Using Latency-Recency Profiles for Data Delivery on the Web

Laura Bright   Louiqa Raschid

University of Maryland
College Park MD 20742
bright@cs.umd.edu, louiqa@umiacs.umd.edu

Abstract

An important challenge to web technologies such as proxy caching, web portals, and application servers is keeping cached data up-to-date. Clients may have different preferences for the latency and recency of their data. Some prefer the most recent data, others will accept stale cached data that can be delivered quickly. Existing approaches to maintaining cache consistency do not consider this diversity and may increase the latency of requests, consume excessive bandwidth, or both. Further, this overhead may be unnecessary in cases where clients will tolerate stale data that can be delivered quickly. This paper introduces latency-recency profiles, a set of parameters that allow clients to express preferences for their different applications. A cache or portal uses profiles to determine whether to deliver a cached object to the client or to download a fresh object from a remote server. We present an architecture for profiles that is both scalable and straightforward to implement at a cache. Experimental results using both synthetic and trace data show that profiles can reduce latency and bandwidth consumption compared to existing approaches, while still delivering fresh data in many cases. When there is insufficient bandwidth to answer all requests at once, profiles significantly reduce latencies for all clients.

**Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002**

1 Introduction

Recent technological advances and the rapid growth of the Internet have increased the number of clients who access objects from remote servers on the Web. Remote data access is often characterized by high latency due to network traffic and remote server workloads. Data at remote servers is often characterized by frequent updates. For example, stock quotes may be updated as frequently as once per minute[12].

The increased popularity of the Internet has resulted in a corresponding increase in the diversity of both the clients and the types of data and services that they request. As a consequence of this increased diversity, clients may have varying preferences about the recency and latency of their requests. For example, a stock trader who requires accurate stock information may be willing to wait longer for the most recent data. On the other hand, if there is heavy congestion on the Internet or at a remote server, a client reading the news may accept slightly stale data if it can be delivered quickly.

To date, there have been many caching solutions developed to improve access latencies and data availability on the Web. However, the effectiveness of all these approaches is limited by cached data that becomes stale as updates are made at remote servers. In addition to the more traditional technologies such as proxy caching, newer technologies such as web portals and application servers also utilize caching. At present, techniques used to keep cached objects up-to-date cannot handle diverse client preferences and may perform poorly with respect to either latency, recency, or bandwidth consumption. To handle the popularity and diversity of the web and to scale to the large number of clients and data sources, it is crucial to develop web caching technologies that can accomodate diverse client preferences with respect to recency and latency.

This paper introduces *latency-recency profiles* to address this limitation of current web technologies. We first describe some existing caching technologies to improve latency and availability. We then present consistency mechanisms that are currently used by these technologies, and discuss their shortcomings. Finally, we introduce latency-recency profiles, our flexible, scalable solution that allows clients to express their preferences with respect to latency and recency of data.

1.1 Web Caching Technologies

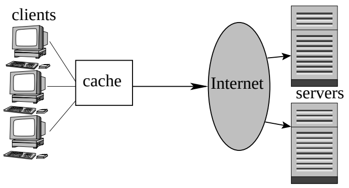

Figure 1: Proxy Cache Architecture

Proxy Caches

Client-side proxy caching is a widely used technique to reduce access latencies on the Web [7, 10, 20]. In this architecture, a cache resides between a group of clients, e.g., a company or university campus, and the Internet. This architecture is shown in Figure 1. A proxy cache stores objects previously requested by clients, and these cached objects may be used to serve subsequent requests. Since multiple clients access objects through the proxy, a proxy cache can leverage commonalities in client requests and reduce access latencies.

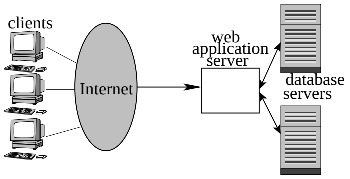

Figure 2: Application Server Architecture

Application Server Caches

Application servers are an emerging technology to improve the performance of data intensive web sites by offloading some functionality from web servers. Many commercial products are available, e.g., Oracle9iAS Web Cache[4], IBM WebSphere[29], and BEA WebLogic[28]. Application servers are well-suited for large scale data handling, and can perform caching to further improve performance. For example, an application server cache can reside between database server and the Internet, and cache components of dynamically generated web pages. The application server can then automatically generate pages without contacting the database server. The Oracle Application Server Web Cache [4] is an example of a product with this functionality. We note that there is typically some cooperation between the database server and the application server, and there may be a high-bandwidth link connecting them. The architecture is shown in Figure 2.

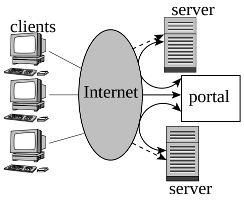

Figure 3: Portal Architecture

Web Portals

Portals make it easier for clients to locate and access relevant information. Earlier portals served primarily as "hubs" to direct users to relevant web sites. Today, however, many portals cache data gathered from other sites, so clients can easily access all relevant information from a single site. According to a recent study by Booz-Allen & Hamilton[26], an increasing number of portals serve as destinations themselves, rather than gateways to other sites, and 60% of Web sessions include at least one access to a portal site. The web portal architecture is shown in Figure 3. The challenge of providing up-to-date data is critical to the success of a portal, in particular, since clients will only visit portals that meet their needs. Any portal that fails to do so risks losing clients to competitor sites. Therefore, it is crucial for portals to keep cached data up-to-date while meeting client preferences with respect to latency and recency.

A challenge for all these technologies is that cached data becomes stale as updates are made at remote servers. To date, several techniques have been proposed to keep cached data consistent with data at remote servers. Most implementations of proxy caches, portals, and application server caches use some combination of these. However, none of these techniques consider diverse client preferences. We describe these existing consistency approaches and discuss their limitations. In the following discussion, we use the generic term "cache" to refer to any of the above technologies.

1.2 Existing Consistency Approaches

Time-to-Live (TTL)

An accepted strategy for maintaining the consistency of cached objects is to assign each object a time-to-live (TTL) [8, 11, 17], i.e., the estimated length of time the object will remain fresh. If the TTL of a requested object has expired, the cache must *validate* the objects (check for updates) at the remote server. Note that this approach requires no cooperation from remote servers. While TTL delivers more recent data to clients, validation adds overhead to client requests and reduces the benefits of caching. Further, it is difficult to accurately estimate an object's TTL. An estimate that is too conservative will improve freshness but result in many unnecessary validations at remote servers, while an estimate that is too optimistic reduces contact with remote servers but may result in many clients receiving stale data. TTL is the most commonly used mechanism in proxy caches, and may also be used by web portals.

