Performance Evaluation of an Optimal Cache Replacement Policy for Wireless Data Dissemination

Jianliang Xu, *Member*, *IEEE*, Qinglong Hu, Wang-Chien Lee, *Member*, *IEEE*, and Dik Lun Lee

Abstract—Data caching at mobile clients is an important technique for improving the performance of wireless data dissemination systems. However, variable data sizes, data updates, limited client resources, and frequent client disconnections make cache management a challenge. In this paper, we propose a gain-based cache replacement policy, *Min-SAUD*, for wireless data dissemination when cache consistency must be enforced before a cached item is used. *Min-SAUD* considers several factors that affect cache performance, namely, access probability, update frequency, data size, retrieval delay, and cache validation cost. This paper employs *stretch* as the major performance metric since it accounts for the data service time and, thus, is fair when items have different sizes. We prove that *Min-SAUD* achieves *optimal* stretch under some standard assumptions. Moreover, a series of simulation experiments have been conducted to thoroughly evaluate the performance of *Min-SAUD* under various system configurations. The simulation results show that, in most cases, the *Min-SAUD* replacement policy substantially outperforms two existing policies, namely, *LRU* and *SAIU*.

Index Terms—Cache replacement, cache consistency, wireless data dissemination, data management, mobile computing, performance analysis.

1 Introduction

Owing to the rapid development of mobile devices, wireless application standards, wireless high-speed networks, and supporting software technologies, we are witnessing a takeoff of wireless data applications in the commercial market. However, constraints of mobile wireless environments, such as scarce bandwidth and limited client resources, remain barriers to the full utilization of the capabilities of mobile computing. Client data caching has been considered a good solution for coping with the inefficiency of wireless data dissemination because it reduces the amount of traffic over the wireless communication channel by answering queries from data cached at the client [3], [9].

In contrast to the typical use of caching techniques in operating systems and database systems, client-side data caching in wireless data dissemination has the following characteristics [21], [33]:

1. Cached data items may have different sizes,
2. Data retrieval delays (i.e., cache miss penalties) are different for different items subject to the broadcast schedule employed, and
3. Data may be constantly updated over time at the server.

In addition, mobile clients may frequently disconnect voluntarily (to save power and/or connection cost) or due to failure. These factors make the design of client cache management for wireless data dissemination a challenge. There are three important issues involved in client cache management:

1. A *cache replacement policy* determines which data item(s) should be deleted from the cache when the free space is insufficient for accommodating an item to be cached [3],
2. A *cache prefetching policy* automatically preloads data items into the cache for possible future access requests [4], and
3. A *cache invalidation scheme* maintains data consistency between the client cache and the server [9].

In this paper, we study the cache replacement problem.

As will be discussed in Section 2, cache replacement policies for wireless environments have been studied only for push-based broadcasts [3], [25], [30]. Furthermore, these previous studies assumed that data items had the same size and ignored data updates and client disconnections. Little work has investigated cache replacement policies in a realistic wireless environment where updates, disconnections, and variable data sizes are common.

In a preliminary study [33], we developed a cache replacement policy, *SAIU*, for wireless on-demand broadcast. *SAIU* took into consideration four factors that affect cache performance, i.e., access probability, update frequency, retrieval delay, and data size. However, an optimal

- *J. Xu is with the Department of Computer Science, RM 707, Sir Run Run Shaw Building, Hong Kong Baptist University, Kowloon Tong, KLN, Hong Kong. Email: xujl@comp.hkbu.edu.hk.*
- *Q. Hu is with the Database Technology Intstitute, IBM Silicon Valley Lab, San Jose, CA 95114. E-mail: qhu@us.ibm.com.*
- *W.-C. Lee is with the Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16801. E-mail: wlee@cse.psu.edu.*
- *D.L. Lee is with the Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, KLN, Hong Kong. E-mail: dlee@cs.ust.hk.*

Manuscript received 16 May 2001; revised 21 Mar. 2002; accepted 29 July 2002.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 114158.

formula for determining the best cached item(s) to replace based on these factors was not given in the preliminary study. Also, the influence of the cache consistency requirement was not considered in *SAIU*.

In this paper, we propose an optimal cache replacement policy, called *Min-SAUD*, which accounts for the cost of ensuring *cache consistency* before each cached item is used. We argue that cache consistency must be required since it is crucial for many applications such as financial transactions and that a cache replacement policy must take into consideration the cache validation delay caused by the underlying cache invalidation scheme. In addition, *Min-SAUD* considers access probability, update frequency, retrieval delay, and data size in developing the gain function which determines the cached item(s) to be replaced.

This paper employs *stretch* as the major performance metric since it accounts for the data service time and, thus, is fair when items have different sizes. The analytical study shows that *Min-SAUD* achieves optimal stretch under the standard assumptions of the independent reference model [17] and Poisson arrivals of data accesses and updates. The adoption of the independent reference model makes sense because it reflects the access behavior on the Web as shown in [11]. On the other hand, Poisson arrivals are usually used to model data access and update processes [22].

We conduct a series of simulation experiments to evaluate the performance of the *Min-SAUD* policy under different system settings. The simulation removes the assumptions made in the analysis so that we can observe the impact of the assumptions during the analysis. The simulation simulates on-demand broadcasts and compares *Min-SAUD* to two cache replacement policies, i.e., *LRU* and *SAIU* [33]. The results show that *Min-SAUD* achieves the best performance under various system configurations. In particular, the performance improvement of *Min-SAUD* over the other schemes becomes prominent when the cache validation delay is significant. This indicates that cache validation cost plays an important role in cache replacement policies.

