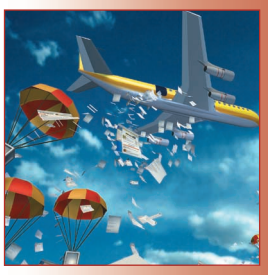

Web Caching in Broadcast Mobile Wireless Environments

Effectively exploiting available communication bandwidth and client resources is vital in wireless mobile environments. One technique for doing so is client-side data caching, which helps reduce latency and conserve network resources. The SliCache generic self-tunable cache-replacement policy addresses these issues by using intelligent slicing of the cache space and novel methods for selecting which objects to purge. Performance evaluations show that SliCache improves mobile clients' access to Web objects compared to other common policies.

Dimitrios Katsaros and Yannis Manolopoulos
Aristotle University

As used by current Web servers, unicast delivery is a waste of resources for dissemination-based applications (those in which a dynamic client population requests data from servers that respond with the requested items). Network and server loads increase with each additional client because data must be transmitted in response to each client request. In contrast, broadcast is becoming an increasingly appealing solution because of its excellent scalability: a single broadcast can satisfy all pending requests for a given item. Yet, scarce bandwidth and client resources leave wireless users with lengthy access delays. Researchers have proposed various scheduling algorithms that attempt to reduce average or maximum latency, but in addressing the "average" mobile client's needs, these approaches sacrifice the individual for the sake of the majority. As Acharya and colleagues have noted, "tuning the performance of the broadcast channel is a zero-sum game."[1] Improving broadcast for any single access distribution hurts performance for other clients. Data caching at the mobile client is therefore vital for reducing the latency experienced by clients and for conserving network resources.

Current proposals for cache replacement in wireless mobile environments are generally limited because they assume

 1089-7801/04/$20.00 Published by the IEEE Computer Society

Related Work in Caching in Wireless Mobile Environments

There are very few proposals for cache-replacement policies in wireless mobile environments, and most of the ones that have been introduced are designed for pure push-based broadcast.

The PIX policy assumes that the client has exact knowledge of the broadcast disk in which the object belongs.[1] Algorithms based on *look-ahead* (knowledge of future broadcasts) — such as the Gray algorithm[2] and the policy proposed by Tassiulas[3] — suffer from high implementation costs; this is primarily because they assume full knowledge of the broadcast schedule.

For on-demand broadcasts, Xu et al. proposed the *minimum stretch integrated with access rates, update frequencies, and cache validation delay* (Min-Saud) policy.[4] Min-Saud also considers access frequency, retrieval cost, object size, and object-update frequency. Tests have shown that it performs best in limited settings, such as the *Independent Reference Model* (which assumes that requests are independent and have fixed probability, thus precluding correlated references) and *Poisson distributions* for object accesses (which assumes that the interarrival times for object access follow the exponential distribution). Yin, Cao, and Cai also proposed an equivalent to Min-Saud,[5] but their design (like Min-Saud's) requires cooperation from the server to set values for several included tunable parameters. Obviously, for a client that frequently disconnects or roams inside a cellular system's coverage area, maintaining these parameters is problematic.

Hara[6] has proposed a cooperative caching policy for mobile ad hoc networks, but it is computationally demanding and assumes knowledge of the network topology.

Researchers have proposed numerous replacement policies for variable-size objects in point-to-point settings,[7] but most of these are tailored for Web proxies. Moreover, our testing[8] has shown that these policies tend to favor either recently accessed (as in the Greedy Dual-Size algorithm[9]) or frequently accessed (as with the Dynamic Aging Least-Frequently-Used policy) objects. In addition, such policies usually favor small objects. When trying to combine recency and frequency, these algorithms tend to employ hard-to-set tunable parameters. Moreover, none of the proposed replacement policies are suitable for wireless environments.

References

1. S. Acharya et al., "Broadcast Disks: Data Management for Asymmetric Communications Environments," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, ACM Press, 1995, pp. 199–210.
2. S. Khanna and V. Liberatore, "On Broadcast Disk Paging," *SIAM J. Computing*, vol. 29, no. 5, 2000, pp. 1683–1702.
3. L. Tassiulas and C. Su, "Optimal Memory Management Strategies for a Mobile User in a Broadcast Data Delivery System," *IEEE J. Selected Areas in Comm.*, vol. 15, no. 7, 1997, pp. 1226–1238.
4. J. Xu et al., "Performance Evaluation of an Optimal Cache Replacement Policy for Wireless Data Dissemination," *IEEE Trans. Knowledge and Data Eng.*, vol. 16, no. 1, 2004, pp. 125–139.
5. L. Yin, G. Cao, and Y. Cai, "A Generalized Target-Driven Cache Replacement Policy for Mobile Environments," *Proc. Int'l Symp. Applications and the Internet* (SAINT '03), IEEE CS Press, 2003, pp. 14–21.
6. T. Hara, "Cooperative Caching by Mobile Clients in Push-Based Information Systems," *Proc. Int'l Conf. Information and Knowledge Management* (CIKM '02), ACM Press, 2002, pp. 186–193.
7. H. Bahn et al., "Efficient Replacement of Nonuniform Objects in Web Caches," *Computer*, vol. 35, no. 6, 2002, pp. 65–73.
8. D. Katsaros and Y. Manolopoulos, "Caching in Web Memory Hierarchies," *Proc. Symp. Applied Computing* (SAC 04), ACM Press, 2004, pp. 1109–1113.
9. P. Cao and S. Irani, "Cost-Aware WWW Proxy Caching Algorithms," *Proc. Usenix Symp. Internet Technologies and Systems* (USITS 97), Usenix Assoc., pp. 193–206.