Always-Use-Cache (AUC)

Another strategy is to serve all requests from a cache, and perform prefetching in the background to keep cached objects up to date. We refer to this approach as Always Use Cache (AUC). Prefetching strategies to maximize the overall recency of a cache are described in [9]. This approach has several limitations. First, in the general case where there is no cooperation from remote servers, the cache has no knowledge of when updates occur. Therefore, AUC typically must poll remote servers to keep cached data up to date. This may consume large amounts of bandwidth and does not scale well to large numbers of objects. Also, there may be some delay between when the update occurs and when the cache checks for updates. During this time, the cache will return stale data. Therefore, while AUC minimizes the latency of client requests, it may perform poorly with respect to recency and consume large amounts of bandwidth, since it does not scale well to large caches or frequently updated objects. AUC is commonly used by web portals and other technologies that maintain copies of objects from many web sites.

Server-Side Invalidation (SSI)

A third approach to maintaining cache consistency is to have servers maintain information about objects stored in client caches, and send invalidation messages to caches when an object is updated. Alternatively, a server may push the updated object to caches. We refer to this approach as SSI. This approach has been studied in the context of proxy caching in [23], and related techniques have been studied by the database community in [2, 19, 25]. SSI requires cooperation from remote servers. When a server sends only invalidation messages to the cache, SSI has performance comparable to TTL assuming TTL estimates are accurate. This approach was shown to be feasible in terms of bandwidth and server load in [23]. However, if servers instead send the updated objects after each update, as is the case with many application server caches[4], the overhead of SSI may be considerably greater.

SSI is often used in application server caches or web portals. It may be feasible in these environments because the number of servers and caches is typically fixed, which facilitates cooperation and reduces scalability concerns. It is unlikely to be a viable alternative in proxy caches because many web servers are either unable or unwilling to implement it.

To summarize, existing approaches to maintain cache consistency may increase the latency of requests, consume excessive bandwidth, or do both. Further, this overhead may be unnecessary in cases where clients will tolerate stale data that can be delivered quickly. A scalable solution that can handle clients and applications with diverse latency and recency requirements is needed.

1.3 Our Solution: Latency-Recency Profiles

In this paper, we present *latency-recency profiles* to overcome the limitations of existing approaches. Profiles comprise a set of parameters that allow clients to express their preferences with respect to latency and recency for their different applications. A client may choose a single profile for all applications. Alternatively, a client can specify a set of application specific profiles. A profile-based downloading strategy (labelled Profile) uses latency-recency profiles to determine whether to download a requested object or to use a cached copy.

Profile is a generalization of TTL and AUC. TTL aims to deliver the most recent data, while AUC aims to minimize latency. The parameters of Profile can be tuned to provide performance anywhere between these two extremes. Profile can also provide an upper bound with respect to latency or recency. Further, unlike SSI, it requires no cooperation from remote servers, and can potentially be used for any data source on the web.

Profile is further distinguished by its scalable architecture. A client's browser can communicate profile parameters to a cache by appending parameters to an HTTP request. Profile incorporates these parameters into the downloading decision. The advantage of this architecture is that it is straightforward to implement. It also eliminates the need for caches to maintain profile information and allows a cache to scale to a large number of clients. Further, clients can easily change their profiles without having to inform the cache.

There are incentives for both clients and caches to support profiles. From the client's perspective, they can reduce access latencies compared to the more traditional TTL when appropriate, while providing greater recency than AUC. From the cache's perspective, profiles can reduce bandwidth consumption compared to TTL, AUC, and SSI, which can reduce the cache's operating costs and can improve access latencies for clients who access data from remote servers.

Another key benefit to using profiles is that they are well-suited to handling brief periods of overload ("surges") which often occur on the web. Surges can be caused by congestion on the link connecting a proxy cache to the internet, or by a large number of requests at a popular server. When there is insufficient bandwidth or server capacity to process all requests during such surges, using profiles can reduce the latency of all requests by downloading only the ones that require the most recent data.

The rest of this paper is organized as follows: Section 2 surveys related work, Section 3 describes the framework for latency-recency profiles. We present experimental results using both synthetic and trace data in Section 4, and conclude in Section 5.

2 Related Work

There has been much research in the web caching community [7, 10, 11, 17, 27] on proxy caching to improve performance on the web. Cache consistency techniques are presented in [8, 11, 17, 23]. [8, 11, 17] present the TTL approach described in the previous section. [23] studies techniques for strong cache consistency, i.e. guaranteeing fresh data. The authors show that server-side invalidation is effective for maintaining strong cache-consistency, however this technique must be implemented by remote servers. [9] considers prefetching locally cached objects to improve the overall recency of a cache. This work assumes that all requests are served from the cache, and does not consider client preferences for recency or latency.

Recent work in the database community considers caching dynamically generated web content [6, 24]. Work in [24] caches components of dynamically generated web pages to exploit overlap in queries. However, this work does not consider updates to the underlying databases, and requires cooperation from remote servers. [6] presents techniques for database-backed web sites to invalidate cached dynamic content. These techniques apply only to servers and are not appropriate for other caching technologies, e.g., client-side proxy caching or portals.

Work in materialized views, e.g.,[16, 18, 30] also considers the problem of keeping data up to date. These works typically assume full knowledge of updates to the underlying database. Work in [14] allows stale data to be incorporated into materialized views by adding an obsolescence cost, and shares our goal of allowing clients to accept stale data in exchange for lower latencies. WebView materialization [21, 22] addresses the problem of computing materialized views for web-accessible databases. Efficient strategies for the database to propagate updates to the WebView are presented in [22]. Our problem differs from [21, 22] since we address updating cached objects at the client side, when we do not know exactly when objects are updated at remote servers.

