Available online at www.sciencedirect.com
SCIENCE DIRECT®
J. Parallel Distrib. Comput. 65 (2005) 583–594
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

A generalized target-driven cache replacement policy for mobile environments

Liangzhong Yin, Guohong Cao*, Ying Cai[1]

Department of Computer Science & Engineering, The Pennsylvania State University, University Park, PA 16802, USA

Received 2 November 2003; received in revised form 10 November 2004; accepted 19 December 2004

Abstract

Caching frequently accessed data items on the client side is an effective technique to improve the system performance in wireless networks. Due to cache size limitations, cache replacement algorithms are used to find a suitable subset of items for eviction from the cache. Many existing cache replacement algorithms employ a value function of different factors such as time since last access, entry time of the item in the cache, transfer time, item expiration time and so on. However, most of the existing algorithms are designed for WWW environment under weak consistency model. Their choices of value functions are based on experience and on a value function which only works for a specific performance metric.

In this paper, we propose a generalized value function for cache replacement algorithms for wireless networks under a strong consistency model. The distinctive feature of our value function is that it is generalized and can be used for various performance metrics by making the necessary changes. Further, we prove that the proposed value function can optimize the access cost in our system model. To demonstrate the practical effectiveness of the generalized value function, we derive two specific functions and evaluate them by setting up two different targets: minimizing the query delay and minimizing the downlink traffic. Compared to previous schemes, our algorithm significantly improves the performance in terms of query delay or in terms of bandwidth utilization depending on the specified target.

Keywords: Cache replacement; Cost model; Value function; Mobile computing; Cache invalidation; Cache consistency model

1. Introduction

With the explosive growth of wireless techniques and mobile devices such as laptops, personal digital assistants, people with battery powered mobile devices wish to access various kinds of services at any time any place. However, existing wireless services are limited by the constraints of wireless networks such as narrow bandwidth, frequent disconnections, and limitations of the battery technology. Thus, mechanisms to efficiently transmit information from the server to a massive number of clients (running on mobile devices) have received considerable attention [4,7,9,14,19].

Caching frequently accessed data items on the client side is an effective technique to improve performance in a mobile environment [4]. Average data access latency is reduced as some data access requests can be satisfied from the local cache thereby obviating the need for data transmission over the scarce wireless links. 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.

Cache replacement algorithms have been extensively studied in the context of operating system virtual memory management and database buffer management [11]. In this context, cache replacement algorithms usually maximize the cache hit-ratio by attempting to cache the items that are most likely to be accessed in the future. However, these algorithms may not be suitable for wireless networks due to a number of reasons [10]: First, the data items may have

* Corresponding author. Fax: +1 814 865 3176.
E-mail addresses: yin@cse.psu.edu (L. Yin), gcao@cse.psu.edu (G. Cao), yingcai@cs.iastate.edu (Y. Cai).
[1] Ying Cai is with Department of Computer Science at Iowa State University.

0743-7315/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2004.12.002

different sizes and then the *least recently used* (*LRU*) policy needs to be extended to handle items of varying sizes. Second, data items may be constantly updated at the server side. Thus the consistency issue shall be considered. That is, data items that tend to be inconsistent earlier should be replaced earlier. Third, the cost to download data items from the server may vary. As a result, the cache hit-ratio may not be the best measurement for evaluating the quality of a cache replacement algorithm.

Aggarwal et al. [3] classifies the existing cache replacement policies into three categories: *direct-extension*, *key-based*, *and function-based*. In the direct-extension category [17], traditional policies such as LRU or FIFO are extended to handle data items of non-homogeneous size. The difficulty with such policies in general is that they fail to pay sufficient attention to the data size. In the key-based policies [21], keys are used to prioritize some replacement factors over others; however, such prioritization may not always be ideal.

Recently, function-based replacement policy has received considerable attention [3,5,18,22,24]. The idea in function-based replacement policies is to employ a function of the different factors such as time since last access, entry time of the data item in the cache, transfer time, data item expiration time and so on. For example, the algorithm [5] proposed by Bolot and Hoschka first explicitly considers the delay to fetch web documents in cache replacement. Their value function employs a weighted function of the transfer time, the document size, and the time since last access. However, the choice of the value function is not justified and there are many unspecified weights. The Hybrid Algorithm (HYB) [22] addresses both latency and bandwidth issues. Their value 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. [18], aims to minimize the response time. Their value 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 value 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 [24] is designed for strong cache consistency model. It uses an optimal value function that can minimize the metric *stretch*.[2] Although the authors proved that their value function is optimal, they did not show how to get such an optimal value function.

These function-based policies are valuable in that they address various aspects of cache replacement. However, these algorithms are designed for a specific metric (target). When the target changes, they have to come up with another function. Furthermore, these functions may not even be optimal. In this paper, we propose a novel approach for cache replacement. We first present a cache access cost model for wireless networks and show how to break-down the data access cost and how to use caching to improve the system performance. Based on the cost model, we propose a generalized value function, and prove that the proposed value function can minimize the access cost in ideal situations. Since our value 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 generalized value function, we derive two specific functions by setting up 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 existing policies under various cache sizes, update time, query generate time, and access patterns.