knowledge of the server's schedule or are too heavily parameterized to adapt easily to changing access patterns (see the sidebar, "Related Work in Caching for Wireless Environments"). To meet the demands of mobile environments, a replacement policy should

- discriminate between objects that are likely to be accessed in the near future and those that are not;
- refrain from using (locally estimated or server-supplied) tunable parameters, which don't adapt well to changing access distributions;
- fairly and effectively account for variable object sizes; and
- refrain from making any assumptions about the server's schedule format because mobile clients will move to areas covered by diverse servers.

To satisfy these goals, we developed a generic cache-replacement policy to facilitate mobile-client caching without making ad hoc assumptions about the network's topology or architecture. SliCache is a self-tunable policy that requires neither knowledge of the server's schedule nor maintenance of any tunable parameters. It combines access *rate* and *recency* metrics to quantify the clients' preferences for objects in the client-side cache. It employs intelligent slicing of the cache space and novel methods for selecting the *replacement victim* — the object purged from the cache when space is needed. In this article, we describe SliCache's design and present performance results that show significant improvements over other policies in terms of *average stretch* — the ratio of an object's access latency to its service time.

The SliCache Policy

In designing our proposed replacement policy, we were guided by a few simple principles. First, we make no attempt to predict the server's *schedule* (that is, the retrieval delay for object delivery) because any successful client-side predictions will cause the server to change its schedule to respond to the modified aggregate access pattern created by object caching. We also want to quantify the client's transient and steady preference for certain objects, but without asking for assistance from the server. Quantifying this preference is a complicated task. The replacement policy should gracefully combine the object's access rate and the time the client most recently accessed it. Finally, the policy must consider the variable object size in order to optimize the stretch metric.

SliCache partitions the cache space into two segments (slices) that are used to accommodate cached objects:

- The *recency-slice* (R-slice) isolates objects that appear only once — so-called "one timers" — in the whole requests stream.
- The *interaccess-slice* (I-slice) stores the objects that comprise the client's working set — those that the client has accessed most frequently or recently.

The slices grow and shrink deliberately according to the request stream's characteristics. SliCache uses different ranking functions in the slices to measure the benefit of keeping an object in cache. Objects accessed at least twice while in the cache reside in the I-slice until they are evicted, whereas objects accessed only once are kept in the R-slice. We call the difference between the current time t_c and the time the client last accessed a given object, the recency interval; the interval between the client's penultimate and most recent access of an object is the *last-interaccess* interval.

Ranking Objects in SliCache

The R-slice employs a ranking function based on the ratio of the object's recency to its size — a simple heuristic designed to accommodate many recently accessed objects. The I-slice's ranking function takes the product of the object's last-interaccess interval and its recency. By using the last interaccess as a measure of access frequency, we assume that the interarrival time till the next request will be drawn from the same distribution as the times observed so far. Thus, we can use the observed interarrival times to decide the popularity of an object that is suited to the individual client. Note that we don't consider size in the I-slice's criteria because we want to avoid any bias toward small objects, which would aggravate the overall stretch.

