A Generalized Target-Driven Cache Replacement Policy for Mobile Environments[*]

Liangzhong Yin, Guohong Cao
Department of Computer Science & Engineering
The Pennsylvania State University
{yin, gcao}@cse.psu.edu

Ying Cai
Department of Computer Science
University of Central Florida
cai@cs.ucf.edu

Abstract

Caching frequently accessed data items on the client side is an effective technique to improve the system performance in a mobile environment. Due to cache size limitations, cache replacement algorithms are used to find a suitable subset of items for eviction from the cache.

In this paper, we propose a generalized cost function for cache replacement algorithms for mobile environment. The distinctive feature of our cost function is that it is general and it can be used for various performance metrics by making the necessary changes. To demonstrate the practical effectiveness of the general cost function, we derive two specific functions to be evaluated by setting two different targets: minimizing the query delay and minimizing the downlink traffic. Detailed experiments are carried out to evaluate the proposed methodology. Compared to previous schemes, our algorithm significantly improves the performance in terms of query delay or in terms of bandwidth utilization depending on the targets.

1 Introduction

Caching frequently accessed data items on the client side is an effective technique to improve performance in a mobile environment [3]. Due to the limitations of the cache size, it is impossible to hold all the accessed data items in the cache. As a result, cache replacement algorithms are used to find a suitable subset of data items for eviction from the cache.

Many cache replacement policies [4, 12, 14, 16] employ a cost function of the different factors such as time since last access, entry time of the data item in the cache, transfer time cost, data item expiration time and so on. For example, the algorithm [4] proposed by Bolot and Hoschka first explicitly considers the delay to fetch web documents in cache replacement. Their cost function employs a weighted function of the transfer time cost, the document size, and the time since last access. However, the choice of the cost function is not justified and there are many unspecified weights. The Hybrid Algorithm (HYB) [14] addresses both latency and bandwidth issues. The cost function employs a weighted exponential function of the access frequency, the size, the latency to the server and the bandwidth to the server. Several constants are used, but exactly how to set these constants to get better performance is not given. The LNC-R-W3-U algorithm, proposed by Shim *et al.* [12], aims to minimize the response time. Their cost function employs a rational of the access frequency, the transfer time, the document size, and the validation rate. The author proved that their cache replacement algorithm could find the document subsets that satisfy the cost function. However, the author did not prove that this algorithm could minimize the response time. The algorithms mentioned above are designed for WWW environment where weak cache consistency model is adopted. These algorithms may not be suitable if strong cache consistency model is needed. The Min-SAUD algorithm [16] is designed for strong cache consistency model. It uses an optimal cost function that can minimize the metric *stretch*[1]. Although the authors proved that their cost function is optimal, they did not show how to get such an optimal cost function.

These policies are valuable in that they address various aspects of cache replacement. However, their choices of cost functions are based on experience. These algorithms are designed for a specific metric (target). When the target changes, they have to come up with another function. In this paper, we propose a different approach for cache replacement. We first present a cache access cost model for mobile environments. We then propose a generalized cost function based on this cost model, and prove that the proposed

[*]This work was supported in part by the National Science Foundation CAREER Award CCR-0092770 and ITR-0219711.

[1]The ratio of the access latency of a request to its service time, where the service time is defined as the ratio of the item size to the broadcast bandwidth.

cost function can optimize the access cost in ideal situation. Since our cost function is general, it can be used for various kinds of performance metrics by making the necessary changes. To demonstrate the practical effectiveness of the general cost function, we derive two specific functions by setting two different targets: minimize the query delay and minimize the downlink traffic. Extensive simulations are provided and used to justify the analysis. The simulation results show that for both targets, our cache replacement policy can significantly improve the performance compared to the LRU algorithm and the LRU-MIN algorithm [1].

The rest of the paper is organized as follows. Section 2 presents the system model. In Section 3, we present the generalized cost function and the cache replacement algorithm. The optimal proof is also provided. Some implementation issues are discussed in Section 4. Section 5 evaluates the performance of the proposed cache replacement algorithm under two different targets. Section 6 concludes the paper.

