SAIU: An Efficient Cache Replacement Policy for Wireless On-demand Broadcasts

Jianliang Xu Qinglong Hu* Dik Lun Lee
Department of Computer Science
HK University of Science and Technology
Clear Water Bay, Hong Kong
{xujl,qinglong,dlee}@cs.ust.hk

Wang-Chien Lee
GTE Laboratories Incorporated
40 Sylvan Road
Waltham, MA 02451, USA
wlee@gte.com

ABSTRACT

In a wireless communication environment, on-demand broadcasting is an attractive information delivery method, while data caching is an important technique used to improve system performance. However, most of the wireless data caching techniques in the literature were designed for push-based broadcast systems. Due to variable item sizes, data updates and frequent client disconnections, the design of client cache management in an on-demand broadcast system becomes a challenge. In this paper, we propose an efficient gain-based cache replacement policy, *SAIU*, for on-demand broadcasts. In *SAIU*, the influence of data size, data retrieval delay, access probability and update frequency is considered together. Simulation-based performance evaluation shows that the proposed policy substantially outperforms the existing strategies.

1. INTRODUCTION

With continuous deployment of satellites and cellular systems, mobile computing and communications are becoming an important part of our life. Broadcast-based information dissemination, for its capability to scale up to an arbitrary number of users, has become a hot research topic in the academia [2, 19]. However, these periodic, push-based, wireless data broadcasts usually are not tailored to a particular user's need. Thus, a user's need may have to be sacrificed to satisfy the needs of the majority. Further, push-based broadcasts are not scalable to the database size and react slowly to workload changes. To alleviate these problems, many recent research studies on wireless data dissemination (e.g., [3, 5, 10, 17]) have proposed to use the systems that support both of broadcast and on-demand services. Thus, in this paper, we call the systems that support and utilize both of broadcast and on-demand services the *wireless on-demand broadcast systems*.

A wireless on-demand broadcast system supports both of broadcast and on-demand services through a broadcast channel and a low-bandwidth uplink channel. Users send on-demand requests through the uplink to the server. In response, the server disseminates the requested data to the clients through the shared broadcast channel. The issue of *on-demand broadcast scheduling* in the context of wireless on-demand broadcast systems has been extensively studied in the past few years [3, 5, 7, 17]. However, wireless data caching, as an important technique used to improve system performance [2, 6, 11], has not been exploited in this context. This paper investigates the issue of cache replacement for wireless on-demand broadcasts.

Cache replacement policies for delay sensitive objects with variable sizes have been investigated in the recent years [1, 4, 14, 16]. However, these studies addressed the cache management issue specific to web proxy caching or query result caching in a data warehousing environment.

Cache replacement policies for the wireless broadcast environment were studied only in push-based broadcasts [2, 13, 18]. Furthermore, these previous studies are based on unrealistic assumptions, such as fixed data sizes, no updates, and no disconnections. In real life applications, data usually have different sizes. For example, the document size on the web generally ranges from several kilo-bytes to several mega-bytes. Some information, such as road traffic and stock prices, may constantly change over time. Also, mobile clients may frequently go to disconnected states voluntarily (to save power) or due to failure. These factors make the design of client cache management a challenge in wireless on-demand broadcasts.

In this paper, an efficient gain-based cache replacement policy, *SAIU*, is proposed for wireless on-demand broadcasts. In *SAIU*, the influence of data item size, data retrieval delay, data access probability and update frequency is considered, and a gain function which integrates these factors together is developed. To evaluate the performance of the proposed cache replacement policy, a series of simulation experiments is conducted to compare *SAIU* with other traditional cache replacement policies, i.e., *LRU* and *LRU-MIN* [1]. The simulation results show that the proposed policy *SAIU* substantially outperforms the existing policies.

Our work differs from the previous work mainly in two aspects: 1) cache replacement is studied in the context of on-demand broadcasts; 2) restrictions on data size, data update and disconnection, made in most of the previous work, were

*The author is now with IBM, Database Technology Institute, San Jose California, USA.

CIKM 2000, McLean, VA USA

relieved to make the proposed scheme practical and realistic. To facilitate our study, we make the following assumptions in this paper: data are updated only at the server-side; and strong data consistency is required by mobile clients.

The rest of this paper is organized as follows. Section 2 describes the performance metrics used in this study and introduces several existing techniques related to the cache management in on-demand broadcasts. The cache replacement policy, *SAIU*, and its implementation issues are described in Section 3. Section 4 introduces the simulation model for performance evaluation. The simulation results are presented in Section 5. Finally, Section 6 concludes the paper.

2. BACKGROUND

Before the study for cache replacement management can be carried out, the related important issues, such as performance metrics, broadcast schedule, and invalidation propagation, should be addressed first. In the following, we give a brief background on the above mentioned issues and describe the metrics and techniques adopted in our study.

2.1 Performance Metrics

In the traditional cache management, cached items are *page*s or *block*s, which are assumed to have the same size. Further, cache miss penalties for all the cached items are the same. Thus, the *cache hit ratio* metric is consistent with the *access latency* metric, i.e., the higher the hit ratio, the shorter the overall access latency. Cache hit ratio is often used to measure the effectiveness of cache replacement policies in the traditional cache management.