Figure 1 illustrates the possible arrangement of

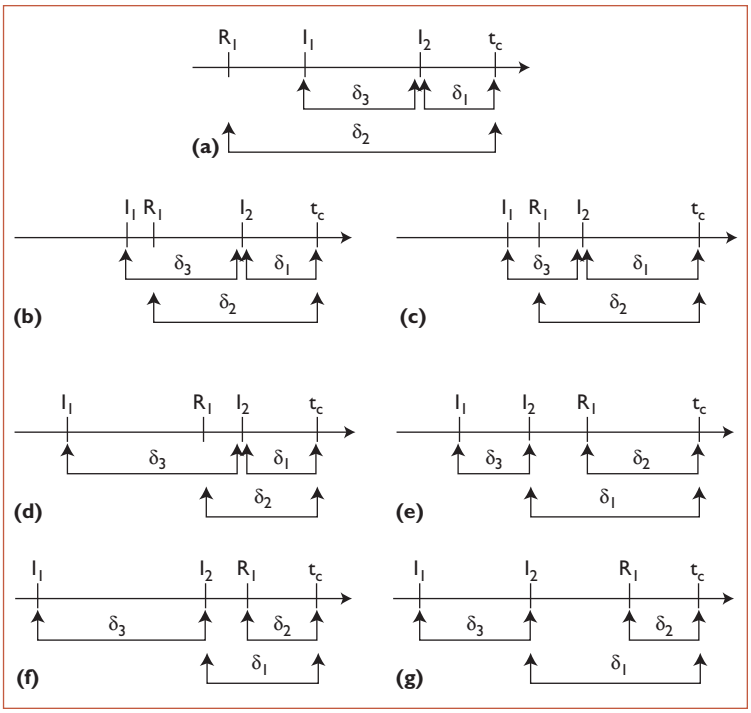

Figure 1. Access to candidate victims, with varying lengths for the R-victim's recency interval δ_2 and for the I-victim's recency interval δ_1 and last-interaccess interval δ_3. (a) Because the access to the R-victim happens before the two accesses to the I-victim, we favor the I-victim and purge the R-victim. (b) The client accesses the R-victim between its two accesses to the I-victim (relative lengths: $\delta_1 < \delta_3 < \delta_2$). Thus, we purge the R-victim. (c) The client accesses the R-victim between its two accesses to the I-victim (relative lengths: $\delta_3 < \delta_1 < \delta_2$), so we again favor the I-victim. (d) The client accesses the R-victim in-between its two accesses to the I-victim (relative lengths: $\delta_1 < \delta_2 < \delta_3$). Thus, we favor the I-victim because it maintains its popularity ($\delta_1 < \delta_3$) and has been accessed more recently than the R-victim ($\delta_1 < \delta_2$). (e) The client accesses the R-victim after its two accesses to the I-victim (relative lengths: $\delta_3 < \delta_2 < \delta_1$). Again, we favor the R-victim because the I-victim loses its popularity ($\delta_1 > \delta_3$) and has been accessed less recently than the R-victim ($\delta_1 > \delta_2$). (f) The client accesses the R-victim after its two accesses to the I-victim (relative lengths: $\delta_2 < \delta_1 < \delta_3$). Thus, we favor the I-victim because it maintains its popularity ($\delta_1 < \delta_3$). (g) The client accesses the R-victim after its two accesses to the I-victim (relative lengths: $\delta_2 < \delta_3 < \delta_1$). As a result, we purge the I-victim because it loses its popularity ($\delta_1 > \delta_3$) and has been accessed less recently than the R-victim ($\delta_1 > \delta_2$).

accesses to the two highest-ranking objects in a slice. We select the replacement victim from these *candidate victims*. The one originating from the R-slice, we call the *R-victim*; the one originating from the I-slice, we call the *I-victim*. Let R_1 be the access time of the R-victim. Let I_1 be the time of the penultimate reference to the I-victim and I_2 be the time of the last reference to it. Let t_c be the current time; δ_1 is the recency interval of the I-victim ($\delta_1 = t_c - I_2$); δ_2 is the recency interval of the R-victim ($\delta_2 = t_c - R_1$); and δ_3 is the last-interaccess interval of the I-victim ($\delta_3 = I_2 - I_1$).

In Figure 1a, the R-victim was accessed before the two accesses to the I-victim. In Figures 1b, 1c, and 1d, the access to R-victim happened between the two accesses to the I-victim. In Figures 1e, 1f, and 1g, the access to R-victim happened after the two accesses to the I-victim. The primary difference between Figures 1b, 1c, and 1d is the relative lengths of the intervals δ_1, δ_2, and δ_3. (Figures 1e, 1f, and 1g share a similar relationship).

In order to select which object will be purged (the R-victim or I-victim), we must estimate whether the I-victim is gradually losing its popularity, while estimating the R-victim's potential for a second reference. To do so, we compare the intervals between δ_1 and δ_3 and between δ_1 and δ_2. A comparison between δ_1 and δ_2 estimates the victims' short-term temporal locality (due to correlated accesses or transient preference to it). By comparing δ_1 and δ_3, we can estimate the I-victim's long-term (steady-state) popularity.

Among the scenarios in Figure 1, we favor the I-victim in all cases in which $\delta_1 < \delta_3$ — that is, in a, c, and e — which indicates that the object maintains its popularity. When $\delta_1 > \delta_3$, we favor the I-victim only when $\delta_1 < \delta_2$, as in Figure 1c. (This lets us protect the cache from objects the client is likely to request only once.) Note that we also favor the I-victim when $\delta_2 > \delta_1$ and $\delta_2 > \delta_3$ (as in Figure 1a) because we consider the R-victim a one-timer.

