, we compare the result of the analysis of lookaround caching with a simulation with similar parameters. The simulated graph had 1500 nodes, an average degree of 3.71, $\alpha$equal to 50, and and average server-client round trip latency of 13.1. In the simulation, the lookahead was fixed to 2 levels, and the radius of caching at 3. The two analytic curves were generated by setting the one-way client–server path length($d$) to 6 and 7, respectively. This corresponds to the average length of client–server round trip lengths being 12 and 14, respectively. Thus, in general, we would expect the curve generated by simulation to be bounded (on either side) by the two curves generated by the analysis. For small cache sizes ($C < 500$), this is the case. The refinements to the basic model generate an extremely accurate measure of the cache latency. When the cache sizes are larger, the analysis overestimates the number of *unique* items that are cached.

In Figure 4, we use the expression for expected round trip latency to evaluate the benefits of lookahead caching. The x-axis shows the size of each cache, the y axis denotes the amount of the cache memory devoted to caching local objects. The round trip latency is shown on the z-axis. We use the nearer caches are checked earlier assumption to evaluate the length of the detour. In this plot, we set the number of popular objects in the network to 80,000, and again, $\alpha$is set to 50. the topology, the large number of popular objects map to a topology that may be too large to simulate at even a coarse scale. The average client–server round trip path length was set to 14. However, the plot clearly shows the benefit of lookaround caching — the *valley* in the plot is directly due to the benefits of lookaround caching. In the plot, the expected round trip latency decreases as the amount of memory devoted to the lookaround cache is increased ($.99 < \eta < 0.5$). However, after a point, as more memory is dedicated to the lookaround cache, the number of objects cached in the network decreases, and the round trip latency increases sharply ($0 < \eta < 0.3$), as there are not enough objects cached in the network, and each lookaround hit causes a large detour. It is interesting to note that at very small values of $\eta$, the round trip latency can increase to greater than the average latency without caching—this is again expected, as there are not enough objects cached, and each detour is greater then the remaining path length to the server. Of course, a trivial optimization, that forbids detours greater than the remaining path length can easily be incorporated.

Thus, we with some extensions to the basic cache model, we can accurately predict the cache behavior. However, the models do not represent the sizes of different transit and stub domains, and the interaction between different domains. Also, the model is devoid of interactions due to different access patterns. In order to study these phenomena in greater detail, we present a set of detailed simulation results in the next section.

# 5 Simulation Environment

## 5.1 Experimental Methodology

The wide area caching behavior was simulated using a locally developed discrete event network simulator called AN-Sim. AN-Sim simulates an active network as defined in [3], and allows for variation of the network topology. This section discusses the various parameters of our simulations.

**Network Topologies.** We simulated several topologies, differing in number of nodes, diameter, average node degree, ratio of transit nodes to stub nodes, etc. Table 2 summarizes the properties for a list of topologies for which we present results. All of these graphs have 1500 nodes, of which 60 are transit nodes.

| Graph | Average Degree | Avg. Num. of S-Domains per Transit node | Avg. Num of S-nodes per S-domain |
|---|---|---|---|
| `Base` | 3.71 | 4 | 6 |
| `More Stub Domains` | 3.14 | 6 | 4 |
| `Fewer Stub Domains` | 4.11 | 2 | 12 |
| `Higher Degree` | 4.53 | 4 | 6 |
| `Lower Degree` | 3.06 | 4 | 6 |

Table 2: Simulated Topologies

**Cache and Server Distributions.** A subset of stub nodes, chosen uniformly at random, was designated to host *popular* servers. The fraction of servers that were popular, and corresponding accesses to the popular servers was varied using a Zipf distribution, i.e. a small fraction $\epsilon$ of the servers received a large fraction $1-\epsilon$ of the requests. We varied $\epsilon$ between 0.1 and 0.2, resulting in a number of popular servers between 150 and 300. The remaining accesses (to non-popular servers) were chosen uniformly at random from the remaining stub nodes. There were 4 billion ($2^{32}$) unique objects in each simulation, the vast majority of which were not accessed. Each object was associated with a particular server, thus each server's content was unique. A subset of objects at each server was designated to be popular, and popular objects were requested 95% of the time. The number of popular objects was fixed at 48 per server (for a total number of popular objects that ranged between 7200 and 14400. The remaining objects were served uniformly at random the remaining 5% of the time.