For cached items with different sizes, obviously cache hit ratio is no longer a reliable performance metric. In the previous work of web proxy caching, *byte hit ratio*, which is the ratio of the total number of bytes hit to the total number of bytes requested, has been introduced to evaluate the cache performance. In an on-demand broadcast system, however, due to the data broadcast scheduling, cache miss penalties vary for different data items. Thus, byte hit ratio cannot reflect the overall system performance either. As a result, two metrics can be used to evaluate the performance of cache management:

- **Access latency:** the elapsed time when a request is submitted to the time when the request is serviced.
- **Stretch [3]:** the ratio of the access latency of a request to its *service time*, where service time is defined as the ratio of the requested item's size to the broadcast bandwidth.

Access latency alone is not a fair performance measure, since it does not count the difference in data size/service time. Generally, for a smaller item, which has a shorter service time, shorter access latency is expected by clients. Therefore, compared with access latency, stretch is a more reasonable performance metric for items with variable sizes. Thus, we employ stretch as the major metric in this study.

2.2 Scheduling Algorithms

Typically, a caching technique is effective in reducing access latency and improving data availability. Data caching may also have a great impact on broadcast schedules, because it reduces client access requests to the server and thus changes the clients' access patterns. On the other hand, caching strategies are also affected by broadcast scheduling algorithms, because different scheduling algorithms may result in different data retrieval delays (i.e. cache miss penalties) for the same data request. Thus, it is mandated to consider broadcast schedule in the design of a caching scheme.

The broadcast scheduling algorithm is vital to the system performance in on-demand broadcasts. As pointed out in [5], a good on-line scheduling algorithm should balance individual and overall performance and can scale well to data set size, client population, and broadcast bandwidth. To this end, several efficient on-demand broadcast scheduling algorithms, such as Longest Wait First (*LWF*), Longest Total Stretch First (*LTSF*) and *RxW*, have been proposed in the literature [3, 5, 19]. In our simulation study, *LTSF* is employed as the scheduling algorithm for on-demand broadcasts, since it shows good performance when client data caching is not considered [3]. In *LTSF*, the data item with the largest total current stretches is chosen for next broadcast, where the current stretch of a pending request is the ratio of the time the request has been in the system to its service time.

2.3 Invalidation Propagation

To maintain cache consistency, periodically propagating *invalidation report* (hereafter denoted as *IR*) is an efficient method for notifying mobile clients of recent data updates at the server [6, 12]. An *IR* consists of the server's updating history up to w broadcast intervals. Every active client listens to the broadcast channel for *IR* and invalidates its cache content accordingly. Under a strong data consistency requirement, a mobile client has to obtain an *effective IR* from the broadcast channel to do validation checking before answering a query from its local cache.

Among the various *IR*-based cache invalidation approaches, *adaptive cache invalidation* algorithms work well under various system workloads [9]. In adaptive cache invalidation algorithms, the updating history window w and content organization of the next *IR* are dynamically decided based on the system workload. In the simulation, we employ the *adaptive invalidation report with adjusting window* (*AAW_AT*) scheme for cache invalidation, because it demonstrates superior performance over other algorithms [9].

3. CACHE REPLACEMENT ALGORITHM

Cache replacement is an important issue to be tackled for cache management in an on-demand broadcast system. In the traditional cache management methods, access probability is the primary factor used to determine a cache replacement policy. In an on-demand broadcast environment, however, three additional factors, namely *data retrieval delay*[1], *data update frequency* and *data item size*, need to be considered in the design of cache replacement policies. In the following, we first provide a design guideline for cache replacement policies based on some of our observations. Then, we introduce a gain-based cache replacement policy, *SAIU*. Finally, we address some of the implementation issues.

3.1 Design Guideline

Before introducing the proposed cache replacement policy, we first discuss the intuitive observations we made:

[1]In this paper, the data retrieval delay is referred to as the data access latency through the on-demand broadcast service for a cache miss. In other words, the data retrieval delay is the cache miss penalty for a data item.

Observation 1: For two cached data items with the same data size, retrieval delay, and update frequency, the one with the lower access probability should be chosen for replacement.
Observation 2: For two cached data items with the same data size, access probability, and update frequency, the one with the lower miss penalty (i.e., shorter data retrieval delay) should be chosen for replacement.
Observation 3: For two data items with the same size, the same retrieval delay and the same access probability, the one with a higher update frequency should be chosen for replacement.
Observation 4: For two data items with different data sizes but the same retrieval delay, access probability, and update frequency, the one with larger data size should be chosen for replacement. The reason is two-fold: i) the larger data item has a longer service time, thus a lower miss penalty in terms of stretch; ii) if the larger data item is removed, the cache can accommodate more data items and satisfy more access requests. Consequently, cache hit ratio and hence the overall access latency could be improved.

Based on the above observations, we provide a design guideline for cache replacement policies:

Design Guidance: *A cache replacement policy should choose the data items with low access probability, short retrieval delay, high update frequency and large data size for replacement.*

3.2 The *SAIU* Replacement Policy

In this subsection, we propose a gain-based cache replacement policy, namely *Stretch · Access-rate · Inverse Update-frequency* (*SAIU*), for on-demand broadcasts. To facilitate the discussion, we define the following notations:

- D: number of data items in the database.
- L_i: data retrieval delay on the broadcast channel for item i, $i = 1, \ldots, D$.
- A_i[2]: access rate for data item i, $i = 1, \ldots, D$.
- U_i: update frequency for data item i, $i = 1, \ldots, D$.
- s_i: size for data item i, $i = 1, \ldots, D$.
- B: bandwidth of the broadcast channel.
- S_i: stretch on the broadcast channel for item i, $S_i = \frac{L_i}{s_i/B}, i = 1, \ldots, D$.