2 The System Model

2.1 Mobile Computing Model

In a mobile computing system, the geographical area is divided into small regions, called cells. Each cell has a *base station* (BS) and a number of *mobile terminals* (MTs). Inter-cell and intra-cell communications are managed by the BSs. The MTs communicate with the BS by wireless links. An MT can move within a cell or between cells while retaining its network connection. An MT can either connect to a BS through a wireless communication channel or disconnect from the BS by operating in the *doze* (power save) mode.

The mobile computing platform can be effectively described under the *client/server* paradigm. A data item is the basic unit for update and query. MTs only issue simple requests to read the most recent copy of a data item. There may be one or more processes running on an MT. These processes are referred to as clients (we use the terms MT and client interchangeably). In order to serve a request sent from a client, the BS needs to communicate with the database server to retrieve the data items. Since the communication between the BS and the database server is through wired links and is transparent to the clients (i.e., from the client point of view, the BS is the same as the database server), we use the terms BS and server interchangeably.

2.2 The Cache Invalidation Model

Frequently accessed data items are cached on the client side. To ensure cache consistency, a cache management algorithm is necessary. Classical cache invalidation strategies may not be suitable for mobile environments due to frequent disconnections and high mobility of mobile clients. It is difficult for the server to send invalidation messages directly to the clients because they often disconnect to conserve battery power and are frequently on the move. For the clients, querying data servers through wireless links for cache invalidation is much slower than wired links because of the latency of the wireless links. As a solution, we use the IR-based cache invalidation approach [3] to maintain cache consistency. In this approach, the server periodically broadcasts an *Invalidation Report (IR)* in which the changed data items are indicated. Rather than querying the server directly regarding the validation of cached copies, the client can listen to these IRs over wireless channels and use the information to invalidate its local cache. More formally, the server broadcasts an IR every L seconds. The IR consists of the current timestamp T_i and a list of tuples (d_x, t_x) such that $t_x > (T_i - w * L)$, where d_x is the data item id, t_x is the most recent update timestamp of d_x, and w is the invalidation broadcast window size. In other words, IR contains the update history of the past w broadcast intervals. However, any client who has been disconnected longer than w IR intervals cannot use the report, and it has to discard all cached items even though some of them may still be valid. Many solutions [10, 11, 15] are proposed to address the long disconnection problem, and Hu *et al.* [10] has a good survey of these schemes.

In the IR-based cache invalidation model, every client, if active, listens to the IRs and invalidates its cache accordingly. To answer a query, the client listens to the next IR and uses it to decide whether its cache is valid or not. If there is a valid cached copy of the requested data item, the client returns the item immediately. Otherwise, it sends a query request to the server through the uplink. Hence, the average latency of answering a query is the sum of the actual query processing time and half of the IR interval. If the IR interval is long, the delay may not be able to satisfy the requirements of many clients. In order to reduce the query latency, Cao [7] proposed to replicate the IRs m times; that is, the IR is repeated every $(\frac{1}{m})^{th}$ of the IR interval. To reduce the packet size, the invalidation report replica, which is called UIR, only contains the invalidation information since last IR report. A client only needs to wait at most $(\frac{1}{m})^{th}$ of the IR interval before answering a query. Hence, latency can be reduced to $(\frac{1}{m})^{th}$ of the latency in the previous schemes (when query processing time is not considered). In this paper, we will apply the UIR-based approach to reduce the query delay of the IR-based cache invalidation model. Although our algorithm is based on this cache invalidation model, it can also work under other models, such as proposed in[3, 10, 11], with no or trivial changes.

We have standard assumptions of the Poisson arrivals of data accesses/updates and the independent reference model. The Poisson arrivals are usually used to model data access and update processes. The independent reference model has been adopted by many researchers [5, 16] and it explains the access behavior well [5].

3 A Generalized Target-Driven Cache Replacement Algorithm

To facilitate our discussion, the following notations are used. Figure 1 further explains the use of these notations.

- n: the number of data items in the database.
- f_i: the cost of fetching data item i to the cache.
- c: the mean cost of validating the consistency of data item in cache.
- v_i: the cost of getting updated data item i from the server.
- a_i: the mean access rate to data item i.
- u_i: the mean update rate of data item i.
- s_i: the size of data item i.
- P_{a_i}: the probability of referencing data item i.
- P_{u_i}: the probability of invalidating cached data item i.
- V: the set of all the cached data items.

