Proactive Power-Aware Cache Management for Mobile Computing Systems

Guohong Cao, *Member*, *IEEE*

Abstract—Recent work has shown that *invalidation report (IR)*-based cache management is an attractive approach for mobile environments. However, the IR-based cache invalidation solution has some limitations, such as long query delay, low bandwidth utilization, and it is not suitable for applications where data change frequently. In this paper, we propose a proactive cache management scheme to address these issues. Instead of passively waiting, the clients intelligently prefetch the data that are most likely used in the future. Based on a novel *prefetch-access ratio* concept, the proposed scheme can dynamically optimize performance or power based on the available resources and the performance requirements. To deal with frequently updated data, different techniques (indexing and caching) are applied to handle different components of the data based on their update frequency. Detailed simulation experiments are carried out to evaluate the proposed methodology. Compared to previous schemes, our solution not only improves the cache hit ratio, the throughput, and the bandwidth utilization, but also reduces the query delay and the power consumption.

Index Terms—Invalidation report, power conservation, query latency, caching, mobile computing.

1 Introduction

With the advent of third generation wireless infrastructure and the rapid growth of wireless communication technology such as Bluetooth and IEEE 802.11, mobile computing becomes possible: People with battery powered mobile devices can access various kinds of services at any time any place. However, existing wireless services are limited by the constraints of mobile environments 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 [3], [13], [23], [25].

Broadcasting has been shown to be an effective data dissemination technique for wireless networks in many studies [3], [13]. With this technique, users access data by simply monitoring the channel until the required data appear on the broadcast channel. To efficiently deliver data on the broadcast channels, content organization and data broadcast scheduling should be based on client access patterns. For example, techniques such as broadcast disks [2] were provided to improve system performance by broadcasting hot data items frequently. To reduce client power consumption, techniques such as indexing [13] were proposed to reduce the client tune-in time. The general idea is to interleave index (directory) information with data on the broadcast channels such that the clients, by first retrieving the index information, are able to obtain the arrival time of the desired data items. As a result, a client can enter doze mode most of the time and only wakes up just before the desired data arrive.

Although broadcasting has good scalability and low bandwidth requirement, it has some drawbacks. For example, since a data item may contain a large volume of data (especially in the multimedia era), the data broadcast cycle may be long. Hence, the clients have to wait for a long time before getting the required data. Caching frequently accessed data items at the client side is an effective technique to improve performance in mobile computing systems. With caching, the data access latency is reduced since some data access requests can be satisfied from the local cache, thereby obviating the need for data transmission over the scarce wireless links. When caching is used, cache consistency must be addressed. Although caching techniques used in file systems such as Coda [20], Ficus [19] can be applied to mobile environments, these file systems are primarily designed for a point-to-point communication environment and they may not be applicable to the broadcasting environment.

Recently, many works [3], [6], [7], [5], [15], [25], [23] have shown that *invalidation report (IR)*-based cache management is an attractive approach for mobile environments. In this approach, the server periodically broadcasts an invalidation report in which the changed data items are indicated. Rather than querying the server directly regarding the validation of cached copies, the clients can listen to these IRs over the wireless channel and use them to validate their local cache. The IR-based solution is attractive because it can scale to any number of clients who listen to the IR. However, the IR-based solution has some drawbacks. First, there is a long query latency associated with this scheme since a client must listen to the next IR and use the report to conclude whether its cache is valid or not before answering a query. Hence, the average latency of answering a query is the sum of the actual query processing time and half of the

• *The author is with the Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16802. E-mail: gcao@cse.psu.edu.*

Manuscript received 16 Feb. 2001; revised 21 Nov. 2001; accepted 24 Jan. 2002.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 115747.

IR interval. If the IR interval is long, the delay may not be able to satisfy the requirements of many clients. Second, even though many clients cache the same updated data item, all of them have to query the server and get the data from the server separately. Although the approach works fine for some *cold* data items, which are not cached by many clients, it is not effective for *hot* data items. For example, suppose a data item is frequently accessed (cached) by 100 clients, updating the data item once may generate 100 uplink (from the client to the server) requests and 100 downlink (from the server to the client) broadcasts. Obviously, it wastes a large amount of wireless bandwidth and battery energy.

In our previous work [4], we addressed the first problem with a UIR-based approach. In this approach, a small fraction of the essential information (called updated invalidation report (UIR)) related to cache invalidation is replicated several times within an IR interval and, hence, the client can answer a query without waiting until the next IR. However, if there is a cache miss, the client still needs to wait for the data to be delivered. Thus, both issues (query delay and bandwidth utilization) are related to the cache hit ratio. In this paper, we propose a proactive cache management scheme to improve the cache hit ratio and, hence, reduce the query delay and improve the bandwidth utilization. Instead of passively waiting, clients intelligently prefetch the data that are most likely used in the future. Based on a novel *prefetch-access ratio* concept, the proposed scheme can dynamically optimize performance or power based on the available resources and the performance requirements. Although caching data at the client site can improve performance and conserve battery energy, caching may not be the best option if the broadcast data are frequently updated, in which case, we propose applying different techniques (broadcasting and caching) [8] to deal with different components of the data items based on their update frequency. Extensive experiments are provided and used to evaluate the proposed methodology. Compared to previous schemes, our solution improves the cache hit ratio, the throughput, and the bandwidth utilization with low power.

The rest of the paper is organized as follows: Section 2 develops the necessary background. In Section 3, we propose techniques which can improve the cache hit ratio and the bandwidth utilization with low power. Section 4 evaluates the performance of the proposed solutions. The next two sections give future research directions and summarize the paper.

2 Preliminaries

When cache techniques are used, data consistency issues must be addressed to ensure that clients see only valid states of the data or at least do not unknowingly access data that are stale according to the rules of the consistency model. Problems related to cache consistency have been widely studied in many other systems such as multiprocessor architectures [10], distributed file systems [17], [20], distributed shared memory [18], and client-server database systems. The notion of data consistency is, of course, application dependent. In database systems, data consistency is traditionally tied to the notion of transaction serializability. In practice, however, few applications demand or even want full serializability and more efforts have gone into defining weaker forms of correctness [16]. In this paper, we use the *latest value* consistency model[1] [1], [4], [23], which is widely used in dissemination-based information systems. In the latest value consistency model, clients must always access the most recent value of a data item. This level of consistency is what would arise naturally if the clients do not perform caching and the server broadcasts only the most recent values of items. When client caching is allowed, techniques should be applied to maintain the latest value consistency.

2.1 The IR-Based Cache Invalidation Model

To ensure cache consistency, the server broadcasts invalidation reports (IRs) 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. 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.

In order to save energy, the client (mobile device) may power off most of the time and only turn on during the IR broadcast time. Moreover, a client may be in the doze mode for a long time and it may miss some IRs. Since the IR includes the history of the past w broadcast intervals, the client can still validate its cache as long as the disconnection time is shorter than $w * L$. However, if the client disconnects longer than $w * L$, it has to discard the entire cached data items since it has no way to tell which parts of the cache are valid. Since the client may need to access some data items in its cache, discarding the entire cache may consume a large amount of wireless bandwidth in future queries. Many solutions [12], [15], [24] are proposed to address the long disconnection problem and Hu and Lee [12] have a good survey of these schemes.

2.2 The UIR-Based Cache Invalidation

In order to reduce the query latency, Cao [4] proposed replicating the IRs m times, that is, the IR is repeated every $(\frac{1}{m})\text{th}$ of the IR interval. As a result, a client only needs to wait at most $(\frac{1}{m})\text{th}$ of the IR interval before answering a query. Hence, latency can be reduced to $(\frac{1}{m})\text{th}$ of the latency in the previous schemes (when query processing time is not considered).

Since the IR contains a large amount of update history information, replicating the complete IR m times may consume a large amount of broadcast bandwidth. In order to save the broadcast bandwidth, after one IR, $m - 1$ *updated invalidation reports* (UIRs) are inserted within an IR interval.

1. The Coda file system [20] does not follow the latest value consistency model. It supports a much weaker consistency model to improve performance. However, some conflicts may require manu-configuration and some updated work may be discarded.

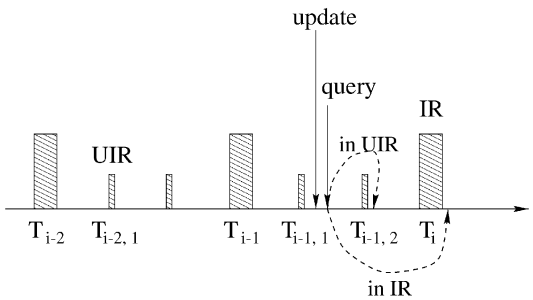

Fig. 1. Reducing the query latency by replicating UIRs.