There is also relevant work in the database community [2, 19, 25] that relaxes the requirement that cached copies be consistent with objects that reside on remote servers. These works allow cached copies of objects to deviate from the data at the server in a controlled way. This requires servers to propagate updates to the client-side cache when a cached value no longer has an acceptable degree of precision; which places a burden on servers and does not scale well to a large number of clients.

3 Latency-Recency Profiles

Our latency-recency profiles allow clients to express their preferences for their applications using a few parameters. Profiles are set individually by each client, and a single client can specify either a single profile or different profiles for different applications. We first discuss several key issues that are crucial to successfully implementing and using profiles. We then describe how clients can choose target latency and recency values, and present a parameterized decision function that can capture the latency-recency tradeoff for a particular client or application. This is the basis of our approach labelled Profile. Finally, we briefly discuss upper bounds provided by our function, and describe how the parameters can be tuned to meet client requirements with minimal overhead for the clients.

3.1 Issues

There are several issues that are important to successfully implementing and using client profiles at a cache. The first issue is *scalability.* An implementation of profiles that requires a cache to store detailed information about each client would add considerable overhead because clients would need to register profiles with the cache, and the cache would need to keep the information up to date. This does not scale well to large numbers of clients. While our solution Profile exploits knowledge of profiles, it does not require the cache to store any profile information. Browsers append the profile parameters to client HTTP requests, and Profile uses these parameters in the decision function. Thus, Profile can easily scale to a large number of clients, with no additional communication overhead between the client and the cache. This scalability is a key benefit to using a parameterized function.

A second issue is *flexibility.* Clients should be able to specify profiles that are appropriate for each of their applications, and they should be able to easily adjust their profiles as needed. To allow clients to use different profiles for different applications, clients can choose a default profile which they can override for specific domain names or URLs. For example, a client requiring the most recent stock quotes may specify that all requests to the domain `finance.yahoo.com` [12] require the most recent data, but that all other requests can tolerate up to 1 update. Clients can easily change their profiles using their browser, without communicating with the cache.

The third issue is *ease of implementation.* It is straightforward to modify a cache to implement Profile. Profile allows clients with diverse profiles to share a cache without adding any overhead to each other's requests. For each individual request, the cache will use Profile to choose how to serve the request based on that client's profile. If clients with different profiles request the same object simultaneously, the cache could serve one client's request from the cache while downloading a fresh copy for the other client.

The final issue relates to *guarantees.* Profile is a generalization of TTL, which guarantees fresh data (assuming TTL estimates are accurate) and AUC, which guarantees low latency. In addition, Profile can support upper bounds on either latency or recency, which other approaches do not support.

3.2 Parameters

Profiles include the following parameters:
Target Latency: The first parameter is a *target latency* (T_L), which is the desired end-to-end latency to download an object. We note that the cache can estimate the latency of downloading an object using techniques described in [1, 15], which have been shown to be reasonably accurate in practice.
Target Recency (Age): Clients specify a target recency T_R. There are many possible recency metrics that could be chosen. In this paper our recency metric is the number of times the object has been updated at the remote server since it was cached. We refer to this metric as *age.*

We briefly discuss our choice of recency metric. There have been many different metrics described and used in the literature, e.g., [2, 9, 14, 19]. One metric is the amount of time elapsed since the cached object became stale[9]. Obsolescence measures age in terms of the number of insertions, deletions, and modifications[14]. Work in [19] considers *age*, the number of times an object has been updated at the remote server. The choice of recency metric depends on the semantics of the application and the types of updates that occur, so each of the above metrics is useful in different circumstances. In this paper we selected *age* as the recency metric[19] because we believe this metric is useful for a variety of applications. In the remainder of this paper, we use the terms *recency* and *age* interchangeably.

3.3 Profile: Parameterized Decision Function and Profile-Based Downloading

We now present the details of Profile. It uses a parameterized function that incorporates client profiles into the decision of whether to download a requested object or to use a cached copy. First, we describe the decision function. We note that there are many different functions that could be used. We chose this particular function because it has several desirable properties. First, it can be tuned to provide an upper bound with respect to latency or recency. Second, when it is impossible to meet both targets, two parameters can be set to reflect a tradeoff, i.e., the relative importance of meeting each of the targets.

Our function first calculates a score for both recency and latency as follows:

$$\texttt{Score(T,x,K)} = \begin{cases} 1 & \text{if } \texttt{x} \leq \texttt{T} \\ K/(x - T + K) & \text{otherwise} \end{cases}$$

`T` is the target value of recency or latency, `x` is the actual value, and `K` is a constant ≥0 that is used to tune the rate at which the score decreases. Let $\texttt{K}_L$ be the `K` value used to control the latency score, and let $\texttt{K}_R$ be the `K` value used to control the recency score. Note that the `K` values are set automatically by the browser based on client preferences, using a graphical interface (described in Section 3.4).

Combined Weighted Score

The decision function is a separable function that combines the scores for recency and latency. It can also be tuned to capture the latency-recency tradeoff for a client or application. This is done by assigning (relative) weights to the importance of latency and recency. The sum of the weights must equal 1. For some applications it may be more important to meet the recency target; for others it may be more important to meet the latency target. Let `w` be the weight assigned to meeting the latency target, and let `(1 - w)` be the weight assigned to meeting the recency target. We compute the *combined score* of an object as follows:

$$\texttt{CombinedScore = (1 - w)*Score(T}_R\texttt{, Age, K}_R\texttt{) + w*Score(T}_L\texttt{,Latency, K}_L\texttt{)}$$

Profile-Based Downloading

Our algorithm Profile uses the combined scoring function to make the decision of whether or not to download an object. When an object is requested, we compute the score of either downloading the object (`DownloadScore`) or using the cached copy (`CacheScore`). The Profile policy is as follows: *When an object is requested, if* `DownloadScore` > `CacheScore`*, the object is downloaded from the remote server. Otherwise the cached copy is used.*

We compute `DownloadScore` for an object as follows: Recall that when an object is downloaded, its `Age` is 0 because the remote server always provides the most recent data. Therefore, $\texttt{Score(T}_R\texttt{, Age, K}_R\texttt{)}$ is always 1.0. `Latency` is the estimated latency of downloading the object from a remote server. Thus, `DownloadScore`, the combined score of downloading an object, is

$$\texttt{DownloadScore = (1-w)*1.0 + w*Score(T}_L\texttt{, Latency, K}_L\texttt{)}$$

We now consider `CacheScore`. Recall that when an object is read from the cache, its `Latency` is 0. Therefore, `Score(`$\mathtt{T}_L$`,Latency, `$\mathtt{K}_L$`)` is always 1.0. `Age` is the estimated age of the cached object. Thus, `CacheScore`, the combined value of using a cached copy of an object, is

$$\mathtt{CacheScore = (1\text{-}w)*Score(T}_R\mathtt{,\ Age,\ K}_R\mathtt{)\ +\ w*1.0}$$

3.4 Choosing a Profile

The success of latency-recency profiles depends on the ease of creating a profile. If setting the parameters is complicated and time consuming, clients will be less inclined to use profiles. We describe an interface that allows clients to express the most appropriate profiles for their applications.