Implementation Issues

In our work with SliCache, we faced two critical implementation issues:

- Measuring time for estimating the interaccess and recency intervals, and
- Managing the metadata recorded for each cached object.

To solve the former, we decided to measure interaccess and recency intervals using *virtual time*, which we calculate from the number of requests posed by the client to the server: after each request, a virtual time clock (a counter created by SliCache and initialized to zero) in each client cache advances by one time unit. The counter is reset to zero every time the cache empties. Different clients' time clocks will reflect different virtual times, depending on the number of requests they've made since they started operation. (Note that our virtual clocks are not related, for instance, to network synchronization clocks; they simply count client requests. In this way, we avoid the trouble of synchronization caused by the client disconnections.)

For each cached object o_i, we need to track its size s_i, the time t_l of the last reference to it, and the time t_p of the penultimate reference to it. The metadata are kept in two separate *max-heaps* — binary trees that store sets of keys in such a way that the key with the maximum value is always found at the max-heap's root. In SliCache, the *R-heap* stores entries for the R-slice objects, and the *I-heap* stores entries for the I-slice objects. The R-heap's sorting key is the ratio

$$\frac{-t_l}{s_i},$$

whereas the I-heap's sorting key is

$$\frac{-1}{(t_c - t_l) * (t_l - t_p)}.$$

The latter's dependence on the current time increases the time complexity, though not significantly; the `build-heap` operation (which transforms a binary tree that doesn't obey the max-heap property into a max-heap) is invoked only on the relatively small I-heap, which holds only the client's working set of objects. In case of ties, we favor the most recently accessed object.

Figure 2 shows our implementation of the SliCache policy. If a requested object o_i is larger than the total cache space ($s_i > cs$), SliCache doesn't consider it for caching. If SliCache finds the object in the I-slice, it simply records the object's updated last-interaccess time and last access time to the heaps (via the `update-statistics` command). If we find the object in the R-slice, SliCache promotes it to the I-slice — the working-object cache. (We assume that an insertion to a slice always records the appropriate metadata — that is, last access time and penultimate access time if the objected was accessed twice or more.) If the object is larger than the free cache space ($s_i > a$), we must evict some objects by identifying

the R-victim and I-victim.

Given that we use max-heaps and that the R-slice sorting key doesn't depend on the current time, the R-victim is always found at the top of the R-heap. For the I-heap, however, we must update the sorting-key values for all objects using the `I-heap-build-heap` command. We then call `evictOne` to purge as many objects as needed to accommodate o_i. Note that `I-heap-build-heap` executes at most once for each call of SliCache, regardless of how many times `evictOne` is called.

Performance Evaluation

We conducted a series of tests comparing SliCache to several other state-of-the-art policies:

- the *Least-Recently-Used* (LRU) caching policy expunges from the cache the object that was least recently referenced,
- the *LRU-k* policy, a generalization of LRU, evicts the object whose *k*-th reference is furthest in the past,
- the *PIX*[1] policy evicts the object with the smallest ratio of access-frequency to broadcast-frequency, and
- the *minimum stretch integrated with access rates, update frequency, and cache validation delay*[2] (Min-Saud) policy can be considered a generalization of PIX that also takes into account the object's size and update frequency.

All of these policies operate on the clients' cache and maintain various data (object size, popularity, cache-entry time, and so on), which they use to make replacement decisions. To evaluate and compare performance among them, we used a simulation model with a single-cell environment, in which one server serves multiple clients. In the rest of this section, we describe how we implemented the considered replacement policies.

The PIX policy is tightly related to the Broadcast-Disks paradigm introduced by Acharya et al.[1] PIX requires knowledge of the disk to which the object belongs, and thus, of its exact broadcast frequency. In our tests, we approximated this frequency with the well-known exponential-smoothing-based formula, which estimates the new value for a variable as a weighted average of its past values. We implemented the Min-Saud policy without its object-update consideration — that is, with negligible object-validation delay. We set all other parameters as indicated by Xu et al.[2] Real, publicly available Web-request streams, such as those at the Internet Traffic Archive (http://ita.ee.lbl.gov/), are limited in that each client performs only a few requests. Moreover, all such streams exhibit the same highly skewed (Zipfian) access pattern, in which a few objects are responsible for the great majority of accesses (following what is known as Zipf's law). Instead, we used synthetically generated data to test the policies for

```
Algorithm SliCache
// Cache space = cs.
// Free cache space = a.
// Request object o_i of size s_i.
begin
if( s_i > cs )     return;
if( I-heap-inHeap(o_i) )
    update-statistics;
else if( R-heap-inHeap(o_i) )
    R-heap-remove(o_i);
    I-heap-insert(o_i);
else
    if( s_i > a )
        I-heap-build-heap();
        while( a < s_i )
            evictOne();
    R-heap-insert( o_i );
    a - = s_i;
end

procedure evictOne
// δ1: recency interval of I-victim (of d_I).
// δ2: recency interval of R-victim (of d_R).
// δ3: last interaccess interval of I-victim (of d_I).
begin
if( empty(I-heap) )
     finalVictim= R-heap-extract-max();
else if( empty(R-heap) )
     finalVictim= I-heap-extract-max();
else
     d_R = R-heap-return-max();
     d_I = I-heap-return-max();
     if( (δ1 > δ3) AND (δ1 > δ2) )
         finalVictim= I-heap-extract-max();
     else
         finalVictim= R-heap-extract-max();
a + = s_finalVictim
end
```