Each UIR only contains the data items that have been updated after the last IR was broadcast. In this way, the size of the UIR becomes much smaller compared to that of the IR. As long as the client downloads the most recent IR, it can use the UIR to verify its own cache. The idea of the proposed technique can be further explained by Fig. 1. In Fig. 1, $T_{i,k}$ represents the time of the kth UIR after the ith IR. When a client receives a query between $T_{i-1,1}$ and $T_{i-1,2}$, it cannot answer the query until T_i in the IR-based approach, but it can answer the query at $T_{i-1,2}$ in the UIR-based approach. Hence, the UIR-based approach can reduce the query latency in case of a cache hit. However, if there is a cache miss, the client still needs to fetch data from the server, which increases the query latency. Next, we propose a cache management algorithm to improve the cache hit ratio and the bandwidth utilization.

3 A Proactive Power-Aware Cache Management Algorithm

In this section, we present techniques which can improve the cache hit ratio, reduce the power consumption, and efficiently deal with frequently updated data.

3.1 Improve the Cache Hit Ratio by Prefetching

In most previous IR-based algorithms, updating a hot data item may result in many cache misses. We address the problem by asking the clients to prefetch data that may be used in the near future. For example, if a client observes that the server is broadcasting a data item which is an invalid entry[2] of its local cache, it is better to download the data; otherwise, the client may have to send another request to the server and the server will have to broadcast the data again in the future. To save power, clients may only wake up during the IR broadcasting period and then how to prefetch data becomes an issue. As a solution, after broadcasting the IR, the server first broadcasts the id list of the data items whose data values will be broadcast next and then broadcasts the data values of the data items in the id list. Each client should listen to the IR if it is not disconnected. At the end of the IR, a client downloads the id list and finds out when the interested data will come and wakes up at that time to download the data. With this approach, power can be saved since clients stay in the doze mode most of the time; bandwidth can be saved since the server may only need to broadcast the updated data once.

Since prefetching also consumes power, it is very important to identify which data should be included in the id list. Based on whether the server maintains information about the client or not, two cache invalidation strategies are used: the *stateful* server approach and the *stateless* server approach. In [5], [7], we studied the stateful server approach. In the proposed solution, a counter is maintained for each data item. The counter associated with a data item is increased by 1 when a new request for the data item arrives. Based on the counter, the server can identify which data should be included in the id list. Novel techniques are designed to maintain the accuracy of the counter in case of server failures, client failures, and disconnections. However, the stateful approach may not be scalable due to the high state maintenance overhead, especially when handoffs are frequent. Thus, we adopt the stateless approach in this paper. Since the server does not maintain any information about the clients, it is very difficult, if not impossible, for the server to identify which data is hot. To save broadcast bandwidth, the server does not answer the client requests immediately; instead, it waits for the next IR interval. After broadcasting the IR, the server broadcasts the id list (L_{bcast}) of the data items that have been requested during the last IR interval. In addition, the server broadcasts the values of the data items in the id list. At the end of the IR, the client downloads L_{bcast}. For each item id in L_{bcast}, the client checks whether it has requested the server for the item or the item becomes an invalid cache entry due to server update. If either of the two conditions is satisfied, it is better for the client to download the current version since the data will be broadcast.

One important reason for the server not to serve requests until the next IR interval is due to energy consumption. In our scheme, a client can go to sleep most of the time and only wakes up during the IR and L_{blist} broadcast time. Based on L_{blist}, it checks whether there are any interested data that will be broadcast. If not, it can go to sleep and only wakes up at the next IR. If so, it can go to sleep and only wakes up at that particular data broadcast time. For most of the server initiated cache invalidation schemes, the server needs to send the updated data to the clients immediately after the update and the clients must keep awake to get the updated data. Here, we trade off some delay for more battery energy. Due to the use of UIR, the delay trade-off is not that significant; most of the time (cache hit), the delay can be reduced by a factor of m, where $(m-1)$ is the number of replicated UIRs within one IR interval. Even in the worst case (for cache miss), our scheme has the same query delay as the previous IR-based schemes, where the clients cannot serve the query until the next IR. To satisfy time constraint applications, we may apply *priority requests* as follows: When the server receives a priority request, it serves the request immediately instead of waiting until the next IR interval.

3.2 An Adaptive Prefetch Approach

The advantage of the approach depends on how hot the requested data item is. Let us assume that a data item is frequently accessed (cached) by n clients. If the server

2. We assume cache locality exists. When cache locality does not exist, other techniques such as profile-based techniques [9] can be used to improve the effectiveness of the prefetch.

broadcasts the data after it receives a request from one of these clients, the saved uplink and downlink bandwidth can be up to a factor of n when the data item is updated. Since prefetching also consumes power, we investigate the trade-off between performance and power and propose an adaptive scheme to efficiently utilize the power in this section.

Each client may have different available resources and performance requirements and these resources, such as power, may change with time. For example, suppose the battery of a laptop lasts three hours. If the user is able to recharge the battery within three hours, power consumption may not be an issue and the user may be more concerned about the performance aspects such as the query latency. However, if the user cannot recharge the battery within three hours and wants to use it a little bit longer, power consumption becomes a serious concern. As a design option, the user should be able to choose whether to prefetch data based on the resource availability and the performance requirement. This can be done manually or automatically. In the manual option, the user can choose whether the query latency or the power consumption is the primary concern. In the automatic approach, the system monitors the power level. When the power level drops below a threshold, power consumption becomes the primary concern. If query latency is more important than power consumption, the client should always prefetch the interested data. However, when the power drops to a threshold, the client should be cautious about prefetching.

There are two solutions to reduce the power consumption. As a simple solution, the client can reduce its cache size. With a smaller cache, the number of invalid cache entries reduces and the number of prefetches drops. Although small cache size reduces prefetch power consumption, it may also increase the cache miss ratio, thereby degrading performance. In a more elegant approach, the client marks some invalid cache entries as *nonprefetch* and it will not prefetch these items. Intuitively, the client should mark those cache entries that need more power to prefetch, but are not accessed too often.

3.2.1 The Adaptive Prefetch Approach

In order to implement the idea for each cached item, the client records how many times it accessed the item and how many times it prefetched the item during a period of time. The *prefetch-access ratio (PAR)* is the number of prefetches divided by the number of accesses. If the PAR is less than 1, prefetching the data is useful since the prefetched data may be accessed multiple times. When power consumption becomes an issue, the client marks those cache items which have $PAR > \beta$ as *nonprefetch*, where $\beta > 1$ is a system tuning factor. The value of β can be dynamically changed based on the power consumption requirements. For example, with a small β, more energy can be saved, but the cache hit ratio may be reduced. On the other hand, with a large β, the cache hit ratio can be improved, but at a cost of more energy consumption. Note that, when choosing the value of β, the uplink data request cost should also be considered.

When the data update rate is high, the PAR may always be larger than β and clients cannot prefetch any data. Without prefetch, the cache hit ratio may be dramatically reduced, resulting in poor performance. Since clients may have a large probability to access a very small amount of data, marking these data items as prefetch may improve the cache hit ratio and does not consume too much power. Based on this idea, when $PAR > \beta$, the client marks δ number of cache entries which have high access rate as $prefetch$.

Since the query pattern and the data update distribution may change over time, clients should measure their access rate and PAR periodically and refresh some of their history information. Assume N^x_{acc} is the number of access times for a cache entry d_x. Assume $N^x_{c\_acc}$ is the number of access times for a cache entry d_x in the current evaluation cycle. The number of access times is calculated by

$$N^x_{acc} = (1 - \alpha) * N^x_{acc} + \alpha * N_{c\_acc},$$

where $\alpha < 1$ is a factor which reduces the impact of the old access frequency with time. A similar formula can be used to calculate PAR.

3.3 Dealing with Frequently Updated Data

The IR-based approach is very useful in applications where data items do not change frequently and, hence, clients can cache these data items and use them to serve queries locally. However, if the data are frequently updated, caching may not be helpful. In this case, broadcasting the data on the air may be a good solution. Following this idea, many indexing techniques [13] have been proposed to address the trade-off between query latency and power consumption. In most of the indexing techniques, the index and the real data are both broadcast. Since some data items may contain a large amount of data (especially in the multimedia era), the clients may have to wait for a long time before getting the required data. In short, the indexing techniques are good for small data size, while the IR-based approach is good for large data size with less update frequency. However, in real life, most applications may not work well with either approach. For example, although the stock price of a company is updated frequently, the company-related news, such as the company profile, financial news, new product release, and broker coverage, may only be updated several times in a day. Since the stock price is updated too often, the IR-based approach is not suitable. Similarly, indexing techniques should not be used to maintain company-related news due to large data sizes that need to be updated. We propose applying multiple techniques to deal with the problem. The central idea is to differentiate the frequently updated data part from others. In other words, a data item can be divided into two *data components*: the hot component and the cold component. The hot component is the data part which is frequently updated, while the cold component is the data part which is not frequently updated. Indexing techniques are used to access those data components that are frequently updated, whereas IR-based techniques are used to access those data components which are not frequently updated. Considering the above example, indexing techniques (or simple broadcasting) are used to

Notations:

- L, w, d_x, t_x: defined before.
- d_x^c, d_x^h: d_x^h is the hot component of d_x. d_x^c is the cold component of d_x.
- $\mathcal{L}$: the set of hot components.
- $\mathcal{L}_{hot}$: the set of hot components of the hot data items.
- D: the set of data items.
- m: $(m-1)$ is the number of replicated *UIR*s within one IR interval.
- $T_{i,k}$: represents the time of the kth UIR after the ith IR.
- L_{data}: an id list of the data items that a client requested from the server.
- L_{bcast}: an id list of the data items that the server received in the last IR interval. Initialized to be empty.

