Power-Aware Cache Management in Mobile Environments

Guohong Cao
Department of Computer Science & Engineering
The Pennsylvania State University
University Park, PA 16802
E-mail: gcao@cse.psu.edu

Abstract

Recent work has shown that *invalidation report (IR)* based cache management is an attractive approach for mobile environments. To improve the cache hit ratio of IR-based approach, clients should proactively prefetch the data that are most likely used in the future. Although prefetching can make use of the broadcast channel and improve cache hit ratio, clients still need to consume power to receive and process the data. In this chapter, we first present a basic scheme to dynamically optimize performance and power based on a novel *prefetch-access ratio* concept. Then, we extend the scheme to achieve a balance between performance and power considering various factors such as access rate, update rate, and data size.

Index Terms: Invalidation report, latency, cache management, power-aware, prefetch, mobile computing.

1 Introduction

With the advent of the third generation wireless infrastructure and the rapid growth of wireless communication technology such as Bluetooth and IEEE 802.11, wireless Internet becomes possible: people with battery powered mobile devices (PDAs, hand-held computers, cellular phones, etc.) can access various kinds of services at any time any place. However, the goal of achieving ubiquitous connectivity with small-size and low-cost mobile devices (clients) is challenged by the power constraints. Most mobile clients are powered by battery, but the rate at which battery performance improves is fairly slow [19]. Aside from major breakthrough in battery technology, it is doubtful that significant improvement can be expected in the foreseeable future. Instead of

trying to improve the amount of energy that can be packed into a power source, we can design power-aware protocols so that the mobile clients can perform the same functions and provide the same services while minimizing their overall power consumption.

Understanding the power characteristics of the wireless network interface (WNI) used in mobile clients is important for the efficient design of communication protocols. A typical WNI may operate in four modes: transmit, receive, idle, and sleep. Many studies [25, 28] show that the power consumed in the receive or idle mode is similar, but are significantly higher than the power consumed in the sleep mode. As a result, most of the work on power management concentrates on putting the WNI into sleep when it is in the idle mode. This principle has been applied to different layers of the network hierarchy [8, 20, 28], and can also be applied to data dissemination techniques such as broadcasting. With broadcasting, mobile clients access data by simply monitoring the channel until the required data appear on the broadcast channel. To reduce the client power consumption, techniques such as indexing [12] 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. As a result, a client can enter sleep 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 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 [23], Ficus [21] can be applied to mobile environments, these file systems are primarily designed for point-to-point communication environment, and may not be applicable to the broadcasting environment.

Recently, many works [3, 6, 5, 4, 14, 30, 26, 29] 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. Since IRs arrive periodically, clients can go to sleep most of time and only wake up when the IR comes. 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 such as long query latency and low cache hit ratio. In our previous work [6], we addressed the long latency problem with a UIR-based approach, where 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. To improve the cache hit ratio, we proposed a proactive cache management scheme [5], where clients intelligently prefetch the data that are most likely used in the future. Prefetching has many advantages in mobile environments since wireless networks such as wireless LANs or cellular networks support broadcasting. When the server broadcasts data on the broadcast channel, clients can prefetch interested data to increase the cache hit ratio without increasing the bandwidth consumption. Although prefetching can make use of the broadcast channel and improve cache hit ratio, clients still need to consume power to receive and process the data. Further, they cannot power off the wireless network interface, which consumes a large amount of power even when it is in the idle mode [24]. Since most mobile clients are powered by battery, it is important to prefetch the right data. Unfortunately, most of the prefetch techniques used in the current cache management schemes [6, 7] do not consider power constraints of the mobile clients and other factors such as the data size, the data access rate, and the data update rate.

To address the power consumption issue, we first present a basic adaptive scheme to save power during prefetch. 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. Then, we extend the basic scheme and present a value-based (VP) scheme, which makes prefetch decisions based on the value of each data item considering various factors such as access rate, update rate, and data size. Finally, we extend the VP scheme and present two adaptive value-based prefetch (AVP) schemes, which can achieve a balance between performance and power based on different user requirements.

The rest of the chapter is organized as follows. Section 2 develops the necessary background. In Section 3, we propose power-aware cache management techniques to balance the tradeoff between performance and power. Section 4 concludes the chapter and points out future research directions.