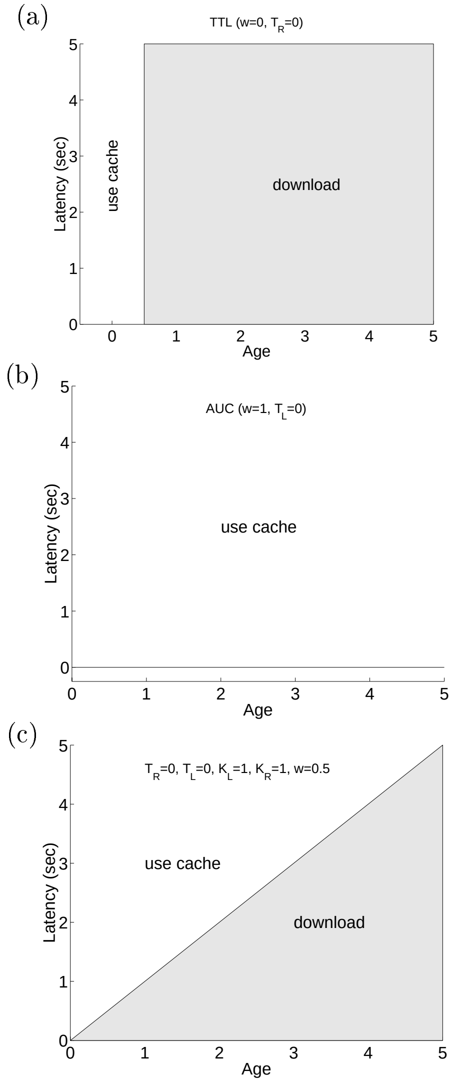

Figure 4: Behavior of (a) TTL (b) AUC (c) Profile with T_R=T_L=0, w=0.5, and K_R=K_L=1

Default Profiles

The default profile has its targets set to provide identical performance to TTL. This corresponds to settings of `w`=0 and T_R=0. Note that with these settings, the T_L value is irrelevant. This TTL setting is what many caches currently provide, e.g., proxy caches [5]. For those clients who wish to explicitly trade recency for improved latency, the browser will present a small number of parameter settings to the client, and let the client choose the settings that best suit their needs for each application. We describe how this choice can be made using the graphical interfaces of Figure 4 and Figure 5.

Figure 4 illustrates the latency-recency tradeoffs of three possible parameter settings. In these graphs, we plot the recency (age) of a cached object as `x` number of updates on the x axis and the latency of downloading the object as `y` seconds on the y axis. If a point `(x,y)` lies in the shaded area, then the object is downloaded. If `(x,y)` lies in the white area, then the object is read from the cache. Figure 4(a) displays the behavior of the default TTL profile (`w`=0, $\mathtt{T}_R$`=0`) to the user. Any object with 0 updates is served from the cache, while any object with 1 or more updates is downloaded. Figure 4(b) displays the behavior of AUC (`w`=1, $\mathtt{T}_L$`=0`), where the client will tolerate any amount of staleness to minimize access latency. AUC always uses the cached object (no shaded area), regardless of the number of updates.

Tuning Profiles

For clients who desire performance between the two extremes of TTL and AUC, a profile with parameters (`w` = 0.5, $\mathtt{K}_R$ and $\mathtt{K}_L$=1, and $\mathtt{T}_R$ and $\mathtt{T}_L$=0) can be chosen. Figure 4(c) displays the behavior of this profile. We see that the decision function captures the latency-recency tradeoff. When objects have higher access latencies, users may tolerate older cached objects (white area). Conversely, as the cached object becomes more stale, users are willing to wait longer to download a fresh object (gray area).

The profiles illustrated in Figure 4 can be tailored further. This is straightforward to do in our framework. For example, consider a client who wishes to receive data with recency of no more than 1 update. Such a client could choose the default TTL as in Figure 4(a), but change the T_R value from 0 to 1. This would result in any object with 2 or more updates being downloaded, rather than 1 or more updates as shown in Figure 4(a).

Upper Bounds

The profiles of Figure 4 do not provide any upper bounds on latency or recency. For clients who desire even greater control over the settings of their profiles, the values for w and K_L and K_R can be chosen to provide upper bounds. Clients do not need to manually choose w and K values. Instead, clients are aided by a graphical interface (similar to Figure 5) that illustrates the tradeoff for settings of w and K values, and allows them to make the appropriate choice.

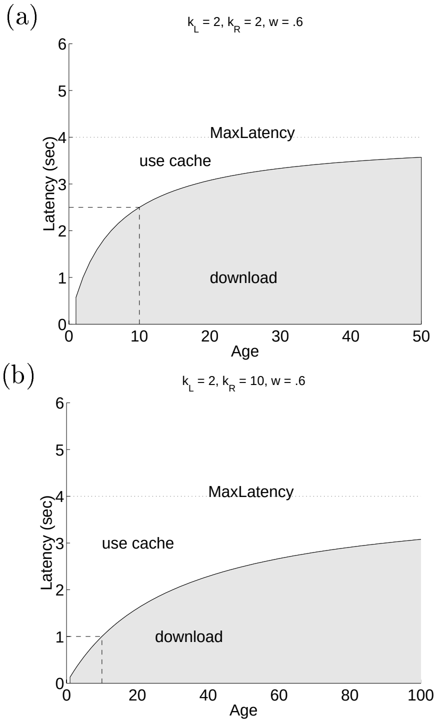

Figure 5: Upper Bounds on the Latency-Recency Tradeoff

An upper bound for either latency or recency can be chosen. In particular, assigning a higher weight to latency (w > 0.5) places an upper bound on the latency of a downloaded request, and assigning a higher weight to age (w < 0.5) places an upper bound on the age of an object delivered to the client from the cache.