(A) At interval time T_i, construct IR_i as follows:

$IR_i = \{< d_x, t_x > | \ (d_x \in D) \wedge (T_i - L * w < t_x^c \leq T_i)\}$;
Broadcast IR_i, $\mathcal{L}$, and L_{bcast};

for each $d_x \in L_{bcast}$ **do**
broadcast data item d_x;
Execute Step **B** if the UIR interval reaches.

$L_{bcast} = \emptyset$;

(B) At interval time $T_{i,k}$, construct $UIR_{i,k}$ and $\mathcal{L}_{hot}$ as follows:

$UIR_{i,k} = \{d_x \mid (d_x \in D) \wedge (T_{i,0} < t_x^c \leq T_{i,k})\}$ $(0 < k < m-1)$;
$\mathcal{L}_{hot} = \{d_x^h \mid d_x$ is a hot item $\}$
broadcast $UIR_{i,k}$ and $\mathcal{L}_{hot}$.

(C) Receives a $request(L_{data})$ from client C_j: $L_{bcast} = L_{bcast} \cup L_{data}$.

Fig. 2. The algorithm at the server.

access the stock prices, whereas IR-based techniques are used to access the company news.

To implement the idea, we modify the UIR-based approach so that it can be used to deal with frequent updates. The idea is to broadcast the frequently updated data components multiple times during an IR interval. This can be done by broadcasting them after each UIR or IR. Since most of the frequently updated data components have small data size, broadcasting them should not add too much overhead. If the client access pattern is known, the hot data components should be broadcast more often than the cold data components to reduce the average query latency. If multiple channels are available, the server can use one channel to deliver the frequently updated data components using indexing techniques and use another channel to deliver the cold components using the UIR-based techniques. The process of dividing a data item into two components should be mutually agreed upon by the clients and the server. When a client needs to serve a query, it has to know where to locate the components of the data item. It may have to access one component from the local cache and download the other one from the broadcast channel.

3.4 The Algorithm

The formal description of the server and the client algorithm is shown in Fig. 2 and Fig. 3, respectively. As shown in Fig. 2, the server is responsible for constructing and broadcasting the IR and UIR at predefined time interval. As shown in Fig. 3, the client validates its cache based on the received IR or UIR. If the client missed the previous IR, it has to wait for the next IR. Each data d_x has two components, d_x^h and d_x^c. If the client has a valid cached copy of the d_x^c, it still needs to get the d_x^h from the UIR or IR in order to serve the query.

4 Performance Evaluation

The performance evaluation includes three parts. In part 1 (Section 4.2), we demonstrate the effectiveness of the proposed approach on improving the cache hit ratio. Part 2 (Section 4.3) demonstrates the effectiveness of the proposed approach on reducing the power consumption. The effectiveness of dealing with frequently updated data is shown in Part 3 (Section 4.4).

4.1 The Simulation Model and System Parameters

In order to evaluate the efficiency of various invalidation algorithms, we develop a model which is similar to that employed in [4], [12], [15]. It consists of a single server that serves multiple clients. The database can only be updated by the server, whereas the queries are made on the client side. As shown in Fig. 4, the server is responsible for sending data and control information on the broadcasting channel. Based on the information, clients download data to serve queries and download control information to help maintain cache consistency. If there is a cache hit, the client can serve the query locally; otherwise, it sends an uplink request to the server requesting the data. On receiving the request, the server puts the data on the broadcast channel.

From the server point of view, the database is divided into two subsets: the *hot* data subset and the *cold* data subset. The hot data subset includes the first 100 items (out of 2,000 items) and the cold data subset includes the remaining data items of the database. From the client point of view, the database is divided into three subsets: the *hot* data subset, the *medium* data subset, and the *cold* data subset. The hot data subset includes 20 randomly (based on the client id) chosen items from the first 100 items. The medium data subset includes the remaining 80 items in the first 100 items of the database. The cold data subset includes the remaining data items of the database. A client has a

Notations:

- $N^x_{acc}, N^x_{c\_acc}, \alpha, \beta, \delta$: defined before.
- $Q_i = \{d_x \mid d_x$ has been queried before $T_i\}$.
- $Q_{i,k} = \{d_x \mid d_x$ has been queried in the interval $[T_{i,k-1}, T_{i,k}]\}$.
- t^c_x: the timestamp of the cached data item d_x.
- T_l: the timestamp of the last received IR.
- N^x_{pre}: the number of prefetch times for a cache entry d_x.
- $N^x_{c\_pre}$: the number of prefetch times for a cache entry d_x in the current evaluation cycle.
- L_{data}: an id list of the data items that a client requested from the server. Initialized to be empty.

(A) When a client C_j receives IR_i, $\mathcal{L}$, and L_{bcast}:

if $T_l < (T_i - L * w)$
then drop the entire cache or go uplink to verify the cache (or use other techniques to deal with long disconnection);
for each data item $< d_x, t_x >$ in the cache **do**
  if $((d_x, t_x) \in IR_i) \wedge (t^c_x < t_x)$
  then invalidate d_x;
for each $d_x \in L_{bcast}$ **do**
  if $(d_x \in L_{data})$
  then download d_x into local cache; Use d_x to answer the previous query; $N^x_{c\_acc}$ ++;
  if d_x is an invalid cache item and the cache item is marked as $prefetch$
  then download d_x into local cache; $N^x_{c\_pre}$ ++;
$T_l = T_i$; **if** $(Q_i \neq \emptyset)$ **then** query (Q_i);
When the evaluation cycle reaches, mark_prefetch().

(B) When a client receives a $UIR_{i,k}$ and $\mathcal{L}_{hot}$:

if missed IR_i **then** break; /* wait for the next IR */
for each data item $< d_x, t_x >$ in the cache **do**
  if $(d_x \in UIR_{i,k})$
  then invalidate d_x;
if $(Q_{i,k} \neq \emptyset)$ **then** query $(Q_{i,k})$.

Procedure query(Q)

$L_{data} = \emptyset$;
for each $d_x \in Q$ **do**
  if d_x is a valid entry in the cache
  then if d^h_x is not in the current $\mathcal{L}_{hot}$ or $\mathcal{L}$
    then delay the query processing until the next IR;
    else use the cache's value and d^h_x to answer the query; $N^x_{c\_acc}$ ++;
  else $L_{data} = L_{data} \cup \{d_x\}$;
send $request(L_{data})$ to the server.

Procedure mark_prefetch()

for each cache entry d_x **do**
  $N^x_{acc} = (1-\alpha) * N^x_{acc} + \alpha * N^x_{c\_acc}$; $N^x_{pre} = (1-\alpha) * N^x_{pre} + \alpha * N^x_{c\_pre}$;
  if $\frac{N^x_{pre}}{N^x_{acc}} < \beta$
  then mark it as $prefetch$;
  else mark it as $non-prefetch$;
Based on the N^x_{access}, mark δ most frequently accessed cache entries as $prefetch$.

Fig. 3. The algorithm at the client.

probability of 75 percent to access the data in its hot set, a probability of 10 percent to access the data in the medium set, and a probability of 15 percent to access the data in the cold set. The traffic pattern is based on the following observations: Suppose some users are interested in getting financial information of some companies. Most people in the software sector are interested in well-known software companies, such as Microsoft and Oracle, but not small software companies, such as CT Holdings. Also, people in the software sector may not have too much interest in companies such as Ford or GM, although they are hot items in the automobile sector. As a result, for clients who are interested in the software sector, Microsoft and Oracle are hot data, Ford and GM are medium data (which are hot data for people interested in the automobile sector), and other small companies are cold data.

