Distributed and Parallel Databases 6, 351–372 (1998)

Energy-Efficient Mobile Cache Invalidation

KUN-LUNG WU, PHILIP S. YU AND MING-SYAN CHEN klwu@watson.ibm.com
IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598

Received ; Accepted

Abstract. Caching data in a wireless mobile computer can significantly reduce the bandwidth requirement. However, due to battery power limitation, a wireless mobile computer may often be forced to operate in a doze or even totally disconnected mode. As a result, the mobile computer may miss some cache invalidation reports. In this paper, we present an energy-efficient cache invalidation method for a wireless mobile computer. The new cache invalidation scheme is called grouping with cold update-set retention (GCORE). Upon waking up, a mobile computer checks its cache validity with the server. To reduce the bandwidth requirement for validity checking, data objects are partitioned into groups. However, instead of simply invalidating a group if any of the objects in the group has been updated, GCORE retains the cold update set of objects in a group if possible. We present an efficient implementation of GCORE and conduct simulations to evaluate its caching effectiveness. The results show that GCORE can substantially improve mobile caching by reducing the communication bandwidth (thus energy consumption) for query processing.

Keywords: mobile computing, energy-efficient caching, cache invalidation, wireless computing

1. Introduction

In a mobile computing environment, a large number of battery-powered, portable machines can be used by many users to query the information and/or database servers from different places through the wireless communication channels [4, 7–9]. These mobile computers may often be disconnected (i.e., powered off) for perhaps prolonged periods of time in order to conserve battery energy. They may also frequently relocate from one cell to another and connect to different data servers at different times. This entire mobile computing information infrastructure could enable the users with unrestricted mobility, and satisfy their information needs at any time and in any place.

Generally speaking, the bandwidth of the wireless channel is very limited. Thus, caching of frequently used data in a mobile computer can be an effective approach to reducing the wireless bandwidth requirement [4]. Once caching is used, however, a cache invalidation strategy is needed to ensure the data cached in a mobile computer are consistent with those stored in the server.

Depending on whether or not the server maintains the state of the mobile clients' cache, Barbara and Imielinski classified cache invalidation strategies into two categories [4]. In the first category, the server knows which data are cached by which mobile computers and is called a *stateful* server. Once a data item is changed, the server sends invalidation messages to the clients that are caching that particular data. The server has to locate the clients. But, disconnected mobile clients cannot be contacted by the server. Thus, a disconnection by a mobile computer automatically means its cache is no longer valid. Moreover, if a mobile

computer wants to relocate, it may have to notify the servers. This implies some restrictions on the freedom of the mobile computer.

In the second category, the server does not know about the state of its clients' cache and is called a *stateless* server. The server does not even know which mobile computers are currently active. To ensure cache consistency, the server simply periodically broadcasts an invalidation report containing the data items that have been updated recently. The mobile clients listen to the broadcast and invalidate their caches accordingly.

In [4], three cache invalidation schemes using different invalidation reports were proposed for the case of a stateless server. However, in these three schemes no attempt was made to check with the server whether or not some of the cached objects are still valid after a reconnection. As a result, when a mobile computer wakes up, it may have to discard the entire cache contents if the disconnection has been too long. This is because the mobile computer does not know whether or not some of its cached objects have been updated since it became disconnected. Discarding the entire cache because of a disconnection can be costly as most of the benefits of caching are lost, especially if most of the cached objects are still valid because they have not been updated for a long period of time.

In this paper, we propose an energy-efficient cache invalidation scheme that salvages as many cached objects as possible after a reconnection, if they are still valid. Unlike the schemes proposed in [4], which do not check cache validity after a reconnection, our schemes check the cache validity with the server, if necessary, and retain as many valid objects as possible. Since checking cache validity not only requires uplink bandwidth but also consumes battery energy, it must be done efficiently. One simple-checking approach is to send all the cached object ids to the server. This is costly because the number of object ids can be large. One possible approach to reducing the overhead of validity checking is to do it at a group level, instead of object level. With such a simple-grouping scheme, however, the entire group must be invalidated if any of the objects in the group has been updated. In this paper, we propose a new scheme called Grouping with COld update-set REtention (GCORE). In GCORE, instead of invalidating the entire group, we retain the cold update set of the objects in the group if all the updated objects (most likely belong to the hot update set) have been included in the most recently broadcasted invalidation report.

Here, the hot update set represents the set of objects that are frequently updated by transactions in the server, while the cold update set is the set of objects that are less frequently updated. In other words, most of the transactions access and update objects belonging to the hot update set. Obviously, some of the objects referenced by queries and cached in a mobile computer may belong to the hot update set and thus may often be invalidated due to changes occurred in the server. Nevertheless, objects that are frequently updated are highly likely to be included in the latest broadcast invalidation report. Therefore, the objects in a group that belong to the cold update set can be retained if the recently updated objects in the group can be correctly identified. GCORE tries to efficiently retain, if possible, the cold update set of a group on the mobile computer. For example, if there are 100 objects in a group and 3 of them (likely to belong to the hot update set) have been recently updated, then GCORE tries to retain the rest of the 97 objects (likely to belong to the cold update set) in the group if they have not been updated.

In GCORE, a mobile computer invalidates its cache upon waking up based on the first received broadcast invalidation report and then checks the validity of the rest of the cache

at a group level. The server maintains a simple group update history that dynamically excludes the hot update set. As a result, the server can efficiently determine whether or not the rest of the group can be retained by any mobile computer without tracking the mobile computer's cache state. In other words, even though certain objects belonging to a group may or may not be in a mobile computer's cache, the validity of the group can be efficiently determined by the server. Detailed descriptions of GCORE along with a simple-checking and a simple-grouping schemes are presented in this paper.

To evaluate and compare the performance of GCORE with these schemes, we developed an event-driven simulator. The effectiveness of caching is computed as the bandwidth requirement for query processing in a mobile computer, including the uplink and downlink communication costs and the broadcast invalidation cost. Lower bandwidth requirement means more energy-efficient caching in a mobile computer because it consumes less energy to receive and send messages. The simulation results show that compared with no-checking (such as the ones presented in [4]) and simple-checking schemes, both GCORE and the simple-grouping schemes significantly improve the caching effectiveness by reducing the bandwidth requirement for query processing. This is particularly true if a mobile computer can be disconnected for a long period of time and most of the data in the server still have not been updated. Furthermore, GCORE requires less energy consumption than the simple-grouping scheme does, especially if the group size is large.

The recent popularity of portable personal computers and personal digital assistants has attracted a lot of interests in mobile computing. Many applications using these devices can benefit from mobile caching in general [4] and GCORE in particular. For example, a large number of mobile users can keep abreast of news updates involving business information, recent sales/profits figures, and company news related to personal investment portfolios. These news items and pieces of information tend not to be updated frequently and can be maintained in a server. Invalidation messages can be broadcast periodically so that the mobile devices can be notified about the changes. After querying the server for his/her interested news and information, a mobile user can turn off the device and turn on again later on to query the server for any updates since the last query. If most of the items are not frequently updated, GCORE can save the energy consumption of the mobile device by reducing the communication overhead with the server. Of course, if the items are updated frequently, such as stock quotes, then mobile caching cannot help, nor can GCORE.