2 Cache Invalidation Techniques

In this section, we define our cache consistency model, and describe techniques to improve the performance of the IR-based cache invalidation model.

2.1 Cache Consistency Model

When cache techniques are used, data consistency issues must be addressed. 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. In this chapter, we use the *latest value* consistency model [2, 6, 16], 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. Note that the Coda file system [23] 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.

When client caching is allowed, techniques should be applied to maintain the latest value consistency. Depending on whether or not the server maintains the state of the client's cache, two invalidation strategies are used: the *stateful* server approach and the *stateless* server approach. In

the stateful server approach, the server maintains the information about which data are cached by which client. Once a data item is changed, the server sends invalidation messages to the clients with copies of the particular data. The Andrew File System [17] is an example of this approach. However, in mobile environments, the server may not be able to contact the disconnected clients. Thus, a disconnection by a client automatically means that its cache is no longer valid. Moreover, if the client moves to another cell, it has to notify the server. This implies some restrictions on the freedom of the clients. In the stateless server approach, the server is not aware of the state of the client's cache. The clients need to query the server to verify the validity of their caches before each use. The Network File System (NFS) [22] is an example of this approach. Obviously, in this option, the clients generate a large amount of traffic on the wireless channel, which not only wastes the scarce wireless bandwidth, but also consumes a lot of battery energy. Next, we present the IR-based cache invalidation model which has been widely used in mobile environments.

2.2 The IR-based Cache Invalidation Model

In the IR-based cache invalidation strategy, the server periodically broadcasts invalidation reports (IRs), which indicates the updated data items. Note that only the server can update the data. To ensure cache consistency, every client, if active, listens to the IRs and uses these IRs to invalidate its cache accordingly. To answer a query, the client listens to the next IR and uses it to decide whether its cache is still valid or not. If there is a valid cached copy of the requested data, the client returns the data immediately. Otherwise, it sends a query request through the uplink (from the client to the server). The server keeps track of the recently updated information and broadcasts an IR every L second. In general, a large IR can provide more information and is more effective for cache invalidation, but a large IR occupies a large amount of broadcast bandwidth and the clients may need to spend more power listening to the IR since they cannot switch to power save mode while listening. In the following, we look at two IR-based algorithms.

2.2.1 The Broadcasting Timestamp (TS) Scheme

The TS scheme was proposed by Barbara and Imielinski [3]. In this scheme, 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.

In order to save energy, an MT may power off most of the time and only turn on during the IR broadcast time. Moreover, an MT may be in the power off mode for a long time to save energy, and hence the client running the MT may miss some *IR*s. Since the IR includes the history of the past w broadcast intervals, the client can still validate its cache as long as its 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 items in its cache, discarding the entire cache may consume a large amount of wireless bandwidth in future queries.

2.2.2 The Bit Sequences (BS) Scheme

In the BS scheme [14], the IR consists of a sequence of bits. Each bit represents a data item in the database. Setting the bit to 1 means that the data item has been updated. The update time of each data item is also included in the IR. To reduce the length of the IR, some grouping methods are used to make one bit coarsely represent several data items. Instead of including one update timestamp for each data item, the BS scheme uses one timestamp to represent a group of data items in a hierarchical manner. Let IR be $\{[B_0, TS(B_0)], ..., [B_k, TS(B_k)]\}$ where $B_i = 1$ means that half of the data items from 0 to 2^i at time $TS(B_i)$ have been updated. The clients use the bit sequences and the time-stamps to decide what data items in their local cache should be invalidated. The scheme is very flexible (no invalidation window size is needed) and it can be used to deal with the long disconnection problem by carefully arranging the bit sequence. However, since the IR represents the data of the entire database (half of the recently updated data items in the database if more than half data items have been updated since the initial time), broadcasting the IR may consume a large amount of downlink bandwidth.

Many solutions [11, 14, 27] are proposed to address the long disconnection problem, and Hu *et al.* [11] has a good survey of these schemes. Although different approaches [3, 14] apply different techniques to construct the IR to address the long disconnection problem, these schemes

maintain cache consistency by periodically broadcasting the IR. The IR-based solution is attractive because it can scale to any number of MTs who listen to the IR. However, this solution has long query latency, since the client can only answer the query after it receives the next IR to ensure cache consistency. Hence, the average latency of answering a query is the sum of the actual query processing time and half of the IR interval.