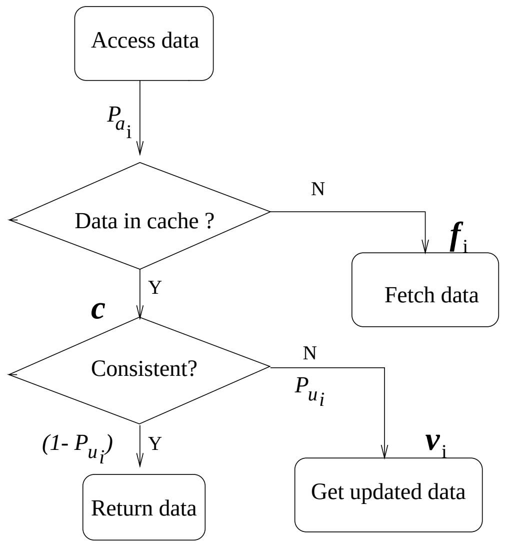


Figure 1. The cache access cost model

Based on the above notations, the cache replacement policy should optimize the following expression

$$max \sum_{i \in V} cost(i)$$
$$where \quad cost(i) = P_{a_i}(f_i - c - P_{u_i} v_i) \tag{1}$$

This cost function can be explained by the cache access cost model shown in Figure 1. If data item i is not in the cache, it will take f_i to fetch data item i into the cache. In other words, if i is in the cache, we can save the access cost by f_i. However, it also takes $(c + P_{u_i} v_i)$ to validate it and get the updated data if necessary. Thus, caching the data can save the cost by $(f_i - c - P_{u_i} v_i)$ per access. Since the access possibility is P_{a_i}, we can conclude that the value $P_{a_i}(f_i - c - P_{u_i} v_i)$ reflects the value of caching data item i.

Based on this cost function, we can build our cache replacement policy $General\_Opt$. Let V denote the set of all the cached data items. Suppose we need to replace data items of size s in order to add a new data item to the cache, our policy finds the victim items set V^* that satisfies the following two conditions:

$$(a) \ \sum_{i \in V^*} s_i \geq s$$

$$(b) \ \forall V_k (V_k \subseteq V \wedge \sum_{i \in V_k} s_i \geq s), \ \sum_{i \in V_k} cost(i) \geq \sum_{i \in V^*} cost(i) \tag{2}$$

Intuitively V^* is the least costly subset of V whose total size is at least s.

Theorem 1 *The General_Opt algorithm replaces the set of items that minimize the total access cost.*

Proof. Suppose A is the set that $General\_Opt$ algorithm finds. B is an arbitrary set whose total size $\sum_{i \in B} s_i \geq s$. Suppose $A \bigcap B = \emptyset$, otherwise we can remove the intersecting elements since their costs are equal under two algorithms. According to the algorithm

$$\sum_{i \in A} cost(i) \leq \sum_{j \in B} cost(j)$$

Let $C = V - A - B$. Let d denote the data item to be brought into the cache. Let $Cost(C)$ and $Cost(d)$ denote the cost of accessing C and d respectively.

After replacing A from the cache, the cost of accessing A (not in cache) is

$$\sum_{i \in A} P_{a_i} f_i$$

and the cost of accessing B (still in cache) is

$$\sum_{j \in B} P_{a_j}(c + P_{u_j} v_j)$$

Thus the total access cost after replacing A is

$$T_A = \sum_{i \in A} P_{a_i} f_i + \sum_{j \in B} P_{a_j}(c + P_{u_j} v_j) + Cost(C) + Cost(d)$$

Similarly the total access cost after replacing B is

$$T_B = \sum_{i \in B} P_{a_i} f_i + \sum_{j \in A} P_{a_j}(c + P_{u_j} v_j) + Cost(C) + Cost(d)$$