The rest of this paper is organized as follows: Section 2 reviews related work. Section 3 describes the system architecture and the performance metric used in this study. The cache replacement policy, *Min-SAUD*, and its optimality analysis and implementation issues are presented in Section 4. Section 5 introduces the simulation model for performance evaluation. The simulation results are presented in Section 6. Finally, Section 7 concludes the paper.

2 Related Work

Mobile wireless environments are limited by narrow bandwidth and frequent disconnections, etc. Client-side data caching is an important technique to improve access performance in such environments [3], [9]. A lot of research has been done on cache consistency, replacement, and prefetching in the past few years. However, most of the previous studies focused on the cache consistency issue, with relatively little research being done on cache replacement and prefetching methods. Given the limited cache on mobile clients, the latter two issues are very important to data access performance. In the following, we briefly review related studies.

2.1 Cache Consistency Algorithms

Barbara and Imielinski were first to address the cache consistency issue for mobile environments. In [9], three invalidation report (*IR*) based schemes, namely, *TS*, *AT*, and *SIG*, were presented. Most of the newly proposed invalidation schemes are variants of these basic *IR* schemes (e.g., [12], [13], [15], [18], [20], [23], [24], [29], [32], [35]). They differ from one another mainly in the organization of *IR* contents and the mechanism of uplink checking. All of these invalidation schemes incur certain cache validation delay for ensuring data consistency before the data is used.

In location-dependent information services, there is yet another kind of cache invalidation, where a previously cached data instance may become invalid when the client moves to a new location. In a previous paper [34], we proposed, analyzed, and evaluated three schemes for this kind of location-dependent cache invalidation. In this paper, we do not consider location-dependent services.

2.2 Cache Replacement and Prefetching Policies

The cache replacement issue for wireless data dissemination was first studied in the *Broadcast Disks* (Bdisk) project. Acharya et al. proposed a cache replacement policy called $\mathcal{PIX}$ [2], [3], in which the data item with the minimum value of p/x was evicted for replacement, where p is the item's access probability and x is its broadcast frequency. Thus, an evicted item either has a low access probability or has a short retrieval delay. In a subsequent study [4], Acharya et al. explored the use of prefetching to further improve data access performance.

Tassiulas and Su presented a cache update policy that attempted to minimize average access latency [30]. In [30], the broadcast channel was divided into time slots of equal size, which were equal to the broadcast time of a single item. Let λ_i be the access rate for item i and $\tau_i(n)$ be the amount of time from slot n to the next transmission of item i. A time-dependent reward (latency reduction) for item i at slot n is given by $r(i,n) = \lambda_i \tau_i(n) + \frac{\lambda_i}{2}$. The proposed *W-step look-ahead* scheme made the cache update decision at slot n such that the cumulative average reward from slot n up to slot $n+W$ was maximized. The larger the window W, the better the access performance, but the worse the complexity of the algorithm.

Caching algorithms for the Bdisk systems were also investigated by Khanna and Liberatore [25]. Different from the previous work, their work assumed that neither knowledge of future data requests nor knowledge of access probability distribution over the data items was available to the clients. The proposed *Gray* algorithm took into consideration the factors of both access history and retrieval delay for cache replacement/prefetching. Theoretical study showed that, in terms of worst-case performance, *Gray* outperformed *LRU* by a factor proportional to $CacheSize/\log CacheSize$.

In summary, existing studies on cache replacement for wireless data dissemination are based on simplifying assumptions, such as fixed data sizes, no updates, and no disconnections, thereby making the proposed schemes impractical for a realistic wireless environment.

2.3 Replacement Policies for Web Proxy Caching

Other related work includes studies on Web proxy caching [6], [31]. Here, we briefly describe some typical solutions for

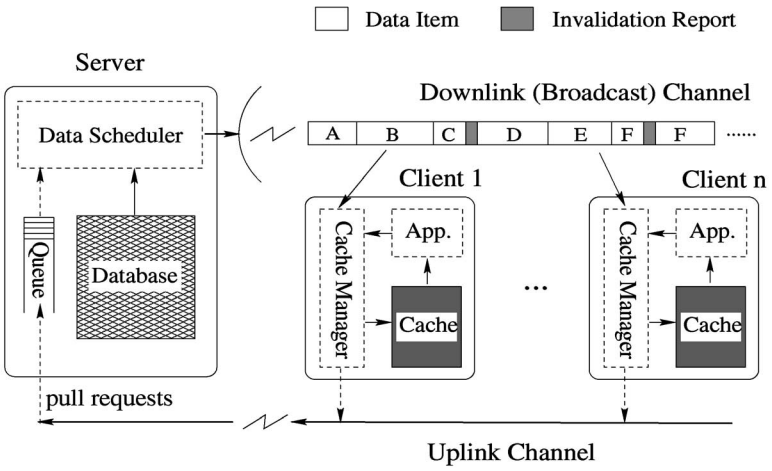


Fig. 1. A generic system architecture.