There have been many papers discussing other aspects of supporting mobile computing, including location management, data replication, communication and other system design issues, such as [1–3, 5, 6, 10, 11]. These citations are by no means exhaustive as many research and development projects are currently being actively conducted to build the national information infrastructure. Our attentions in this paper specifically focus on the issues of supporting energy-efficient caching in a wireless mobile computing environment, closely related to [4, 7]. Effective caching and other issues for supporting mobile computing are very important in the future for providing information services to users at any time and in any place.

The rest of the paper is organized as follows. Section 2 describes the cache invalidation schemes for a stateless server. Section 3 presents the simulation model. Section 4 discusses the simulation results. Finally, Section 5 summarizes the paper.

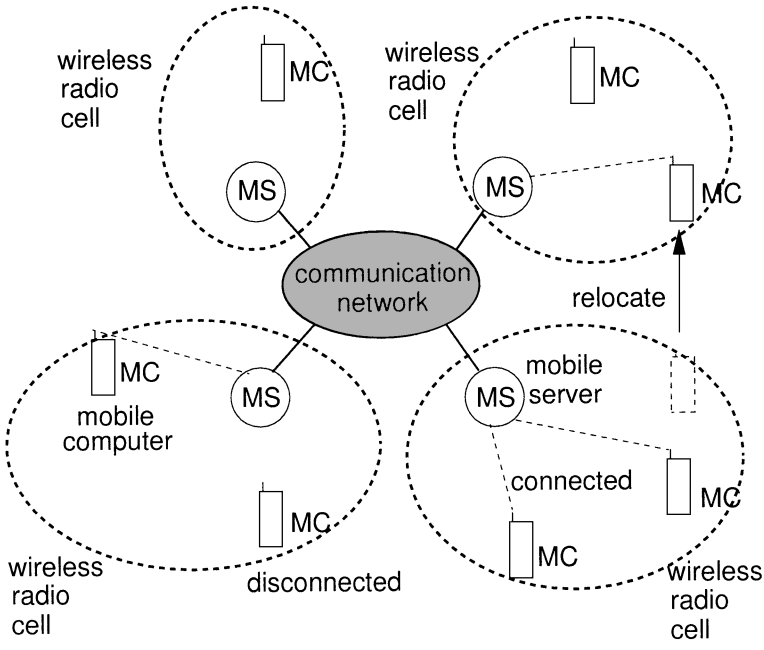


Figure 1. A wireless mobile computing environment.

2. Cache invalidation schemes

In this section, we describe the details of GCORE for energy-efficient mobile caching. For comparison purposes, we also present a no-checking scheme, a simple-checking scheme and a simple-grouping scheme.

Figure 1 shows a generic wireless mobile computing environment, similar to the one described in [4, 8]. There are multiple wireless radio cells. Each cell has a mobile server station that is equipped with wireless communication capability. The mobile server station is simply called mobile server in this paper and it stores a complete copy of the database. The mobile servers are connected through a communication network (typically wired). A mobile computer can connect to a server (uplink) through a wireless communication channel. It can disconnect from the server by operating in a dose mode (consumes significantly less energy) or a power-off mode. It can move from one cell to another cell. The server can communicate with a particular mobile computer through a wireless channel, if the mobile computer is not powered off. We assumed that data are only updated in the servers. Mobile computers only read the data and do not update them. To ensure cache consistency, the server periodically broadcasts invalidation reports and all the mobile computers, if active, listen to the reports and invalidate their caches accordingly. Database is assumed to be completely replicated in the mobile servers, so that when a mobile computer moves to another cell, it receives similar invalidation reports. However, it is possible that the communication cells may not overlap and hence a mobile device cannot communicate with any server when moving between cells. In this case, the mobile device can simply disallow the user to access the cashed objects

or allows the access but signals that the cached objects may be obsolete. In this paper, we assume that a mobile device can always communicate with the server when it wakes up.

Frequently accessed data objects by queries are cached in a mobile computer's cache. Here, we assumed that the cache at the mobile computer is a nonvolatile memory such as a hard disk. After a disconnection period, the content of the cache can be retrieved. The server keeps track of the object ids that are recently updated and broadcasts an invalidation report every L seconds. The most recent invalidation report broadcasted is denoted as *IR* in this paper. *IR* consists of the current timestamp T and a list of (o_i, t_i) such that $t_i > (T - w \times L)$, where o_i is an object id and t_i is its corresponding most recent update timestamp, and w is the invalidation broadcast window. In other words, *IR* contains the update history of the past w broadcast intervals.

2.1. No-checking caching scheme

Due to the constraint of limited energy, a mobile computer is usually required to operate in a doze mode (not active) or even to be completely disconnected for a prolonged period of time. As a result, a mobile client may miss certain invalidation messages broadcasted by the server. Upon missing an invalidation report, a mobile computer may have to discard the entire cache, since it does not know which parts of the cache is valid. This simple scheme is called the no-checking scheme in this paper and is similar to the broadcasting timestamp scheme proposed in [4]. Figure 2 shows the query processing algorithm for a

Query Processing Using the No-Checking Scheme:

```
if (T_lb < (T − w × L))
    invalidate the entire cache;
else
    {
        for ∀o_i ∈ IR
            {
                if (o_i is in the cache) and (t_i > t_i^c)
                    invalidate o_i;
            }
    }
∀q ∈ QL
    {
        if all the objects referenced by q are in the cache
            process q;
        else
            send the missed object ids to the server;
    }
T_lb = T;
```

Figure 2. Algorithm for query processing using the no-checking scheme.

mobile computer after receiving an invalidation report. Throughout this paper, we assumed that all the queries are batched in a query list, *QL*, and are not processed until a mobile computer invalidates its cache after receiving an invalidation report. Also, the timestamp of the latest invalidation report received, denoted by T_{lb}, is also reliably maintained so that after a mobile computer wakes up from a disconnection, it knows the timestamp of the latest report that it received.

In figure 2, t_i^c is the timestamp of the cached copy of object o_i. For those objects that are missed in the cache, the object ids are sent to the server and the server then sends back the data and the associated update timestamps to the mobile computer. They will be again cached in the mobile computer. For the no-checking scheme shown in figure 2, if a mobile computer has been disconnected for more than w broadcast intervals, then it must discard the entire cache contents once it reconnects. This can significantly increase the wireless bandwidth requirement between the mobile computer and the server as most of the objects subsequently referenced by queries result in cache misses.

2.2. Simple-checking caching scheme

Note that even after a long period of disconnection, if most of the cached data still have not been updated by the server, they should be retained in the cache after the mobile computer turns on. This can significantly reduce the bandwidth requirement because the mobile computer can use its local cached data. But, in order to retain cached data, it is necessary to identify which cached data are still valid and which should be purged.