So,

$$\begin{aligned} T_A - T_B &= (\sum_{i \in A} P_{a_i} f_i + \sum_{j \in B} P_{a_j}(c + P_{u_j} v_j)) \\ &\quad -(\sum_{i \in B} P_{a_i} f_i + \sum_{j \in A} P_{a_j}(c + P_{u_j} v_j)) \\ &= (\sum_{i \in A} P_{a_i} f_i - \sum_{j \in A} P_{a_j}(c + P_{u_j} v_j)) \\ &\quad -(\sum_{i \in B} P_{a_i} f_i - \sum_{j \in B} P_{a_j}(c + P_{u_j} v_j)) \\ &= \sum_{i \in A} cost(i) - \sum_{j \in B} cost(j) \\ &\leq 0 \end{aligned}$$

Thus, the $General\_Opt$ algorithm replaces a set that minimize the total access cost. □

Based on the generalized cost function, we can derive specific cost function for a specific metric. For example, suppose we want to minimize the query delay, f_i will be the delay to fetch item i after the query is generated; c is the delay to validate the cached item; v_i is the delay to get the updated item i from the server after cache validation. We have also derived other specific cost functions and we will evaluate their performances in Section 5.

4 Implementation Issues

In the $General\_Opt$ algorithm, the optimization problem defined by Equation 2 is essentially the 0/1 knapsack problem, which is known to be *NP-hard*. Although there is no optimal solution to the problem, when the data sizes are relatively small compared to the cache size [12], a well-known heuristic can obtain the sub-optimal solution, and we adopt this heuristic; that is, throw out the cached data item i with the minimum $\frac{cost(i)}{s_i}$ value until the free cache space is sufficient to accommodate the incoming data item.

4.1 Parameter Estimation

In the actual implementation, f_i, v_i, P_{a_i}, and P_{u_i} are usually not constant. We have to estimate these parameters accurately to capture the temporal locality of data access. In the following, we provide techniques to estimate the value of these parameters.

We adopt the exponential aging method, which is adopted in TCP ([13]) to estimate the round-time delay, to estimate f_i and v_i. It combines both the history data and the current observed value to estimate the parameters. Whenever an access or validation is completed, f_i and v_i are re-calculated as following:

$$f_i = \alpha * f_i^{new} + (1 - \alpha) * f_i^{old}$$

$$v_i = \alpha * v_i^{new} + (1 - \alpha) * v_i^{old}$$

P_{a_i} and P_{u_i} can be derived from a_i and u_i. Since P_{a_i} is proportional to a_i, P_{a_i} can be replaced by a_i directly. Let T_{a_i} be the time of access and T_{u_i} be the time of invalidation. Since we assume the Poisson arrivals of data accesses and updates, the possibility that cache invalidation happens before next access is [16]:

$$P_{u_i} = Pr(T_{u_i} < T_{a_i}) = \frac{u_i}{a_i + u_i}$$

So, the cost function in Equation 1 can be replaced by the following cost function[2]:

$$cost(i) = a_i(f_i - c - \frac{u_i}{a_i + u_i} v_i) \qquad (3)$$

We cannot simply use the above aging technique to estimate a_i and u_i since the access rate and the update rate should still be "aged" in the absence of access to a data item. We apply similar techniques used by Shim *et al.* [12] to estimate a_i and u_i. This method uses K most recent samples to estimate a_i and u_i as follows.

$$a_i = \frac{K}{T - T_{a_i}(K)}$$

$$u_i = \frac{K}{T - T_{u_i}(K)}$$

where T is the current time, $T_{a_i}(K)$ and $T_{u_i}(K)$ are the time of the K^{th} most recent access and update. If less than K samples are available, all the available samples are used. It is shown by Shim *et al.* [12] that K can be as small as 2 or 3 to achieve the best performance. Thus the spatial overhead to store recent access and update time is relatively small.

4.2 Cache Insertion and Removal

A priority queue is needed so that the data item with the least $cost(i)/s_i$ value can be quickly found and removed. We implement the priority queue based on a heap. With the help of the heap, remove and insert operations can be performed in $O(logN)$ time, where N is the total number of cached items. Due to data item access and parameter re-evaluation, the key value of the item within the heap maybe changed, and then its position should also be changed to

[2]This cost function is a generalized version of the cost function proposed in [16], which is proven to be able to optimize the metric *stretch*. We can derive virtually the same cost function proposed in [16] by assigning the specifi c cost values to the parameters in our cost function.

reflect its current value. A pointer is used to record its position in the heap. In case of a value change, the item can be found through this pointer in $O(1)$ time and $O(logN)$ time is needed to adjust its position.