4.1.1 The Server

The server broadcasts IRs (and UIRs in our algorithm) periodically to the clients. The server assigns the highest

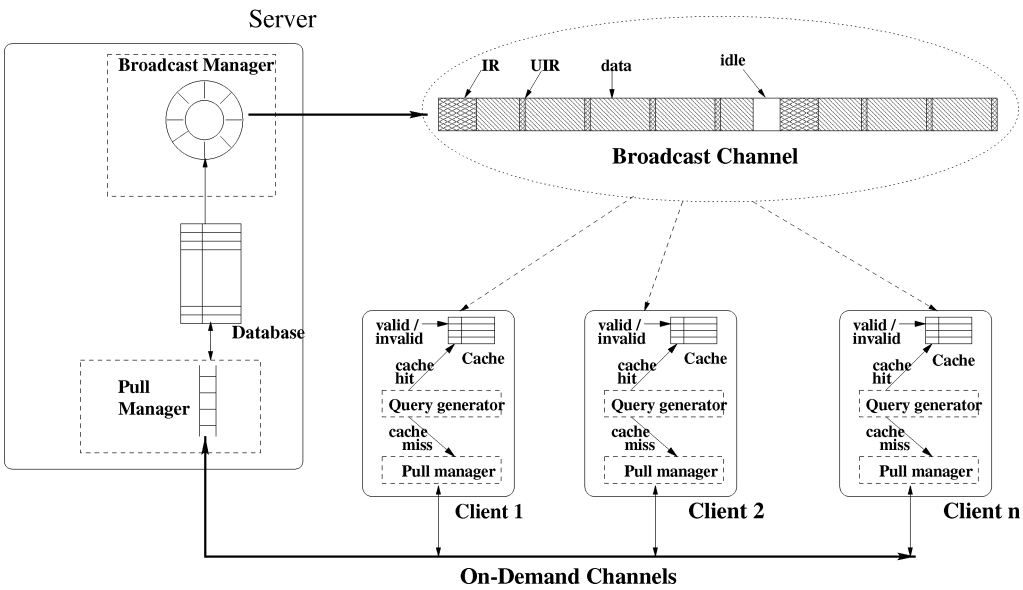

Fig. 4. The simulation model.

priority to the IR/UIR broadcasts and equal priorities to the rest of the messages. This strategy ensures that the IRs (or UIRs) can always be broadcast over the wireless channels with the broadcast interval specified by the parameter L (or $\frac{L}{m}$). All other messages are served on a FCFS (first-come-first-serve) basis. It is possible that an IR or UIR time interval reaches while the server is still in the middle of broadcasting a packet. We use a scheme similar to the beacon broadcast in IEEE 802.11 [11], where the server defers the IR or UIR broadcast until it finishes the current packet transmission. However, the next IR or UIR should be broadcast at its originally scheduled time interval.

The server generates a single stream of updates separated by an exponentially distributed update inter-arrival time. All updates are randomly distributed inside the hot data subset and the cold data subset, whereas 33.3 percent of the updates are applied to the hot data subset. In the experiment, we assume that the server processing time (not data transmission time) is negligible and the broadcast bandwidth is fully utilized for broadcasting IRs (and UIRs) and serving client's data requests.

4.1.2 The Client

Each client generates a single stream of read-only queries. Each new query is generated following an exponentially distributed time. The client processes generated queries one by one. If the referenced data are not cached on the client side, the data ids are sent to the server for fetching the data items. Once the requested data items arrive on the channel, the client brings them into its cache. To simplify the presentation and simulation, we assume no data item is marked as *nonprefetch* (e.g., δ is equal to the cache size) except in Section 4.3.1, where the effects of power consumption are evaluated. We do not model disconnections except Section 4.2.3. The default system parameters are listed in Table 1.

4.2 Simulation Results: Improving the Cache Hit Ratio

4.2.1 The Cache Hit Ratio

Fig. 5a shows the cache hit ratio as a function of the number of clients. As can be seen, the cache hit ratio of our algorithm increases as the number of clients increases, but the cache hit ratio of the TS algorithm does not change with the number of clients; e.g., TS (n = 1) and TS (n = 100) have the same cache hit ratio. When the number of clients in our algorithm drops to 1, the cache hit ratio of our algorithm is similar to the TS algorithm. In the TS algorithm, a client only downloads the data that it has requested from the server. However, in our algorithm, clients also download the data which may be accessed in the near future. Considering 100 clients, due to server update, one hot data item may be changed by the server and the clients may have to send requests to the server and download the data from the server. In the TS algorithm, it may generate 100 cache misses if all clients need to access the updated data. In our algorithm, after a client sends a request to the server, other clients can download the data. In other words, after one cache miss, other clients may be able to access the data from their local cache. Certainly, this ideal situation may not always occur, especially when the cache size is small or the accessed data item is a cold item. However, as long as some downloaded data items can be accessed in the future, the cache hit ratio of our algorithm will be increased. Due to cache locality, a client has a large chance to access the invalidated cache items in the near future, so downloading these data items in advance should be able to increase the cache hit ratio. As the number of clients decreases, clients have less opportunity to download data requested by others and, hence, the cache hit ratio decreases. This explains why our algorithm has similar cache hit ratio when the number of clients drops to 1. Fig. 5b shows the cache hit ratio under different cache sizes when the number of clients is 100. Based on the above explanation, it is easy to see that the cache hit ratio of our algorithm is always higher than that of

TABLE 1
Simulation Parameters

Number of clients (n)	100
Broadcast interval (L)	20 seconds
Broadcast bandwidth	100000 bits/s
Cache size (c)	20 to 500 items
Mean query generate time (T_q)	30s to 200s
Broadcast window (w)	10 intervals
UIR replicate times ($m-1$)	4 (5 - 1)
Hot data items	20 out of the first 100 items
Medium data items	the remainder of the first 100 items
Cold data items	remainder of DB
Hot data access prob. (p_h)	0.75
Medium data access prob.	0.10
Mean update arrival time ($T_u/T_c/T_h$)	0.001s to 1000s
Hot data update prob.	0.33
Hot component size	64 bits
Data size (S)	10000 bytes
Database size	2000 items

the TS algorithm for one particular cache size (cache size is 20 items, 100 items, or 500 items).

From Fig. 5b, we can see that the cache hit ratio grows as the cache size increases. However, the growing trend is different between the TS algorithm and our algorithm. For example, in the TS algorithm, when the update arrival time is $1s$, the cache hit ratio does not have any difference when the cache size changes from 20 data items to 500 data items. However, in our algorithm, under the same situation, the cache hit ratio increases from about 33 percent to 61 percent. In our algorithm, clients may need to download interested data for future use, so a large cache size may increase cache hit ratio. However, in the TS algorithm, clients do not download data items that are not addressed to them. When the server updates data frequently, increasing the cache size does not help. This explains why different cache size does not affect the cache hit ratio of the TS algorithm when $T_u = 1s$.

As shown in Fig. 5b, the cache hit ratio drops as the update arrival time decreases. However, the cache hit ratio of the TS algorithm drops much faster than our algorithm. When the update arrival time is $10,000s$, both algorithms have similar cache hit ratio for one particular cache size. With $c = 500\ items$, as the update arrival time reaches $1s$, the cache hit ratio of our algorithm still stays around 61 percent, whereas the cache hit ratio of the TS algorithm drops below 7 percent. This can be explained as follows: When the update arrival time is very low (e.g., $1s$), most of the cache misses are due to hot data access; when the update arrival time is very high (e.g., $10,000s$), most of the cache misses are due to cold data access. Since our algorithm is very effective at improving cache performance when accessing hot data, the cache hit ratio of our algorithm can be significantly improved when the update arrival time is low. However, as the mean update arrival time drops further ($T_u < 1s$), the cache hit ratio of our algorithm drops

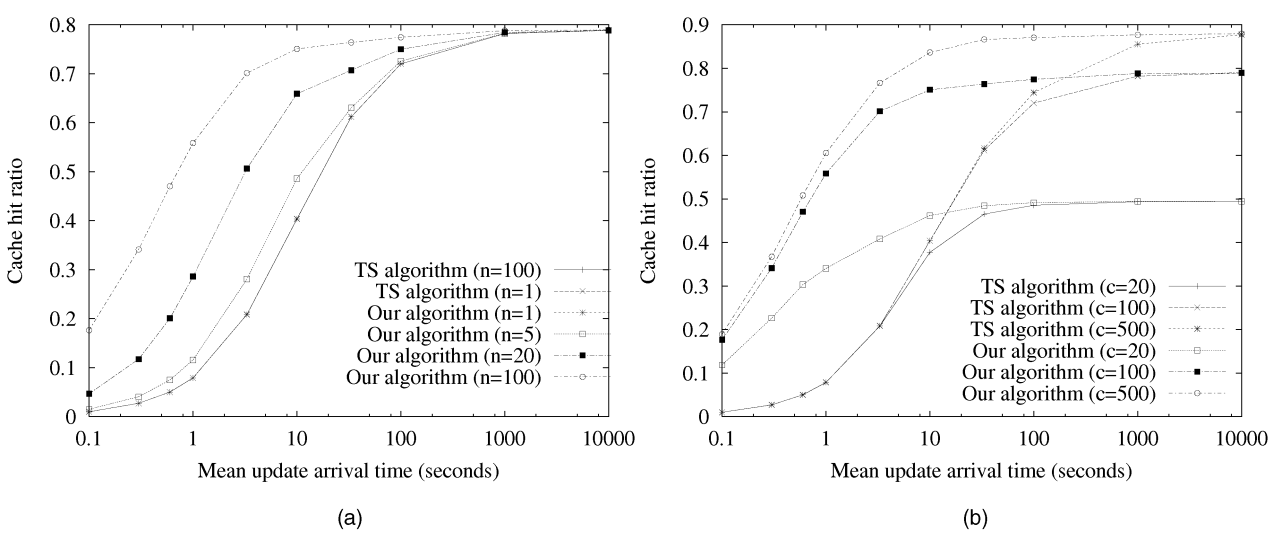

Fig. 5. A comparison of the cache hit ratio ($T_q = 100s$). (a) Shows the cache hit ratio as a function of the number of clients when the cache size is 100 items. (b) Shows the cache hit ratio under different cache sizes when the number of clients is 100.