2.3 The UIR-based Cache Invalidation Model

In order to reduce the query latency, we [6] proposed to replicate the IRs m times; that is, the IR is repeated every $(\frac{1}{m})^{th}$ of the IR interval. As a result, 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).

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. 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 Figure 1. In Figure 1, $T_{i,k}$ represents the time of the k^{th} UIR after the i^{th} 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 present a cache management algorithm to improve the cache hit ratio and the bandwidth utilization.

2.4 Using Prefetch to Improve Cache Hit Ratio and Bandwidth Utilization

In most previous IR-based schemes, 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

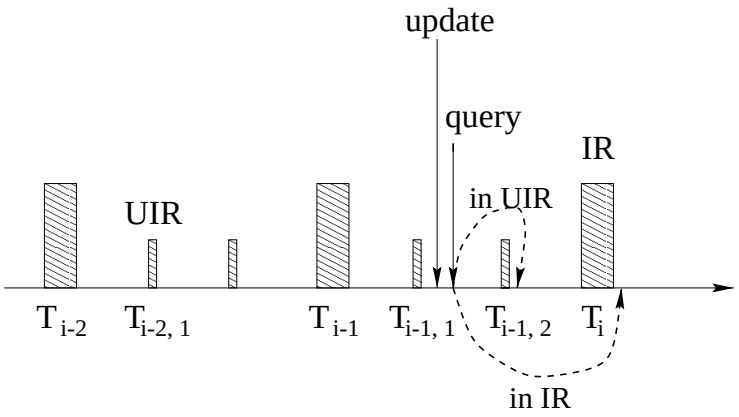

Figure 1: Reducing the query latency by replicating UIRs

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.

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[1] 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 sleep 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,

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

two cache invalidation strategies are used: the *stateful* server approach and the *stateless* server approach. In [4, 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 chapter. 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 any 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 tradeoff some delay for more battery energy. Due to the use of UIR, the delay tradeoff 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.

2.4.1 Remarks

Prefetching has been widely used to reduce the response time in the Web environment. Most of these techniques [13, 15, 18] 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. Although the objective of prefetching is the same: to improve cache hit ratio and reduce the response time, there are many differences between our prefetch technique and the existing work. First, prefetch in the Web environment will increase the web traffic, but not in our UIR-based model due to the broadcast environment. Second, the prefetch technique used in UIR is not a simple prefetch. We consider power consumption issues and carefully design L_{blist} so that clients can still stay sleep most of time. With our careful design, most clients stay sleep when not prefetching, but the clients that need to prefetch still consume power to download and process the data. Next, we present techniques to further reduce this part of power consumption.

3 Techniques to Optimize Performance and Power

In this section, we first present a basic scheme to optimize performance and power, then extend it to consider various factors such as access rate, update rate, and data size.

3.1 The Basic Scheme

The advantage of the prefetch 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 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 tradeoff between performance and power, and propose an adaptive scheme to efficiently utilize

the power in this subsection.

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 *non-prefetch* 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.

The basic 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 *non-prefetch*, 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 and resulting in poor performance. Since clients may have a large probability to access a very small amount of data, marking these data items as pre-fetch may improve the cache hit ratio and does not consume too much power. Based on this idea, when $PAR > \beta$, the client marks N_p 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_{acc}^x is the number of access times for a cache entry d_x. Assume $N_{c\_acc}^x$ 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

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

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

Although the basic adaptive prefetch scheme can achieve a better tradeoff between performance and power, it does not consider varying data size and the data update rate. Also, there is no clear methodology as to how and when N_p should be changed. In the next two subsections, we will address this problem by an adaptive value-based prefetch (AVP) scheme, which consists of two parts. The first part is the value-based prefetch (VP) scheme, which identifies valuable data for prefetching. The second part is the adaptive value-based prefetch (AVP) scheme, which determines how many data items should be prefetched.