5 Performance Evaluation

In this section, we evaluate the performance of the proposed methodology. To compare with other algorithms, we use two specific targets and apply them to our general function. The first target is to minimize the query delay. The second target is to minimize the downlink traffic.

5.1 The Simulation Model

In the simulation, we model a single server that maintains a collection of n data items. A number of clients access these data items. The UIR cache invalidation model is applied for wireless data dissemination.

5.1.1 The Client Model

The client query model is similar to what have been used in our previous studies [7, 6, 17]. Each client generates a single stream of read-only queries. The mean query generate time for each client is T_{query}. The access pattern follows $Zipf$ distribution [18] with parameter θ. In our simulation, data items with smaller id will have higher access rate.

Similar to [2], we partition the data items into disjoint regions of $RegionSize$ items each. The access possibility of any item within a region follows uniform distribution. The Zipf distribution is applied to these regions.

5.1.2 The Server Model

The server broadcasts cache invalidation information (IR and UIR) periodically. If the server receives requests from clients, it will serve the requests during the next IR interval on an FCFS (first-come-first-service) basis. There are totally n data items at the server side. The data size varies from s_{min} to s_{max}, and has the following two types of distributions.

- **Random:** The distribution of data size falls randomly between s_{min} and s_{max}.
- **Increase:** The size (s_i) of the data item (i) grows linearly as i increases; i.e. $s_i = s_{min} + (i-1) * \frac{s_{max}-s_{min}}{n-1}$.

The combination of size distribution and Zipf access pattern defines the joint distribution of access frequency and item size. The choices of the size distributions are based on previously published trace analyses. Some analyses [8, 9] show that small data items are accessed more frequently than large items; while a recent web trace analysis [5] shows that the correlation between data item size and access frequency is weak and can be ignored.

The server generates a single stream of updates separated by an exponentially distributed update inter-arrival time with mean value of T_{update}. The data items in the database are divided into hot data subset and cold data subset. Within the same subset, the update is uniformly distributed, where 80% of the updates are applied to the hot data subset. In the experiment, we assume that the server processing time is negligible, and the broadcast bandwidth is fully utilized for broadcasting IR and UIR, and serving clients' data requests. Most of the system parameters are listed in Table 1. The second column lists the default values of these parameters. In the simulation, we may change the parameters to study the impact of these parameters. The ranges of these parameters are listed in the third column.

Parameter	Default value	Range
Database size (n)	3000 items	
RegionSize	50 items	
Number of clients	100	
s_{min}	0.5 KB	
s_{max}	20 KB	
Mean update time T_{update}	100 seconds	
Hot update prob.	0.8	
Hot subset percentage	0.2	
Broadcast interval (L)	20 seconds	
Broadcast window (w)	10 interval	
Broadcast bandwidth	144 kb/s	
Relative cache size	10% of total database size	1% to 50%
Mean query generate time T_{query}	100 seconds	
Zipf distribution parameter θ	0.9	0 to 1

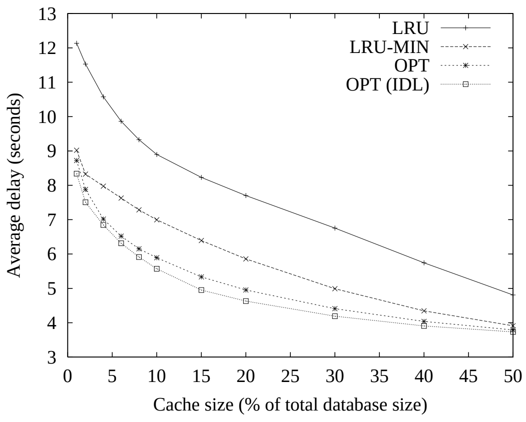
Table 1. Simulation parameters and their default values

Since the client caches are only partially full at the initial stage, the effectiveness of the different algorithms may not be truly reflected. In order to get a better understanding of the true performance for each algorithm, we collect the result data only after the system becomes stable, which is defined as the time when the client caches are full.