cache replacement in Web proxy caching. Based on Web traces, it was observed that small documents are accessed frequently [1], [28]. To deal with small document retrieval, the *LRU-MIN* cache replacement policy was proposed [1]. A generalized *LRU* replacement policy that considered document size was also investigated [6]. More recently, cost-based cache replacement policies have been developed [10], [28]. Bolot and Hoschka developed a cost-based algorithm which considers document retrieval delay [10]. Shim et al. proposed a cost-based replacement algorithm, *LNC-R-W3-U*, which explicitly considers document retrieval delay and document validation cost [28]. However, these existing studies did not give analytical justifications for the cost functions used, but solely relied on trace-based simulation, which was valid only for the particular traces used, but failed to give insightful observations. Moreover, the characteristics of Web access behavior may evolve from time to time. For example, the clients' preference to access small documents observed in [1] could not be observed in more recent studies [11]. In addition, the Web and mobile environments differ in many aspects, e.g., the data delivery and update propagation methods are quite different. None of the studies on Web proxy caching employed the *stretch* performance measure, which is the primary metric used in this paper (see Section 3.2).

In a recent paper [19], Hosseini-Khayat explored the cache replacement problem for data items with variable sizes and different cache miss penalties. However, [19] only studied the *offline* version of the problem in which the entire sequence of future requests is known in advance. In contrast, this paper focuses on *online* cache replacement policies.

More recently, Chang and Chen [16] investigated caching strategies for transcoding proxies. Transcoding is a transformation that is used to convert a multimedia object from one form to another, which trades off object fidelity for size. A weighted transcoding graph was devised to manage multiple versions of objects in the proxy cache. This paper does not consider object transcoding.

3 Background

In this section, we give a brief description of the system architecture and the performance metric adopted in this study.

3.1 System Architecture

Fig. 1 depicts a generic architecture of the wireless data dissemination systems studied. We assume that the system employs on-demand broadcasts for data dissemination. That is, the clients send pull requests to the server through the uplink channel. In response, the server disseminates the requested data items to the clients through the broadcast channel based on a scheduling algorithm [5], [7], [8]. The clients retrieve the items of their interest off the air by monitoring the broadcast channel.

Push-based broadcast is a common alternative to on-demand broadcast for wireless data dissemination [3]. In push-based broadcasts, a fixed set of data are periodically broadcast based on precompiled data access patterns. In fact, push-based broadcasts can be seen as a special case of on-demand broadcasts, where uplink cost is zero and data scheduling is based on the aggregate access patterns. Consequently, the result presented in this paper can also be applied to push-based broadcasts.

As illustrated, there is a cache management mechanism in a client. Whenever an application issues a query, the local cache manager first checks whether the desired data item is in the cache. If it is a cache hit, the cache manager still needs to validate the consistency of the cached item with the master copy at the server. To validate the cached item, the cache manager retrieves the next invalidation report from the broadcast channel (see below for details). If the item is verified as being up-to-date, it is returned to the application immediately. If it is a cache hit but the value is obsolete or it is a cache miss, the cache manager sends a pull request to the server, which will schedule the broadcast of the desired data. When the requested data item arrives on the wireless channel, the cache manager returns it to the application and retains a copy in the cache. The issue of cache replacement arises when the free cache space is not enough to accommodate a data item to be cached. Since validation is important to ensure data consistency and the delay it causes cannot be neglected, we develop, in this paper, an optimal cache replacement scheme that incorporates the validation delay in determining the cached item(s) to be replaced.

Techniques based on *invalidation report (IR)* have been proposed to address the cache consistency issue [9], [20].

*IR*s are interleaved with the broadcast data and periodically broadcast on the broadcast channel. An *IR* consists of the server's update history up to the most recent w broadcast intervals (w can be a constant or a variable). Every mobile client maintains the timestamp T_l of the last cache validation. Thus, upon reception of an *IR*, a client checks to see whether its T_l is within the coverage of the *IR* received. If it is, the client starts the invalidation process in accordance with the type of the *IR* received. Otherwise, it drops the cache contents entirely (when w is a constant) [9] or ignores the *IR* and sends its T_l to the server in order to enlarge w of the next *IR* (when w is a variable) [20].

3.2 Performance Metrics

In operating systems and database systems, cached items usually have the same size (e.g., a *page* or a *block*). In these systems, since cache miss penalties for all cached items are the same, 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. Consequently, the cache hit ratio is often used to measure the effectiveness of cache replacement policies in traditional cache management.

For applications in which cached items have different sizes, the cache hit ratio is obviously no longer a reliable performance metric. In the previous work on Web proxy caching, the *byte hit ratio*, which is the ratio of the total number of bytes hit to the total number of bytes requested, was introduced to evaluate cache performance. In a wireless data dissemination system, however, subject to the broadcast schedule employed, cache miss penalties differ for different data items. Thus, the byte hit ratio cannot fairly reflect the overall system performance either. In this paper, we use *stretch* to evaluate the performance of cache replacement policies:

- **Stretch** [5]: 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.

Generally, for a smaller item, which has a shorter service time, a shorter access latency is expected by users. In contrast, users can tolerate a longer latency for a larger item. Since access latency does not count the difference in data size/service time, it is not a fair performance metric. Stretch overcomes such a shortcoming in performance measure. Thus, this study aims to optimize the overall stretch performance of a cache replacement policy while keeping access latency as short as possible. On the other hand, as we will discuss in Section 7, the proposed analysis technique can be extended to optimize other performance metrics such as access latency and cache hit ratio.