Following the design guideline proposed in the last subsection, we define a gain function for each item i:

$$gain(i) = \frac{L_i \cdot A_i}{s_i \cdot U_i} \quad (1)$$

If we multiply Equation (1) by the broadcast bandwidth (which is a constant B), it becomes $gain(i) = \frac{L_i \cdot A_i}{s_i/B \cdot U_i} = \frac{S_i \cdot A_i}{U_i}$. Thus we call it *SAIU*. The proposed policy *SAIU* works as follows. To find space for the kth accessed data item, the algorithm removes the cached data item i with the minimum $gain(i)$ value until the free space is sufficient to accommodate the incoming item.

3.3 Implementation Issues

In this subsection, we address three critical implementation issues, namely *heap management*, *estimate of running parameters*, and *maintenance of cached item attributes*.

[2]Note that A_i is for one client only.

3.3.1 Heap Management

A (binary) min-heap data structure is used to implement the *SAIU* policy. The key field for the heap is the $gain(i)$ value for each cached data item i. When the events of cache replacement occur, the root item of the heap is deleted. This operation is repeated until sufficient space is obtained for the incoming data item. Let N denote the number of cached items and M the victim set size. Every deletion operation has a complexity of $O(logN)$. An insertion operation also has a $O(logN)$ complexity. Thus, the time complexity for every cache replacement operation is $O(MlogN)$. In addition, when an item's *gain* value is updated, its position in the heap needs to be adjusted. The time complexity for every adjustment operation is $O(logN)$. The practical complexity of *SAIU* is further investigated by simulation in Section 5.5.

3.3.2 Estimate of Running Parameters

Several parameters are involved in computation of the $gain(i)$ function. Among these parameters, s_i can be obtained when item i arrives. In most cases, U_i, L_i, and A_i, are not available to the clients. Thus, we need to employ some estimate methods to obtain these data.

A well-known exponential aging method is used to estimate U_i, L_i, and A_i [16]. Initially, U_i and L_i are set to 0. When a new update on item i arrives, U_i is updated according to the following formula:

$$U_i = \alpha_u/(t^c - t_i^{lu}) + (1 - \alpha_u) \cdot U_i \quad (2)$$

where t^c is the current time, t_i^{lu} is the timestamp of the last update on item i, and α_u is a factor to weight the importance of the most recent update with that of the past updates. The larger the α_u, the more important the recent updates.

Similarly, when a query for item i is answered by the server, L_i is re-evaluated as follows:

$$L_i = \alpha_s \cdot (t^c - t_i^{qt}) + (1 - \alpha_s) \cdot L_i \quad (3)$$

where t_i^{qt} is the query time and α_s is a weight factor for the running L_i estimate.

U_i and L_i, estimated at the server-side, are piggybacked to the clients when data item i is delivered; t_i^{lu} is also piggybacked so that the client can continue to update U_i based on the received *IR*s. The client caches the data item as well as its U_i, t_i^{lu} and L_i values. The maintenance of these parameters (along with some other parameters) will be discussed in the next subsection.

Different clients may have different access patterns, while some of their data accesses are answered by cache. It is difficult for the server to know the real access pattern for each client. Consequently, access arrival rate A_i is estimated at the client-side. The following method is used to estimate A_i. When a query for item i is issued by a client, if there is no information about item i maintained in the cache, A_i is set to 0 and t_i^{la} is set to the current time; otherwise, A_i is updated using the following formula:

$$A_i = \alpha_a/(t^c - t_i^{la}) + (1 - \alpha_a) \cdot A_i \quad (4)$$

where t_i^{la} is the timestamp of the last access to item i, α_a is a constant factor used to weight the most recent access for the running access frequency estimate.

3.3.3 Maintenance of Cached Item Attributes

To realize the *SAIU* policy, a total of six parameters, namely s_i, U_i, t_i^{lu}, L_i, A_i, and t_i^{la}, are needed for each cached data item. Thus, we refer them as the *cached item attributes* (or simply call them *attributes*). To obtain these attributes efficiently, one may store the attributes for all data items in the client cache. Obviously, this strategy does not scale up to the database size. In the other extreme, one may retain the attributes only for the cached data items. However, this will cause the so-called "starvation" problem, as observed in [14, 16], which states that a newly cached data item i could be selected as the first few candidates for replacement since it has only incomplete information[3]. If the cached item attributes are evicted from the cache together with the data item i, then upon re-accessing item i, these attributes must be collected again from scratch. Consequently, item i is likely to be evicted again.

Similar to [14], we employ a heuristic to maintain the cached item attributes. The attributes for the currently cached data items are kept in the cache. Let $GAIN_{min}$ be the minimum *gain* value for the currently cached data items. For those data items that are not cached, we only retain the attributes for an item j whose $gain(j)$ is larger than $GAIN_{min}$. This heuristic is adaptive to the cache size. If the cache size is large, it can accommodate more data items and hence a relatively small $GAIN_{min}$ value. As a result, attributes for more data items can be retained in the cache. On the other hand, if the cache size is small, less data items are contained and the $GAIN_{min}$ value is relatively large, thus less attributes are kept.