**Access Patterns** We simulated several different access patterns at clients. Accesses are only generated at stub nodes. In the uniform access pattern, a server is chosen using the server distribution given above, and an object is chosen at the server, again using the distribution given above. We also simulated a more skewed access pattern. In this case, correlated accesses originated at clients with a given probability. Each set of correlated accesses have an average length of 16. During correlated accesses, the source-destination pair was fixed. The probability that a particular datagram is accessed more than once within the correlated access sequence was varied. Correlated accesses are not consecutive, but are generated with a random temporal offset (from current time). In the third access pattern simulated, a set of servers is associated with each node. A fraction of all accesses from the node is then directed towards servers in this (per node) set. We also simulated the case in which the set of servers is associated with a particular domains, instead of individual nodes.

These access patterns —and indeed all of our simulation parameters— are consistent with what is known about access to objects in the World Wide Web. Among the "invariants" found by Arlitt and Williamson in their study of server logs [2] was that the number of distinct objects requested is between .3% and 2.1% of the total number of requests, i.e. each requested object is requested 50 to 300 times in a trace on the average. They also found that the vast majority of all requests are for objects that would fit inside a single IP datagram, and that at least 75% of all requests to each server they studied are nonlocal.

**Cache Sizes** The size of the cache at each caching node is proportional to the node degree. This is equivalent to having a fixed amount of caching per edge (interface) incident upon the node. For each caching simulation, the total number of cache slots in the network was held constant. Thus, in cases where the total number of caching nodes were small (like in Transit-Only caching, or caching only in stub nodes connected to transit nodes), the relative size of the cache (compared to cache size when each node has a cache) is large. We should note that equal number of cache slots in the network is not equivalent to equal number of cache slots on any given path. Also, when each node in the network maintains a cache, there may be a substantial fraction of caches, that are only on leaf nodes, useful only when there is a repeat request from a host on the leaf. However, we did not endeavor to equalize the number of cache slots encountered on individual paths.

**Limitations of Experiments.** Our experimental setup has some known limitations. The simulator does not enforce link rates, and thus lost datagrams due to full buffers are not represented in the results. The lookaround algorithm does not generate actual packet traffic in the simulation. Also, the space required for storing the list of cached items at neighbors is not accounted for. None of these is particularly serious.

### 5.2 Performance Metrics

We use the following metrics to evaluate the performance of network caches.

- **Round trip length.** We measure the request-response round trip length, i.e. the number of nodes of the network traversed in completing a transaction. This is perhaps the simplest and "truest" measure of network cache performance.
- **Fraction of queries that generate cache hits** After an initial start up period, the cache performance stabilizes. We measure the fraction of queries that are serviced by cache hits. Note that queries served by caches not only reduce access latencies and conserve bandwidth, but also reduce server load.
- **Fraction of thrashing nodes** We define a node $i$ to be thrashing if the number of flushes at $i$ is greater than a certain fraction of hits at $i$. Precisely, if flushes outnumber hits by greater than 10%, then a node is marked as thrashing.

## 6 Simulation Results

### 6.1 Summary of results

We have simulated cache performance for the topologies specified in table 2 across a wide range of cache sizes, server distributions, and access parameters. Transit nodes are not able to take advantage of correlated accesses from particular stubs. Thus, in general, the active mechanisms outperform all other methods (including Transit-Only caching) for correlated accesses. For non-correlated accesses Transit-Only caching scheme performs best (in terms of average round trip length).when there is low or no correlation in accesses. However, the AN schemes are always within 10% of the round trip length of the Transit-Only caching scheme.