There are several possible approaches to identifying valid cache entries after a disconnection, ranging from the most straightforward to the more sophisticated, such as GCORE proposed in this paper. They involve different trade-offs. To accurately identify the valid cache entries, a mobile computer can simply send to the server all the cached object ids and their corresponding timestamps. However, this requires a lot of uplink bandwidth as well as battery energy. On the other hand, the mobile computer can send group ids and group timestamps (the latest update time in a group) and the validity can be checked at the group level. This reduces the uplink bandwidth requirement. But, a single object updated in a group essentially invalidates the entire group. As a result, the amount of cached objects salvaged after a reconnection may be quite small, resulting in many future cache misses. GCORE tries to combine the advantages of both schemes by salvaging as many cached objects as close to the simple-checking scheme while consumes as little uplink costs as close to the simple-grouping scheme.

Note that a straightforward way of checking cache validity, either at the object level or the group level, is to simply send the object or group ids and their timestamps to the server. This is typically used for object version checking in a general distributed system. However, such a scheme can be unnecessarily costly in a wireless mobile computing environment because it requires uplink communication cost to send both object ids and timestamps. Instead, it is sufficient to just send T_{lb} with object ids or group ids to the server. This is because once a mobile computer processes a new *IR*, all the valid cache entries can be viewed as being timestamped at that moment. The server can check the validity of an

Query Processing Using the Simple-Checking Scheme:

```
for ∀o_i ∈ IR
    {
        if (o_i is in the cache) and (t_i > t_i^c)
            invalidate o_i;
    }
if (T_lb < (T − w × L))
    {
        send T_lb and all the object ids that are not yet invalidated to the server;
        wait for the validity report from the server;
        invalidate the cache according to the validity report;
    }
∀q ∈ QL
    {
        if all the objects referenced by q are in the cache
            process q;
        else
            send the missed object ids to the server;
    }
T_lb = T
```

Figure 3. Algorithm for query processing using the simple-checking scheme.

object or a group of objects based on T_{lb}, the timestamp of the latest broadcast report received by a mobile computer. Therefore, in all the three checking schemes discussed in this paper, the object timestamps or the group timestamps are not sent during validity checking.

For the simple-checking scheme of this paper, only T_{lb} and all the object ids that are still not yet invalidated by a mobile computer are sent to the server. The server then compares T_{lb} with the object update timestamp stored in the server and sends a validity report back to the mobile computer. This validity report can simply be a bit vector, with each bit indicating yes or no for the corresponding object. Figure 3 shows the algorithm for query processing after a mobile computer receives an invalidation report. It first invalidates its cache according to *IR*. Then, if the mobile computer just wakes up from a disconnection and $T_{\text{lb}} < (T - w \times L)$, it sends the cached objects that are not yet invalidated to the server for validity checking. Note that since *IR* contains the object ids that were updated during the past w broadcast intervals, no validity checking is necessary if $T_{\text{lb}} \geq (T - w \times L)$. After receiving the validity report from the server, the mobile computer then processes the queries. As mentioned above, the uplink communication overhead can be very large for the simple-checking scheme, especially if the cache size is large and the mobile computer is frequently disconnected.

2.3. *Simple-grouping caching scheme*

In order to reduce the uplink communication costs for validity checking, the entire database can be partitioned into a number of groups and a mobile computer checks its cache validity at the group level. The grouping function can be simply a modulo function, resulting in equal-sized groups. Or it can be different for different types of objects, resulting in different group sizes. Whatever grouping function is chosen, it must be agreed upon between the server and the mobile computer. Data objects belonging to a group may or may not be in a mobile cache. But, the group validity checking is not affected by the grouping function or by the fact that some of the group objects are not in a mobile cache. Note that, the grouping of objects is used only for cache validity checking after a mobile computer reconnects, it is not used by the server for broadcasting invalidation reports. Broadcast invalidation reports contain all the object ids and their most recent update timestamps in the last w broadcast intervals.

The algorithm for simple grouping is similar to the one for simple checking, except that it sends much less information to the server and as a result requires much less uplink bandwidth. Figure 4 shows the query processing algorithm using simple-grouping scheme. It is the same as GCORE, to be described next, in sending the group ids and T_{lb} to the server for group validity checking. However, it differs from GCORE in the way the server determines whether or not a group is valid. In the simple-grouping scheme, if any object

Query Processing Using Simple-Grouping and GCORE schemes:

```
for ∀o_i ∈ IR
    {
       if (o_i is in the cache) and (t_i > t_i^c)
          invalidate o_i;
    }
if (T_lb < (T − w × L))
    {
       send T_lb and all the group ids in the cache to the server;
       wait for the group validity report from the server;
       invalidate the cache according to the group validity report;
    }
∀q ∈ QL
    {
       if all the objects referenced by q are in the cache
          process q;
       else
          send the missed object ids to the server;
    }
T_lb = T
```

Figure 4. Algorithm for query processing using simple-grouping and GCORE schemes.

within a group is updated after T_{lb}, then the entire group is considered to be invalid. Thus, the amount of cached objects that can be retained may be small after a reconnection, resulting in a large amount of uplink and downlink costs due to cache misses from future query processing.

2.4. Grouping with cold update-set retention (GCORE)

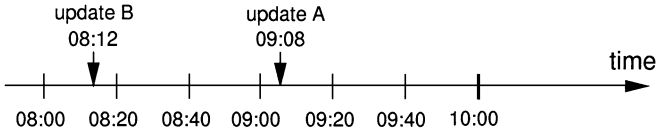
To avoid unnecessarily discarding of a group, we present a grouping with cold update-set retention scheme to improve the caching effectiveness for wireless mobile computing. Similar to the simple-grouping scheme, the server partitions the entire database objects into a number of groups. So, it incurs relatively small uplink costs for validity checking. However, unlike the simple-grouping scheme, GCORE tries to salvage a group if possible so that the future downlink costs due to cache misses can be significantly reduced. The query processing algorithm for a mobile computer using GCORE is shown in figure 4.

In addition to grouping, the server also dynamically identifies hot update set that has been updated recently in a group and excludes it from the group when checking the validity of the group. If all the updated object ids in a group have already been included in *IR*, these objects will be invalidated by the mobile computer when it receives *IR*. With the hot update set excluded from a group, the server can conclude that the objects that are not updated in the group can be retained in the cache and validate the rest of the group as a result. This scheme is therefore referred to as grouping with cold update-set retention in this paper. GCORE is energy efficient because it incurs both low uplink costs for validity checking and low downlink costs as it retains more cached objects.