4. SIMULATION MODEL

This section describes the simulation model used for performance evaluation. The simulation model is implemented using *CSIM* [15]. A single cell environment is considered. The model consists of a single server and $NumClient$ clients[4], where the clients continuously query data items from the server and the server continuously broadcasts data items to the clients based on the on-demand requests.

The default system parameter settings are given in Table 1. The database is a collection of $DbSize$ data items and is partitioned into disjointed *region*s, each with $RgnSize$ items. Data items have sizes varying from s_{min} to s_{max}. The following two types of size distributions are considered [8]:

- **Increasing Distribution (INCRT)**: $size_i = s_{min} + \frac{(i-1)(s_{max}-s_{min}+1)}{DbSize}$, $i = 1, \ldots, DbSize$.
- **Decreasing Distribution (DECRT)**: $size_i = s_{max} - \frac{(i-1)(s_{max}-s_{min}+1)}{DbSize}$, $i = 1, \ldots, DbSize$.

Combined with the skewed access pattern, *INCRT* and *DECRT* represent clients' favor of frequently querying smaller items and larger items, respectively (see details in Section 4.1). The broadcast channel has a bandwidth of $BcastBW$. The average delay for uplink request messages is very small. Furthermore, compared with the broadcast access latency, the setup time of an uplink request message is negligible. Similar to the previous work [3, 5], we do not consider the setup time and service time for uplink messages in the simulation. *IR*s are broadcast periodically on the broadcast channel with an interval of $BcastInt$.

[3]It may incorrectly produce a relatively smaller *gain* value.
[4]Each client could be further treated as an aggregate of clients with lower access rates.

Parameter	Setting	Meaning
$NumClient$	100	# of clients in a cell
$DbSize$	2,000 items	# of items in database
$RgnSize$	20 items	# of items per db region
s_{min}	2 KB	Minimum data item size
s_{max}	2,000 KB	Maximum data item size
$BcastBW$	1,000 Kbps	Broadcast bandwidth
$BcastInt$	2.0 s	IR broadcast interval
$ConMsgSize$	64 bytes	Control msg. size in IR

Table 1: Default System Parameter Settings

Parameter	Setting	Meaning
$ThinkTime$	10 s	Mean think time
p	0.1	Disconnection probability
$CacheRatio$	5%	Ratio of cache size to db size
$ParaSize$	4 bytes	Size of cached parameter
$DiscTime$	200 s	Mean disconnection time
θ	0.95	Zipf distribution parameter

Table 2: Default Client Parameter Settings

4.1 Client Model

Each client is simulated by a process and runs a continuous loop that generates a stream of queries. After the current query is finished, the client waits for a period of $ThinkTime$ and then makes the next query request. The model assumes, when a client is in the *thinking* state, it has a probability of p to enter the disconnected state every *IR* broadcast interval. The time that a client is in a disconnected state follows an exponential distribution with a mean of $DiscTime$. Each client has a cache of $CacheSize$, which is a $CacheRatio$ ratio of $DbSize$. Cache size is defined as $CacheSize = \frac{s_{max}+s_{min}-1}{2} \times DbSize \times CacheRatio$. In order to maintain fairness to different caching schemes, the $CacheSize$ parameter includes both the space needed for storing item attributes and the space available for storing data. Each cached parameter occupies $ParaSize$ bytes.

The client access pattern follows a $Zipf$ distribution (with parameter θ) [20]. The data items are sorted such that the item 0 is the most frequently accessed, and the item $DbSize - 1$ is the least frequently accessed. In other words, with an *INCRT* sized setting the clients access the smallest item most frequently, while with a *DECRT* sized setting the clients access the largest item most frequently. $Zipf$ distributions are frequently used to model non-uniform access patterns. The probability of accessing any item within a database region is uniform and the $Zipf$ distribution is applied to these regions. Table 2 summarizes the default client parameter settings.

4.2 Server Model

Parameter	Setting	Meaning
$UpdateTime$	100 s	Update inter-arrive time
$U\text{-}Cold/U\text{-}Hot$	80/20	Cold/Hot update pattern

Table 3: Default Server Parameter Settings

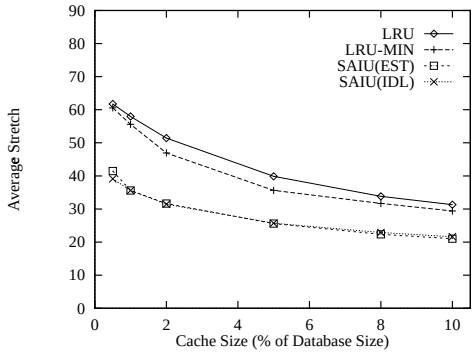

Figure 1: Stretch Performance of Various Cache Sizes (INCRT)