3.2 The Value-Based Prefetch (VP) Scheme

To consider the effects of data size, we need to introduce a new performance metric. One widely used performance metric is the response time, i.e., the time between sending a request and receiving the reply. It is a suitable metric for homogeneous settings where different data requests have the same "size". However, the data requirements of users and applications are inherently diverse, and then to encapsulate all responses into a single-size broadcast would be unreasonably wasteful. Therefore, unlike some previous work [5, 10], we do not assume that the data items have the same size. When data requests are heterogeneous, response time alone is not a fair measure given that the individual requests significantly differ from each another in their *service time*, which is defined as the time to complete the request if it was the only job in the system. We adopt an alternate performance measure, namely the *stretch* [1] of a request, defined to be *the ratio of the response time of a request to its service time*. The rationale behind this choice is based on our intuition; i.e., clients with larger jobs should be expected to be in the system longer than those with smaller requests. The drawback of minimizing response time for heterogeneous workloads is that it tends to improve the system performance of large jobs since they contribute the most to the response time. Minimizing stretch, on the other hand, is more fair to all job sizes. Note that in broadcast systems, the service time for a request is the requested data size divided by the bandwidth. For simplicity, we remove the constant bandwidth factor and use the data size to represent the service time.

Next, we present a value-based function which allows us to gauge the worth of a data item when making a prefetch decision. The following notations are used in the presentation:

- p_{a_i} : the access probability of data item i
- p_{u_i}: the probability of invalidating cached data item i before next access.
- f_i : the delay of retrieving data item i from the server
- s_i : the size of data item i
- v : the cache validation delay

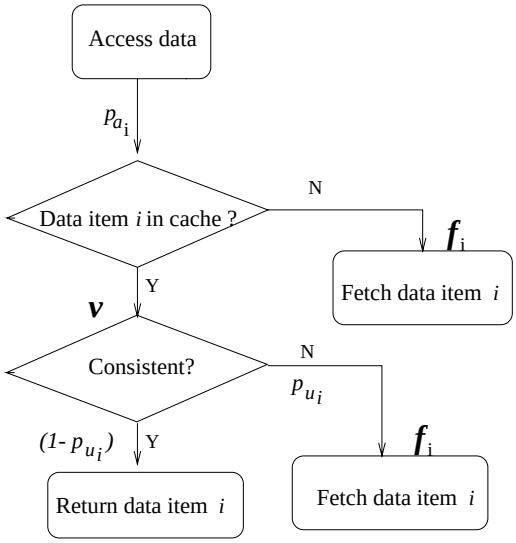

Figure 2: The data access cost model

The value function is used to identify the data to be prefetched. Intuitively, the ideal data item for prefetching should have a high access probability, a low update rate, a small data size, and a high retrieval delay. Equation (1) incorporates these factors to calculate the value of a data item i.

$$value(i) = \frac{p_{a_i}}{s_i}(f_i - v - p_{u_i} \cdot f_i) \tag{1}$$

This value function can be further explained by the data access cost model shown in Figure 2. If item i is not in the cache, in terms of the stretch value, it takes f_i/s_i to fetch item i into the cache. In other words, if i is prefetched to the cache, the access cost can be reduced by f_i/s_i. However, it also takes $((v + P_{u_i} f_i))/s_i$ to validate the cached item i, and update it if necessary. Thus, prefetching the data can reduce the cost by $((f_i - v - p_{u_i} \cdot f_i))/s_i$ for each access. Since the access probability is p_{a_i}, the value of prefetching item i is $\frac{p_{a_i}}{s_i}(f_i - v - p_{u_i} \cdot f_i)$.

The VP scheme decides which data item should be prefetched based on the value function. The VP scheme is defined as follows. Suppose a client can prefetch N_p data items, the VP scheme prefetches the N_p items which have the highest value based on the value function. Note that VP is not responsible for determining how many items (N_p) should be prefetched. N_p is determined by the adaptive scheme, which will be discussed in Section 3.3.

Theorem 1 *Prefetching items with high value can achieve lower stretch than any other prefetch*

schemes given that the number of prefetches is limited.

Details of the proof and how to estimate the parameters can be found in [29]. The proposed value-based function is calculated in terms of stretch since the performance metric is stretch. Actually, this value-based function can be easily extended for other performance metrics. For example, if the performance metric is query delay, the value function will be changed to $value(i) = p_{a_i}(f_i - v - p_{u_i} \cdot f_i)$. Similar techniques can be used to prove that this value function can minimize the query delay.