4 An Optimal Cache Replacement Algorithm

Cache replacement policy plays a central role in cache management. Traditional cache replacement policies (e.g., *LRU*), while suitable for cached items with the same size and miss penalty, do not perform well in wireless data dissemination [33]. In the following, we first introduce a new gain-based cache replacement policy, *Min-SAUD*. Then, we show that the proposed policy results in the optimal access performance in terms of *stretch*. Finally, we address some of the implementation issues.

4.1 The *Min-SAUD* Replacement Policy

In this section, a gain-based cache replacement policy, *Minimum Stretch integrated with Access rates, Update frequencies, and cache validation Delay* (denoted as *Min-SAUD*), is proposed for the wireless data dissemination systems under cache consistency. To facilitate our discussion, the following notations are defined (note that these parameters are for one client only):

- D: the number of data items in the database.
- C: the size of the client cache.
- $\overline{a_i}$: mean access arrival rate of data item i,

$$i = 1, 2, \ldots, D.$$

- $\overline{u_i}$: mean update arrival rate of data item i,

$$i = 1, 2, \ldots, D.$$

- x_i: the ratio of update rate to access rate for data item i, i.e., $x_i = \overline{u_i}/\overline{a_i}$, $i = 1, 2, \ldots, D$.
- p_i: access probability of data item i, $p_i = \overline{a_i}/\sum_{k=1}^{D} \overline{a_k}$ for $i = 1, 2, \ldots, D$.
- l_i: access latency of data item i, $i = 1, 2, \ldots, D$.
- b_i: retrieval delay from the server (i.e., cache miss penalty) for data item i, $i = 1, 2, \ldots, D$.
- s_i: size of data item i, $i = 1, 2, \ldots, D$.
- v: cache validation delay, i.e., access latency of an effective invalidation report.
- d_k: the data item requested in the kth access,[1] $d_k \in \{1, 2, \ldots, D\}$.
- C_k: the set of cached data items after the kth access, $C_k \subseteq \{1, 2, \ldots, D\}$.
- U_k: the set of cached data items that are updated between the kth access and the $(k+1)$th access, $U_k \subseteq C_k$.
- V_k: the set of victims chosen to be replaced in the kth access, $V_k \subseteq (C_{k-1} - U_{k-1})$.

The key issue for cache replacement is to determine a victim itemset, V_k, when the free space in the client cache is not enough to accommodate the incoming data item in the kth access. In [33], we have observed that a cache replacement policy should choose the data items with low access probability, short data retrieval delay, high update frequency, and large data size for replacement. As described in the Introduction, a cache replacement policy should also take into account the cost of cache validation. Thus, in *Min-SAUD*, a gain function incorporating these factors is defined for each cached item i:[2]

$$gain(i) = \frac{p_i}{s_i}\left(\frac{b_i}{1 + x_i} - v\right). \quad (1)$$

1. The client accesses are assumed to be numbered sequentially.
2. The gain function was derived while we were trying to prove the optimality of a heuristic gain function that we initially defined for cache replacement.

The idea is to maximize the total gain for the data items kept in the cache. Thus, to find space for the kth accessed data item, the *Min-SAUD* policy identifies the optimal victim itemset, V_k^*, $V_k^* \subseteq (C_{k-1} - U_{k-1})$, such that

$$V_k^* = arg \min_{V_k \subseteq (C_{k-1}-U_{k-1})} \sum_{i \in V_k} gain(i) \tag{2}$$

$$s.t. \quad \sum_{i \in V_k} s_i \geq \sum_{j \in (C_{k-1}-U_{k-1})} s_j + s_{d_k} - C. \tag{3}$$

It is easy to see that *Min-SAUD* reduces to $\mathcal{PIX}$ when *Bdisk* is used and when data items have equal size and are read-only. This is because, under that circumstance, the data retrieval delay of an item is inversely proportional to its broadcast frequency [3]. Therefore, *Min-SAUD* can be considered a generalization of $\mathcal{PIX}$.

4.2 Analysis of the *Min-SAUD* Policy

In this section, we show that *Min-SAUD* is an optimal cache replacement policy in terms of stretch. The independent reference model [17] is assumed in the analysis. To facilitate our analysis, we assume that the arrivals of data accesses and updates for data item i follow the Poisson processes. Specifically, t_i^a and t_i^u, the interarrival times for data accesses and updates of data item i, follow exponential distributions with means of $\overline{a_i}$ and $\overline{u_i}$, respectively. In other words, the density functions for t_i^a and t_i^u are $f(t_i^a) = \overline{a_i} e^{-\overline{a_i} t_i^a}$ and $g(t_i^u) = \overline{u_i} e^{-\overline{u_i} t_i^u}$, respectively. Further, we assume that the access latency of the cache is zero since it is negligible compared to the access latency to the server.