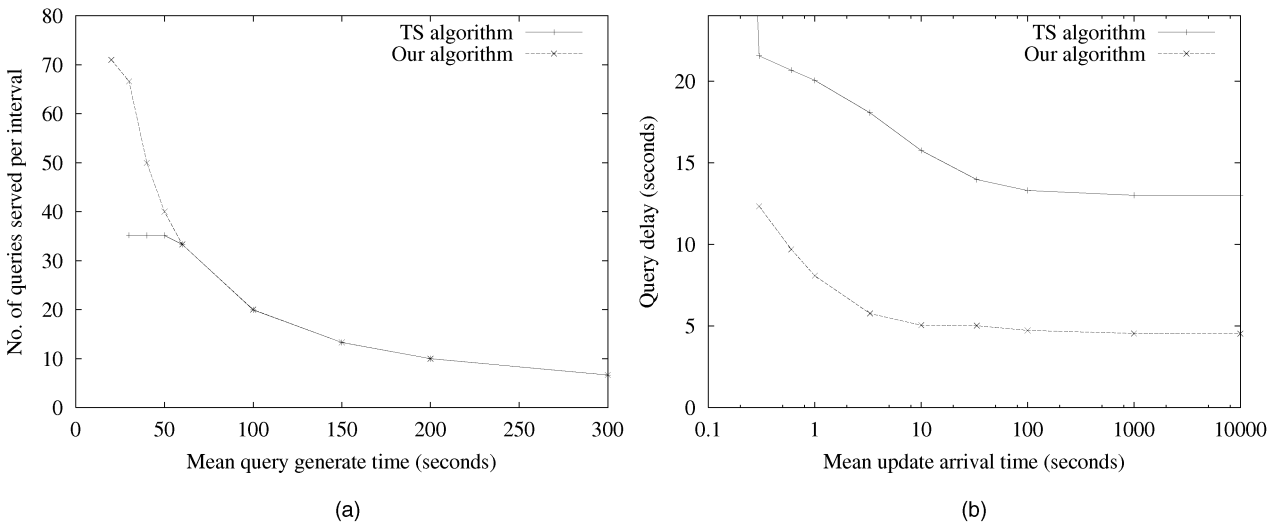

Fig. 6. A comparison of the throughput and query delay. (a) Shows number of queries served per IR interval ($T_u = 3s$, $c = 100\ items$). (b) Shows the query delay as a function of the mean update arrival time ($T_q = 100s$, $c = 100\ items$).

much faster than before. At this time, the hot data changes so fast that the downloaded hot data may be updated before the client can use it, hence failing to improve the cache hit ratio. Note that, when the update arrival time is very high, the cache performance depends on the LRU policy and it is very difficult to further improve the cache hit ratio except by increasing the cache size.

4.2.2 The Throughput and the Query Delay

Since the broadcast bandwidth is fixed, the server can only transmit a limited amount of data during one IR interval and it can only serve a maximum number (θ) of queries during one IR interval. However, the throughput (the number of queries served per IR interval) may be larger than θ since some of the queries can be served by accessing the local cache. Since our algorithm has a higher cache hit ratio than the TS algorithm, our algorithm can serve more queries locally and the clients send fewer requests to the server. As shown in Fig. 6a, when the query generate time reduces to $30s$, the number of requests in the TS algorithm is larger than θ and some queries cannot be served. As a result, the throughput of the TS algorithm remains at 35, whereas the throughput of our algorithm reaches 70. In the TS algorithm, since the broadcast channel has already been fully utilized when $T_q = 60s$, further reducing the query generate time does not increase the throughput. When the query generate time is low, the broadcast channel has enough bandwidth to serve client requests and, hence, both algorithms can serve the same number of queries (although they have difference query latency).

In the case of a cache miss, the TS algorithm and our algorithm have the same query delay. However, in the case of a cache hit, our algorithm can reduce the query delay by a factor of m. As shown in Fig. 6b, as the mean update arrival time increases, the cache hit ratio increases and the query delay decreases. Since our algorithm has a higher cache hit ratio than the TS algorithm, the query delay of our algorithm is shorter than the TS algorithm. For example, with $T_u = 10,000s$, our algorithm reduces the query delay by a factor of 3 compared to the TS algorithm.

Fig. 7 shows how the cache locality affects the query delay. When $p_h = 0$, 90 percent of the data accesses are uniformly distributed among the 4,900 data items. Since the cache size is too small, the cache hit ratio is almost zero and then the TS algorithm and our algorithm have similar query delay. Since data updates can reduce the cache hit ratio, both algorithms have lower query delay when $T_u = 1,000s$ than that when $T_u = 10s$. As the hot data access probability increases, the query delay of both algorithm drops, but the query delay of our algorithm drops much faster. When $p_h = 0.9$, clients only access the first 100 items and the cache hit ratio is close to 100 percent in our algorithm. As a result, the query delay of our algorithm drops to almost 2s, whereas the query delay of the TS algorithm is still larger than 10s.

4.2.3 The Effects of Disconnections

In order to evaluate the effects of disconnections, we change the simulation model as follows: Each client generates a

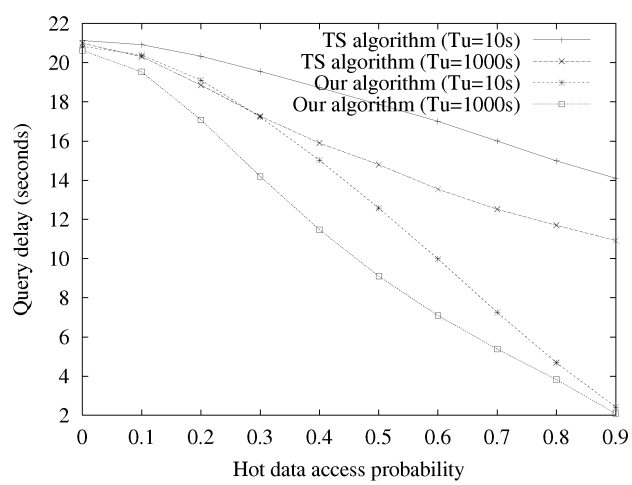

Fig. 7. The effects of hot data access probability (p_h) on the query delay ($c = 100\ items$, $T_q = 100s$).

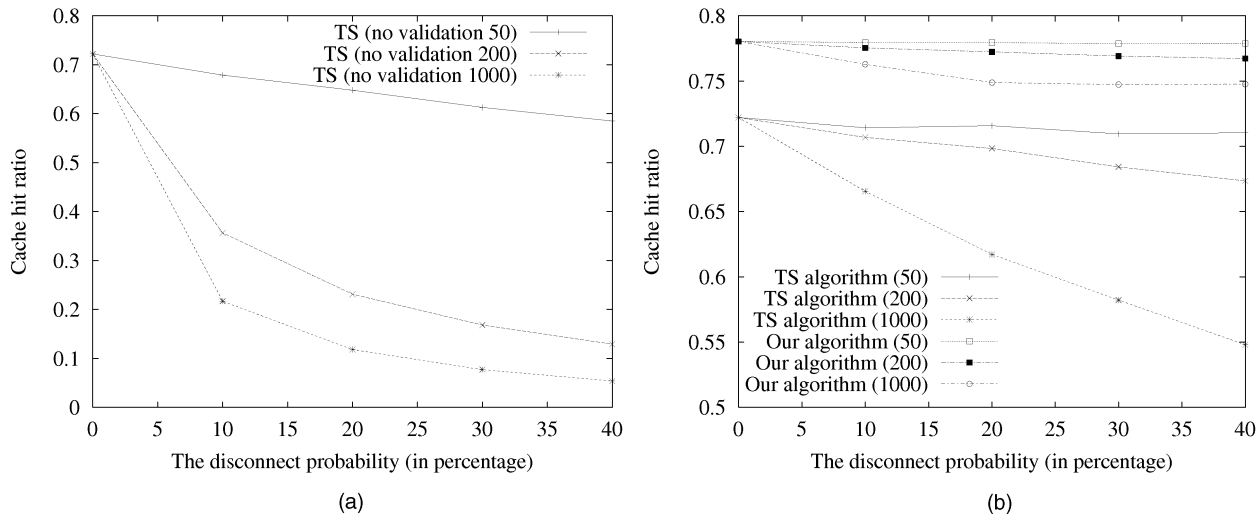

Fig. 8. The effects of disconnection on cache hit ratio ($T_q = 100s$, $T_u = 100s$, $c = 100\ items$).

single stream of read-only queries. Each new query is separated from the completion of the previous query by either an exponentially distributed query generate time or an exponentially distributed disconnection time (with mean of T_{dis}). A client can only enter the disconnection mode when the outstanding query has been served.