For the base graph, there are on average 4.92 stub nodes, and 3.32 transit nodes on each path. In this graph, there are 60 transit nodes, and 1440 stub nodes. Thus, in general, a much larger fraction of total transit nodes and transit caches are traversed for each access. Also, under Transit-Only caching, the average cache is 25 times larger than the average cache under AN caching. Thus, the large caches lead to large gains due to multiplexing at the transit nodes—especially if the accesses are not correlated. In the subsequent sections, we present plots, and details of a small cross section of results from our experiments. Except where otherwise noted, our results are for the base graph topology, and the cache replacement policy is LRU.

### 6.2 Variation in Cache Size

We consider the performance of the caching schemes, as we vary the nominal cache size between 4–48 cache slots per interface. The modulo, and no AN methods use nominal cache size, as these methods cache at all nodes. The corresponding cache sizes for Transit-Only caching are 33 to 397, and for Stubs-connected-to-Transit (SCT) caching are 15 to 188. All of the caching mechanisms, except SCT, show a smooth decrease in number of hops traversed per round trip as the cache size is increased. In section 5.1, we defined a set of correlated accesses to be a set of accesses in which the source-destination pair is fixed. Such correlated accesses are not consecutive, but are generated with a temporal offset. Thus, between correlated accesses, there can be other accesses generated from the client. The average size of a set of correlated accesses was 8, and the probability of initiating a set of correlated accesses was 0.25. Note that just correlated access do not imply better caching performance, as there are sufficient items in each server to fill up caches in any path. In Figure 5, the probability of repeating an accesses within a set of correlated accesses is 0.1. The modulo cache radius was fixed at 3. The improvement due to modulo caching is seen for small sizes cache. For cache size 48, the "no AN" (cache everywhere) mechanism performs better than the modulo caching as there are enough cache slots available, such that 3 is too large a radius for modulo caching. The lookaround schemes perform better than all but the transit-only caching scheme, and the performance of the two-level lookaround scheme[5] is within 10% of the Transit-Only scheme in all cases. It should be noted that the number of cache slots per interface for the two-level lookaround scheme is an order of magnitude smaller than the transit-only scheme. Also, the average degree of the transit nodes is much greater than the average degree of the graph. Thus the total size of the transit nodes caches are 25 *times* larger than the total size of the modulo caches. Comparatively, the average total size of the SCT caches are 4.25 times greater than the modulo caches. As the access are more correlated, as shown in Figure 6, the modulo caches with lookaround outperform all other cache mechanisms—including the transit-only caching. Also significant in Figures 5 and 6 is the behavior of the caches located only at the stubs connected to transit nodes. Their performance improvements are negligible beyond 12 cache slots per interface, and as such this method does not scale well with increase in cache size.

For the same variation in cache sizes experiment, Figure 7 shows the fraction of queries that generate cache hits. Once again, all the methods except caching at stub nodes connected to transit nodes show increases with increase in cache size. The SCT method actually results in a proportionately large fraction of hits—but the large number of hops in the round trip suggests that more hits occur in the stub node to which the *server* is connected, and not the

[5] In all cases, the lookaround is restricted to within the same domain.