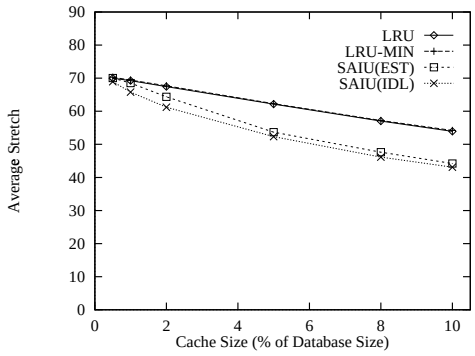

Figure 2: Stretch Performance of Various Cache Sizes (DECRT)

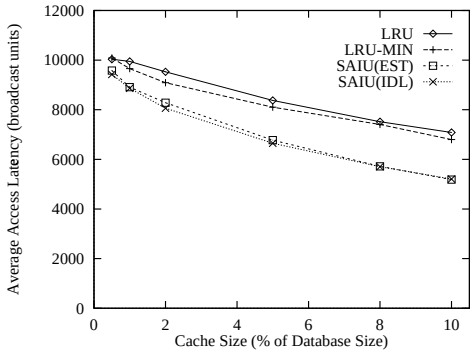

Figure 3: Access Latency of Various Cache Sizes (INCRT)

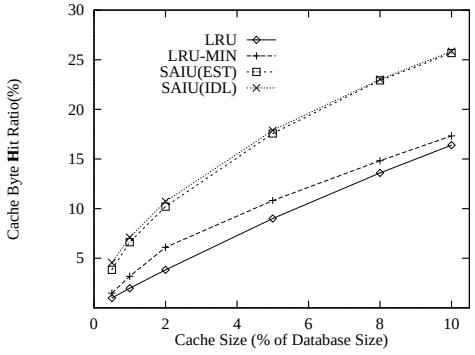

Figure 4: Cache Byte Hit Ratio of Various Cache Sizes (INCRT)

The server is modeled by a single process. Table 3 gives the server parameter settings. The clients' requests are buffered at the server, if necessary, and an infinite queue buffer is assumed. After broadcasting the current item, the server chooses an outstanding request from the buffer as the next candidate, according to the scheduling algorithm used. Overheads of scheduling and request processing at the server are not considered in the model.

Data updates are generated by the server process with an exponentially distributed update inter-arrival time with a mean of $UpdateTime$. A $Cold/Hot$ update pattern is assumed in the simulation model. Specifically, the uniform distribution is applied to all the database regions. Within a region, $U\text{-}Cold\%$ of the updates are for the first $U\text{-}Hot\%$ items and $U\text{-}Hot\%$ of the updates are for the rest. For example, we assume in our experiments that, within a region, 80% of the updates occur on the first 20% data items (i.e., update-hot items) and 20% of the updates occur on the rest 80% of data items (i.e., update-cold items).

5. PERFORMANCE EVALUATION

This section explores the performance of the proposed cache replacement policy by simulation. The primary performance metric employed in this study is *average stretch*. *Average access latency* and *byte hit ratio* are also provided to show some system performance wherever appropriate. *Broadcast unit*, which is defined as the ratio of the minimum item size to the broadcast bandwidth, is used to measure access latencies. In the experiment, we employ *AAW_AT* [9] to propagate invalidation information and use *LTSF* [3] for on-demand broadcast scheduling.

The results are obtained when the system is in a stable state, i.e., for each client at least 5000 queries after its cache is full, so that the warm-up effects of the client cache and the broadcast channel are eliminated. For the exponential aging estimate method, we set $\alpha_a = \alpha_s = \alpha_u = 0.25$ [2, 16]. Unless it is mentioned explicitly, the broadcast bandwidth is fully utilized.

As stated before, several factors determine the overall cache performance in a wireless on-demand broadcast system. Changing the cache size, broadcast bandwidth, item size ratio and update rate can have different impacts on the performance. In what follows, we first evaluate the robustness of the proposed cache replacement policy under various workloads. Then, we analyze the time complexity for the *SAIU* policy. The *LRU* and *LRU-MIN* [1] policies are also included for comparisons in the experiments. Due to space limitation, we show only part of the experimental results.

5.1 Experiment #1: Impact of the Cache Size

In this subsection, we vary the cache size to investigate the performance of the cache replacement schemes. The simulation results are shown in Figure 1 through Figure 4. To estimate data retrieval delays, access and update frequencies, *SAIU(EST)* uses the exponential aging method (Equations (2), (3) and (4)). *SAIU(IDL)* is assumed to have perfect knowledge about data access and update frequencies.

As shown in Figure 1 and Figure 2, the *SAIU* outperforms the *LRU* and *LRU-MIN* policies significantly in terms

of stretch. Specifically, in an *INCRT* sized setting (Figure 1), the average improvement of *SAIU* over *LRU* and *LRU-MIN* is 35.5% and 31.1%, respectively; in a *DECRT* sized setting (Figure 2), the average improvement of *SAIU* over *LRU* and *LRU-MIN* is 12.1%. The improvement for *INCRT* is greater than that for *DECRT*. This is mainly because *SAIU* can cache more frequently accessed items in the former setting, whereas *SAIU* has to balance between caching more items and caching more frequently accessed items in the latter setting. Furthermore, in the latter setting, when the cache size is less than 2% of the database size (see Figure 2), *SAIU* does not improve the performance much since the cache can accommodate only several data items.