Figure 2. Pseudocode for the SliCache policy. The `evictOne` *procedure selects the replacement victim, whereas SliCache is responsible for placing the cached objects into the correct slice.*

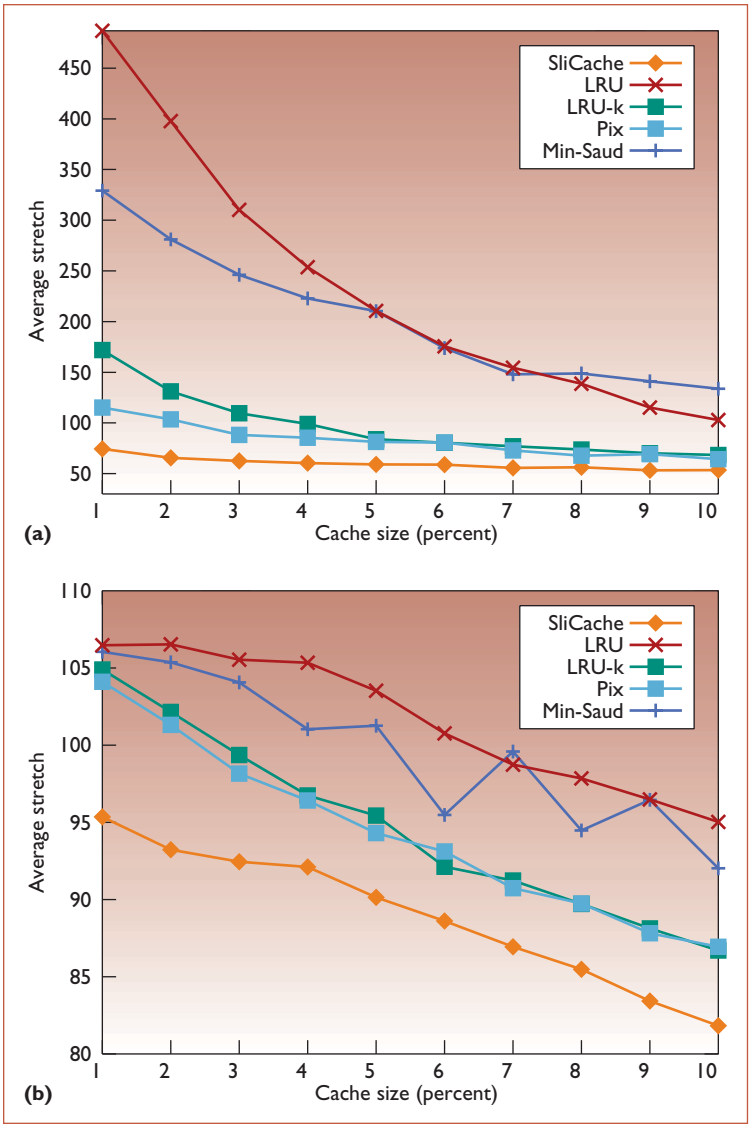

Figure 3. Average stretch versus cache size. (a) For the increasing size distribution, the average stretch achieved by SliCache is up to 36 percent better than PIX, the second-best performing policy. (b) For the decreasing size distribution, SliCache achieves stable 10-percent performance gains over its competitors.

various data and access distributions (uniform and Zipfian) and for steady-state caches.

System Parameters

Our simulated system is an *infrastructured* wireless system consisting of a single server (base station) and several clients roaming inside the cell it serves. The base station communicates via a wired network to database servers or other stations. This architecture supports an *uplink channel*, through which mobile clients place their requests, and a *broadcast channel* (1,000 Kbps in our tests) through which the server transmits requested objects from the single queue it maintains. The client keeps listening to the broadcast channel until it gets the requested objects.

The database is a collection of DB objects and is partitioned into Regions-Group (RNG) disjoint regions that each hold an equal number of objects. By default, the database is set to 10 regions that store up to 200 objects apiece. The default cache size is equal to 5 percent of the database size. The access probability p_i for each region is determined by a *Zipfian distribution* with parameter θ (set to 0.90 by default):

$$p_i = \frac{(1/i)^{\theta}}{\sum_{i=1}^{n}(1/i)^{\theta}}, 1 \le i \le RNG$$

Objects range in size from *minSize* (2 Kbytes) to *maxSize* (2,000 Kbytes). Within a given region, all objects have an equal probability of being accessed. Again following Xu et al., we considered the following distributions:

- Increasing (Incrt): $size_i = minSize + \frac{(i-1) * (\max Size - \min Size + 1)}{DB}$,

  where $1 \le i \le DB$.
- Decreasing (Decrt): $size_i = maxSize - \frac{(i-1) * (\max Size - \min Size + 1)}{DB}$,

  where $1 \le i \le DB$.