In Figure 5, w = 0.6 and T_L, T_R=0. The upper bound on latency is 4 seconds. The choice of K_L and K_R in Figure 5 (a) and 5 (b) illustrate the latency-recency tradeoff that the clients can select that controls how the latency asymptotically approaches the upper bound of 4 seconds. The choice of values makes Profile more aggressive to download data as reflected by the larger shaded area.

4 Experiments

We use both trace data and synthetic data to compare Profile against three algorithms, TTL, AUC, and SSI. Our simulation models the proxy cache architecture of Figure 1.[1] We first describe the details of these algorithms. We then describe the details of both the trace and synthetic datasets. Finally, we present our results. Our key results are as follows:

- Profile significantly reduces bandwidth consumption compared to all approaches for both trace and synthetic data. Compared to TTL, Profile reduces bandwidth consumption with only a slight increase in the amount of stale data delivered to clients (trace data). Profile also provides better recency than AUC (trace and synthetic data).
- Profile can benefit from an increased cache size more than either TTL or AUC (trace data). AUC cannot deliver recent data when the cache size is large, while TTL cannot utilize a larger cache size to reduce latency. Profile can exploit increasing cache size to reduce both age and latency.
- In the presence of surges, Profile improves latencies for *all* clients, even for clients who require the most recent data.

4.1 Algorithms

We consider the following algorithms:

- TTL: This is the cache consistency mechanism currently used in most proxy caches [5, 8, 11, 17]. Cached objects are assigned a TTL value which is an estimate of how long they will be fresh in the cache. The TTL approach guarantees that all cached objects are up-to-date if the TTL estimate is accurate. It uses two parameters, UpdateThreshold and DefaultMax; these are explained in Section 4.2.
- AUC: We implemented a modified version of the prefetching strategy presented in [9]. We relax their assumption that all objects must be in the cache. Instead, we refresh objects that are currently in the cache in a round robin manner. On a cache miss, objects are downloaded from a remote server. This strategy has the advantage of being straightforward to implement at the cache, and was shown to be near optimal in [9]. Objects are validated in the background at a specified PrefetchRate, and only validated objects that have been updated at the remote server are downloaded.
- Profile: This is implemented as was described in Section 3. The decision function uses the *estimated latency* of downloading objects, and the *estimated age* of cached objects. We describe how to compute these for the trace data in Section 4.2. The settings of the profile parameters are described with the results in Section 4.4.
- SSI-Msg: We consider two variations of SSI. In the first, *SSI-Msg* the server sends invalidation *messages* to a cache whenever an object is updated,

[1] These results also apply to portals, if we do not consider the additional time to send data from a portal to a client.

but does not send the actual object to the cache. If the cached object is subsequently requested, the updated object is downloaded from the server. Note that this approach is comparable to TTL with accurate expiration times. This approach was shown to consume a comparable amount of bandwidth to TTL in [23].

- SSI-Obj: In the second variation, *SSI-Obj*, the server sends all updated objects to the cache. This consumes more bandwidth than *SSI-Msg* but guarantees that all cached objects will be up-to-date, which reduces the latency of requests.

4.2 Data

We now describe the trace and synthetic data used in our experiments.

4.2.1 Trace Data

We used trace data from NLANR[13]. This data was gathered from a proxy cache in the United States in January 2002. We considered approximately 3.7 million requests made over a period of 5 days. We performed preprocessing on the NLANR trace data to prepare it for the experiments. Specifically, the trace data did not report on object modification or expiration times, which we need to make downloading decisions and to determine the recency of cached objects. Our solution to this problem was to create an "augmented" trace using the workload from the original NLANR trace data. Over a period of 5 days, we replicated the trace workload by sending requests to the servers in the traces at (approximately) the same time of day as in the original workload. The requests were made from the domain `umiacs.umd.edu` which is connected to its ISP via a high speed DS3 line with a maximum bandwidth of 27 Mbps. When each requested object arrived, we logged the latency of the request and the time the object was last modified (when available). We used the logging mechanism provided by the Squid cache[5], but did not cache any objects. This augmented trace data provided the information we needed for this study.

In our trace-based experiments, we cached only objects that had last modified information available and were not labelled uncacheable. For the TTL algorithm, to estimate the TTL of an object, we use the policy implemented in Squid[5]. When an object's last-modified timestamp is available, Squid estimates the lifetime of an object using the adaptive TTL technique[8, 17]. In adaptive TTL, an object's `TTL` is estimated to be proportional to the age of the object at the time it was cached. The exact value depends on a parameter `UpdateThreshold`. We used an `UpdateThreshold` of 0.05, which is representative of values used in practice [5]. We calculate `TTL` = (`CurrentTime` - `LastModifiedTime`) * `UpdateThreshold`. An object's `ExpirationTime` = `CurrentTime` + `TTL`. If this estimate exceeds a default maximum value `DefaultMax`, then an object's `TTL` is estimated as `DefaultMax`. As in the Squid cache implementation we use a `DefaultMax` of 3 days.

For the Profile algorithm, we need estimates of the latency and recency of objects to make a downloading decision. We estimated the latency of an object as the average latency over all previous requests, which was shown to perform well in [1]. To estimate the age of cached objects, we defined `UpdateInterval` as `ExpirationTime - LastModifiedTime`, and defined the age of a cached object as (`CurrentTime - LastModifiedTime`)/`UpdateInterval`.

For AUC, all cache hits were served directly from the cache, and we validated objects in the background at a specified `PrefetchRate`. We considered AUC with two different prefetch rates, 60 objects per minute (AUC-60) and 300 objects per minute (AUC-300). Note that for TTL and Profile we did not perform any prefetching in this study.

On a cache hit, we need to determine if an object is fresh or stale. For all schemes, an object was fresh if the object's `last-modified` time was unchanged since the previous request. For TTL and Profile, an object was stale if its `last-modified` time had changed. For AUC, we also need to consider the effects of prefetching. If the object's `last-modified` time had changed and was more recent than the time the object was last prefetched, the object was stale. Otherwise it was fresh.