The access cost, in terms of stretch, for a data item is the product of its access probability and its stretch. Recall that the stretch of a data item is the ratio of its access latency to its service time, where the service time can be derived by the ratio of the item size to the broadcast bandwidth. With a fixed broadcast bandwidth, we can ignore the bandwidth factor and define the *relative stretch* of a data item as the ratio of its access latency to its size. Without loss of generality, assuming k accesses to the data items have taken place, we have the *relative* access cost S_k for any caching strategy after the kth access as follows:

$$S_k = \sum_{1 \leq i \leq D} p_i \cdot \frac{l_i}{s_i}. \tag{4}$$

Under the data consistency requirement, even when a query generates a cache hit, the client still needs to wait for the arrival of the effective *IR*, which is the next *IR* containing the update information necessary for the validation of the cached copy. Let $Pr(U_i)$ be the probability that data item i is updated during the period from the current time to the arrival time of the effective *IR* of the next query on item i. Thus, (4) can be rewritten as:

$$\begin{aligned} S_k &= \sum_{i \in C_k} \frac{p_i \cdot l_i}{s_i} + \sum_{i \notin C_k} \frac{p_i \cdot l_i}{s_i} \\ &= \sum_{i \in C_k} \frac{p_i \cdot l_i}{s_i} Pr(U_i) + \sum_{i \in C_k} \frac{p_i \cdot l_i}{s_i} (1 - Pr(U_i)) + \sum_{i \notin C_k} \frac{p_i \cdot l_i}{s_i}. \end{aligned} \tag{5}$$

The above equation consists of three terms, corresponding to three cases, namely,

1. a cache hit but an obsolete copy,
2. a cache hit and an up-to-date copy, and
3. a cache miss.

The access latency l_is in these three cases are $v + b_i$, v, and b_i, respectively:

$$l_i = \begin{cases} v + b_i & \text{if } i \in C_k \text{ and an obsolete copy;} \\ v & \text{if } i \in C_k \text{ and an up-to-date copy;} \\ b_i & \text{if } i \notin C_k. \end{cases} \tag{6}$$

We are now going to derive $Pr(U_i)$. We use $Pr(U_i')$ to denote the probability that data item i is updated during the period from the current time to the arrival time of the next query on data item i. We expect the chance that item i is updated during the *IR* waiting period but not updated between the current time and the arrival time of the next query on item i is very slim, thus $Pr(U_i)$ can be approximated by $Pr(U_i')$:

$$\begin{aligned} Pr(U_i) \doteq Pr(U_i') &= Pr(t_i^u < t_i^a) = \int_{t_i^a=0}^{\infty} \int_{t_i^u=0}^{t_i^a} f(t_i^a) g(t_i^u) dt_i^u dt_i^a \\ &= \frac{\overline{u_i}}{\overline{u_i} + \overline{a_i}}. \end{aligned} \tag{7}$$

Therefore, combining (5), (6), and (7), we obtain

$$\begin{aligned} S_k = &\sum_{i \in C_k} \left(\frac{p_i(v + b_i)}{s_i} \cdot \frac{\overline{u_i}}{\overline{u_i} + \overline{a_i}} \right) + \sum_{i \in C_k} \left(\frac{p_i \cdot v}{s_i} \cdot \frac{\overline{a_i}}{\overline{u_i} + \overline{a_i}} \right) + \\ &\sum_{i \notin C_k} \frac{p_i \cdot b_i}{s_i} = \sum_{i \in C_k} \frac{p_i(v + \frac{\overline{u_i} b_i}{\overline{u_i} + \overline{a_i}})}{s_i} + \sum_{i \notin C_k} \frac{p_i \cdot b_i}{s_i}. \end{aligned} \tag{8}$$

The following theorem proves that *Min-SAUD* is an optimal cache replacement policy:

Theorem 1. *The replacement policy* Min-SAUD *gives better access cost, in terms of stretch, than any other replacement policy.*

Proof. We derive the optimality of the *Min-SAUD* policy by showing that the cost S_k using this policy is always the minimum for all k. The proof uses the induction method.

Suppose S_w is the optimal cost for some $k = w$ given any cache replacement policy. Let V_{w+1} be the victim set chosen to make room for d_{w+1}. Therefore, we obtain $C_{w+1} = C_w - U_w \cup \{d_{w+1}\} - V_{w+1}$. Hence,

$$\begin{aligned} S_{w+1} &= \sum_{i \in C_{w+1}} \frac{p_i \left(v + \frac{\overline{u_i} b_i}{\overline{u_i} + \overline{a_i}} \right)}{s_i} + \sum_{i \notin C_{w+1}} \frac{p_i \cdot b_i}{s_i} \\ &= S_w + \sum_{i \in U_w} \left(\frac{p_i}{s_i} \left(\frac{b_i}{1 + x_i} - v \right) \right) - \frac{p_{d_{w+1}}}{s_{d_{w+1}}} \left(\frac{b_{d_{w+1}}}{1 + x_{d_{w+1}}} - v \right) \\ &\quad + \sum_{i \in V_{w+1}} \left(\frac{p_i}{s_i} \left(\frac{b_i}{1 + x_i} - v \right) \right) \\ &= B + \sum_{i \in V_{w+1}} \frac{p_i}{s_i} \left(\frac{b_i}{1 + x_i} - v \right), \end{aligned} \tag{9}$$

where

$$B = S_w + \sum_{i \in U_w} \left(\frac{p_i}{s_i} \left(\frac{b_i}{1 + x_i} - v \right) \right) - \frac{p_{d_{w+1}}}{s_{d_{w+1}}} \left(\frac{b_{d_{w+1}}}{1 + x_{d_{w+1}}} - v \right).$$

Since a cache replacement policy cannot control the value of B, (9) implies that the lowest access cost is achieved when the items with the lowest

$$\sum_{i \in V_{w+1}} \frac{p_i}{s_i} \left(\frac{b_i}{1 + x_i} - v \right)$$

are chosen as the victims. This is exactly what the *Min-SAUD* policy does. As the *Min-SAUD* policy makes this optimal decision for every replacement, subject to the restriction of cache capacity, no other policy can provide a lower access cost. □

4.3 Implementation Issues

In this section, we first address three critical implementation issues, namely, heap management, estimate of running parameters, and maintenance of cached item attributes, for the *Min-SAUD* policy. Finally, the client cache access mechanism is described.