Fig. 8 shows the effects of disconnections on cache hit ratio. The x-axis represents the disconnect probability. We measure three mean disconnection times $50s$, $200s$, and $1,000s$. Fig. 8a shows the cache hit ratio of the original TS algorithms [3]. When the mean disconnection time is $50s$, which is less then $w * L$ seconds, clients can still validate their cache by using the IR and the cache hit ratio does not reduce too much as the disconnection time (or probability) increases. However, if a client disconnects longer than $w * L$ seconds (e.g., $200\,s$, $1,000\,s$), it has to discard the whole cache. As shown in the figure, the cache hit ratio drops dramatically. Previous techniques [3], [12], [15], [24] can be used to deal with the long disconnection problem. To simplify the presentation and simulation, in the following comparisons, we assume that the client, when reconnected, will send a request to the server to verify the validity of the local cache. Fig. 8b compares the cache hit ratio of the TS algorithm and our algorithm under this modification. Since the client keeps the valid cache items even though it has been disconnected for a long time, the cache hit ratio of both algorithms drops slowly compared to Fig. 8a. As the mean disconnection time increases to $1,000s$, the cache hit ratio of the TS algorithm still drops very fast, even with the modification. On the contrary, our algorithm performs a little bit better. This can be explained by the fact that our algorithm allows prefetch, by which the clients can update their invalid cache entries.

4.3 Simulation Results: Reducing the Power Consumption

In the IR/UIR-based cache invalidation approaches, clients can enter doze mode most of time and only wake up during IR and UIR broadcasting time. Since the IR size and the UIR size are very small compared to the data size, the number of prefetches represents a large part of the energy consumption. Thus, we use the number of prefetches per IR interval as a performance metric to evaluate the energy consumption.

To simplify the simulation, we choose β to be 2. As explained in Section 3.2, β should be larger than 1. After a cache miss, the client sends an uplink request to ask for the data and downloads the data to its local cache. Although the uplink request packet size is much smaller than the downloaded data size, sending a uplink request may also consume a large amount of energy. This is due to the connection setup delay and the distance between the base station and the client. To simplify the presentation and simulation, we suppose the power consumption of downloading a data item is similar to that of sending an uplink request and $\beta = 2$ can be used to optimize the power consumption. During the simulation, we found that α and the refreshment cycle time does not affect the performance too much.[3] Thus, we fix these two parameters, i.e., $\alpha = 0.2$, the refreshment cycle is 10 times the mean query generate time. The *no-nonprefetch* algorithm and *no-prefetch* algorithm are modifications of our algorithm. The *no-nonprefetch* algorithm (i.e., δ is infinitely large) does not mark any data item as *nonprefetch*. The *no-prefetch* algorithm (i.e., $\delta = 0$ and $\beta = 0$) does not perform any prefetch. Also, we do not consider the frequent data update problem (i.e., $\mathcal{L} = \emptyset$).

4.3.1 The Effects of δ (The Number of Cache Entries to Prefetch)

As shown in Fig. 9a, the number of prefetches increases as the mean update arrival time decreases for the no-nonprefetch approach and our algorithm (with $\delta \neq 0$). When the mean update arrival time decreases, data are updated more frequently and more clients have cache misses. As a result, the server broadcasts more data during each IR and the clients prefetch more data to their local

3. Our simulation is always in a steady state. In order to see the effects of α, the client access pattern should be changed. That is, the hot data subset needs to be changed with time. Since this is not the major concern of the paper, we did not model it.

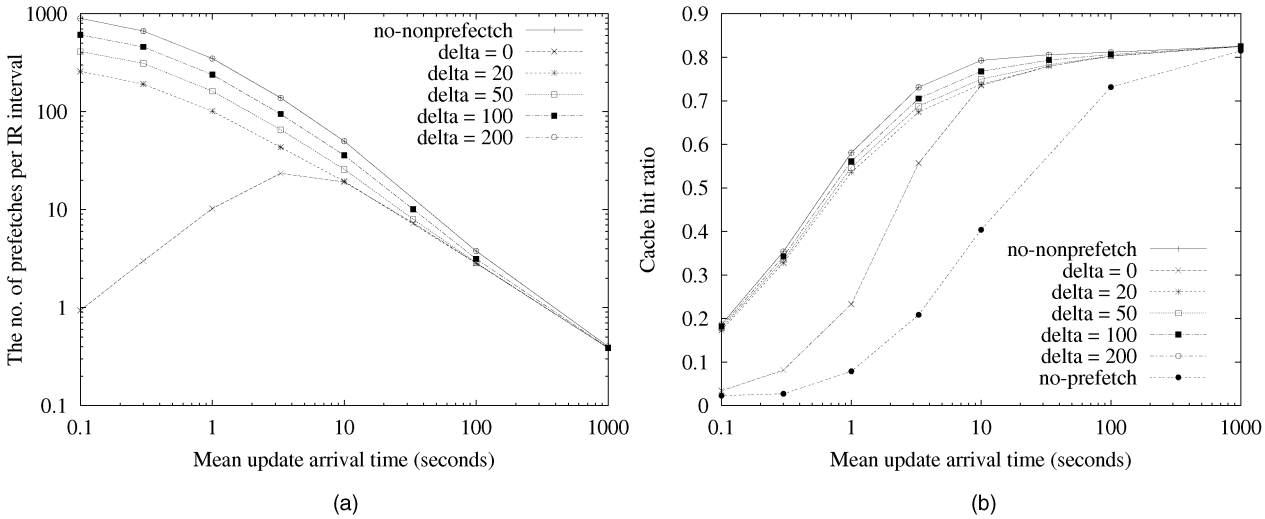

Fig. 9. The effects of δ ($T_q = 100s$, $c = 200\ items$, $n = 100$).

cache. However, with $\delta = 0$, the number of prefetches starts to drop when the mean update arrival time is lower than $3s$. This can be explained by the fact that the number of prefetches in our algorithm with $\delta = 0$ is determined by PAR. As the mean update arrival time drops below $3s$, most cache entries have $PAR < \beta$ and they are marked as *nonprefetch*. Since δ represents the number of cache entries that are marked as *prefetch*, the number of prefetches increases as δ increases. When δ equals the cache size, the number of prefetches in our algorithm ($\delta = 200$) and the no-nonprefetch approach becomes equal.

Prefetching data into the local cache can improve the cache hit ratio. When some data items are marked as nonprefetch, the cache hit ratio may be reduced. As shown in the Fig. 9b, the cache hit ratio drops as δ decreases. Note that, when $\delta = 200$, our algorithm has the same cache hit ratio as the no-nonprefetch approach. When the mean update arrival time is high, there are not too many data updates and most of the queried data can be served locally. When the mean update arrival time decreases, data is updated more frequently. With prefetching, the cache hit ratio drops more slowly. For example, the prefetching approaches have a similar cache hit ratio when the mean update arrival time drops from $1,000s$ to $10s$. On the other hand, without prefetching, the cache hit ratio may be significantly dropped. For example, the no-prefetch approach has a very low cache hit ratio when $T_u = 10s$, but relatively high cache hit ratio when $T_{update} = 1,000s$. When the mean update arrival time drops below $0.3s$, the cache hit ratio of the our approach with $\delta = 0$ becomes similar to that of the no-prefetch approach.

From Fig. 9, we can see that the number of prefetches is related to the cache hit ratio. For example, the no-nonprefetch approach has the highest number of prefetches and it has the highest cache hit ratio. The no-prefetch does not have prefetch overhead, but it has the lowest cache hit ratio. When $\delta = 0$, the number of prefetches in our approach significantly drops at $T_u = 3s$ and its cache hit ratio also dramatically drops. Although there is a trade-off between the cache hit ratio and the number of prefetches, our approach outperforms the no-nonprefetch approach in general. For example, when $T_u = 0.1s$, our approach with $\delta = 20$ has a similar cache hit ratio to the no-nonprefetch approach, but it reduces the number of prefetches by a factor of 4. It's interesting to see that our approach with $\delta = 0$ has negligible prefetch overhead when the mean update arrival time is very low or very high and it has high cache hit ratio when the mean update arrival time is high.

4.3.2 The Effects of Cache Size on a Simple Solution

The number of prefetches can be reduced by reducing δ. As an alternative, the number of prefetches can also be reduced by simply reducing the cache size. Fig. 10 shows the effects of cache size on the cache hit ratio and the number of prefetches. Similarly to Fig. 9, there is a trade-off between the cache hit ratio and the number of prefetches. Although reducing the cache size can always reduce the prefetch overhead, we do not recommend this solution since it may dramatically reduce the cache hit ratio. For example, with $T_u = 1,000s$, as the cache size reduces from 200 items to 20 items, the cache hit ratio is almost reduced by half. Moreover, when $T_u = 1,000s$, the number of prefetches is very low (below 1) and the power consumption is mainly determined by other factors, such as serving query from local cache. Thus, further reducing the number of prefetches cannot save too much energy and the cache hit ratio may be dramatically reduced.

4.4 Simulation Results: Handling Frequently Updated Data

In this section, we compare the performance of the following algorithms:

- *Indexing:* In the indexing approach [13], data and indexes are periodically broadcast on the air. Even though adding index can save power, it also increases the query latency. A simple broadcasting, which does not broadcast the indexes, provides the lower bound on query delay. The lower bound is used to compare with the proposed approach.