The performance of access latency and cache byte hit ratio for an *INCRT* sized setting is shown in Figure 3 and Figure 4, respectively. We can see that *SAIU* also performs much better than *LRU* and *LRU-MIN* in terms of these two metrics. Similar results are obtained for a *DECRT* sized setting.

Due to estimation of the running parameters, in most cases, the performance of *SAIU(EST)* is slightly worse than that of *SAIU(IDL)*, but they are very close. Similar results are observed for other parameter settings. In the subsequent subsections, we plot *SAIU(EST)* only to improve clarity of the figures.

5.2 Experiment #2: Impact of the Broadcast Bandwidth

As pointed out in Section 3, cache replacement policies should take data retrieval delay into consideration. In this subsection, the cache replacement algorithms are evaluated under various bandwidth. The average stretches for *INCRT* and *DECRT* are shown in Figure 5 and Figure 6, respectively. Similar curves are obtained for the access latency performance. We do not plot them due to space limitation.

As the bandwidth increases, the performance for all the strategies becomes better, as expected. When the bandwidth is increased from 200 Kbps to 1500 Kbps, the improvement of *SAIU* over *LRU* becomes greater (from 32.0% to 33.8% for *INCRT* and from 10.8% to 15.7% for *DECRT*). This implies that *SAIU* can utilize the bandwidth more effectively. When the bandwidth reaches 2000 Kbps, for *INCRT* the improvement of *SAIU* over *LRU* is only 10.8%. This could be explained as follows. In this case, it is observed that the broadcast channel is under a light utilization. Therefore, the data retrieval latency and hence the stretch (i.e., cache miss penalty) is very small. On the other hand, for a cache hit, under a strong data consistency assumption, a client has to retrieve an effective *IR* before answering a query locally. Because we fix the *IR* broadcast interval (i.e. 2 seconds) in all the experiments, in this case the waiting time for an *IR* becomes significant for a cache hit. This becomes more significant in terms of stretch since *SAIU* favors smaller data items for caching. As a result, *SAIU*, though with a much higher hit ratio[5], cannot achieve a greater improvement over *LRU* and *LRU-MIN* on the overall stretch and access latency performance in this case.

5.3 Experiment #3: Influence of the Item Size Ratio

[5]To save space, the cache hit ratio performance is not shown here.

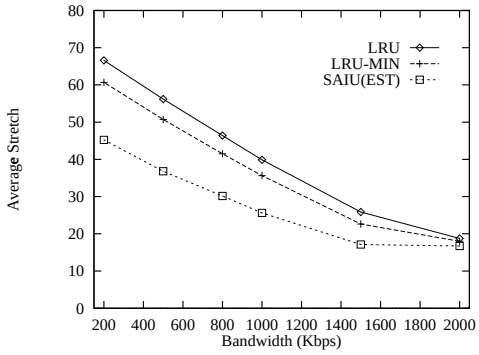

Figure 5: Stretch Performance for Various Broadcast Bandwidth (INCRT)

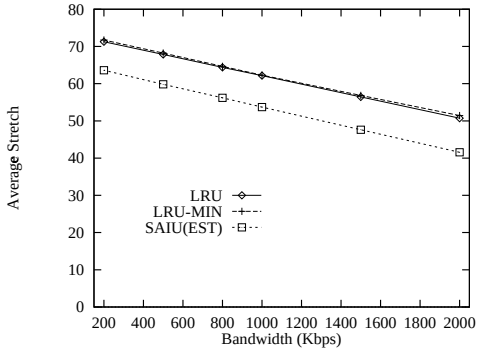

Figure 6: Stretch Performance for Various Broadcast Bandwidth (DECRT)

Item size is another important parameter in the *SAIU* policy. This subsection investigates the sensitivity of this algorithm to the ratio of the maximum item size to the minimum item size (Figure 7 and Figure 8). In the experiments, we fix the maximum item size to 2,000 KB and vary the minimum item size from 2,000 KB to 0.2 KB, i.e., the size ratio is varied from 1 to 10000. In order to make a fair comparison, when the minimum item size is decreased, the cache size is shrunk accordingly, following the formula: $CacheSize = \frac{s_{max}+s_{min}-1}{2} \times DbSize \times CacheRatio$.

From Figure 7 and Figure 8, we can see that *SAIU* adapts well to various item size ratios. *SAIU*, in all cases, improves the stretch performance substantially over *LRU* and *LRU-MIN*. Notice that as the size ratio increases, the stretch performance for all strategies decreases first and increases again for *INCRT*, whereas it almost remains unchanged for *DECRT*. The reason is as follows. In an *INCRT* sized setting, the larger the size ratio, the smaller the frequently accessed items. Therefore, as the size ratio increases from 1 to 100, the access latency improves tremendously, and thus a better stretch is achieved. However, after the size ratio reaches to some value (1000 in the experiment), the access latency cannot improve much, hence due to much smaller service times, the stretch begins to degrade again. In a *DECRT* sized setting, since clients access largest items more frequently, the above phenomenon cannot be observed. Consequently, its stretch performance is almost the same.