To facilitate a mobile computer to salvage many of its local cache contents without incurring high uplink costs, the server needs to maintain a more sophisticated data structure for the group update history. In this paper, we present an efficient implementation for GCORE. It maintains for each group the object update history of the past W broadcast intervals ($W \geq w$), consisting of a list of object ids and their most recent update timestamps, and the most recent update time of the group. This group update history is maintained in `group_table[]` (see figure 5 for its definition). In addition, it also maintains the number of distinct objects that were most recently updated between $(T - W \times L)$ and $(T - w \times L)$ to speed up the group validity checking.

```
struct group_table_entry
{
    double time;  /* most recent update timestamp of a group          */
    int total_wW; /* total number of distinct objects most recently   */
                  /* updated between (T - W * L) and (T - w * L)      */
    struct pair *uplist; /* pointer to a list of pairs (o, t)'s for   */
                         /* objects updated in the previous W         */
                         /* broadcast intervals                       */
} group_table[];
```

Figure 5. Data structure for maintaining group update history.



group_table[1]:

time = 09:08;
total_wW = 1;
uplist-->(A, 09:08)-->(B, 08:12)-->NULL;

Group[1] = {A, B, C, D, E, F}; w = 3; W = 6

Current time = 10:00

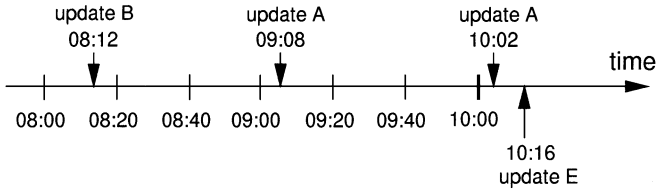
T = 10:00

Figure 6. Example of a `group_table[]`.

As an example, figure 6 shows a snapshot of a group table entry, `group_table[1]`. In figure 6, group 1 contains objects A, B, C, D, E, and F. A broadcast interval is 20 min, $w = 3$ and $W = 6$ in this example. Object A was updated at 09:08 and object B was updated at 08:12. The server keeps track of the update history of the past 6 broadcast intervals. Thus, both objects A and B and their update timestamps are maintained in `group_table[1].uplist`. It also shows that the most recent update to this group was at time 09:08. Since *IR* contains the update history of the past 3 broadcast intervals, the number of distinct objects that were most recently updated between $(T - W \times L)$ and $(T - w \times L)$ is 1. Namely, `group_table[1].tot_wW` at the moment is 1 (the number of distinct objects most recently updated between 08:00 and 09:00 is 1).

For every object updated, the server updates `group_table[]`. Assume that object a is updated at time t and it belongs to group g. The server first sets `group_table[g].time` to t. Next, it checks to see if this object has already been in `group_table[g].uplist`. If yes, its corresponding update time is changed to t. Otherwise, a new pair is created for this object. As a continuing example, figure 7 shows the changes to `group_table[1]` at time 10:19 after object A and E were updated at 10:02 and 10:16, respectively. The update timestamp of object A is changed to the most recent update time of 10:02 and a new pair (E, 10:16) is inserted into the update list pointed to by `group_table[1].uplist`. Of course, the group update time is also changed to 10:16 to reflect the most recent update to the group.

Every time the server broadcasts an invalidation report, it also updates `group_table[]` accordingly. For each group, the server removes the pairs in `group_table[].uplist` that have update times less than $T - W \times L$. This would eliminate any objects that were updated before $T - W \times L$, and limit the amount of history that the server must maintain under GCORE. In addition, the server also updates `group_table[].total_wW`, which keeps the number of distinct objects that were most recently updated between $T - W \times L$



group_table[1]:

time = 10:16;
total_wW = 1;
uplist-->(E, 10:16)-->(A, 10:02)-->(B, 08:12)-->NULL;

Group[1] = {A, B, C, D, E, F}; w = 3; W = 6

Current time = 10:19

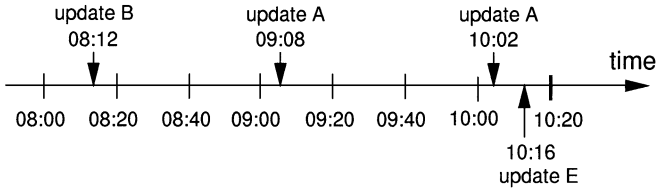
T = 10:00

Figure 7. Changes to `group_table[]` due to an object update.

and $T - w \times L$. The maintenance of `group_table[].total_wW` is for a fast group validity checking. If no object was most recently updated during $T - W \times L$ and $T - w \times L$ and $T_{lb} > (T - W \times L)$, it means if there are any objects in the group updated since T_{lb}, their ids have been included in *IR*. Notice that in GCORE (see figure 4), the cache validity checking is done after the mobile computer first invalidates its cache based on *IR*, the most recent invalidation report. As a result, when a just-woke-up mobile computer sends a group validity checking to the server, it can ensure that those recently updated hot data have already been invalidated in the mobile computer's cache.

Continuing the example of figures 6 and 7, we show the changes to `group_table[1]` at time 10:20 when a new *IR* is broadcasted in figure 8. Object *B* is now discarded from the update history since `group_table[1]` only keeps track of the update history between 08:20 and 10:20 now. Also, `group_table[1].tot_wW` becomes 0 now since object *A* has been again updated at 10:02. Thus, the number of distinct objects most recently updated between 08:20 and 09:20 is zero.

The server checks the validity of a group by examining whether or not all the objects updated since the mobile computer becomes disconnected have been included in the latest invalidation report *IR*. If yes, then the group can be retained by the mobile computer. Otherwise, the entire group is invalid. This can be achieved by first checking if `group_table[].time` $< T_{lb}$. If yes, the group is valid since the group is last updated before the mobile computer becomes disconnected. If not, the server further checks if `group_table[].total_wW` equals to 0 and $T_{lb} > (T - W \times L)$. If yes, this group is also valid since the updated object ids are all included in *IR*. Otherwise (either `group_table[].tot_wW` is greater than 0 or $T_{lb} < (T - W \times L)$), this group is invalid. Based on the example shown in figure 8, if a mobile computer was disconnected at time 09:02 and woke up at 10:18, the server can valid group 1 if the mobile computer checks for



group_table[1]:

time = 10:16;
total_wW = 0;
uplist-->(E, 10:16)-->(A, 10:02)-->NULL;

Group[1] = {A, B, C, D, E, F}; w = 3; W = 6

Current time = 10:20

T = 10:20

Figure 8. Changes to `group_table[]` due to invalidation a new invalidation broadcast.

its validity after 10:20. In this case, the T_{lb} sent to the server by the mobile computer would be 09:00. All the object ids that are updated between 09:20 and 10:20 would be included in the *IR*. As a result, the rest of group 1 can still be retained in the cache since there is no object that was most recently updated between 09:00 and 09:20. Note that object *A* has been further updated at 10:02 and has been included in the *IR*.

3. Simulation model

In order to evaluate the performance of GCORE, an event-driven simulator was developed to model a server and a mobile computer. In the simulation model, we assumed that database objects are only updated in the server by transactions and queries are read-only and are processed in the mobile computer. If the referenced data objects are not cached in the mobile computer, it sends the requested object ids to the server and the server sends back the object data. The effectiveness of caching is measured by the communication bandwidth requirement for query processing. This communication requirement includes the receiving of the broadcast invalidation reports, the uplink communication for validity checking and asking for missed objects, and the downlink communication for sending the data to the mobile computer. High bandwidth requirement means less effective caching and more energy consumption.