The smaller objects are more popular for the Incrt distribution, whereas the larger objects are more popular for the Decrt distribution. Due to space restrictions, we focus on the Incrt results here, but the Decrt results are similar.

All 50 simulated clients follow the same access pattern, generating requests for objects according to the Zipfian distribution and never disconnecting. This setting — homogeneous clients and no think time (between query response and next request) or disconnection time — is ideal for the PIX and Min-Saud policies because it allows them to continuously monitor the broadcast channel for the current parameter values supplied by the server. Moreover, the tunable parameter values remain stable and relatively accurately predictable because the clients have identical and stable access pat-

terns. This is significant because it lets PIX and Min-Saud, which heavily rely on tunables, achieve their best performance.

The server in our tests ran the R × W scheduling algorithm,[3] which takes into account each object's popularity and the time since the last broadcast in selecting which object to broadcast next. R × W performed best for a large number of data and access distributions, achieving the lowest average and worst access time among its competitors. Client requests were buffered at the server, whose queue we assumed to be infinite, thus ensuring that the scheduling algorithm knows the exact number of requests for each object in order to make scheduling decisions. We further assumed zero latency for fetching objects from cache. We obtained results for the system in stable state; that is, each client finished at least 4,000 queries after its cache was full, thus eliminating the warm-up effect on the client cache and broadcast channel.

To measure the cache's efficiency, we measured requests' average stretch.[4] Neither hit ratio nor byte-hit ratio is appropriate in wireless mobile Web environments because the latency for retrieving objects depends on the broadcast schedule; these two metrics can overestimate the cache's performance in wireless environments. Average latency — elapsed time between the start of a request and the end of the response to it — is also an unfair metric because it disregards the variable object size, which translates into variable service time.

Experimental Results

We examined our proposed caching scheme's performance for varying cache sizes and *skew* — which measures the relative preference to each object — in the access distribution (zero skew implies a uniform access pattern), as well as for different client access patterns and different server schedules. Policies (such as Min-Saud) normalize an object's caching profit according to its size, thus showing a bias toward small files and completely failing when this preference is absent. SliCache, on the other hand, provides a graceful balance between recency and frequency, while treating small and large objects evenly (in the I-slice) and thus minimizing the stretch metric. Moreover, its implementation cost is low in terms of space reserved for metadata and processing time; our experiments showed that the average number of cached objects in the I-slice is equal to 23 percent of the total number of cached objects.

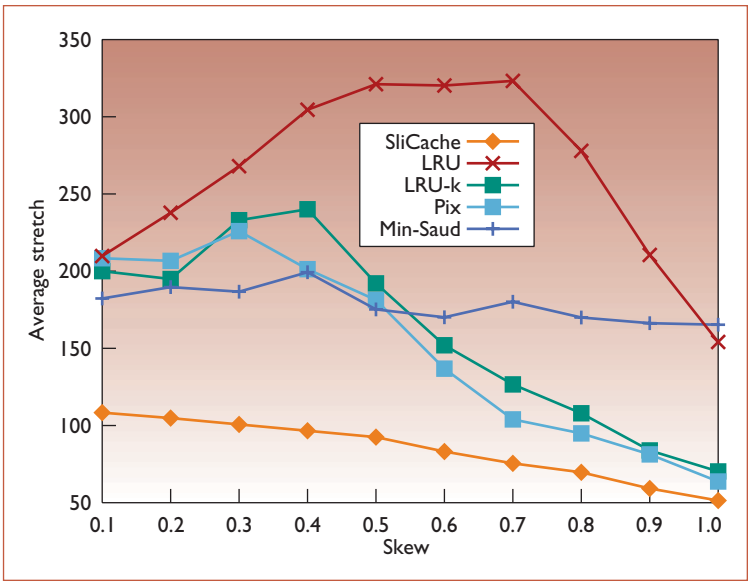

Figure 4. Average stretch versus Zipfian skew. As the skew increases (such that fewer objects are responsible for greater percentage of access), all policies achieve lower average stretch. The results show that SliCache is the clear winner, especially for low and moderate skew.