The rest of the paper is organized as follows. Section 2 presents the system model. In Section 3, we present the generalized value 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

[2] 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.

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 wireless networks 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 invalidation report-based cache invalidation approach [4] 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 [14,16,23] are proposed to address the long disconnection problem, and Hu et al. [14] 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 [9] 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 those in [4,14,16].

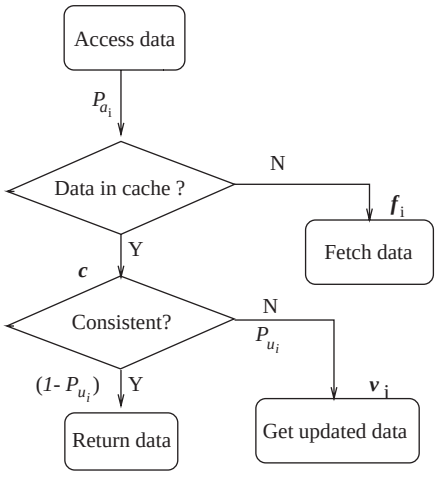

Fig. 1. The cache access cost model.

Similar to other researchers, we assume that data accesses/updates follow Poisson distribution and the independent reference model [11]. The Poisson arrivals are usually used to model data access and update processes [15]. The independent reference model has been adopted by many researchers [6,24] and it explains the access behavior well [6].

3. A generalized target-driven cache replacement algorithm

To facilitate our discussion, the following notations are used. Fig. 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.

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

$$max \sum_{i \in V} value(i),$$

where

$$value(i) = P_{a_i}(f_i - c - P_{u_i} * v_i). \quad (1)$$

This value function can be explained by the cache access cost model shown in Fig. 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 value 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 set (V^*) of items to be replaced which satisfies the following two conditions:

$$\text{(a)} \quad \sum_{i \in V^*} s_i \geqslant s,$$

$$\text{(b)} \quad \forall V_k \left(V_k \subseteq V \wedge \sum_{i \in V_k} s_i \geqslant s \right), \quad \sum_{i \in V_k} value(i) \geqslant \sum_{i \in V^*} value(i). \tag{2}$$

Intuitively V^* is the least valuable 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 the *General_Opt* algorithm found. B is an arbitrary set whose total size $\sum_{i \in B} s_i \geqslant s$. $A \bigcap B$ is assumed to be an empty set $\emptyset$: otherwise, we can remove the intersecting elements since their values are equal under both algorithms. According to the algorithm

$$\sum_{i \in A} value(i) \leqslant \sum_{j \in B} value(j).$$

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,

$$\begin{aligned} T_A - T_B &= \left(\sum_{i \in A} P_{a_i} * f_i + \sum_{j \in B} P_{a_j} * (c + P_{u_j} * v_j) \right) \\ &\quad - \left(\sum_{i \in B} P_{a_i} * f_i + \sum_{j \in A} P_{a_j} * (c + P_{u_j} * v_j) \right) \\ &= \left(\sum_{i \in A} P_{a_i} * f_i - \sum_{j \in A} P_{a_j} * (c + P_{u_j} * v_j) \right) \\ &\quad - \left(\sum_{i \in B} P_{a_i} * f_i - \sum_{j \in B} P_{a_j} * (c + P_{u_j} * v_j) \right) \\ &= \sum_{i \in A} value(i) - \sum_{j \in B} value(j) \leqslant 0. \end{aligned}$$

Thus, the *General_Opt* algorithm replaces a set of data times which can minimize the total access cost. □

Based on the generalized value function, we can derive specific value 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 can also derived other specific value functions such as minimizing the downlink traffic as shown in Section 5.

4. Implementation issues

In the *General_Opt* algorithm, the optimization problem defined by Eq. (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 size is relatively small compared to the cache size [18], we can use heuristics to obtain sub-optimal solutions. The heuristic we will use is: throw out the cached data item i with the minimum $\frac{value(i)}{s_i}$ value until the free cache space is sufficient to accommodate the incoming data.

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 use the exponential aging method, which has been adopted in TCP [20] to estimate the round trip 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 recalculated as following:

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

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

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 data accesses and updates follow Poisson distribution, the probability that the cache invalidation happens before the next data access is [24]:

$$P_{u_i} = Pr(T_{u_i} < T_{a_i}) = \int_0^{\infty} \int_0^{T_{a_i}} a_i e^{-a_i T_{a_i}} u_i e^{-u_i T_{u_i}} \, dT_{u_i} \, dT_{a_i} = \frac{u_i}{a_i + u_i}.$$

So, the value function of Eq. (1) can be replaced by the following:

$$value(i) = a_i * \left(f_i - c - \frac{u_i}{a_i + u_i} * v_i \right). \tag{3}$$

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. [18] to estimate a_i and u_i. This method uses K most recent samples to estimate a_i and u_i as follows

$$a_i = \frac{K}{T - T_{a_i}(K)}, \tag{4}$$

$$u_i = \frac{K}{T - T_{u_i}(K)}, \tag{5}$$

where T is the current time, $T_{a_i}(K)$ and $T_{u_i}(K)$ are the time of the Kth most recent accesses and updates. If less than K samples are available, all the available samples are used. Shim et al. [18] showed that K can be as small as 2 or 3 to achieve the best performance. Thus, the spatial overhead of storing recent accesses and updates is relatively small.