A total of D objects are in the database. Of the D objects, β portion of them are hot update set, while $(1-\beta)$ portion of them are cold update set. Data in the hot update set are randomly chosen from the D objects. The number of objects updated by a transaction is uniformly distributed between $U/2$ and $3U/2$ objects, where U is the mean. Of the data objects updated by a transaction, α fraction of them are from the hot update set, and the rest from the cold update set. Update transaction arrival is a Poisson process with rate λ_u.

In order to focus on the cache invalidation effect, we assumed that a cache miss is resulted only from invalidation. It does not result from a query referencing an object that is replaced by another object. In other words, we assumed that all the queries in a mobile computer reference a fixed subset of objects that are initially cached. The cache size is B objects. These B objects are randomly chosen from the D objects in the database. Some of the cached objects may be invalidated because they have been updated by transactions in the server. The objects referenced by a query are randomly chosen from these B objects and the number of objects referenced by a query is uniformly distributed between $Q/2$ and $3Q/2$, where Q is the mean.

The probability of a mobile computer becoming disconnected in the next broadcast interval given that it is active now is denoted as P_{disc}. The length of disconnection is uniformly distributed between $L_{\text{disc}}/2$ and $3L_{\text{disc}}/2$, where L_{disc} is the mean. When a mobile computer is active, query interarrival times are exponentially distributed with mean $1/\lambda_{\text{q}}$ seconds. Queries are batched in QL and are not processed until a mobile computer receives a broadcast invalidation report. We accumulated the communication costs for query processing in a mobile computer for a period of time (50,000 broadcast intervals), and then compute its average bandwidth requirement. For the computation of communication costs, the size of an object id is O_{id} bits and the size of each object is O bytes. The size of a group id is G_{id}. In the simulations, we assumed that $O = 256$ bytes, $O_{\text{id}} = 64$ bits and $G_{\text{id}} = 64$ bits. The size of a timestamp is 256 bits. Notation and its definition for all the simulation parameters are summarized in Table 1. The default values used in the simulations, if not otherwise specified, are included in the parentheses.

Broadcast invalidation requires communication bandwidth every L seconds for all schemes even when the mobile computer is disconnected. However, it consumes energy for a mobile computer only when it is active. For the no-checking scheme, the communication costs for query processing can be rather high both for uplink and downlink costs resulted from cache misses, especially if the disconnection length is longer than the broadcast invalidation window. Note that for all the caching schemes there are uplink costs due to a mobile computer requesting missed objects in its cache. For the other checking schemes, the uplink costs for cache validity checking are added to the total bandwidth costs. For the simple-checking scheme, the uplink costs for validity checking is due to the mobile computer sending the object ids and T_{lb} to the server. For the simple-grouping and GCORE schemes, the uplink costs for validity checking is due to the mobile computer sending the group ids and T_{lb}. Depending on the group size, the uplink costs of the simple-checking scheme can be substantially larger than those of the simple-grouping and GCORE schemes.

4. Simulation results

This section evaluates and compares the effectiveness of various caching schemes described in Section 2. We compute the communication costs due to query processing in the mobile computer, including the costs of receiving the broadcast invalidation reports, the uplink costs of checking the cache validity and requesting missed objects, and the downlink costs of the validity report and the data objects due to cache misses. The bandwidth requirement presented is the average of the communication costs over 50,000 broadcast intervals.

Table 1. System and workload parameters.

Notation	Definition (default values)
D	Database size in the server (100,000 objects)
B	Cache size in a mobile computer (5000 objects)
λ_u	Update transaction arrival rate in the server, Poisson interarrival time distribution (0.01 jobs/s)
λ_q	Query arrival rate in a mobile computer, Poisson interarrival time distribution (0.1 jobs/s)
U	Mean number of objects updated by a transaction (5 objects)
Q	Mean number of objects referenced by a query (20 objects)
$\alpha - \beta$	Reference skew by the update transactions (90%-10%)
G	Group size (200 objects)
w	Number of broadcast intervals for an invalidation report, or broadcast invalidation window (10)
W	Number of broadcast intervals for maintaining the update history by GCORE (60)
L	Length of a broadcast interval (20 s)
O	Object size (256 bytes)
O_{id}	Object id size (64 bits)
G_{id}	Group id size (64 bits)
T	Timestamp for the current broadcast invalidation report
T_{lb}	The timestamp of the latest broadcast invalidation report received by a mobile computer before it went to sleep
P_{disc}	Conditional probability of a disconnection in the next broadcast interval given that a mobile computer is active now (0.2)
L_{disc}	Mean length of a disconnection (200 s)
E_{ratio}	Ratio of energy consumption for sending over receiving messages by a mobile computer

Since the bandwidth requirement would be smaller if the mobile computer is often disconnected (similar to the case of a sleeper in [4]), we plotted the bandwidth requirement for 1000 queries. By doing this, the true effectiveness of caching is manifested in bandwidth requirement. In addition, since the energy consumptions for sending requests uplink and receiving data from downlink may be different, we also show the relative energy consumption between the simple-grouping scheme and GCORE. Unless otherwise specified, the default values for most of the simulations are provided in Table 1.

4.1. Comparisons of all four schemes

The impact of broadcast invalidation window. First, we examine the impact of w, relative to L_{disc}, on the bandwidth requirement. Figure 9 shows the total bandwidth requirement per 1000 queries for the four different schemes. In this experiment, the mobile computer is disconnected relatively infrequently, namely $P_{disc} = 0.1$. (Sensitivity analysis on P_{disc} will be presented next.) The mean disconnection length L_{disc} is 400 s, which is equivalent to 20 broadcast intervals. Since disconnection length is uniformly distributed between $L_{disc}/2$ and $3L_{disc}/2$, for this experiment a mobile computer may power off for a period of

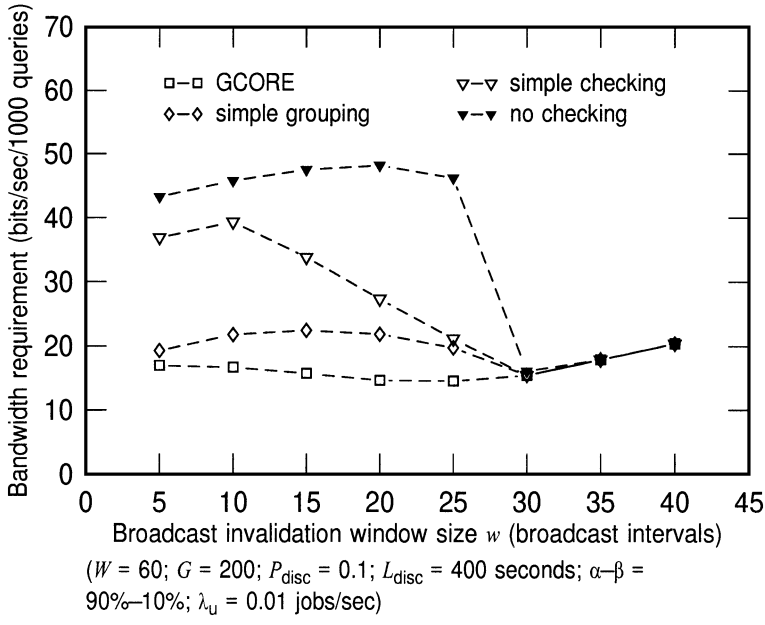


Figure 9. Impact of broadcast invalidation window size.

between 10 and 30 broadcast intervals. For GCORE and the simple-grouping schemes, the group size was 200 objects.