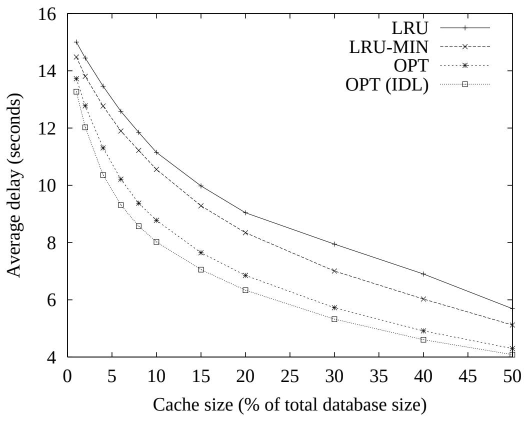
5.2 The Evaluated Algorithms

Four cache replacement algorithms are compared in our simulations.

- **LRU:** Keep removing the item that was used the least recently until there is enough space in the cache.



(a) Increase



(b) Random

Figure 2. The average query delay as a function of the cache size

- **LRU-MIN** [1]: Suppose the incoming data size is S and there is not enough space in the cache. The algorithm finds the list of items in the cache with size at least S and remove the least recently used items from the list. If the list is empty, the algorithm finds the list of items with size at least $S/2$ and keep removing items in the list according to the LRU order. Similarly, if more space are needed, try the items of size at least $S/4$.

- **OPT:** This is our algorithm. It keeps removing the item with least $cost(i)/s_i$ value where the cost function is defined by Equation 3.

- **OPT (IDL):** We also simulate an ideal case, where the access rate and the update rate are known as *a prior*. This defines an upper bound for our algorithm.

5.3 Simulation Results: Minimizing the Query Delay

Suppose our target is to minimize the query delay. Then as shown in Section 3, in Equation 3, f_i will be the delay to fetch item i after the query is generated; c is the delay to validate the cached item; v_i is the delay to get the updated item i from the server after cache validation.

5.3.1 The Average Delay under Different Cache Size

Figure 2 shows the average query delay as a function of the cache size. The total database size is fixed. We change the relative cache size from 1% of total database size to 50% of total database size to study the effect of cache size on the average delay.

The "Increase" distribution favors small data items. A large number of data items can still be saved in the cache even when the cache size is small. As a result, the cache hit-ratio is higher and the query delay is lower. This can be verified by Figure 2, where the query delay under "Increase" size pattern is smaller than that under "Random" pattern.

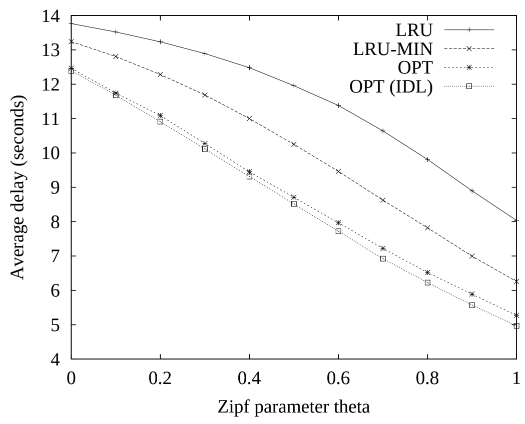
Generally speaking, the average query delay drops as the cache size increases. However, our algorithms always outperform LRU and LRU-MIN. For the "Random" size distribution (Figure 2 (b)), OPT (IDL) can outperform LRU by 28% when the relative cache size is 10% and 33% when the relative cache size is 30%. Although OPT is not as good as OPT (IDL), its average query delay is still 21% less than that of LRU algorithm and 17% less than that of LRU-MIN algorithm when the relative cache size is 10%.

For the "Increase" size distribution, there are correlation between access rate and data size. So those algorithms that consider data size will have better performance than those that do not. For example, in Figure 2 (a), the difference between LRU algorithm and other algorithms is much larger than that in Figure 2 (b).