3.3 The Adaptive Value-based Prefetch (AVP) Scheme

Due to limitations of battery technology, the energy available for a mobile client is limited and must be used prudently. If the prefetched data item is not accessed or is invalidated before it is accessed, the energy spent on downloading this item will be wasted. To avoid wasting power, it is important that clients only download the data with high value, but such a strict policy may adversely affect the performance of the system and increase the query delay.

Each client may have different available resources and performance requirements, and these resources such as power may change over time. Since N_p controls the number of data to be prefetched and then affects the tradeoff between performance and power, we propose adaptive schemes to adjust N_p to satisfy different client requirements.

3.3.1 The Value of N_p

When N_p reduces to 0, there will be no prefetch. As N_p increases, the number of prefetches increases and the power consumption also increases. Since the maximum number of data items to be prefetched is limited by the cache size, N_p is also limited by this number. Intuitively, the query delay decreases as the number of prefetches increases. However, this is not always true considering the overhead to maintain cache consistency. In our cache invalidation model, a client needs to wait for the next IR to verify the cache consistency. This waiting time may increase the query delay compared to the approaches without prefetch. The cost has been quantified in Equation 1, where v is the cache invalidation delay. Due to the cost of v, the value of a data item may be negative.

If $value(i)$ is negative, prefetching item i not only wastes power but also increases the average stretch. Therefore, N_p should be bounded by N_p^{max}, which is limited by the client cache size and the data value; i.e., a client will not prefetch items with negative values.

The tradeoff between performance and power can be achieved by adjusting N_p. In the following subsections, we present two adaptive schemes: the AVP_T (T for Time) scheme which dynamically adjusts N_p to reach a target battery life time, and the AVP_P (P for Power) scheme which dynamically adjusts N_p based on the remaining power level.

3.3.2 AVP_T: Adapting N_p to Reach a Target Battery Life

A commuter normally knows the amount of battery energy and the length of the trip between home and office. With these resource limitations, the commuter wants to achieve the lowest query delay. This is equivalent to the problem of adapting N_p to reach a target battery life and minimize the average stretch. Suppose a battery with E joule lasts T_1 seconds when $N_p = N_p^{max}$, and T_2 seconds when $N_p = 0$. It is possible to adjust N_p to reach a target battery life time $T \in [T_1, T_2]$. In AVP_T, the client monitors the power consumed in the past. If it consumes too much power in the past and cannot last T seconds, N_p is reduced. On the other hand, it increases N_p when it found that it has too much power left. Certainly, N_p should be bounded by N_p^{max}.

3.3.3 AVP_P: Adapting N_p based on the Power Level

When the energy level is high, power consumption is not a major concern and then trading off energy for performance may be a good option if the user can recharge the battery soon. On the other hand, when the energy level is low, the system should be power-aware to prolong the system running time to reach the next battery recharge time. Based on this intuition, the AVP_P scheme dynamically changes N_p based on the power level. Let a_k be the percentage of energy left in the client. When a_k drops to a threshold, the number of prefetches should be reduced to some percentage, say $f(a_k)$, of the original value. Some simple discrete function can be as follows:

$$f(a_k) = \begin{cases} 100\% & 0.5 < a_k \leq 1.0 \\ 70\% & 0.3 < a_k \leq 0.5 \\ 50\% & 0.2 < a_k \leq 0.3 \\ 30\% & 0.1 < a_k \leq 0.2 \\ 10\% & a_k \leq 0.1 \end{cases} \quad (2)$$

At regular interval, the client re-evaluates 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 in the cache, which have the maximum value, as prefetchable. In this way, the number of prefetches can be reduced to prolong the system running time. Because this is a discrete function, N_p does not need to be frequently updated and the computation overhead is low.

Note that a simple policy which is neither too aggressive nor conservative might result in similar average stretch and lifetime as the VAP_P scheme if the battery runs out before recharge. However, if the user recharges the battery frequently, this simple policy may not be a good option since it saves power at the cost of delay, but power consumption is not a concern at this time. In contrast, our adaptive scheme tries to tradeoff power for performance at the beginning, and become power-aware when the client cannot recharge in time.