In general, because the broadcast invalidation cost is proportional to the number of objects included in the invalidation report, it increases as w increases for all four schemes. However, if w is greater than the disconnection length, all cached objects can be validated with the invalidation report *IR*, and no checking is ever needed. Therefore, if $w \geq 30$ broadcast intervals, all four schemes require exactly the same communication bandwidth. This is because the maximum disconnection length is 30 broadcast intervals. For the cases where $w < 30$ broadcast intervals, both GCORE and simple grouping require significantly less bandwidth than simple-checking and no-checking schemes. Obviously, validity checking helps retain some of the cached objects and thus substantially reduces bandwidth requirements. Even with a simple-checking scheme where the uplink cost can be substantial, it still can improve the overall cache effectiveness by retaining many cached objects.

The impact of disconnection probability. If the benefits of retaining cached objects are not realized by queries, e.g., a mobile computer may be disconnected most of the time, it may not pay off to perform validity checking, especially for the simple-checking scheme. Here, we examine the impact of P_{disc}, the conditional probability of a mobile computer disconnected in the next broadcast interval given that it is active now. Figure 10 shows the total bandwidth requirement for the four different caching schemes. For this experiment, w was 10 broadcast intervals (or 200 s) and L_{disc} was 400 s. Namely, a disconnection can last for between 200 and 600 s. Thus, the first invalidation report a mobile computer

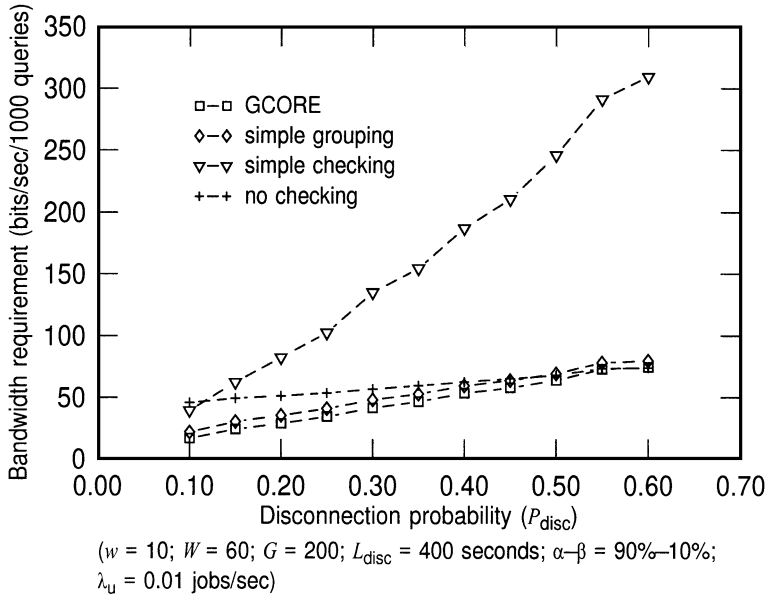


Figure 10. Impact of disconnection probability on the bandwidth requirement for query processing.

receives after a reconnection does not contain enough information for validating cached objects. It must either discard the entire cache, as in the case of no-checking scheme, or check the validity of cached objects with the server.

As indicated from figure 10, as P_{disc} increases the overhead of cache validity checking starts to outweigh the benefit of retaining cached objects. This is particularly true for the case of simple checking, as it needs high uplink bandwidth to send all the object ids. In this figure, simple checking is beneficial only for the case of very small P_{disc}. Figure 10 clearly illustrates the importance of reducing the uplink communication overhead for cache validity checking. Both GCORE and simple grouping have much smaller uplink costs, and as a result can still be better than the no-checking scheme for a larger P_{disc}. But, as a mobile computer is disconnected most of the time, it does not pay to check the validity of cached objects since they are not likely to be reused anyway. In such cases, it is better to just use the no-checking scheme.

The impact of disconnection length. Mobile caching can be very effective if a mobile computer occasionally disconnects only for a small time interval. This is because (1) most of the updated data objects will be included in the upcoming broadcast invalidation report, reducing the needs to do validity checking; and (2) less data objects would be updated by the transactions in a small period of time. Figure 11 shows the total bandwidth requirements for the four different schemes. All four schemes require little bandwidth if L_{disc} is 100 s. In this experiment, the broadcast invalidation interval was 200 s. With $L_{\text{disc}} = 100$ s, a mobile computer only disconnects between 50 and 150 s. Thus, there is no need to do validity checking. However, as the disconnection length increases, the advantage of

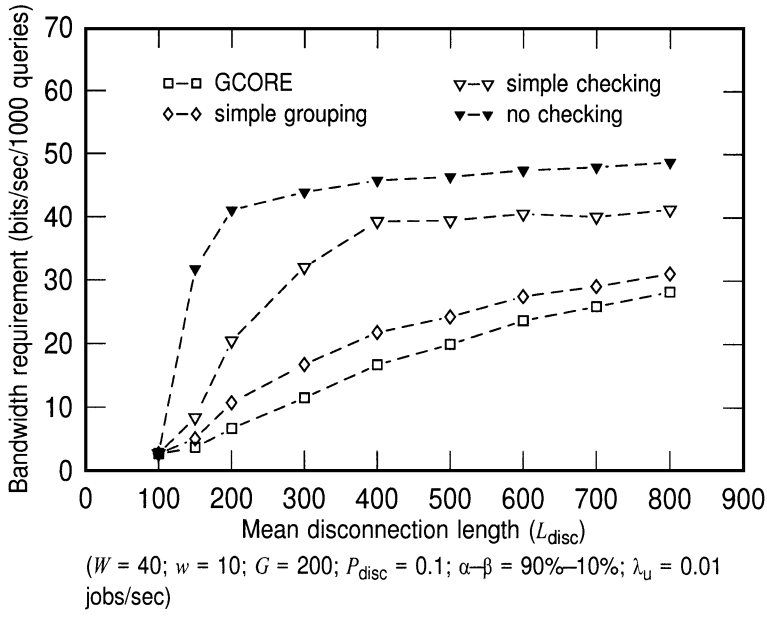


Figure 11. Impact of disconnection length on the bandwidth requirement for query processing.

performing validity checking starts to appear. In figure 11, the performance advantage of GCORE, compared to the no-checking scheme, becomes very substantial once the mean disconnection length exceeds 150 s.

Note that the total bandwidth requirement would have been much higher if a mobile computer sends both the object or group ids and their timestamps for validity checking, especially for the simple-checking scheme where the uplink costs dominate the total costs. In our experiments in this paper, as explained in Section 2.2, only a single T_{lb} is sent with the object ids or group ids. It is important to pay special attention to communication overhead in wireless mobile computing as communication (uplink or downlink) consumes energy as well as wireless bandwidth.