4.2.2 Synthetic Data

To complement our trace results and study the performance of profiles, we also performed simulation studies using synthetic data, where we control updates at remote servers, and use more accurate age information. We used the following parameters to generate the synthetic data:

- *Update Interval* is the average length of time between consecutive updates. In our simulation this value ranged from once every 10 minutes to once every 2 hours.

- *Estimated Latency* is the expected end-to-end latency of downloading the object from the remote server. We modeled the latencies of objects using latency distributions from NLANR traces[13]. To reduce the effects of network and server errors in this data we considered only requests with latencies of less than 5000 msec. The distribution of these values was highly skewed, with a median of approximately 200 msec and a mean of approximately 500 msec. 90% of the requests had latencies less than 1400 msec.

- *Workload* is the average number of requests per minute. We report on a workload of 8 requests/sec (480 requests/minute), which is representative of many cache workloads[13]. We ran simulations for 6 hours of simulation time for a total of ≈172800 requests.

- *World Size:* We considered a world of 100,000 objects with a popularity following a Zipf-like distribution. The ith most popular object had popularity proportional to $1/i^{\alpha}$, where α is a value between 0 and 1.0. We generated a distribution with α= 0.7, which was typical of traces analyzed in [3].

We note that for TTL and Profile, for the synthetic data we assumed that the cache had accurate expiration times (TTL estimates) for all objects. We use the trace data to compare TTL, AUC, and Profile in the real world case where estimates are often inaccurate.

4.3 Setup and Metrics

We implemented our simulation environment in C++. We ran simulations and experiments with trace data on a Sparc 20 workstation running Solaris 2.6. We assumed the cache was initially empty.

For the synthetic data, we ran simulations for 2 hours of simulation time to warm up the cache, then ran them for an additional 6 hours. For the trace data, we used the first 12 hours of the trace to warm up the cache, then collected data on the remainder of the trace. We repeated each simulation 10 times to verify the accuracy of our results, and validated that our results satisfied the 95% confidence intervals. For both trace and synthetic data, we consider cache sizes ranging from 1% of the world size to an infinite cache. We first report on results for an infinite cache. We then consider the effects of varying cache size on the performance of all approaches. We used the Least Recently Used (LRU) policy to replace objects when the cache was full; this is commonly used in practice [5].

We report on the following metrics:

- Validation messages (vals): This is the number of messages that were sent between cache and remote servers. For TTL, AUC, and Profile, a validation message is sent from the cache to a server to check for updates. The requested object was only downloaded if it had actually been updated. For SSI-Msg, a validation message is sent from a server to a cache to invalidate cached objects. Messages are typically much smaller than the actual objects. We note that for SSI-Obj, the server sends the actual objects to the cache, so no messages are sent.

- Downloads (Useful Validations): This is the number of requested objects that were validated and subsequently downloaded because they were stale in the cache. For SSI-Obj, this includes all objects that were updated at remote servers and sent to the cache. For SSI-Msg,TTL and Profile, this includes requested cached objects that were not sufficiently recent in the cache. For AUC, this includes objects that were prefetched (validated) in the background and were downloaded because they were stale.

	SSI-Obj	SSI-Msg	TTL
Val. Msgs	0	161768	67170
Downloads	161768	67170	67170
AvgAge	0	0	0
StaleHits	0	0	0

	AUC-60	AUC-300	Profile
Val. Msgs	21600	100797	15932
Downloads	16158	75833	15932
AvgAge	2.09	0.61	1.38
StaleHits	112492	74548	96281

Table 1: Results for Experiments with Synthetic Data

- Useless Validations: For the trace data, we also report on useless validations. These are objects that were in the cache and were validated at the remote server, but had not actually been modified since they were cached. Since useless validations add unnecessary latency to requests, it is important to minimize this number.

- Stale Hits: For the trace data, this is the number of objects served from the cache (without validation), but that had actually been updated at the remote server.

- Age: This is the average age of objects delivered to clients, i.e., the number of times they were updated at the remote server. Objects that were downloaded from a server always had an age of 0.

- Latency: This is the average latency of the requests in msec.

4.4 Comparison of Profile to Existing Technologies

Our first set of results shows the benefits of using Profile for an infinite cache. We first show simulation results using synthetic data. We then use the trace data to compare Profile to TTL and AUC. The trace data reflects the situation when TTL estimates are inaccurate, which is often the case in practice. We do not study SSI on the traces since it requires server cooperation.

In these experiments, all clients used a single profile with T_R =1 update and T_L=1 second. The other settings were w = 0.5 and K_L, K_R = 1. Recall that with w=0.5, neither latency nor recency is favored in the tradeoff.

	TTL	AUC-60	AUC-300	Profile
Val. msgs	151367	378312	1891560	92943
Useful vals	24898	933	2810	22896
Useless vals	122074	279349	327776	67601
Avg Est.Age	0	18.4	11.1	0.87
Stale Hits	4282	31285	22897	7704

Table 2: Results for Experiments with Trace Data

The number of validations and downloads for the simulation study with synthetic data is shown in Table 1. The first observation is that SSI-Obj consumes the most bandwidth because it sends a large number of objects to the cache to keep the cache up to date. This is shown in the Downloads row. SSI-Msg and AUC-300 also consume significant amounts of bandwidth compared to Profile. While they download fewer objects than SSI-Obj, they still send many validation messages. In contrast, Profile performs fewer validations *and* fewer downloads than all other approaches. We will use the trace data to further quantify the bandwidth savings of Profile relative to TTL and AUC.

The average ages of objects and number of stale hits are also shown in Table 1. These results show that while AUC-60 and Profile have a comparable number of downloads, AUC-60 does so at the cost of delivering significantly less recent data. AUC-60 delivers objects with an average age of 2.09 updates compared to 1.38 updates for Profile. AUC-60 also provides nearly 20% more stale hits than Profile. AUC-300 provides better recency (0.61) than Profile. However, it does so at the cost of validating 600% more objects than Profile (100797 vs. 15932) and downloading nearly 500% more objects(75833 vs. 15932).