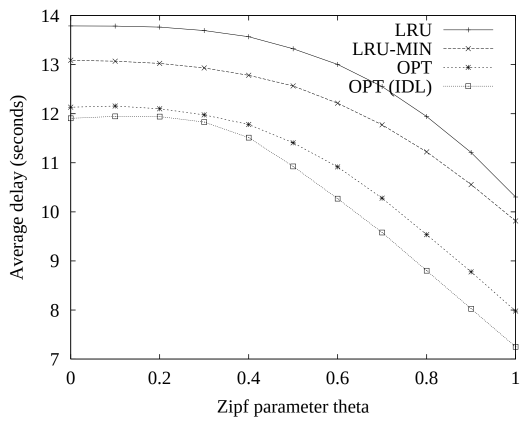
5.3.2 The Average Delay under Different Access Pattern (θ)

The Zipf parameter θ determines the "skewness" of the access distribution. Figure 3 shows the effect of the access pattern on the system performance. When $\theta = 0$, the "Random" and "Increase" distribution almost generate the same result. This is because the access is uniform and there is no favor on the size of data items. As θ grows, the average delay of the "Increase" distribution drops faster than the "Random" distribution since more items can be cached in the "Increase" distribution.

As shown in Figure 3, OPT and OPT (IDL) constantly outperform LRU and LRU-MIN. In Figure 3 (a), on average, OPT outperforms LRU by 22% and outperforms LRU-MIN by 11%. In Figure 3 (b), on average, OPT outperforms



(a) Increase



(b) Random

Figure 3. The average query delay as a function of Zipf parameter θ

LRU by 18% and outperforms LRU-MIN by 12%.

5.4 Simulation Results: Minimizing the Downlink Traffic

The downlink bandwidth determines the amount of data the server can broadcast in one IR interval. If we reduce the downlink traffic, the server can handle more requests, and hence, the server can serve more clients or clients can make more requests. In order to minimize the downlink traffic, we change the general cost function to meet this specific requirement. As a result, f_i will be all the downlink bandwidth needed to fetch item i to cache, c is the downlink bandwidth needed for cache invalidation, and v_i is the downlink bandwidth needed to download the data.

Similar to Section 5.3, we compare the performances of four algorithms, LRU, LRU-MIN, OPT and OPT (IDL). Due to space limitation, we only show the results of the "Random" distribution due to the similarity between the "Random" distribution and the "Increase" distribution. The performance is measured by the average downlink traffic, which is the overall downlink traffic divided by the number of queries.

5.4.1 The Average Downlink Traffic Under Different Cache Sizes

Figure 4 shows that our algorithm always outperforms other algorithms. The OPT (IDL) outperforms LRU or LRU-MIN by more than 26% on average. The OPT is not as good at OPT (IDL), but it still outperforms LRU or LRU-MIN by more than 21% on average.

5.4.2 The Average Downlink Traffic under Different Access Pattern (θ)

Figure 5 shows the impact of access pattern on the average downlink traffic. When θ is small, the access is uniformed

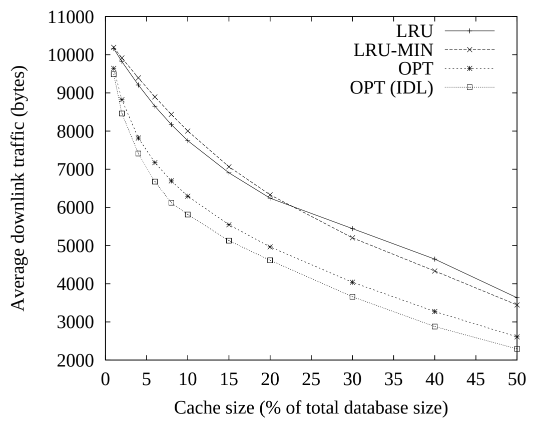


Figure 4. The average downlink traffic as a function of the cache size

distributed. The performances of four algorithms are similar because the cache hit-ratio is very low and there is less room for performance improvement. When θ increases, more accesses are focused on few items. As a result, it became important to cache the right data items and hence the performance difference between four algorithms increases. When $\theta = 1$, compared to LRU (which now performs better than LRU-MIN), OPT can reduce downlink traffic by about 1.5K per query (21%) and OPT (IDL) can reduce about 1.9K per query (27%).