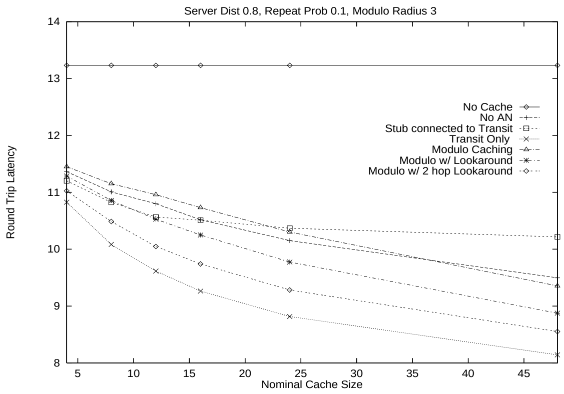


Figure 5: Round Trip Latency with varying cache size, Low Access Correlation

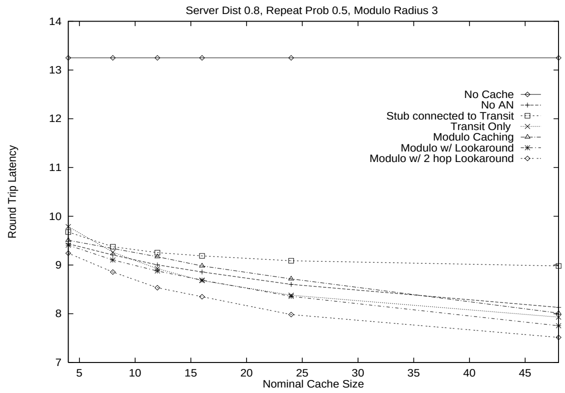


Figure 6: Round Trip Latency with varying cache size

client. This is not unexpected—the *gateway* stub node connected to the transit domain for a busy server will experience a lot of traffic due to the busy server, and as such, will cache a large part of that data as well. Figure 8 shows the fraction of thrashing nodes for the same experiment with high access correlation. In general a higher fraction of AN caches thrash compared Transit-Only and SCT caches. This is partly due to the mismatch in individual cache sizes. However, in this case, less thrashing does not translate to better performance, as the AN caches outperform all other cache mechanisms under high access correlation.

### 6.3 Variation in Server Distribution

We have used a Zipf distribution on accesses to popular servers (i.e. 100-$x$% of the accesses are to $x$% of the nodes). However, it is not clear exactly what fraction of the nodes should be considered to be *servers*. In this experiment (Figure 9), we vary the fraction of nodes that are servers from 1–50%. Even when a large fraction of nodes are servers, the cache performances are not affected. Thus, wide-area caching seems robust in face of widely varying server locations, and distributions. It is interesting to note that round trip latencies for Transit-Only cache schemes do not improve much when the server distributions are extremely skewed—less than 10% of the nodes are servers. The other schemes improve as the number of popular objects (which is a multiple of the number of servers) decrease, but in case of Transit-Only caches, even if all the objects are cached everywhere, the query has to reach the transit nodes before it is serviced.

### 6.4 Variation in Topology

We consider the effectiveness of the various cache policies as the underlying topologies are varied. The round trip latency is inversely proportional to the graph degree, and directly proportional to the average size of the stub domains. The two level lookaround algorithm performs equal to the Transit-Only cache method for the base graph (with probability of a repeat equal to 0.3). As topologies are varied, the performances are comparable—with

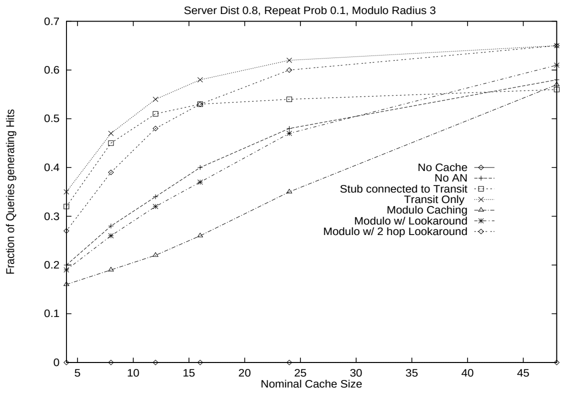


Figure 7: Fraction of Cache Hits with varying cache size

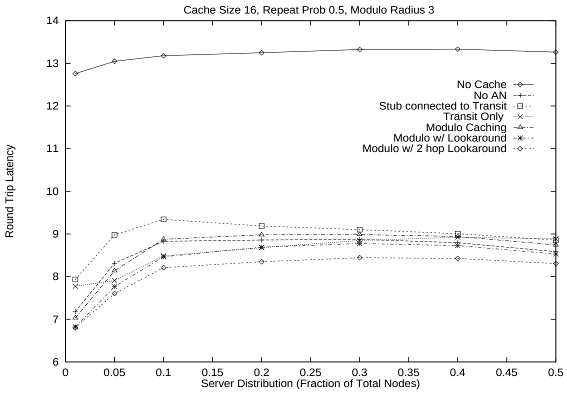


Figure 8: Fraction of Thrashing Caches with varying cache size

the AN methods performing better as the number of stub domains increase. As the repeat probabilities increase, the AN mechanisms perform better than all other mechanisms in all topologies.