Our trace results further compare TTL, AUC, and Profile in the real-world case where TTL estimates are often inaccurate. Table 2 shows the number of validations for TTL, AUC, and Profile, for an infinite cache. The first observation is that both variants of AUC validate significantly more objects than either TTL or Profile. Recall that AUC validates objects at the specified `PrefetchRate`. TTL also validates many more objects than Profile.

The number of useful and useless validations are shown in the second and third lines of Table 2[2]. We note that for AUC, for a fair comparison we measured useful and useless validations only for prefetched objects that were subsequently requested.

A key observation is that TTL performs nearly twice as many useless validations as Profile (122074 vs. 67601). In these cases, TTL adds latency to requests without improving the recency. In contrast, Profile can significantly reduce the number of useless validations (by ≈ 60,000) with only a small increase in the number of stale hits (≈ 4000 more than TTL). Another key observation is that both variants of AUC perform many more useless validations than either TTL or Profile. Further, AUC performs very few useful validations for objects that are subsequently requested (less than 3000 for AUC-3000 vs. 24898 for Profile). Thus, AUC can consume large amounts of bandwidth to keep the cache refreshed while doing little to improve the recency of data delivered to clients.

Since the trace data does indicate how many times servers actually modified objects, we must estimate the age, i.e., number of times a stale object was updated at a server since it was cached. We compute age= (`CurrentTime` - `LastModifiedTime`)/`UpdateInterval`, where `UpdateInterval` is estimated as defined in Section 4.2. While this is an estimate, it gives an idea of how out of date the stale objects were.

Both variants of AUC prefetch a large number of objects, while still delivering many stale objects to clients. The average estimated age of the stale hits is shown in the last line of Table 2, and show that AUC can deliver very out of date objects. This is because the prefetching strategy for AUC prefetches all objects with equal frequency, which may cause frequently updated objects to become very out of date. While this prefetching strategy is near optimal for minimizing the number of stale hits[9], our results clearly show that AUC may nevertheless result in very stale data. Thus, prefetching may not be appropriate for applications that cannot tolerate stale data, especially when the data is updated frequently.

4.5 Effect of Cache Size

We now use the trace data to measure the performance of TTL, AUC, and Profile for varying cache sizes. We varied our relative cache size from 1% of the world size to 100% of the world size (i.e., an infinite cache). We show that Profile can better utilize cache size to reduce latency (compared to TTL) and to reduce age (compared to AUC).

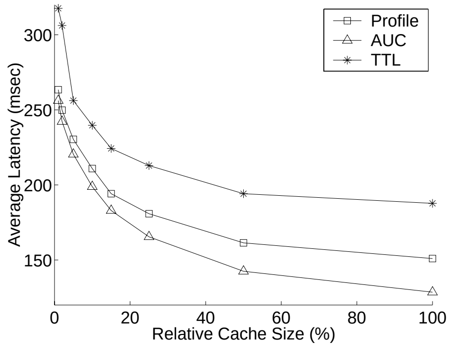

Figure 6: Effect of Cache Size on Average Latency

The average latency for Profile and the baseline algorithms are plotted in Figure 6. The first observation is that both Profile and AUC better utilize in-

[2] In some cases the trace did not contain a last modified date to determine if a validation was useful or useless, therefore the sum of these values is less than the total validations.

creased cache size to reduce latency. While increasing the cache size increases the number of objects that can be cached, objects that expire in the cache must always be validated for TTL. Increasing the cache size does not decrease the number of stale objects in the cache, so TTL does not benefit significantly from a larger cache. In contrast, Profile and AUC can benefit more from an increased cache size. While objects in the cache may be stale, they may still be useful to some clients.

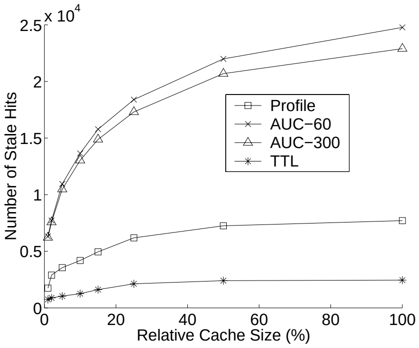

Figure 7: Effect of Cache Size on Number of Stale Hits

Figure 7 shows the number of stale hits. As the cache size increases, the stale hits for both AUC-60 and AUC-300 increase dramatically. This is because prefetching for AUC does not scale well and a greater number of client requests are being serviced by (possibly stale) cached objects, so the number of stale hits increases. This shows that the reduced latency of AUC comes at the high cost of delivering very stale data. In contrast, the number of stale hits for Profile increases by a much smaller amount. In summary, AUC cannot utilize a large cache size to reduce age and delivers very stale data. Similarly TTL cannot utilize a larger cache size to reduce latency. In contrast, Profile is flexible and can exploit increasing cache size to reduce both age and latency.

4.6 Effect of Surges

Under normal workloads, there is typically sufficient bandwidth and server capacity to handle all requests. However, from time to time networks or servers may experience "surges", i.e., a period of time during which the available resource capacity exceeds the demand. During surges, many request will be backlogged and their processing may be delayed significantly. As an example, we consider the case where there is insufficient bandwidth between a cache and the servers. In this case, the servers will attempt to deliver many objects simultaneously, which will cause delays delivering the objects to the cache. This could occur in a proxy cache if a surge in remote requests saturates the bandwidth between the Internet and the cache. It could also occur in an application server cache if many clients make requests to the server simultaneously.

Our final experiment is a simulation that compares Profile to TTL in the presence of surges. A surge is represented by a capacity ratio. The capacity ratio is the ratio of available resources per second to the resources required per second. For example, during a surge period, if a server can handle 10 requests per second and requests arrive at the rate of 20 requests per second, then the capacity ratio during this period is 1/2. A capacity ratio of 1 means there are sufficient resources to handle all requests, and requests will incur no extra delay as a result of the surge. However, if this ratio is less than 1, performance can severely degrade.

We consider two groups of clients. The first group, `MostRecent`, has $\mathrm{T}_R = 0$ updates and $\mathrm{T}_L = 1$ sec. The second, `LowLatency`, has $\mathrm{T}_R = 1$ update and $\mathrm{T}_L = 0$ sec. For both groups, the other settings were $\mathtt{w} = 0.5$ and $\mathtt{K}_L$, $\mathtt{K}_R = 1$. In our simulation, we considered a surge with duration 30 seconds. The request rate is 100 requests per second. We vary the available capacity from 20 to 100 objects per second, i.e., the capacity ratio varies from 0.2 to 1.0. For simplicity, we assume no requested objects are evicted from the cache during the surge period. We warmed up the cache for 10000 requests at a non-surge workload of 8 requests/sec, then began the surge period and gathered data.