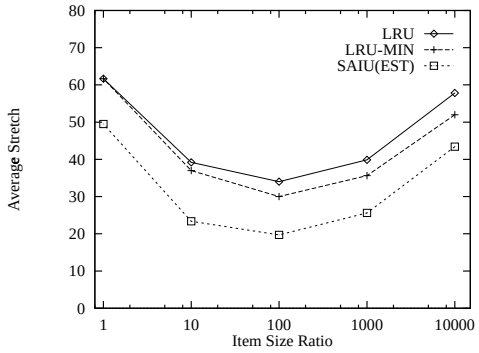

Figure 7: Stretch Performance of Various Item Size Ratios (INCRT)

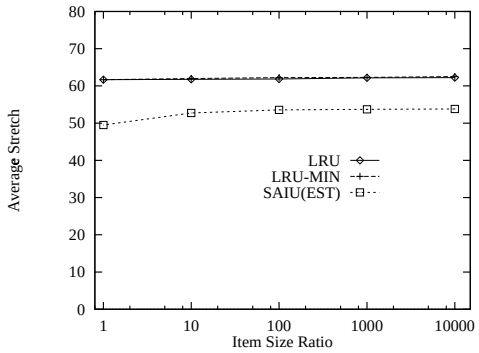

Figure 8: Stretch Performance of Various Item Size Ratios (DECRT)

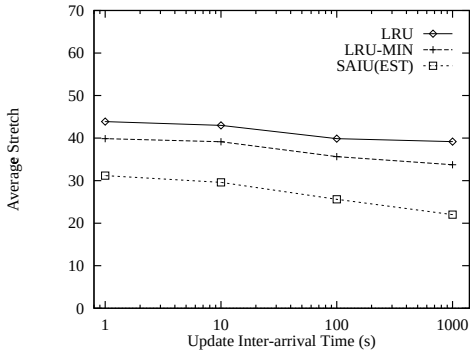

Figure 9: Performance Under Different Update Frequencies (INCRT)

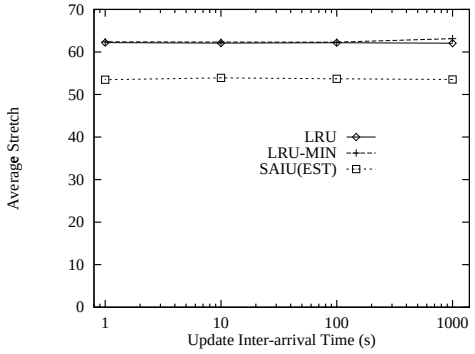

Figure 10: Performance Under Different Update Frequencies (DECRT)

5.4 Experiment #4: Influence of the Update Frequency

The influence of data update frequency is investigated in this subsection. Figure 9 and Figure 10 show the stretch performance when the update inter-arrival time is varied from 1 second to 1000 seconds. The curves obtained for the access latency performance have similar trends.

From Figure 9 and Figure 10, we can see that *SAIU* improves the stretch performance greatly in all cases and shows good adaptiveness for various update frequencies. In an *INCRT* sized setting, as the update inter-arrival time increases, the stretches for all the strategies become slightly worse. *SAIU* degrades more greatly (29.5%) than *LRU*(7.9%) and *LRU-MIN*(15.4%). This implies, from another angle, that *SAIU* is a more effective cache policy, since the more effective a cache policy, the more influence of data updates on the cache performance. We do not observe similar phenomenon in a *DECRT* sized setting. This is because for *DECRT* only a few data items can be accommodated in the client cache, and hence data updates hardly affect the cache performance.

5.5 Experiment #5: Algorithm Complexity

The previous results have shown that *SAIU* has a much better overall system performance than *LRU* and *LRU-MIN*. This subsection studies the time complexity of replacement operations for the *SAIU* algorithm by simulation. The *LRU* algorithm is included as a yardstick. Recall that a heap structure is used to implement *SAIU* (Section 3.3). *LRU* is implemented also with a heap structure in the simulation[6]. As can be seen, the time complexity during replacement consists of two parts: the removal of the victims and the insertion of the incoming item. The time complexity for the average case and the worst case is measured in terms of the number of the item nodes that are visited in the heap during every replacement.

As shown in Figure 11, the algorithm complexity for $INCRT$ is higher than that for $DECRT$ in both LRU and $SAIU$. This is because in an $INCRT$ sized setting more small data items are preferentially kept in the cache, thus the heap size is much larger than that for $DECRT$, which leads to a worse complexity. From Figure 11, it is also interesting to find that for both $INCRT$ and $DECRT$ the $SAIU$ policy has a slightly better average complexity than the LRU policy. The reason is that $SAIU$ chooses relatively large items for replacement, thus a smaller victim set and hence a better complexity is observed.

6. CONCLUSION

In this paper, we have investigated the cache replacement issue in wireless on-demand broadcasts. An efficient cache replacement policy, *SAIU*, was proposed. Different from the previous work, *SAIU* considered a real life application environment, and was developed by taking into consideration various factors that affect cache management, such as data

[6]A linked list implementation of LRU will result in significant complexity for item insertion operations, thus not employed here.

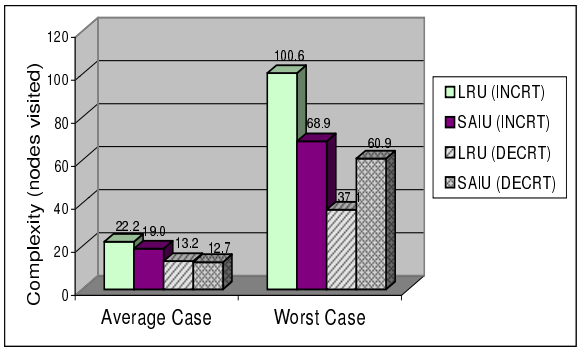

Figure 11: Comparison of Algorithm Complexity

item size, retrieval delay, access probability, and update frequency.