### 6.5 Spatial Access Patterns

In these experiments, we consider a different access pattern. Associated with stub node is a set of *preferred* servers, and a fraction of queries generated by the node is always directed to the preferred set. Figure 11 shows the round trip latencies as the number of servers in the preferred set is varied from 2–12. The probability of accessing a server in the preferred set was 0.25, and the probability of a repeat access was 0.3. The two level lookaround and the Transit-Only cache schemes perform the best, with the lookaround schemes being better if the number of preferred servers are small. In fact, the round trip latency for Transit-Only caching is nearly constant in this case for any number of servers in the preferred groups as the accesses are nearly uniform at the transit domains. The performance of the lookaround schemes deteriorate as the number of servers increase because the locality in accesses are lost, and even inside the stub domains the accesses seem nearly uniform.

We also consider spatial access patterns in cases when the preferred set is common to the entire stub domain. In Figure 12, there is a 0.5 probability of accessing a server in the preferred set, but the probability of repeat is nil. The effect of the number of servers in the preferred set is clear: the round trip latencies increase with increase in the number of preferred servers. In these cases when there is a per domain preference for certain data, the caching at stub nodes connected to transit node scheme works quite well: within 1 hop per round trip of optimal in all cases.

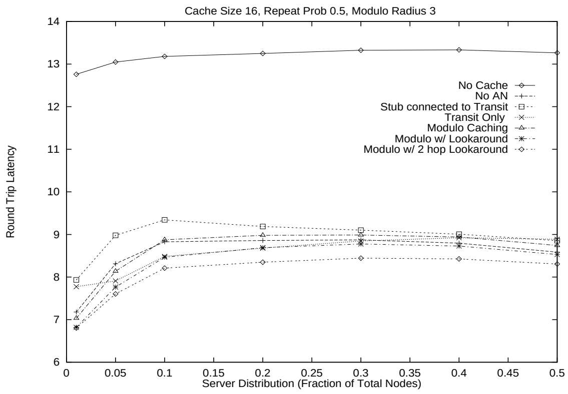


Figure 9: Round Trip Latency with varying server distributions

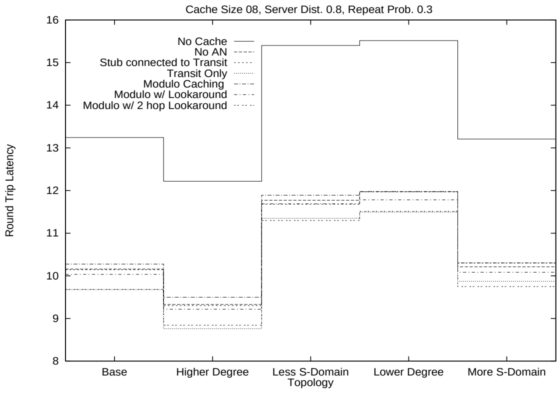


Figure 10: Round Trip Latency for different topologies

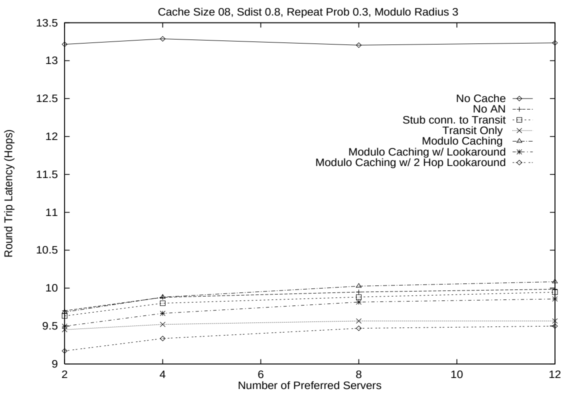


Figure 11: Round Trip Latency with different number of preferred servers per node