One concern about implementing the algorithm is the computational overhead. Eqs. (4) and (5) imply that the value for each item in the cache needs to be recalculated whenever a cache replacement is necessary. This computational overhead it very high. To reduce the computational overhead, we propose the following approximation method. Whenever a cache replacement is necessary, instead of recalculating the value of every data item, we only recalculate the values of the first $2^{\delta} - 1$ items in the heap (The heap structure will be described in the following section.) For data items in the heap, this can guarantee that the value of the δ least valuable items will be recalculated. Most likely, the items to be replaced will be among them because their values are relatively small. Simulation results (Fig. 3 (a)) verifies that $\delta = 3$ can provide satisfying performance and the computational overhead is very small.

4.2. Cache insertion and removal

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

4.3. The client management algorithm

The client-side cache management algorithm is shown in Fig. 2.

5. Performance evaluations

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 generalized function. The first target is to minimize the query delay; whereas the second is to minimize the downlink traffic.

5.1. The simulation model

In the simulation, a single server maintains a collection of n data items and a number of clients access these data items. The UIR cache invalidation model is adopted for data dissemination.

5.1.1. The client model

The client query model is similar to what have been used in our previous studies [8,25]. 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 [26], which has been frequently used [6,12] to model non-uniform distribution. In the Zipf distribution, the access probability of the ith ($1 \leqslant i \leqslant n$) data item is represented as follows

$$P_{a_i} = \frac{1}{i^{\theta} \sum_{k=1}^{n} \frac{1}{k^{\theta}}},$$

```
(A) When a client generates a query for data item i:
    updates a_i;
    if i is valid in the cache then
        wait for the validation report, UIR or IR;
        listen_invalidation_report();
        if i was not updated in the last cache invalidation then
            adjust the position of i in the heap;
            return i from the cache;
        else
            go to step (B);
    else
        go to step (B);

(B) When there is a cache miss for item i:
    send out a request and wait for the next IR;
    listen_invalidation_report();
    get i from the broadcast channel;
    if i is already in the cache then //data was updated at the server
        update data i in the cache;
        adjust the position of i in the heap;
    else if there is enough free space then
        insert item i into the cache and the heap;
    else
        while there is not enough space do
                // remove the item with the least value/s value
                remove the top item from the heap and clear the related data from the cache;
        insert i into the cache and the heap;

(C) listen_invalidation_report()
    listen to the IR or UIR from the server;
    for any item i that has been updated since last invalidation report
        update the mean update rate u_i;
        change the cached item to be invalid if i is in cache.
```

Fig. 2. The client cache management algorithm.

where $0 \leqslant \theta \leqslant 1$. When $\theta = 1$, it is the strict Zipf distribution. When $\theta = 0$, it becomes the uniform distribution. Large θ results in more "skewed" access distribution.

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 serves the requests during the next IR interval on a FCFS (first-come-first-service) basis. There are n data items at the server side. The data size varies from $s_{\min}$ to $s_{\max}$, which follows the following two types of distributions:

- *Random*: The distribution of the 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 data size distribution and Zipf access pattern defines the joint distribution of access frequency and data size. The choices of the data size distributions are based on previously published trace analyses. Some analyses [12,13] showed that small data items are accessed more frequently than large items; while a recent web trace analysis [6] showed 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 interarrival time with mean value of T_{update}. The data items in the database are divided into hot (frequently accessed) 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, and the ranges of these parameters are listed in the third column.

Experiments are run using different workloads and system settings. The performance analysis presented here is designed to compare the effects of different workload parameters such as mean update arrival time, mean query generate time, and system parameters such as cache size and Zipf parameter θ on the performance of our and other algorithms.

Table 1
Simulation parameters and their default values

Parameter	Default value	Range
Database size (n)	3000 items	
Region size	50 items	
Number of clients	100	
s_{min}	0.5k	
s_{max}	20k	
Mean update time (T_{update})	100 s	10–10000 s
Hot update prob.	0.8	
Hot subset percentage	0.2	
Broadcast interval (L)	20 s	
Broadcast window (w)	10 interval	
Broadcast bandwidth	144 kb/s	
Relative cache size	10% of total database size	1–50%
Mean query generate time (T_{query})	100 s	30–300 s
Zipf distribution parameter θ	0.9	0–1

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. For each workload parameter (e.g., the mean update arrival time, or the mean query generate time), the mean value of the measured data is obtained by collecting a large number of samples such that the confidence interval is reasonably small. In most cases, the 95% confidence interval for the measured data is less than 10% of the sample mean.

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.
- *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 is needed, try the items of size at least $S/4$. This algorithm is shown to perform very well when the data size is different.
- *OPT*: This is our algorithm. It keeps removing the item with least $value(i)/s_i$ value where the value function is defined by Eq. (3).
- *OPT (IDL)*: We also simulate an ideal case, where the access rate and the update rate are known as *a priori*. 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. As shown in Eq. (3), f_i is 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.