4.3.1 Heap Management

In *Min-SAUD*, the optimization problem defined by (2) and (3) is essentially the $0/1$ knapsack problem, which is known to be *NP-hard*. Thus, a well-known heuristic for the knapsack problem is adopted to find a suboptimal solution for *Min-SAUD*:[3]

> Throw out the cached data item i with the minimum $gain(i)/s_i$ value until the free cache space is sufficient to accommodate the incoming item.

This heuristic can obtain the optimal solution when the data sizes are relatively small compared to the cache size [28].

A (binary) min-heap data structure is used to implement the *Min-SAUD* policy. The key field for the heap is the $gain(i)/s_i$ value for each cached data item i. When cache replacement occurs, 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 an $O(logN)$ complexity. Thus, the time complexity for every cache replacement operation is $O(MlogN)$. In addition, when an item's $gain(i)/s_i$ 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 *Min-SAUD* is further investigated by simulation experiments in Section 6.5.

4.3.2 Estimate of Running Parameters

Several parameters are involved in computation of the $gain(i)$ function. Among these parameters, p_i is proportional to $\overline{a_i}$, s_i can be obtained when item i arrives, and v is a system parameter. In most cases, $\overline{u_i}$, b_i, and $\overline{a_i}$ are not available to the clients. Thus, we need methods to estimate these values.

3. We do not distinguish *Min-SAUD* and this heuristic in the rest of this paper.

A well-known *exponential aging* method is used to estimate $\overline{u_i}$ and b_i. Initially, $\overline{u_i}$ and b_i are set to 0. When a new update on item i arrives, $\overline{u_i}$ is updated according to the following formula:

$$\overline{u_i} = \alpha_u / (t^c - t_i^{lu}) + (1 - \alpha_u) \cdot \overline{u_i}, \tag{10}$$

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 those of the past updates. The larger the α_u value, the more important the recent updates.

Similarly, when a query for item i is answered by the server, b_i is reevaluated as follows:

$$b_i = \alpha_s \cdot (t^c - t_i^{qt}) + (1 - \alpha_s) \cdot b_i, \tag{11}$$

where t_i^{qt} is the query time and α_s is a weight factor for the running b_i estimate. $\overline{u_i}$ and b_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 $\overline{u_i}$ based on the received *IR*s. The client caches the data item as well as its $\overline{u_i}$, t_i^{lu}, and b_i values. The maintenance of these parameters (along with some other parameters) will be discussed in the next section.

Different clients may have different access patterns, while some of their data accesses are answered by the cache. It is difficult for the server to know the real access pattern that originated from each client. Consequently, the access arrival rate $\overline{a_i}$ is estimated at the client-side. The exponential aging method might not be accurate because it does not age the access rate for the time period since the last access. Therefore, a *sliding average* method is employed in the implementation [28]. We keep a sliding window of k most recent access timestamps $(t_i^1, t_i^2, \cdots, t_i^k)$ for item i in the cache. The access rate $\overline{a_i}$ is updated using the following formula:

$$\overline{a_i} = \frac{k}{t^c - t_i^k}, \tag{12}$$

where t^c is the current time and t_i^k is the timestamp of the oldest access to item i in the sliding window. When fewer than k access timestamps are available for item i, the mean access rate $\overline{a_i}$ is estimated using the maximal number of available samples.

To reduce the computational complexity, the access rates for all cached items are not updated during each replacement. Instead, in the implementation, we update the mean access rate when the data item is accessed. In addition, similarly to [28], we employ an "aging daemon" which periodically generates dummy accesses to all data items. If, for an item, the time period since the last access is larger than an *aging threshold*, we use the current time to age its access rate. The advantages for the periodic aging approach are two-fold. First, the aging accounts for the time since the last access and, hence, is able to catch up the changing workload. Second, with infrequent periodic aging, only minimal reorganization of the heap structure is required. The settings of the estimate parameters are described in Section 6.

Item heap: maintains item attributes and pointers to real data for cached data
Attribute heap: maintains the attributes for non-cached items
Input: the key for the requested data item i
Output: the requested data item i
Procedure:

```
1: if cache hit then
2:    wait for completion of the cache validating procedure;
3:    for all item j's that have been updated since last cache invalidation do
4:       if item j's attributes are cached then
5:          update t_j^lu according to the received IR and u_j using Equation (10);
6:          move the node for item j from the item heap to the attribute heap if item j is
            cached;
7:       end if
8:    end for
9:    if still cache hit then
10:                        // a cache hit and a valid copy
11:      adjust item node i's position in the item heap;
12:      return item i to the application;
13:   end if
14: end if
15: send to the server a request to schedule broadcast of item i;
16:                 // a cache hit but an invalid copy or a cache miss
17: monitor the broadcast channel until item i arrives;
18: if there is no sufficient space to cache item i then
19:    while free space is insufficient do
20:       remove the data pointed by the root of the item heap;
21:       move the node for that item from the item heap to the attribute heap;
22:    end while
23: end if
24: store item i and insert a node for it in the item heap (maybe moved from the attribute
    heap);
25: return item i to the application;
```

Fig. 2. Algorithm 1: Client Cache Access Mechanism.