4.2. Comparisons of GCORE with simple grouping

In this section, we compare GCORE with the simple-grouping scheme in a more detailed level. Figure 12 shows both schemes with different update rates λ_u and query sizes Q. For this experiment, the group size was 200 objects. Two different update rates, 0.01 and 0.05 transactions/s, were used. For a given group size, as more objects in a group are updated (with a higher λ_u), it is more important not to invalidate the entire group simply because an object in the group was updated. Thus, the advantage of GCORE over simple grouping becomes more evident as transaction update rate increases. Moreover, the benefits of retaining cached objects can be better realized if more of the retained objects are subsequently

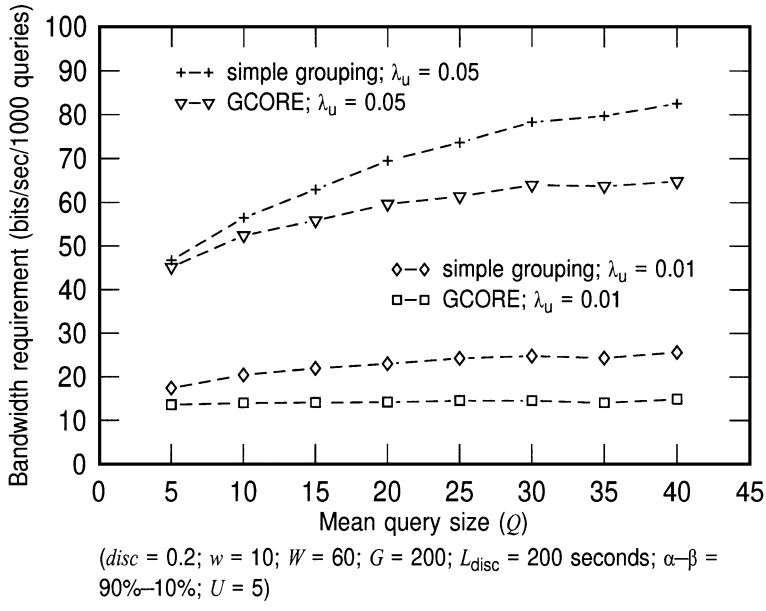


Figure 12. Impact of query size and transaction rate on GCORE and simple grouping.

accessed by queries. This is clearly illustrated by the increasing gap between GCORE and the simple-grouping scheme as Q increases, especially for the cases of $\lambda_u = 0.05$.

The impact of group size. For both simple grouping and GCORE, group size is an important design parameter. With a larger group size, less uplink bandwidth is needed since there are smaller number of groups. However, it becomes more likely that the entire group or most of it may be invalidated since more objects in the group are likely to be updated. Figure 13 shows the bandwidth requirements of GCORE and simple grouping for various group sizes and update rates. In general, as group size increases, the uplink cost decreases but the downlink cost increases. Thus, the total bandwidth first decreases and then increases as group size increases. For the cases with low update rates (such as those with $\lambda_u = 0.01$ in figure 13), the advantage of GCORE over simple grouping increases as group size increases. However, it may not be true for the cases with high update rates. This is because GCORE can retain the cold update set of a group only if the updated objects of the group are all captured in the most recent invalidation report. Namely, there is no distinct object that is most recently updated between time $T - W \times L$ and $T - w \times L$. If a group contains a large number of objects and the update rate is high, then it is less likely that all the updated objects will be captured in *IR*. As a result, GCORE would have less distinctive advantage over simple grouping.

The impact of α and β. The benefits of retaining cold update set by GCORE derive from the fact that the hot update set constitutes a small portion of the data objects accessed by a mobile client. To see the impacts of the hot set sizes on the performance of GCORE and simple grouping, we varied β from 1% to 25%. Figure 14 shows the downlink bandwidth

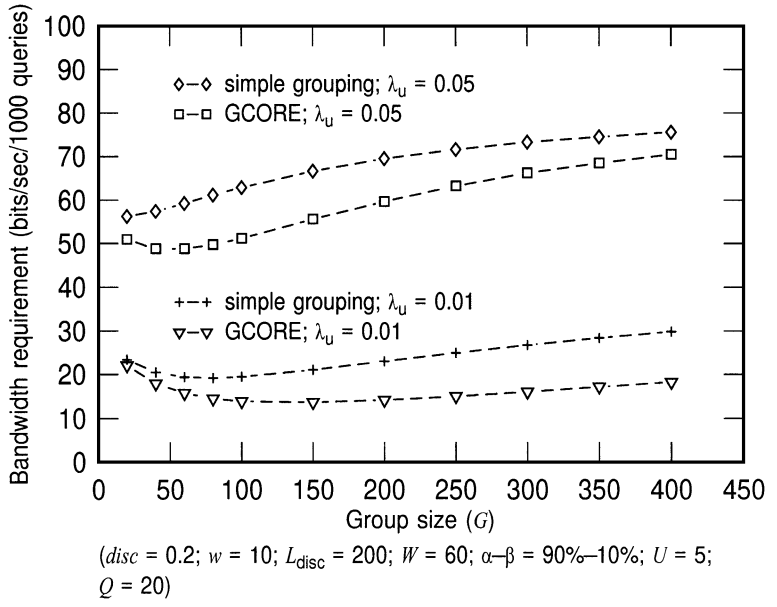


Figure 13. Impact of group size on bandwidth.

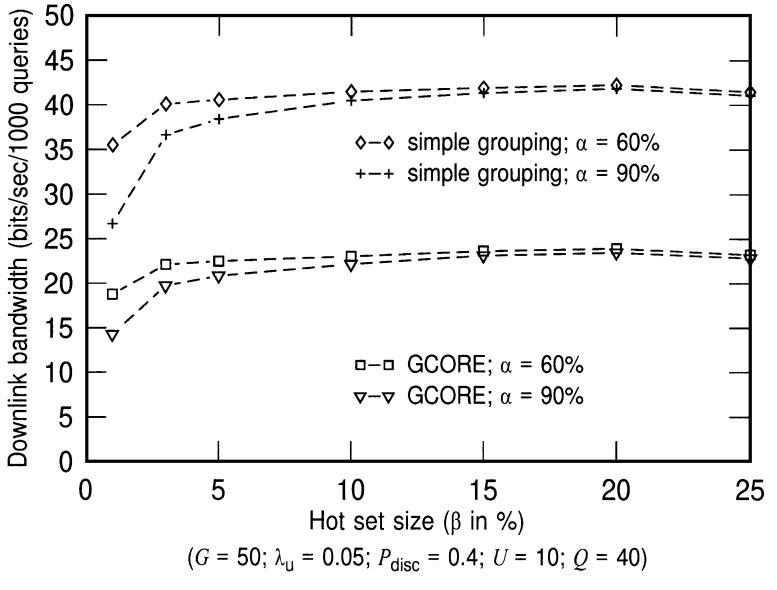


Figure 14. Impact of hot set sizes on downlink bandwidth.

of these two schemes for the cases of α being 60% and 90%. In general, the smaller β is, the smaller the downlink bandwidth requires. However, there is little increase in downlink bandwidth as β goes beyond 10%. For all the cases, GCORE is better than the simple-grouping scheme by a substantial margin because of retention of the cold update set. Moreover, as more transactions are updating data objects in the hot set, less objects in the cold set will be changed. Thus, the downlink bandwidth is smaller for the cases of $\alpha = 90\%$ for both schemes. The benefits of a higher α is more noticeable for the simple-grouping scheme than for GCORE, especially when β is very small (see the cases of $\alpha = 90\%$ and $\beta = 1\%$). This is because an entire group is less likely to be invalidated if transaction updates are highly concentrated on a very small hot set.