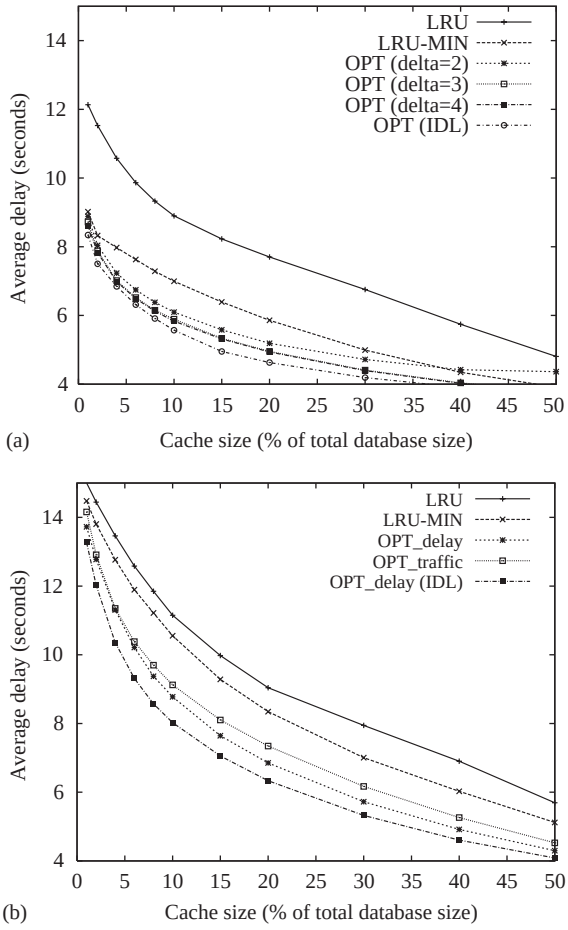

Fig. 3. The average query delay as a function of the cache size: (a) Increase; (b) Random.

In the simulation results, we show the average query delay as a function of different factors such as cache size, mean query generate time, etc. The average query delay is the total query delay divided by the number of queries.

5.3.1. The average delay under different cache size

Fig. 3 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.

To study the effect of δ, we compare the simulation results of OPT when δ is 2, 3 and 5 in Fig. 3(a). As described in Section 4.1, for δ is 2, 3 or 5, our algorithm will recalculate the value of the first 3, 7 or 31 items, respectively, in the heap. It is obvious that our algorithm with bigger δ will get better performance at the cost of more computational overhead. As shown in Fig. 3(a), OPT ($\delta = 3$) has similar performance to OPT ($\delta = 5$). The computational overhead

of OPT ($\delta = 3$) is about 3/4 less. This shows that satisfying performance can be achieved with small δ value such as 3. Therefore, in the following of the paper, we will choose $\delta = 3$, and refer the OPT ($\delta = 3$) algorithm as OPT.

In Fig. 3 (b), we also include the average delay of the algorithm (discussed in Section 5.4) whose target is to minimize the downlink traffic. Here, this algorithm is denoted as OPT_traffic. As we have derived two algorithms for two different metrics: the delay and the downlink traffic, we want to cross-compare each algorithm under the other algorithm's target metric. Fig. 3(b) shows that the performance of OPT_traffic is not as good as OPT when the performance metric is delay. See Section 5.4.1 for further discussion about the cross-comparison.

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 Fig. 3, 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 (Fig. 3(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 and 17% less than that of LRU-MIN when the relative cache size is 10%.

For the "Increase" distribution, there are correlation between access rate and data size. Thus, the algorithms that consider data size will have better performance than those that do not. For example, in Fig. 3 (a), the difference between LRU and other algorithms is much larger than that in Fig. 3 (b).

5.3.2. The average delay under different update arrival time

The mean update arrival time (T_{update}) determines how frequently the server updates its data items. As shown in Fig. 4, our algorithms are much better than LRU and LRU-MIN. For example, in Fig. 4(b), when the update arrival time is 10 s, the average query delay of OPT is 18% less than that of the LRU-MIN algorithm. When the update arrival time is in the range of 1000–10, 000 s, the performance of OPT is about 16% better than that of LRU-MIN. Of course, OPT (IDL) performances better. On average, it is about 25% better than LRU-MIN. Similar results can be found in Fig. 4(a).

5.3.3. The average delay under different query generate time

Fig. 5 shows the average query delay as a function of T_{query}. As explained before, each client generates queries according to the mean query generate time. The generated queries are served one by one. If the queried data is in the local cache, the client can serve the query locally; otherwise, the client has to request the data from the server. If the client cannot process the generated query due to waiting for the server reply, it queues the generated queries. Since the broadcast bandwidth is fixed, the server can only transmit a limited amount of data during one IR interval. If the server receives more queries than it can broadcast during one IR interval, some queries are delayed to the next IR interval. If the server receives more queries than it can broadcast during each IR interval, many queries may not be served, and the query delay may be out of bound. As we can see from Fig. 5, the delay of OPT and OPT (IDL) is much less than that of LRU and LRU-MIN. This is due to the reason that OPT and OPT (IDL) use the cache space more effectively and the number of queries sent to the server can be reduced, and hence, the server is less likely to be overwhelmed by the clients' requests.