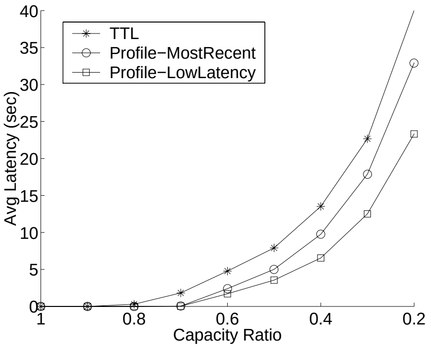

Figure 8: Avg. Latency during a 30-sec. surge period

Figure 8 plots the average latencies of all requests. For TTL, all requests are treated equally, and all objects that have expired in the cache are downloaded. As expected, the latency is very high, especially when the capacity ratio is below 0.5. However, Profile can distinguish between the two groups of clients and better serve their requests. As expected, the latency for some `LowLatency` clients is significantly lower than for TTL. This is because, when a requested object is in the cache, stale data can be delivered to the `LowLatency` clients. Consequently, there is more available bandwidth to serve the other clients. Thus, the latency for the `MostRecent` clients also decreases compared to TTL. Thus, using Profile during a surge can significantly improve access latencies for *all* clients, not just those that can tolerate stale data.

5 Conclusions and Future Work

The growing diversity of clients and services on the Web requires data delivery techniques that can accomodate diverse client needs with minimal overhead to clients, caches, and web servers. This paper introduced *latency-recency profiles*, a scalable, tunable approach to delivering web data that is straightforward to implement for both clients and caches, and requires no cooperation from remote web servers. Our results show that profiles can significantly reduce bandwidth consumption compared to existing technologies, while also providing greater flexibility than TTL or AUC.

In future work, we plan to explore prefetching strategies that exploit profiles. For example, objects for which clients require the most recent data should be kept more up-to-date. We will also consider the effects of wide variations in profiles on performance. Finally, we plan an implementation of profiles in Squid[5], and plan to use it to further study the effectiveness of profiles.

6 Acknowledgements

This research is supported by National Science Foundation grant DMI9908137 and a National Physical Science Consortium fellowship. Bobby Bhattacharjee, Selcuk Candan, Mike Franklin, Avigdor Gal, and Samir Khuller provided useful feedback. We thank Duane Wessels and the National Laboratory for Applied Network Research for the trace data used in this study. NLANR is supported by National Science Foundation grants NCR-9616602 and NCR-9521745.

References

[1] S. Adali, K.S. Candan, Y. Papakonstantinou, and V.S. Subrahmanian. Query caching and optimization in distributed mediator systems. *Proc. SIGMOD*, 1996.

[2] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM. TODS Vol. 15, no. 3*, 1990.

[3] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. *Proc. IEEE INFOCOM*, 1999.

[4] Oracle9iAS Web Cache. *http://otn.oracle.com/products/ias/web_cache/content.html*.

[5] Squid Proxy Cache. *http://www.squid-cache.org*.

[6] K.S. Candan, W.-S. Li, Q. Luo, W.-P. Hsiung, and D. Agrawal. Enabling dynamic content caching for database-driven web sites. *Proc. SIGMOD*, 2001.

[7] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. *Proc. USENIX Symposium on Internet Technologies and Systems*, 1997.

[8] V. Cate. Alex - a global filesystem. *Proc. USENIX File System Workshop*, 1992.

[9] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. *Proc. SIGMOD*, 2000.

[10] E. Cohen, B. Krishnamurthy, and J. Rexford. Improving end-to-end performance of the web using server volumes and proxy filters. *Proc. SIGCOMM*, 1998.

[11] A. Dingle and T. Partl. Web cache coherence. *Proc. 5th WWW Conf.*, 1996.

[12] Yahoo! Finance. *http://finance.yahoo.com*.

[13] National Laboratory for Applied Network Research. *ftp://ircache.nlanr.net/Traces*.

[14] A. Gal. Obsolescent materialized views in query processing of enterprise information systems. *Proc. CIKM*, 1999.

[15] J.R. Gruser, L. Raschid, V. Zadorozhny, and T. Zhan. Learning response time for web sources using query feedback and application in query optimization. *VLDB Journal 9(1):18-37*, 2000.

[16] H. Gupta. Selection of views to materialize in a data warehouse. *Proc. ICDT*, 1997.

[17] J. Gwertzman and M. Seltzer. World wide web cache consistency. *Proc. USENIX Technical Conf.*, 1996.

[18] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. *Proc. SIGMOD*, 1996.

[19] Y. Huang, R. Sloan, and O. Wolfson. Divergence caching in client-server architectures. *Proc. PDIS*, 1994.

[20] T. Kroeger, D. Long, and J. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. *Proc. USENIX Symposium on Internet Technologies and Systems*, 1997.

[21] A. Labrinidis and N. Roussopoulos. Webview materialization. *Proc. SIGMOD*, 2000.

[22] A. Labrinidis and N. Roussopoulos. Update propagation strategies for improving the quality of data on the web. *Proc. VLDB*, 2001.

[23] C. Liu and P. Cao. Maintaining strong cache consistency on the world wide web. *Proc. ICDCS*, 1997.

[24] Q. Luo and J. Naughton. Form-based proxy caching for database-backed web sites. *Proc. VLDB*, 2001.

[25] C. Olston, B.T. Loo, and J. Widom. Adaptive precision setting for cached approximate values. *Proc. SIGMOD*, 2001.

[26] H. Rozanski and G. Bollman. The great portal payoff. *Booz-Allen & Hamilton, http://www.bah.com*.

[27] P. Scheuermann, J. Shim, and R. Vingralek. A unified algorithm for cache replacement and consistency in web proxy servers. *Proc. WebDB*, 1998.

[28] BEA WebLogic. *http://www.bea.com*.

[29] IBM WebSphere. *http://www-3.ibm.com/software/webservers/*.

[30] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. *Proc. SIGMOD*, 1995.