The impact of energy consumption ratio. Sending requests uplink by a mobile computer consumes not only wireless bandwidth but also battery energy. So does receiving data from the server. However, the energy requirement for a mobile computer for uplink communication and receiving data from downlink may be quite different. Uplink communication may consume significantly more energy because it requires the mobile computer to send messages. Here we compare simple grouping and GCORE in energy consumption for query processing. We assumed that 1 unit of energy is needed by the mobile computer for 1 bit/s of downlink communication. For sensitivity analysis, we examined cases of energy consumption ratio for uplink over downlink being 1, 5 and 10. Figure 15 shows the energy consumption for both schemes. For this experiment, $\lambda_u = 0.01$ was used. For the three different ratios, GCORE is more energy efficient than simple grouping. More importantly, as the ratio increases, the optimal group size becomes larger. This is because larger group

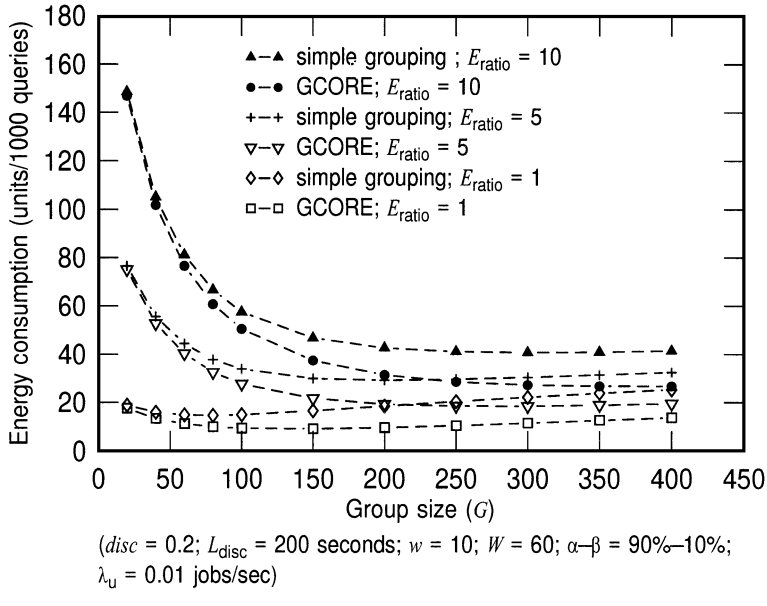


Figure 15. Impact of group size on energy consumption.

size means less number of group ids to be sent uplink. If uplink communication consumes much more energy than downlink, then a larger group size should be used. Note that if the uplink/downlink energy ratio is greater than 1, then the optimal group size for total bandwidth requirement (see figure 13) is less than that for energy consumption.

5. Summary

In this paper, we studied the issues of efficiently caching frequently accessed data in a wireless mobile computer. Because of the limited battery power, a mobile computer may be forced to power off for saving energy. Once disconnected, it may miss some of the cache invalidation reports broadcasted by the server. As a result, when it wakes up again, the entire cache may have to be discarded, significantly reducing the benefits of caching. We presented an energy-efficient cache invalidation scheme, called grouping with cold update-set retention GCORE. It allows a mobile computer to disconnect to save energy, but still retains most of the caching benefits by salvaging most of the cached objects if they are still valid. As a result, less energy is consumed due to communication between the mobile computer and the server. It uses a simple data structure to facilitate the dynamic exclusion of recently updated objects (likely to belong to the hot update set) from a group so that the rest of the group (likely to belong to the cold update set) can be retained in the cache. Upon waking up, a mobile computer checks its cache validity with the server at a group level to save uplink costs.

Simulations were conducted to evaluate the performance of GCORE. We compared GCORE with a no-checking, a simple-checking and a simple-grouping scheme. The results show that, compared with no checking and simple checking, both simple grouping and GCORE require much less bandwidth for processing queries, particularly if a mobile computer is occasionally disconnected for a long period of time and most of the data objects are infrequently updated. Lower bandwidth requirement consumes less energy and thus is more energy efficient. Furthermore, GCORE is more energy efficient than the simple-grouping scheme because it can retain more objects in a mobile cache after a reconnection.

Notes

1. A preliminary version of this paper was presented in the 12th International Conference on Data Engineering.
2. This work was done when Ming-Syan Chen was with IBM Watson Research Center. He is now with the Dept. of Electrical Engineering, National Taiwan University, Taipei, Taiwan.

References

1. A. Acharya and B.R. Badrinath, “Checkpointing distributed applications on mobile computers,” in Proc. of Int. Conf. on Parallel and Distributed Information Systems, 1994, pp. 73–80.
2. R. Alonso and H. Korth, “Database system issues in nomadic computing,” in Proc. of ACM SIGMOD Int. Conf. on Management of Data, 1993, pp. 388–392.
3. B.R. Badrinath and T. Imielinski, “Replication and mobility,” in Proc. of the 2nd Workshop on the Management of Replicated Data, 1992.

4. D. Barbara and T. Imielinski, “Sleepers and workaholics: Caching in mobile distributed environments,” in Proc. of ACM SIGMOD Int. Conf. on Management of Data, 1994, pp. 1–12.
5. Y. Huang, P. Sistla, and O. Wolfson, “Data replication for mobile computers,” in Proc. of ACM SIGMOD Int. Conf. on Management of Data, 1994, pp. 13–24.
6. Y. Huang and O. Wolfson, “Object allocation in distributed databases and mobile computers,” in Proc. of Int. Conf. on Data Engineering, 1994, pp. 20–29.
7. T. Imielinski and B.R. Badrinath, “Querying in highly mobile distributed environments,” in Proc. of Very Large Data Bases, 1992, pp. 41–52.
8. T. Imielinski and B.R. Badrinath, “Mobile wireless computing,” Communications of the ACM, vol. 37, no. 10, pp. 18–28, Oct. 1994.
9. T. Imielinski, S. Viswanathan, and B.R. Badrinath, “Energy efficient indexing on air,” in Proc. of ACM SIGMOD Int. Conf. on Management of Data, 1994, pp. 25–36.
10. R.H. Katz, “Adaptation and mobility in wireless information systems,” IEEE Personal Communications, pp. 6–17, First Quarter 1994.
11. P. Krishna, N.H. Vaidya, and D.K. Pradhan, “Location management in distributed mobile environments,” in Proc. of Int. Conf. on Parallel and Distributed Information Systems, 1994, pp. 81–88.