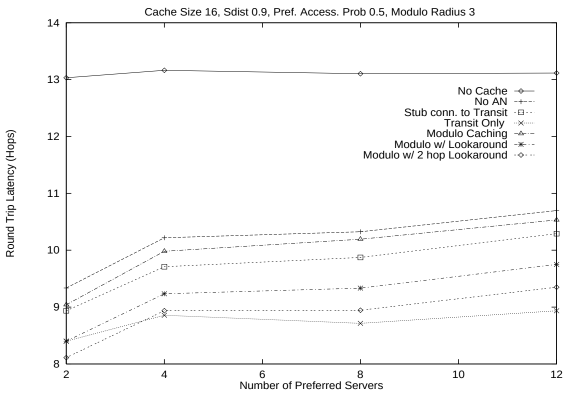


Figure 12: Round Trip Latency with different number of preferred servers per domain

## 7 Concluding Remarks

We have developed and evaluated mechanisms for caching objects within the network. Our active caching mechanisms allow transparent, self-organizing, location of objects where they can be of benefit to reduce access latency. These methods contrast with traditional wide-area network caching that relies on fixed and limited locations for caches.

We have used both simulation and an analytic model to evaluate performance. The simulation uses access patterns that are consistent with studies of Web access; the analytic models are simplified for tractability, though still provide reasonable estimates of access latency. The models could be extended using information about uniqueness of items and/or more complex mathematical modeling to remove simplifying assumptions.

Our results show that active caching is beneficial across a range of network topologies and access patterns, and is especially effective when access patterns exhibit significant locality characteristics.

Our work on network caching has been motivated by the development of applications for active networking. For active networking to succeed, application drivers are critical to demonstrate performance benefits that outweigh the development and deployment costs. Caching represents an example of an application that may be well suited to active networking, since active caching can exploit access patterns indicated by the request-response paths that pass through the network. We have also considered active networking for application-specific congestion control in the context of a specific service architecture [3].

## References

[1] Squid internet object cache. http://squid.nlanr.net/Squid/.

[2] M. Arlitt and C. Williamson. Web server workload characterization: The search for invariants. In *Proceedings of ACM SIGMETRICS '95*, May 1995.

[3] Samrat Bhattacharjee, Kenneth. L. Calvert, and Ellen W. Zegura. An Architecture for Active Networking. In *Proceedings of High Performance Networking 97*, 1997.

[4] C. M. Bowman, P. Danzig, D. Hardy, U. Manber, M. Schwartz, and D. Wessels. Harvest: A scalable, customizable discovery and access system. Technical Report CU-CS-732-94, University of Colorado - Boulder, 1995.

[5] H. Braun and K Claffy. Web traffic characterization: An assessment of the impact of caching documents from NCSA's web server. In *Electronic Proceedings of the Second World Wide Web Conference '94: Mosaic and the Web*, October 1994.

[6] P. Danzig, M. Schwatrz, and R. Hall. A case for caching file objects inside internetworks. In *Proceedings of ACM SIGCOMM '93*, September 1993.

[7] J. Gwertzman and Margo Seltzer. The case for geographical push caching. In *Hot Operating Systems*, 1995.

[8] A. Lopez-Ortiz and Daniel M. German. A multicollaborative push-caching http protocol for the WWW. In *World Wide Web Conference (WWW5)*, 1995. Poster Presentation.

[9] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1), 1997.

[10] D. Wessels and K. Claffy. Internet Cache Protocol (ICP), version 2. Technical report, IETF Network Working Group, May 27 May 1997. draft-wessels-icp-v2-03.txt.

[11] E. Zegura and K. Calvert. Georgia Tech Internet Topology Models. http://www.cc.gatech.edu/projects/gtitm.