4 Conclusions and Future Work

Prefetching can be used to improve the cache hit ratio and reduce the bandwidth consumption in our UIR-based cache invalidation model. However, prefetching consumes power. In mobile environments where power is limited, it is essential to correctly identify the data to be prefetched in order to provide better performance and reduce the power consumption. In this chapter, we first presented a basic scheme to dynamically optimize performance or power based on a novel *prefetch-access ratio* concept. Then, we extended it to achieve a balance between performance and power considering various factors such as access rate, update rate, and data size. Although we explored various adaptive approaches, many issues still need further investigation. For example, the channel state information [9] can be used when making prefetch decisions, i.e., N_p can be dynamically adjusted based on the channel state. If the target battery life time is not known or only known with some probability, AVP_T needs to be extended to factor into these uncertainties.

If the battery recharge cycle or the user profile is known or known with a high probability, how to enhance AVP_T and AVP_P needs further investigation.

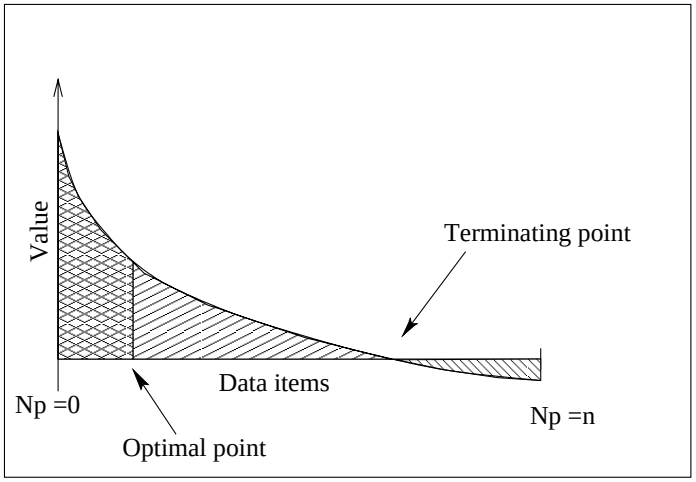

Figure 3: Prefetch value of the data items

Currently, we are investigating the optimal value of N_p. Figure 3 shows the effects of N_p (n is the database size) when data items are sorted according to their values from high to low. Suppose the value of the data can be correctly identified, then increasing N_p means more data items with high value will be prefetched. If the uplink power consumption is much higher than the downlink power, prefetching the data that will be access in the future can actually reduce the power consumption because the uplink power consumption can be saved. Thus, as N_p increases from 0, the system performance increases while the power consumption is not increased or even decreased. Improving performance without sacrificing power will continue until N_p reaches the *optimal point* shown in Figure 3.

After N_p passes through the optimal point, the system performance can still be improved. However, the power consumption will be increased, because some less valuable data are prefetched. Although prefetching these data can improve the performance, they are not frequently accessed and more likely to be invalidated, and hence prefetching them costs more power than getting them when being requested.

Intuitively, the system performance should be improved when more data are prefetched if the power consumption is not an issue. However, this may not be true if the IR-based cache invalidation

model is used to ensure the latest value consistency model. In our stretch value function (Equation 1), v is the cache invalidation delay. This v can make the value of a data item be negative, and hence prefetching the data will increase the average stretch compared to not prefetching it. Therefore, N_p should not be increased beyond a certain point (terminating point) even if the goal is to achieve the best performance.

In summary, before reaching the optimal point, prefetching can save power and improve performance. Between the optimal point and the terminating point, there is a tradeoff between performance and power. After the terminating point, the system should not prefetch. However, how to determine the optimal point and the terminating point still need further investigation. In some cases, e.g., when priority requests are not allowed in the UIR-based model, or when using difference cache consistency model, the terminating point may not exist.

References

[1] S. Acharya and S. Muthukrishnan, Scheduling On-Demand Broadcasts: New Metrics and Algorithms, *ACM MobiCom'98*, pp. 43–54, Oct. 1998.

[2] S. Acharya, M. Franklin, and S. Zdonik, Disseminating Updates On Broadcast Disks, *Proc. 22nd VLDB Conf.*, Sept. 1996.

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

[4] G. Cao, On Improving the Performance of Cache Invalidation in Mobile Environments, *ACM/Baltzer Mobile Networks and Application (MONET)*, vol. 7, no. 4, pp. 291–303, Aug. 2002.