For the "Increase" distribution, the average delay is much less than the "Random" distribution for the same reason mentioned in Section 5.3.1. As T_{query} increases, the average query delay decreases since less queries are generated and the server can serve the queries more quickly. As shown in Fig. 5(a), the average delay increases slightly when T_{query} is larger than 150 s. This is because the chance that cached items are invalid increases as T_{query} increases. As mentioned

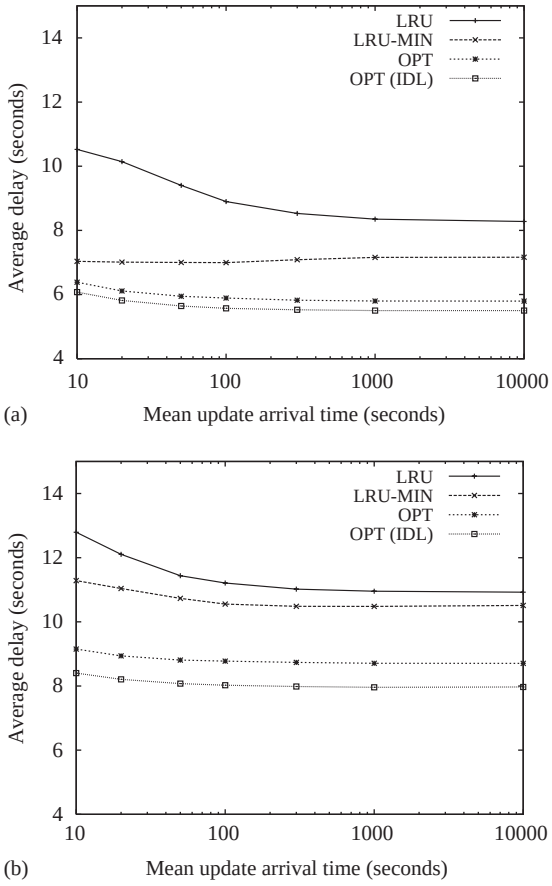

Fig. 4. The average query delay as a function of the mean update time arrival time: (a) Increase; (b) Random.

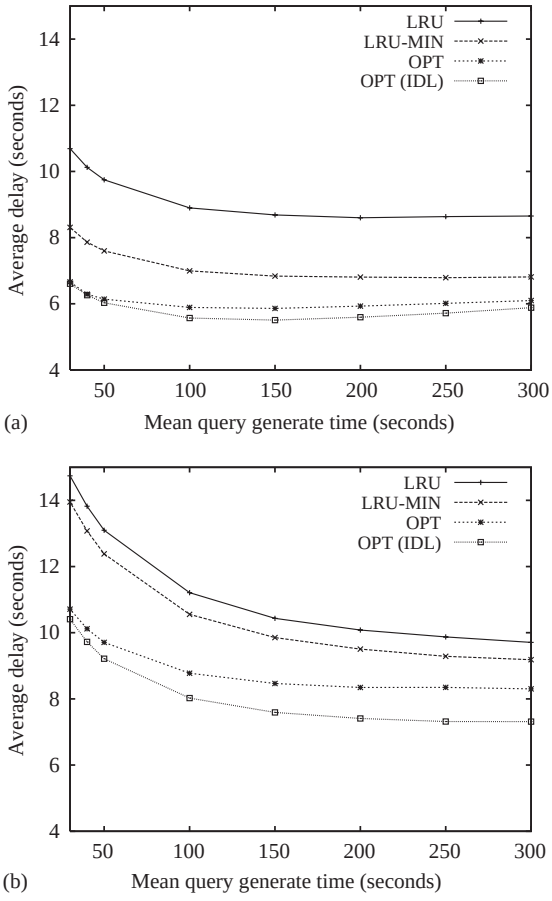

Fig. 5. The average query delay as a function of the mean query generate time: (a) Increase; (b) Random.