Performance for various cache sizes. The graphs in Figure 3 illustrate the replacement policies' performance, measured as the average stretch they achieve with varying cache size. Figure 3a shows the average stretch for *increasing size distribution* (when clients show a preference toward smaller objects), whereas Figure 3b shows the average stretch for *decreasing size distribution* (when clients prefer larger objects). From Figure 3, we see that SliCache can reap significant performance benefits with even very small caches. It shows 10- to 36-percent performance gains over PIX (the second-best performing policy) for increasing size distribution (see Figure 3a), and stable 10-percent gains for decreasing size distributions (see Figure 3b). Stretch decreases monotonically with increasing cache size for all policies except Min-Saud. This happens because Min-Saud purges some large but popular objects from cache to accommodate smaller objects — an inherent shortcoming of weighting by object size. This effect has been observed for many Web proxy-replacement policies.[5]

Performance for various skew values. Our next experiment targeted the policies' performance for varying degrees of access skew. Figure 4 presents the results obtained as the Zipfian distribution's θ parameter varies from 0.1 (almost uniform) to 1 (highly skewed). This figure shows results analo-

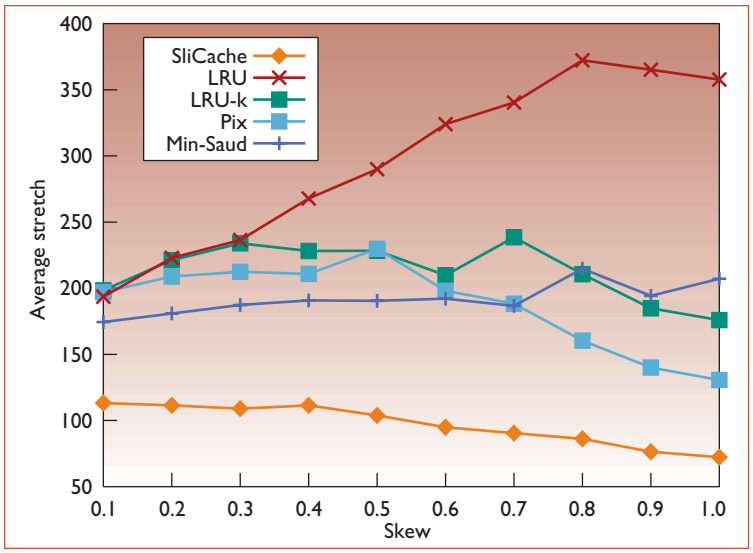

Figure 5. Average stretch versus access skew for heterogeneous clients. The performance gap between SliCache and its competitors widens, especially for moderate and high skew values, compared to their performance with homogeneous clients. The existence of heterogeneous clients causes an increase in the number of popular objects, which in turn affects the broadcast schedule's composition and stability.

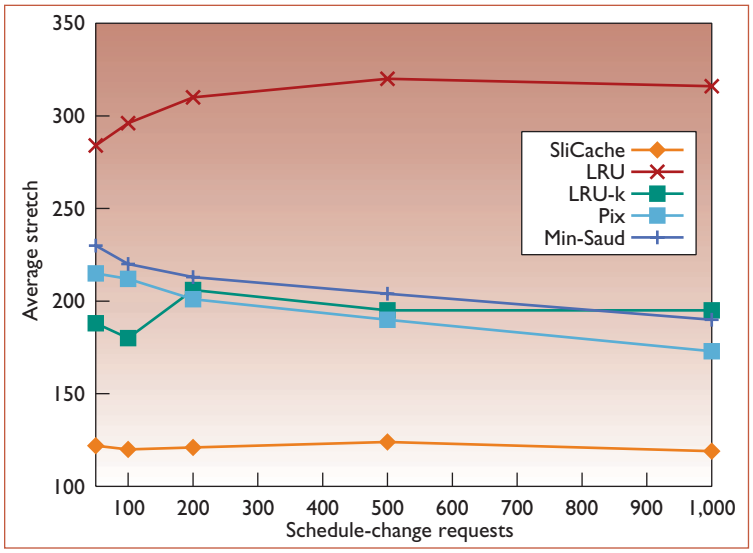

Figure 6. Average stretch versus number of serviced requests before broadcast schedule changes. The server switches between R × W and MRF schedules, thus making the prediction of an object's broadcast frequency quite difficult, affecting the policies that rely on such predictions.

gous to those we obtained for various cache sizes: SliCache performs considerably better than its competitors. Indeed, it shows gains of more than 25 percent for small skew values and more than 10 percent for larger values. For low skew values (0.5 or less), the Min-Saud performs better than PIX; for larger skew values, PIX does better.

Performance for heterogeneous clients. Figure 5 shows the results for testing the policies with non-homogeneous clients and increasing size distribution. For this experiment, we considered three groups, containing the same number of clients. The first group follows the increasing size distribution; thus, its most preferred objects are in the first region; the second group shows a preference shifted at RNG/3, relative to the first group; the third group shows a preference shifted at RNG/3, relative to the second group.