4.3.3 Maintenance of Cached Item Attributes

To realize the *Min-SAUD* policy, a number of parameters must be maintained for each cached data item. They are $s_i, \overline{u_i}, t_i^{lu}, b_i, \overline{a_i}$, and t_i^ks. We refer to these parameters as the *cached item attributes* (or simply *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 [26], [28], 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 (it may incorrectly produce a relatively smaller $gain$ value). If the cached item attributes are evicted from the cache together with data item i, then, upon reaccessing item i, these attributes must be collected again from scratch. Consequently, item i is likely to be evicted again.

Similarly to [26], we employ a heuristic to maintain the cached item attributes. The attributes for the currently cached data items are kept in the cache. Let N_c be the number of cached items. For those data items that are not cached, we only retain the attributes for N_c items with the largest $gain(j)/s_j$ values. Since the attributes themselves can be viewed as a kind of special data, as in the management for cached data, a separate heap is employed to manage the attributes for noncached data. This heuristic is adaptive to the cache size. When the cache size is large, it can accommodate more data items and, hence, attributes for more noncached data can be retained in the cache. On the other hand, when the cache size is small, fewer data items are contained and, thus, fewer attributes are kept.

4.3.4 Description of Client Cache Access Mechanism

We have discussed the various implementation issues for the *Min-SAUD* policy; we now show the client cache access mechanism in Fig. 2. Whenever an application issues a query for data item i, this cache access procedure is invoked.

5 Simulation Model

The simulation model used for performance evaluation is similar to that used in a previous study [33]. It is implemented using *CSIM* [27]. A single cell environment is considered. The model consists of a single server and $NumClient$ clients.[4] On-demand broadcast is employed for wireless data dissemination.

The default system parameter settings are given in Table 1. The database is a collection of $DatabaseSize$ data items and is partitioned into disjointed regions, each with $RegionSize$ items. The data access pattern and update pattern are applied on the regions (see Sections 5.1 and 5.2 for details). Data item sizes vary from s_{min} to s_{max} and have the following three types of distributions:

4. Each client could be further treated as an aggregate of clients with lower access rates.

TABLE 1
Default System Parameter Settings

Parameter	Setting	Meaning
$NumClient$	200	Number of aggregate clients in a cell
$DatabaseSize$	2,000 data items	Number of data items in the database
$RegionSize$	20 data items	Number of data items per database region
s_{min}	1KB	The minimum data item size in the database
s_{max}	100KB	The maximum data item size in the database
$BroadcastBW$	115 Kbps	Channel bandwidth from the server to the clients
$UplinkBW$	14.4 Kbps	Channel bandwidth from the clients to the server
$BroadcastInt$	20 seconds	Broadcast interval for invalidation reports
$ConMsgSize$	512 bytes	Size of control message in an invalidation report

- **INCRT**:

$$size_i = s_{min} + \frac{(i-1)(s_{max} - s_{min} + 1)}{DatabaseSize}, \quad i = 1, \ldots, DatabaseSize,$$

- **DECRT**:

$$size_i = s_{max} - \frac{(i-1)(s_{max} - s_{min} + 1)}{DatabaseSize}, \quad i = 1, \ldots, DatabaseSize,$$

- **RAND**:

$$size_i = s_{min} + \lfloor prob() \cdot (s_{max} - s_{min} + 1) \rfloor, \quad i = 1, \ldots, DatabaseSize,$$

where $prob()$ is a random function uniformly distributed between 0 and 1. Combined with the skewed access pattern, *INCRT* and *DECRT* represent clients' preference for frequently querying smaller items and larger items, respectively; *RAND* models the case where no correlation between the access pattern and data size exists (see Section 5.1 for further details).

*IR*s are broadcast periodically on the broadcast channel with an interval of $BroadcastInt$. The broadcast channel has a bandwidth of $BroadcastBW$. It works in a *preempt-resume* manner with *IR*s having the highest broadcast priority and all other messages having equal priority. This strategy ensures that *IR*s can always reach the clients in time [20]. The uplink channel has a bandwidth of $UplinkBW$.

5.1 Client Model

Each client is simulated by a process running 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.[5] The $ThinkTime$ parameter allows the cost of client processing relative to data broadcast time to be adjusted, thus it can be used to model workload processing as well as the relative speeds of the client CPU and the broadcast medium [3]. 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 size $CacheSize$, which is defined as

$$0.5 \times (s_{max} + s_{min} - 1) \times DatabaseSize \times CacheSizeRatio.$$

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 skewness parameter θ [36]. The data items are sorted such that item 0 is the most frequently accessed and item $DatabaseSize - 1$ is the least frequently accessed. In other words, with the *INCRT* size setting, the clients access the smallest item most frequently; with the *DECRT* size setting, the clients access the largest item most frequently. $Zipf$ distributions are frequently used to model nonuniform access patterns. The probability of accessing any item within a database region is uniform, while accesses to the database regions follow the $Zipf$ distribution. Table 2 summarizes the default client parameter settings.

5.2 Server Model

The server is modeled by a single process. Table 3 gives the server parameter settings. Requests from the clients are buffered at the server, assuming an infinite queue buffer is used. After broadcasting the current item, the server chooses an outstanding request from the buffer as the next candidate according to the scheduling algorithm used. Compared with queuing delay and data transmission delay, the overhead of broadcast scheduling and request processing at the server is negligible. Therefore, they are not considered in the model.