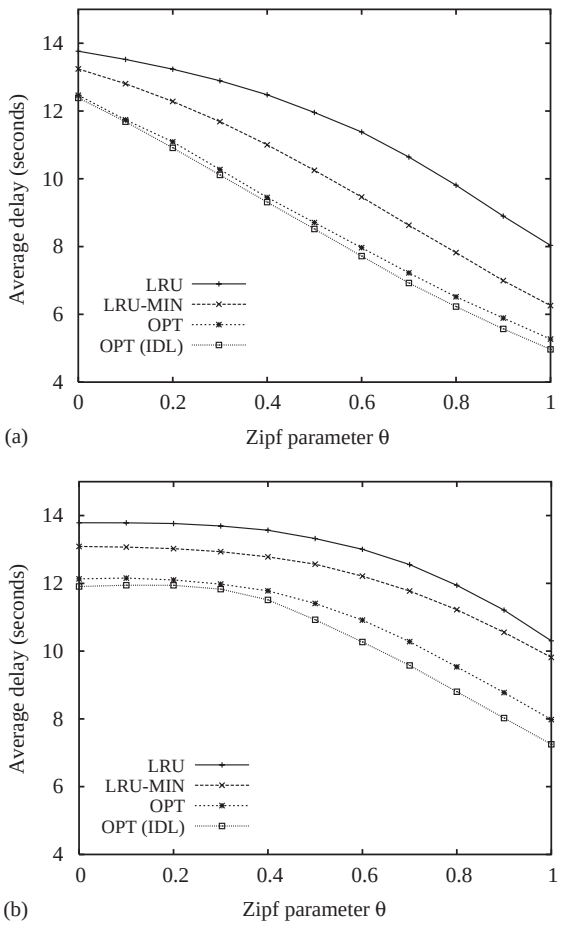

Fig. 6. The average query delay as a function of Zipf parameter θ: (a) Increase; (b) Random.

before, the average query delay is the total delay divided by the number of queries. Thus, the average query delay increases as T_{query} is larger than 150 s. The same trend exists in the OPT of Fig. 5(b) although it is less obvious. The reason is as following. There are two conflicting factors that affect the average query delay when T_{query} increases: (1) the server receives less number of requests and the average query delay tends to decrease. (2) the chance of a cached item being invalid increases, and the average query delay tends to increase. In the "Random" distribution, the server receives more requests than in the "Increase" distribution. When T_{query} increase, the effect of the first factor is bigger than that of the second factor. As a result, the increasing trend in Fig. 5 (b) is less obvious or does not exist.

5.3.4. The average delay under different access pattern (θ)

The Zipf parameter θ determines the "skewness" of the access distribution. Fig. 6 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 the size of data items does not matter. 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 Fig. 6, OPT and OPT (IDL) constantly outperform LRU and LRU-MIN. In Fig. 6(a), on average, OPT outperforms LRU by 22% and outperforms LRU-MIN by 11%. In Figure 6(b), on average, OPT outperforms 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 that 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 generalized value function to meet this specific requirement as follows: 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

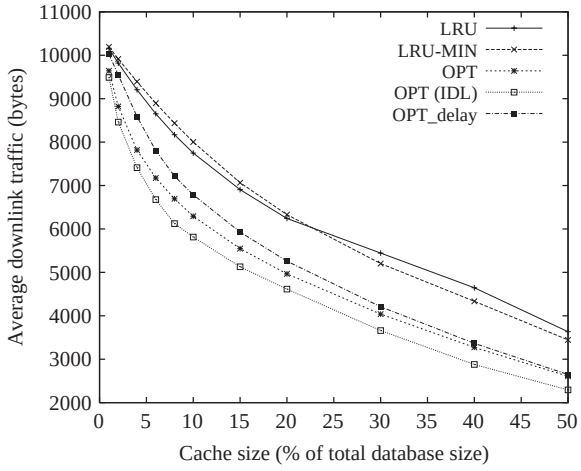

Fig. 7. The average downlink traffic as a function of the cache size.

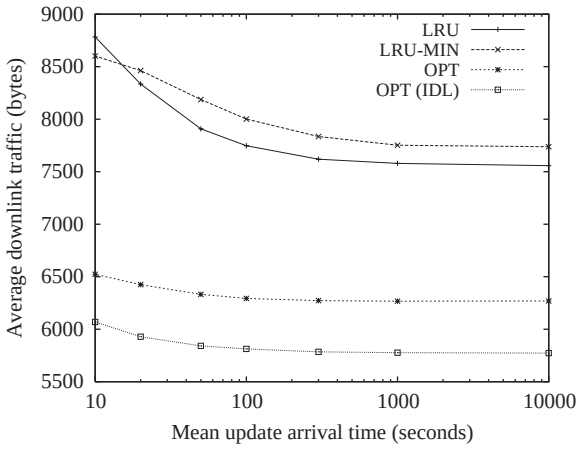

Fig. 8. The average downlink bandwidth as a function of the update arrival time.

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

We also include the simulation results of the delay-optimal algorithm (denoted as OPT_delay in Fig. 7) used in Section 5.3. As can be seen, although OPT_delay is good at reducing the query delay, it is outperformed by our OPT algorithm in term of downlink traffic, because OPT is specifically designed to minimize the downlink traffic. Another interesting thing is that the LRU-MIN algorithm does not show its advantage over LRU. These results, together with the results shown in Fig. 3 (b), show that it is possible for one specific algorithm to work better than others in terms of one specific metric, but not others.

Fig. 7 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. For simplicity, we will not show OPT_delay in later presentations.

5.4.2. The average downlink traffic under different update arrival time

Fig. 8 shows the average downlink bandwidth as a function of the update arrival time T_{update}. The trend here is similar to that in Fig. 4. Compared to LRU or LRU-MIN, OPT (IDL) can reduce the downlink traffic by 2.3k per query (29%) on average, whereas the OPT algorithm can reduce the download traffic by about 1.8k per query (22%).