We conducted a series of simulation experiments to evaluate the performance of *SAIU*. The results demonstrated that the *SAIU* performs substantially better than the well known *LRU* and *LRU-MIN* policies under various workloads, especially for clients which favor access to comparatively smaller items. Through the analysis in Section 5.5, it is not difficult to see that the time complexity of *SAIU*, $O(MlogN)$, is reasonable. Therefore, it is concluded that the *SAIU* replacement policy could be for practical use in on-demand broadcasts.

In a further study, we are incorporating the factor of cache validation delay in the design of an optimal caching policy under the strong data consistency requirement. We plan to conduct simulations for clients with heterogeneous access patterns and to extend the cache replacement policy to a cache admission policy for client disk caching. Prefetching techniques can be combined into the current scheme. Since data caching and broadcast scheduling affect each other, we shall investigate scheduling algorithms which co-operate with client cache management schemes to achieve better performance. Furthermore, the proposed cache replacement policy, *SAIU*, could be easily extended to apply to other remote caching environments. Studies for the application of *SAIU* to push-based broadcasts and web proxies would be an interesting topic.

7. REFERENCES

[1] M. Abrams, C. R. Standridge, G. Abdulla, S. Williams, and E. Fox. Caching proxies: Limitations and potentials. In *Proceedings of the 4th International WWW Conference*, pages 119–133, Dec. 1995.

[2] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast disks: Data management for asymmetric communications environments. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 199–210, San Jose, California, May 1995.

[3] S. Acharya and S. Muthukrishnan. Scheduling on-demand broadcasts: New metrics and algorithms. In *Proceedings of the 4th International Conference on Mobile Computing and Networking (MobiCom'98)*, pages 43–54, Dallas, TX, October 1998.

[4] C. Aggarwal, J. L. Wolf, and P. S. Yu. Caching on the world wide web. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):94–107, January/February 1999.

[5] D. Aksoy and M. Franklin. R x W: A scheduling approach for large-scale on-demand data broadcast. *IEEE/ACM Transactions on Networking*, 7(6):846–860, 1999.

[6] D. Barbara and T. Imielinksi. Sleepers and workaholics: Caching strategies for mobile environments. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 1–12, Minneapolis, Minnesota, May 1994.

[7] A. Datta, D. E. VanderMeer, A. Celik, and V. Kumar. Broadcast protocols to support efficient retrieval from databases by mobile users. *ACM Transactions on Database Systems (TODS)*, 24(1):1–79, 1999.

[8] S. Hameed and N. H. Vaidya. Efficient algorithms for scheduling data broadcast. *ACM/Baltzer Journal of Wireless Network*, 5(3):183–193, 1999.

[9] Q. L. Hu and D. L. Lee. Cache algorithms based on adaptive invalidation reports for mobile environments. *Cluster Computing*, 1(1):39–48, Feb. 1998.

[10] Q. L. Hu, D. L. Lee, and W.-C. Lee. Performance evaluation of a wireless hierarchical data dissemination system. In *Proceedings of the 5th International Conference on Mobile Computing and Networking (MobiCom'99)*, pages 163–173, Seattle, Washington, August 1999.

[11] T. Imielinski and B. R. Badrinath. Wireless mobile computing : Challenges in data management. *Communication of ACM*, 37(10), 1994.

[12] J. Jing, A. K. Elmagarmid, A. Helal, and R. Alonso. Bit-sequences: A new cache invalidation method in mobile environments. *ACM/Baltzer Mobile Networks and Applications*, 2(2):115–127, 1997.

[13] V. Liberatore. Caching and scheduling for broadcast disk systems. Technical Report 98-71, Institute for Advanced Computer Studies, University of Maryland at College Park(UMIACS), Dec. 1998.

[14] P. Scheuermann, J. Shim, and R. Vingralek. WATCHMAN: A data warehouse intelligent cache manager. In *Proceedings of the 22nd VLDB Conference*, pages 51–62, Mumbai, India, Sep. 1996.

[15] H. Schwetman. *CSIM user's guide (version 18)*. MCC Corporation, http://www.mesquite.com, 1998.

[16] J. Shim, P. Scheuermann, and R. Vingralek. Proxy cache design: Algorithms, implementation and performance. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):549–562, July/August 1999.

[17] C.J. Su, L. Tassiulas, and V. J. Tsotras. Broadcast scheduling for information distribution. *ACM/Baltzer Journal of Wireless Networks*, 5(2):137–147, 1999.

[18] L. Tassiulas and C. J. Su. Optimal memory management strategies for a mobile user in a broadcast data delivery system. *IEEE Journal on Selected Areas in Communications*, 15(7):1226–1238, Sep. 1997.

[19] J. W. Wong. Broadcast delivery. *Proceedings of IEEE*, 76(12):1566–1577, 1988.

[20] G.K. Zipf. *Human Behaviour and the Principle of Least Effort*. Addison-Wesley, Massachusetts, 1949.