Because the first group of clients follows the increasing size distribution, its most popular items are in region 1, followed by those in region 2, and so on. Therefore, the most popular items for the j-th group ($2 \leq j \leq 3$) are those in region $\lfloor (j - 1)*RNG/3 \rfloor$, followed by the items in region $\lfloor (j - 1)*RNG/3 \rfloor + 1$, and so on (where *RNG* denotes the total number of disjoint collections of database objects, as defined earlier).

Because we considered an increasing size distribution, for which the first of a total of *RNG* disjoint contains smaller objects, it is obvious that the first group of clients shows a preference for small objects; the third group is biased toward large objects, and the second group shows a preference for medium-sized objects. We observe that there is a wider performance gap for moderate and high skew values between SliCache and the second-best performing policy (the Min-Saud for low skew and PIX for high skew values) in this case than there was with homogeneous clients. We expected this because the existence of heterogeneous clients causes an increase in the number of popular objects, which in turn affects the broadcast schedule's composition and stability. Unlike Min-Saud and PIX, SliCache caches the per-client popular objects rather than striving to predict the broadcast schedule.

Performance for varying server schedules. We also ran tests using the policies with a server with varying schedule, homogeneous clients, and moderate access skew (0.55). The server switched between the R × W and *most requests first* (MRF) scheduling algorithms after every λ requests. Figure 6 shows the performance results for different values of λ, illustrating the vulnerability of PIX and Min-Saud to the change of the scheduling policy, as well as SliCache's insensitivity to this change. LRU and

LRU-k showed the worst performance for larger λ values because these policies quickly purge infrequently accessed objects from cache. Adding to their problems, MRF broadcasts the infrequently accessed objects rarely, thus degrading the performance of these policies.

Conclusions

In this article, we did not consider integrating cache-coherency issues into our caching policy. Yet, our unmodified policy can be effectively combined with a strong cache-consistency technique, such as invalidation reports,[6] in which the server periodically sends out lists of modified objects and the cache never serves an object without reading the invalidation report first. Alternatively, the cache could employ proactive mechanisms such as prefetching.[7]

In the future, we plan to investigate how to exploit information, such as time-to-live, that is associated with each object. We would like to integrate this into SliCache's replacement decisions in order to achieve weak cache consistency.

Acknowledgments

This research was funded through the bilateral program of scientific cooperation between Greece and Turkey (Γ.Γ.Ε.Τ. and from TUBITAK grant number 102E021).

References

1. S. Acharya et al., "Broadcast Disks: Data Management for Asymmetric Communications Environments," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, ACM Press, 1995, pp. 199–210.
2. J. Xu et al., "Performance Evaluation of an Optimal Cache Replacement Policy for Wireless Data Dissemination," *IEEE Trans. Knowledge and Data Eng.*, vol. 16, no. 1, 2004, pp. 125–139.
3. D. Aksoy and M. Franklin, "R × W: A Scheduling Approach for Large-Scale On-Demand Data Broadcast," *IEEE/ACM Trans. Networking*, vol. 7, no. 6, 1999, pp. 846–860.
4. S. Acharya and S. Muthukrishnan, "Scheduling On-Demand Broadcasts: New Metrics and Algorithms," *Proc. IEEE/ACM Conf. Mobile Computing* (MobiCom '98), ACM Press, 1998, pp. 43–54.
5. D. Katsaros and Y. Manolopoulos, "Caching in Web Memory Hierarchies," *Proc. Symp. Applied Computing* (SAC '04), ACM Press, 2004, pp. 1109–1113.
6. G. Cao, "A Scalable Low-Latency Cache Invalidation Strategy for Mobile Environments," *IEEE Trans. Knowledge and Data Eng.*, vol. 15, no. 5, 2003, pp. 1251–1265.
7. A. Nanopoulos, D. Katsaros, and Y. Manolopoulos, "A Data Mining Algorithm for Generalized Web Prefetching," *IEEE Trans. Knowledge and Data Eng.*, vol. 15, no. 5, 2003, pp. 1155–1169.

Dimitrios Katsaros is a PhD candidate in computer science at Aristotle University, Greece. His research interests include caching, replication, prefetching, and content delivery over the Internet, data management and delivery for mobile and pervasive computing, and data mining. Katsaros received a BSc in Informatics from Aristotle University. He is coeditor of *Wireless Information Highways* (Idea Inc., to be published in 2005). Contact him at dkatsaro@csd.auth.gr.

Yannis Manolopoulos is a professor in the Department of Informatics at Aristotle University. His research interests include access methods and query processing for databases, data mining, and performance evaluation of storage subsystems. Manolopoulos received both a BEng in electrical engineering and a PhD in computer engineering from Aristotle University. He has published more than 140 papers in refereed scientific journals and conference proceedings. He is vice-chair of the Greek Computer Society as well as a member of the IEEE and the ACM. Contact him at manolopo@skyblue.csd.auth.gr.