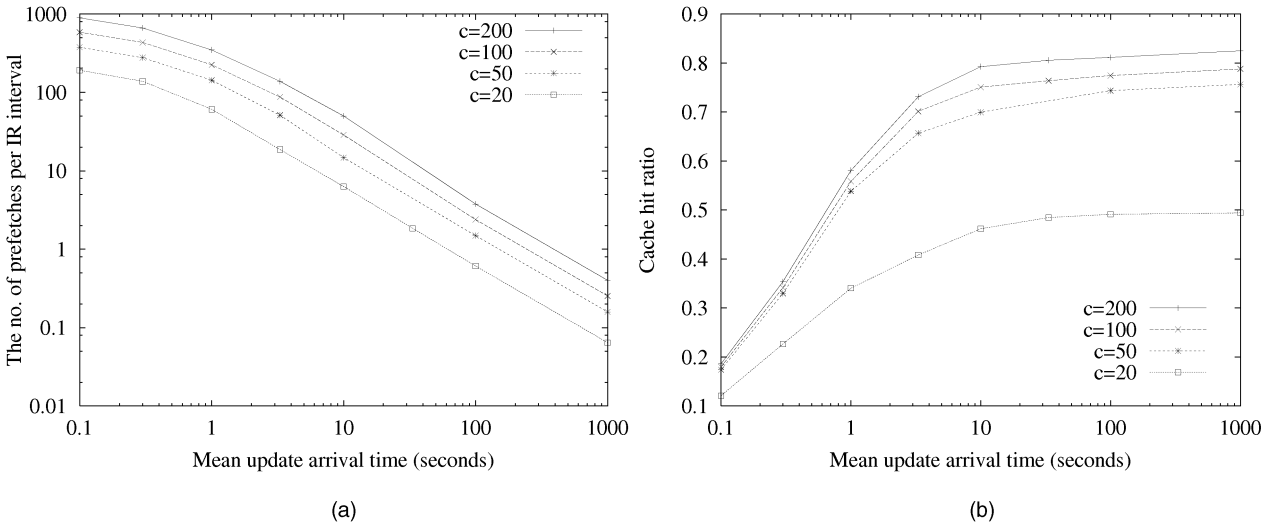

Fig. 10. The effects of cache size ($T_q = 100s$, $n = 100$). (a) Shows the number of prefetches as a function of the mean update arrival time. (b) Shows the cache hit ratio as a function of the mean update arrival time.

- *Ideal UIR:* This is the algorithm presented in Section 3.4. To simplify the comparisons, we do not consider energy consumption issues (i.e., β is equal to the cache size). Since the server may not be able to know which data are the hot data, it is difficult to find $\mathcal{L}_{hot}$. Thus, we name it *ideal* UIR. Note that techniques proposed in [22] can be used to identify the hot data items and to approximate the ideal UIR algorithm.
- *Hybrid UIR:* $\mathcal{L}_{hot} = \emptyset$ is the only difference between the hybrid UIR approach and the ideal UIR approach.
- *Pure UIR:* In this approach, data items are not divided into hot and cold components.

Fig. 11a shows the query delay as a function of the data size for the indexing approach under different update arrival time. As can be seen, the query delay is proportional to the data size. When the data size is small, the delay is small. When the data size increases to 10,000 bytes, the delay is too high to be tolerable. Thus, the indexing approach is not suitable for applications which have very large data sizes and have strict delay requirements. As can be seen, the query delay of the indexing approach is not affected by the mean update arrival time.

Fig. 11b shows the query delay as a function of the mean update arrival time for the pure UIR approach under different data item sizes ($S = 1,000$ bytes, $S = 3,000$ bytes, and $S = 1,000$ bytes). When the mean update arrival time decreases, the data is updated more frequently and more clients have cache misses. As a result, the query delay increases. As the mean update arrival time becomes very small, many clients have cache misses and their requests form a queue at the server. Since it takes a longer time to send a large data item than a small data item, clients may need to wait for a longer time if the data size is large. If the server receives more than the maximal served queries (θ)

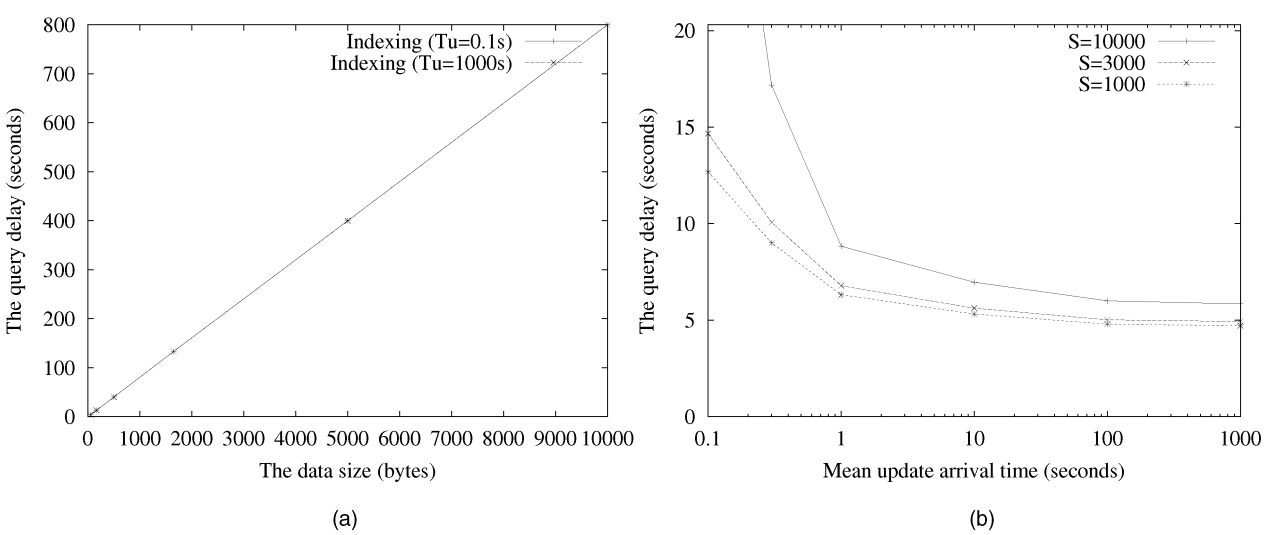

Fig. 11. A comparison of the query delay. (a) Shows the query delay as a function of the data size. (b) Shows the query delay as a function of the mean update arrival time for the pure UIR approach ($T_q = 40s, c = 100\ items$, $n = 100$).

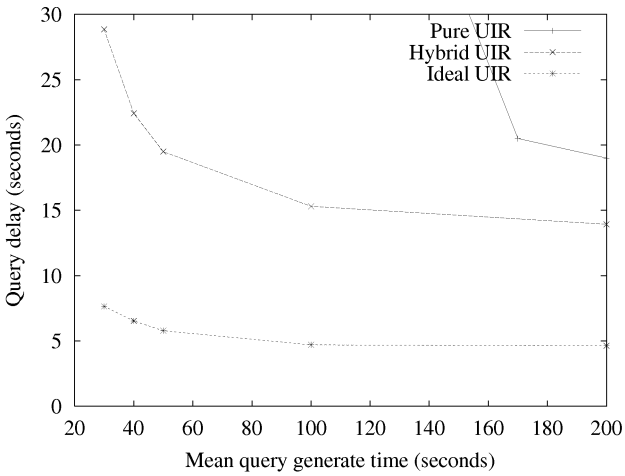

Fig. 12. A comparison of the query delay ($T_c = 100s, T_h = 0.01s$, $c = 100\ items$, $n = 100$).

during each IR interval, many queries may not be served and the query delay may be out of bound. Due to the data size difference, the value of θ varies. For example, with $T_u = 0.1s$, the query delay of $S = 1,000$ is still less than $15s$, but the query delay of $S = 10,000$ becomes infinitely high.

Fig. 12 compares the query delay of the pure UIR approach, the hybrid UIR approach, and the ideal UIR approach. As can be seen, the ideal UIR approach outperforms the hybrid approach, which outperforms the pure UIR approach. In the ideal approach, most of the queries can be served after a UIR is broadcast and the query delay is very low. In the hybrid approach, most of the queries can only be served after the IR is broadcast and the query delay becomes longer compared to the ideal UIR. The query delay of the pure UIR approach becomes infinitely high when T_q drops below $100s$. This is due to the fact that the cache hit ratio of the pure UIR approach is much smaller than the other two. Since the pure UIR approach does not differentiate between the hot component and the cold component, the mean update arrival time equals the hot component update arrival time, $T_h = 0.01s$, and its cache hit ratio is near 0. In the hybrid UIR approach and the ideal UIR approach, the mean update arrival time is $T_c = 100s$ and their cache hit ratio is pretty high. Moreover, in the pure UIR approach, broadcasting the IR and UIR occupies a large amount of bandwidth since many data items are updated and their *id*s must be added to the IR and UIR.