6 Conclusions

In this paper, we propose a generalized cost function for cache replacement algorithms in mobile environments. Based on this generalized cost function, we derive two cost functions to satisfy two specific targets: minimize the query

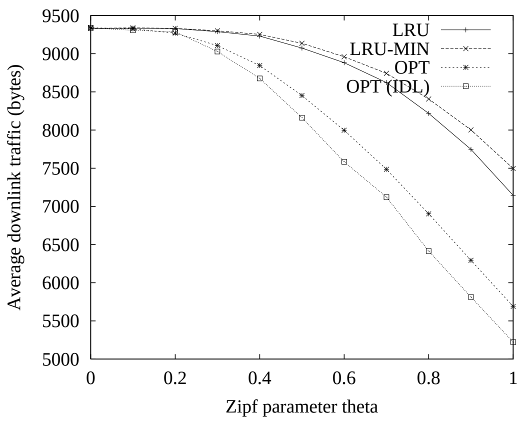


Figure 5. The average downlink traffic as a function of θ

delay and minimize the downlink traffic. Detailed experiments are carried out to evaluate the effectiveness of these cost functions. In both simulations, our cache replacement policy can significantly improve the performance compared to the LRU algorithm and the LRU-MIN algorithm.

References

[1] M. Abrams, C. Standridge, G. Abdulla, S. Williams, and E. Fox, "Caching Proxies: Limitations and Potential," *Fourth International World-Wide Web Conf.*, Dec. 1995.

[2] S. Acharya, M. Franklin, and S. Zdonik, "Prefetching from a Broadcast Disk," *IEEE*, pp. 267–285, 1996.

[3] D. Barbara and T. Imielinski, "Sleepers and Workaholics: Caching Strategies for Mobile Environments," *ACM SIGMOD*, pp. 1–12, 1994.

[4] J. Bolot and P. Hoschka, "Performance Engineering of the World Wide Web: Application to Dimensioning and Cache Design," *Fifth International World-Wide Web Conf.*, 1996.

[5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web Caching and Zipf-like Distributions: Evidence and Implications," *Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, 1999.

[6] G. Cao, "Proactive Power-Aware Cache Management for Mobile Computing Systems," *IEEE Transactions on Computer*, June 2002.

[7] G. Cao, "A Scalable Low-Latency Cache Invalidation Strategy for Mobile Environments," *IEEE Transactions on Knowledge and Data Engineering*, to appear (a preliminary version appeared in MOBICOM00).

[8] C. Cunha, A. Bestavros, and M. Crovella, "Characteristics of WWW client-based traces," *Technical Report TR-95-010, Boston University*, June 1995.

[9] S. Glassman, "A Caching Relay for the World Wide Web," *Computer Networks and ISDN Systems*, vol. 27, 1994.

[10] Q. Hu and D. Lee, "Cache Algorithms based on Adaptive Invalidation Report for Mobile Environments," *Cluster Computing*, pp. 39–48, Feb. 1998.

[11] J. Jing, A. Elmagarmid, A. Helal, and R. Alonso, "Bit-Sequences: An adaptive Cache Invalidation Method in Mobile Client/Server Environments," *Mobile Networks and Applications*, pp. 117–129, 1997.

[12] J. Shim, P. Scheuermann, and R. Vingralek, "Proxy Cache Algorithms: Design, Implementation, and Performance," *IEEE Transactions on Knowledge and Data Engineering*, vol. 11, July/August 1999.

[13] W.R. Steven, "TCP/IP illustrated," *Addison-Wesley*, vol. 3, 1996.

[14] R. Wooster and M. Abrams, "Proxy Caching that Estimates Page Load delays," *Proc. Sixth International World-Wide Web Conf.*, 1997.

[15] K. Wu, P. Yu, and M. Chen, "Energy-efficient caching for wireless mobile computing," *The 20th Intl. Conf. on Data Engineering*, pp. 336–345, Feb. 1996.

[16] J. Xu, Q. Hu, W. Lee, D. Lee, "Performance Evaluation of an Optimal Cache Replacement Policy for Wireless Data Dissemination under Cache Consistency," *2001 Int'l conference on parallel processing*, Sept. 2001.

[17] L. Yin, G. Cao, C. Das, and A. Ashraf, "Power-Aware Prefetch in Mobile Environments," *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2002.

[18] G. Zipf, "Human Behavior and the Principle of Least Effort," *Addison-Wesley*, 1949.