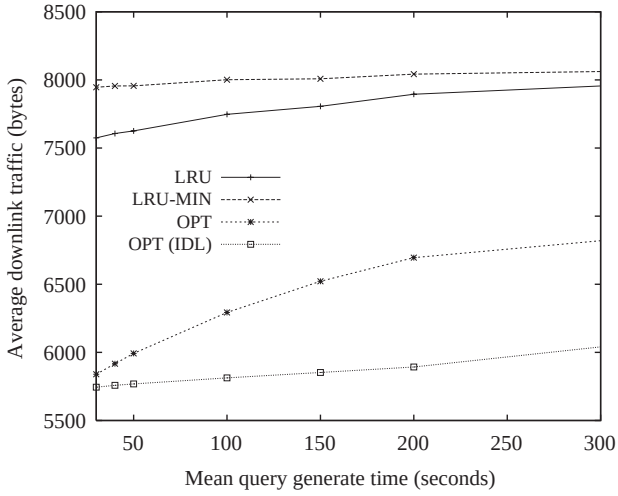

Fig. 9. The average downlink traffic as a function of the query generate time.

5.4.3. The average downlink traffic under different query generate time

Fig. 9 shows the relationship between the downlink traffic and the query generate time T_{query}. As can be seen, the average downlink traffic increases when T_{query} increases. This can be explained by the following two reasons: (1) when T_{query} increases, the possibility that a cached item is invalidated increases; (2) when T_{query} decreases, fewer queries are generated in the same IR interval, and hence, the chance that two or more clients generate the same query decreases. Note that if several clients request for the same data item during the same IR interval, the server only broadcasts the data item once. As less broadcasting data is shared, the average downlink traffic increases. Not surprisingly, OPT outperforms LRU and LRU-MIN while OPT (IDL) always performs the best.

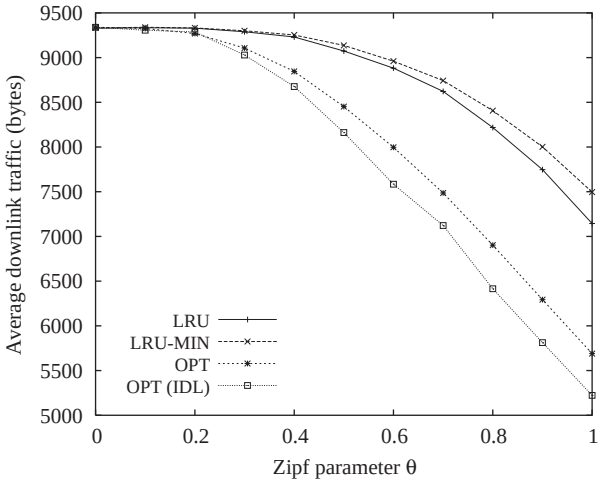

Fig. 10. The average downlink traffic as a function of θ.

5.4.4. The average downlink traffic under different access pattern (θ)

Fig. 10 shows the impact of data access pattern on the average downlink traffic. When θ is small, the access pattern is uniformed distributed. The performance of these four algorithms is 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 and hence the performance difference among these four algorithms increases. When $\theta = 1$, compared to LRU (which now performs better than LRU-MIN), OPT can reduce downlink traffic by about 21% and OPT (IDL) can reduce the traffic by about 27%.

6. Conclusions

In this paper, we proposed a generalized value function for cache replacement algorithms in wireless networks. Based on this generalized value function, we derived two value functions to satisfy two specific targets: minimize the query delay and minimize the downlink traffic. Detailed experiments have been carried out to evaluate the effectiveness of these value functions. In both simulations, our cache replacement policy can significantly improve the system performance compared to existing algorithms under various cache sizes, update time, query generate time and access patterns.

Our algorithm solves the problem of *how to provide cache replacement algorithm given an optimization target*. There are still some existing problems to be solved, for example, how to set the optimization target? There may be several optimization targets, and it is a challenge to find a way to balance these optimization targets. As future work, we will address these issues and propose cache replacement algorithms when multiple optimization targets exist.

Acknowledgments

We would like to thank the editor and the anonymous referees whose insightful comments helped us to improve the presentation of the paper. This work was supported in part by the National Science Foundation (CAREER CNS-0092770 and ITR-0219711).