5 Future Works

Although prefetching has been widely used to reduce the response time in the Web environment, most of these techniques [14] concentrate on estimating the probability of each file being accessed in the near future. They are designed for the point-to-point communication environment, which is different from our broadcasting environment. As we know, the presented simple and effective prefetching technique is the first applied to IR-based cache invalidation strategies. To further improve the performance, techniques based on user profile [9], which indicates the general information types that a user is interested in receiving, will be studied in the future.

The proposed PAR approach did not consider the effects of varying data size and the data transmission rate. However, the ideal data item to be prefetched should have a high access probability, low update rate, small size, and a long retrieval delay. Along this line, we can design prefetch functions which incorporate these factors. Since many of these factors are usually not constant, techniques such as those used in [21] can be applied to estimate these parameters.

By reducing the number of items marked for prefetch, we can reduce the number of prefetches. Let a_k be the percentage of the energy left. It is desired to reduce the amount of prefetches to some percentage ($f(a_k)$) of the original when a_k drops to a threshold. A simple discrete function can be as follows:

$$f(a_k) = \begin{cases} 100\% & 0.5 < a_k \leq 1.0 \\ 70\% & 0.3 < a_k \leq 0.5 \\ 50\% & 0.2 < a_k \leq 0.3 \\ 30\% & 0.1 < a_k \leq 0.2 \\ 10\% & a_k \leq 0.1. \end{cases} \quad (1)$$

At regular intervals, the client reevaluates the energy level a_k. If a_k drops to a threshold value, $N_p = N_p \cdot f(a_k)$. The client only marks the first N_p items, which have the maximum prefetch value, as prefetch. In this way, the number of prefetches can be reduced to prolong the system running time.

The proposed data component concept can be extended to a mixed data access mechanism for mobile environments. In this mechanism, caching, broadcasting, and pull-based delivery are used together to minimize the access time and energy consumption. Logically speaking, data are stored in a hierarchy of media where the most frequently accessed data are cached in the client, the commonly requested (or frequently updated) data subset is temporarily stored on the broadcast channels, and the rest of the data must be pulled from the server via explicit client requests. IRs or UIRs are broadcast to help clients validate their cache. Data caching and push-based data access alleviate pull-based request considerably since most frequently accessed data are retrieved either from the client's cache or from the broadcast channel. On the other hand, requests for infrequently accessed data can always be served on the point-to-point channels. When a user issues a query, the client first searches its cache. If there is a valid copy in the cache, an answer replies immediately. Otherwise, the client attempts to obtain the data item from the server site. The hot spot of the uncached data can be obtained and broadcast by asking the clients to monitor the cache misses and piggyback the information to the server on some subsequent pull requests.

6 Conclusions

IR-based cache invalidation techniques have received considerable attention due to their scalability. However, they have some drawbacks such as long query delay, low bandwidth utilization, and unsuitability for applications where data change frequently. In this paper, we proposed

solutions to deal with these issues. By deferring the service requests, the server supports prefetches, which are used to improve the cache hit ratio. Based on a novel *prefetch-access ratio* concept, the proposed scheme can dynamically optimize performance or power based on the available resources and the performance requirements. Based on a data component concept, different techniques (indexing and caching) are applied to deal with different components of the data according to their update frequency. Simulation results showed that the proposed algorithm can cut the query delay by a factor of 3 and double the throughput compared to the TS algorithm. Simulation results also demonstrated the effectiveness of the prefetch-access ratio concept and the data component concept.

ACKNOWLEDGMENTS

The author would like to thank the editors and the anonymous referees whose insightful comments helped him to improve the presentation of the paper. This work was supported in part by US National Science Foundation CAREER Award CCR-0092770.

REFERENCES

[1] S. Acharya, M. Franklin, and S. Zdonik, "Disseminating Updates on Broadcast Disks," *Proc. 22nd VLDB Conf.*, Sept. 1996.
[2] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik, "Broadcast Disks: Data Management for Asymmetric Communication Environments," *Proc. ACM SIGMOD*, pp. 199-210, May 1995.
[3] D. Barbara and T. Imielinski, "Sleepers and Workaholics: Caching Strategies for Mobile Environments," *Proc. ACM SIGMOD*, pp. 1-12, 1994.
[4] G. Cao, "A Scalable Low-Latency Cache Invalidation Strategy for Mobile Environments," *Proc. ACM Int'l Conf. Mobile Computing and Networking (MobiCom)*, pp. 200-209, Aug. 2000.
[5] G. Cao, "On Improving the Performance of Cache Invalidation in Mobile Environments," *ACM/Baltzer Mobile Networks and Application (MONET)*, to appear.
[6] G. Cao, "Updated Invalidation Report: A Scalable Low-Latency Cache Invalidation Strategy for Mobile Environments," *IEEE Trans. Knowledge and Data Eng.*, to appear (a preliminary version appeared in *Proc. ACM MobiCom '00*).
[7] G. Cao and C. Das, "On the Effectiveness of a Counter-Based Cache Invalidation Scheme and Its Resiliency to Failures in Mobile Environments," *Proc. 20th IEEE Symp. Reliable Distributed Systems (SRDS)*, pp. 247-256, Oct. 2001.
[8] G. Cao, Y. Wu, and B. Li, "A Mixed Data Dissemination Strategy for Mobile Computing Systems," *Lecture Notes in Computer Science, Advances in Web-Age Information Management*, pp. 408-416, Springer-Verlag, 2001.
[9] M. Cherniack, M. Franklin, and S. Zdonik, "Expressing User Profiles for Data Recharging," *IEEE Personal Comm.*, pp. 6-12, Aug. 2001.
[10] A. Datta, D. Vandermeer, A. Celik, and V. Kumar, "Broadcast Protocols to Support Efficient Retrieval from Databases by Mobile Users," *ACM Trans. Database Systems*, vol. 24, no. 1, pp. 1-79, Mar. 1999.
[11] The editors of IEEE 802.11, "Wireless LAN Media Access Control (MAC) and Physical Layer (PHY) Specifications," 802.11 Wireless Standards (http://grouper.ieee.org/groups/802/11), 1999.
[12] Q. Hu and D. Lee, "Cache Algorithms Based on Adaptive Invalidation Reports for Mobile Environments," *Cluster Computing*, pp. 39-48, Feb. 1998.
[13] T. Imielinski, S. Viswanathan, and B. Badrinath, "Data on Air: Organization and Access," *IEEE Trans. Knowledge and Data Eng.*, vol. 9, no. 3, pp. 353-372, May/June 1997.
[14] Z. Jiang and L. Kleinrock, "An Adaptive Network Prefetch Scheme," *IEEE J. Selected Areas in Comm.*, vol. 16, no. 3, pp. 1-11, Apr. 1998.
[15] 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. 115-127, 1997.
[16] H. Korth, "The Double Life of the Transaction Abstraction: Fundamental Principle and Evolving System Concept," *Proc. Very Large Data Base Conf. (VLDB)*, Sept. 1995.
[17] E. Levy and A. Silbershatz, "Distributed File Systems: Concepts and Examples," *ACM Computing Surveys*, vol. 22, no. 4, Dec. 1990.
[18] B. Nitzberg and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *Computer*, vol. 24, no. 8, Aug. 1991.
[19] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G.J. Popek, "Resolving File Conflicts in the Ficus File System," *Proc. USENIX Summer 1994 Technical Conf.*, pp. 183-195, 1994.
[20] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere, "Coda: A Highly Available File System for a Distributed Workstation Environment," *IEEE Trans. Computers*, vol. 39, no. 4, Apr. 1990.
[21] J. Shim, P. Scheuermann, and R. Vingralek, "Proxy Cache Algorithms: Design, Implementation, and Performance," *IEEE Trans. Knowledge and Data Eng.*, vol. 11, no. 4, July/Aug. 1999.
[22] K. Stathatos, N. Roussopoulos, and J. Baras, "Adaptive Data Broadcast in Hybrid Networks," *Proc. 23rd VLDB Conf.*, 1997.
[23] K. Tan, J. Cai, and B. Ooi, "Evaluation of Cache Invalidation Strategies in Wireless Environments," *IEEE Trans. Parallel and Distributed Systems*, vol. 12, no. 8, pp. 789-807, Aug. 2001.
[24] K. Wu, P. Yu, and M. Chen, "Energy-Efficient Caching for Wireless Mobile Computing," *Proc. 20th Int'l Conf. Data Eng.*, pp. 336-345, Feb. 1996.
[25] J. Yuen, E. Chan, K. Lam, and H. Leung, "Cache Invalidation Scheme for Mobile Computing Systems with Real-Time Data," *ACM SIGMOD Record*, Dec. 2000.

Guohong Cao received the BS degree from Xian Jiaotong University, Xian, China. He received the MS degree and PhD degree in computer science from the Ohio State University in 1997 and 1999, respectively. Since Fall 1999, he has been an assistant professor of computer science and engineering at Pennsylvania State University. His research interests include mobile computing, wireless networks, and distributed fault-tolerant computing. He was a recipient of the Presidential Fellowship at the Ohio State University. He is a recipient of the US National Science Foundation CAREER award. He is a member of the IEEE and the IEEE Computer Society.