[5] G. Cao, Proactive Power-Aware Cache Management for Mobile Computing Systems, *IEEE Transactions on Computer*, vol. 51, no. 6, pp. 608–621, June 2002.

[6] G. Cao, A Scalable Low-Latency Cache Invalidation Strategy for Mobile Environments, *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 5, September/October 2003 (A preliminary version appeared in 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, *The 20th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pp. 247–256, Oct. 2001.

[8] J. C. Chen, K. M. Sivalingam, P. Agrawal and R. Acharya, Scheduling Multimedia Services in A Low-Power MAC for Wireless and Mobile ATM Networks, *IEEE/ACM Transactions on Multimedia*, vol. 1, no. 2, June 1999.

[9] Savvas Gitzenis and Nicholas Bambos, Power-Controlled Data Prefetching/Caching in Wireless Packet Networks, *INFOCOM*, 2002.

[10] V. Grassi, Prefetching Policies for Energy Saving and Latency Reduction in a Wireless Broadcast Data Delivery System, *ACM MSWIM*, 2000.

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

[12] T. Imielinski, S. Viswanathan, and B. Badrinath, Data on Air: Organization and Access, *IEEE Transactions on Knowledge and Data Engineering*, vol. 9, no. 3, pp. 353–372, May/June 1997.

[13] Z. Jiang and L. Kleinrock, An Adaptive Network Prefetch Scheme, *IEEE Journal on Selected Areas in Communications*, vol. 16, no. 3, pp. 1–11, April 1998.

[14] 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.

[15] A. Joshi, On Proxy Agents, Mobility, and Web Access, *Mobile Networks and Applications*, vol. 5, no. 4, pp. 233–241, Dec. 2000.

[16] A. Kahol, S. Khurana, S. Gupta, and P. Srimani, An Efficient Cache Management Scheme for Mobile Environment, *The 20th Int'. Conf. on Distributed Computing Systems*, pp. 530–537, April 2000.

[17] M. Kazar, Synchronization and Caching Issues in the Andrew File System, *USENIX Conf.*, pp. 27–36, 1988.

[18] V. Padmanabhan and J. Mogul, Using Predictive Prefetching to Improve World Wide Web Latency, *Computer Communication Review*, pp. 22–36, July 1996.

[19] R. Powers, Batteries for Low Power Electronics, *Proc. IEEE*, vol. 83, no. 4, pp. 687–693, April 1995.

[20] B. Prabhakar, E. Uysal-Biyikoglu, and A. El Gamal, Energy-Efficient Transmission over a Wireless Link via Lazy Packet Scheduling, *IEEE INFOCOM'01*, March 2001.

[21] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G.J. Popek, Resolving file conflicts in the Ficus file system, *Proc. of the USENIX Summer 1994 technical Conference*, pp. 183–195, 1994.

[22] S. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, Design and Implementation of the Sun Network File System, *Proc. USENIX Summer Conf.*, pp. 119–130, June 1985.

[23] 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 Transactions on Computers*, vol. 39, no. 4, April 1990.

[24] M. Stemm and R. Katz, Measuring and Reducing Energy Consumption of Network Interfaces in Hand-Held Devices, *IEICE Trans. on Communications*, vol. 80, no. 8, pp. 1125–1131, Aug. 1997.

[25] M. Stemm and R. H. Katz, Measuring and Reducing Energy Consumption of Network Interfaces in Handheld Devices, *IEICE Transactions on Communications*, vol. E80-B, no. 8, August 1997.

[26] K. Tan, J. Cai, and B. Ooi, Evaluation of Cache Invalidation Strategies in Wireless Environments, *IEEE Transactions on Parall and Distributed Systems*, vol. 12, no. 8, pp. 789–807, 2001.

[27] K. Wu, P. Yu, and M. Chen, Energy-Efficient Caching for Wireless Mobile Computing, *The 20th Int'l Conf. on Data Engineering*, pp. 336–345, Feb. 1996.

[28] Y. Xu, J. Heidemann and D. Estrin, Geography-informed Energy Conservation for Ad Hoc Routing, *Mobicom'01*, July 2001.

[29] L. Yin, G. Cao, C. Das, and A. Ashraf, Power-Aware Prefetch in Mobile Environments, *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pp. 571–578, 2002.

[30] 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.