References

[1] M. Abrams, C. Standridge, G. Abdulla, S. Williams, E. Fox, Caching proxies: limitations and potential, Fourth International World-Wide Web Conference, December 1995.
[2] S. Acharya, M. Franklin, S. Zdonik, Prefetching from a Broadcast Disk, IEEE, (1996) 267–285.
[3] C. Aggarwal, J. Wolf, P. Yu, Caching on the World Wide Web, IEEE Trans. Knowledge Data Eng. 11 (January/February 1999).
[4] D. Barbara, T. Imielinski, Sleepers and workaholics: caching strategies for mobile environments, ACM SIGMOD, 1994, pp. 1–12.
[5] J. Bolot, P. Hoschka, Performance engineering of the World Wide Web: application to dimensioning and cache design, Fifth International World-Wide Web Conference, 1996.
[6] L. Breslau, P. Cao, L. Fan, G. Phillips, S. Shenker, Web caching and zipf-like distributions: evidence and implications, The 18th Annual Joint Conference of the IEEE Computer and Communications Societies, 1999.
[7] G. Cao, On Improving the performance of cache invalidation in mobile environments, ACM/Baltzer Mobile Networks and Application (MONET) 7 (4) (August 2002) 291–303.
[8] G. Cao, Proactive power-aware cache management for mobile computing systems, IEEE Trans. Comput. (June 2002).
[9] G. Cao, A scalable low-latency cache invalidation strategy for mobile environments, IEEE Trans. Knowledge Data Eng. 15 (5) (September/October 2003) 1251–1265 (a preliminary version appeared in ACM MobiCom'00).
[10] P. Cao, S. Irani, Cost-Aware WWW proxy caching algorithm, Proceedings of the Usenix Symposium on Internet Technologies and System, 1997.
[11] E. Coffman, P. Denning, Operating System Theory, Prentice-Hall, Englewood Cliff, NJ, 1973.
[12] C. Cunha, A. Bestavros, M. Crovella, Characteristics of WWW client-based traces, Technical Report TR-95-010, Boston University, June 1995.
[13] S. Glassman, A caching relay for the World Wide Web, Comput. Networks ISDN Systems 27 (1994).
[14] Q. Hu, D. Lee, Cache algorithms based on adaptive invalidation report for mobile environments, Cluster Comput. (February 1998) 39–48.
[15] R. Jain, The Art of Compute System Performance Analysis, Wiley, New York, 1991.
[16] J. Jing, A. Elmagarmid, A. Helal, R. Alonso, Bit-Sequences: an adaptive cache invalidation method in mobile client/server environments, Mobile Networks Appl. (1997) 117–129.
[17] J. Pitkow, M. Recker, A simple yet robust caching algorithm based on dynamic access patterns, Proceedings of the Second International World Wide Web Conference, 1994.
[18] J. Shim, P. Scheuermann, R. Vingralek, Proxy cache algorithms: design, implementation, performance, IEEE Trans. Knowledge Data Eng. 11 (July/August 1999).
[19] H. Song, G. Cao, Cache-miss-initiated prefetch in mobile environments, IEEE International Conference on Mobile Data Management (MDM), January 2004 (an enhanced version accepted by Computer Communications).

[20] W.R. Steven, TCP/IP Illustrated, vol. 3, Addison-Wesley, Reading MA, 1996.
[21] S. Williams, M. Abrams, C. Standridge, G. Abdulla, E. Fox, Removal policies in network caches for World-Wide Web documents, Proceedings of the ACM Sigcomm, 1996.
[22] R. Wooster, M. Abrams, Proxy caching that estimates page load delays, Proceedings of the Sixth International World-Wide Web Conference, 1997.
[23] K. Wu, P. Yu, M. Chen, Energy-efficient caching for wireless mobile computing, The 20th International Conference on Data Engineering, February 1996, pp. 336–345.
[24] J. Xu, Q. Hu, W. Lee, D. Lee, Performance evaluation of an optimal cache replacement policy for wireless data dissemination under cache consistency, 2001 International Conference on Parallel Processing, September 2001.
[25] L. Yin, G. Cao, Adaptive power-aware prefetch in wireless networks, IEEE Trans. Wireless Commun. 3 (5) (September 2004) 1648–1658 (a preliminary version appeared in ICDCS'02).
[26] G. Zipf, Human Behavior and the Principle of Least Effort, Addison-Wesley, Reading, MA, 1949.

Liangzhong Yin received the B.E. degree and the M.E. degree in computer science and engineering from the Southeast University, Nanjing, China, in 1996 and 1999, respectively. He received his Ph.D. degree in the Department of Computer Science & Engineering at the Pennsylvania State University in 2004. He currently works for Motorola Inc. His research interests include wireless/ad hoc networks and mobile computing.

Guohong Cao received his B.S. degree from Xian Jiaotong University, Xian, China. He received the M.S. degree and Ph.D. degree in computer science from the Ohio State University in 1997 and 1999 respectively. Since then, he has been with the Department of Computer Science and Engineering at the Pennsylvania State University, where he is currently an Associate Professor. His research interests are mobile computing, wireless networks, and distributed fault-tolerant computing. His recent work has focused on data dissemination, cache management, network security and resource management in wireless networks. He is an editor of the IEEE Transactions on Mobile Computing and IEEE Transactions on Wireless Communications, has served as a co-chair of the workshop on Mobile Distributed Systems, and has served on the program committee of numerous conferences. He was a recipient of the Presidential Fellowship at the Ohio State University in 1999, and a recipient of the NSF CAREER award in 2001.

Ying Cai received his Ph.D. in computer science from the University of Central Florida in 2002. While studying at this university, Dr. Cai was the chief architect at nStor/StorLogic leading the effort to develop network storage technology. He developed the first remote RAID management system back in 1996 and the product was licensed by several major companies including SGI, Adaptec, and NEC. Currently, Dr. Cai is an assistant professor in the Department of Computer Science at Iowa State University. His research interests include wireless networks, mobile computing, and multimedia systems.