The server process generates data updates with an exponentially distributed update interarrival time having 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, $Update\text{-}Cold\%$ of the updates are for the first $Update\text{-}Hot\%$ items and $Update\text{-}Hot\%$ of the updates are for the rest. For example, we assume in the experiments that, within a region, 80 percent of the updates occur on the first 20 percent of data items (i.e., update-hot items) and 20 percent of the updates occur on the remaining 80 percent of data items (i.e., update-cold items).

5. Since a query is initiated after the completion of the last query, the access arrivals do not follow the Poisson process. Thus, the analytical assumptions are relaxed in the simulation.

TABLE 2
Default Client Parameter Settings

Parameter	Setting	Meaning
$ThinkTime$	10 seconds	Mean think time between consecutive queries
p	0.1	Probability of client disconnection per IR interval
$CacheSizeRatio$	5%	Ratio of the cache size to the database size
$ParaSize$	4 bytes each	Overhead for each cached parameter in replacement policies
$DiscTime$	200 seconds	Mean disconnection time
θ	0.80	Zipf distribution parameter

TABLE 3
Default Server Parameter Settings

Parameter	Setting	Meaning
$UpdateTime$	80 seconds	Mean update inter-arrive time for the database
$Update\text{-}Cold/Update\text{-}Hot$	80/20	Setting for the Cold/Hot update pattern

6 Performance Evaluation

This section evaluates the performance of the proposed cache replacement policy by simulation. *Average stretch* is the primary performance metric employed in this study. In the experiments, we employ *LTSF* [5] as the on-demand broadcast scheduling algorithm and *AAW_AT* [20] as the cache invalidation scheme since they demonstrated superior performance over other schemes [5], [20]. In *LTSF*, the data item with the largest total current stretch is chosen for the 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. In *AAW_AT*, the updating history window w and content organization of the next *IR* are dynamically decided based on the system workload.

The results are obtained when the system has reached a stable state, i.e., each client has issued at least 5,000 queries after its cache is full, so that the warm-up effect on the client cache and the broadcast channel is eliminated. For the exponential aging estimate method, we set $\alpha_s = \alpha_u = 0.25$ [3], [28]. For the sliding average method, the aging period is set to 10,000 seconds and the aging threshold is set to roughly the product of the average access latency and the database size. Unless it is mentioned explicitly, the broadcast bandwidth is fully utilized.

The overall cache performance in a wireless data dissemination system is determined by several factors, such as cache size, cache validation delay, access skewness, and item size ratio. In the following, we first explore the robustness of the proposed cache replacement policy, *Min-SAUD*, under various workloads. Then, we analyze the time complexity of the *Min-SAUD* policy. The *LRU* policy is included as a yardstick in the performance evaluation. We also compare *Min-SAUD* to *SAIU*, which makes use of a cost function of $b_i \cdot \overline{a_i}/(s_i \cdot \overline{u_i})$ to determine the victims [33].

6.1 Experiment #1: Impact of the Cache Size

This section investigates the performance of the cache replacement schemes under different cache sizes. The simulation results are shown in Fig. 3. To estimate data retrieval delays and access and update frequencies, *Min-SAUD(EST)* uses (10), (11), and (12). *Min-SAUD(IDL)* and *SAIU(IDL)* are assumed to have perfect knowledge of data access and update frequencies. Moreover, in *Min-SAUD(IDL)* and *SAIU(IDL)*, the cache space used for storing the cached attributes is not counted.

In Fig. 3, it is obvious that *Min-SAUD* achieves the best stretch performance. On average, the improvement of *Min-SAUD(IDL)* over *LRU* for *INCRT*, *RAND*, and *DECRT* is 30.7 percent, 21.7 percent, and 11.8 percent, respectively, and the improvement of *Min-SAUD(IDL)* over *SAIU(IDL)* for *INCRT* and *RAND* is 24.6 percent and 8.9 percent, respectively. *Min-SAUD(IDL)* and *SAIU(IDL)* have a similar performance for *DECRT*. The improvement follows a decreasing order of *INCRT*, *RAND*, and *DECRT*. This can be explained as follows: First, since *Min-SAUD* takes into consideration the data size, it caches more frequently accessed items in the *INCRT* size setting, whereas it has to balance between caching more items and caching more frequently accessed items in the other two settings, especially for *DECRT*. Thus, *Min-SAUD* outperforms *LRU* to a greater extent for *INCRT* than for *RAND* and *DECRT*. Second, because the influence of cache validation delay on the stretch performance follows a decreasing order of *INCRT*, *RAND*, and *DECRT* (see the next section for details), *Min-SAUD*, which takes into consideration the cache validation delay, improves performance the most for *INCRT* and the least for *DECRT*. As the cache size increases, the improvement of *Min-SAUD* over *LRU* and *SAIU* becomes more significant (for example, for *INCRT*, from 14.3 percent to 42.1 percent over *LRU* and from 20.1 percent to 31.5 percent over *SAIU*). This implies that *Min-SAUD* can utilize the cache space more effectively.

Since the system parameters (i.e., access and update frequencies) in *Min-SAUD(EST)* are only estimates and they occupy some cache space, it shows that *Min-SAUD(EST)* performs slightly worse (within 10 percent) than *Min-SAUD(IDL)*. But still, in most cases, it outperforms *SAIU(IDL)*, which has perfect knowledge of data access and update frequencies.

The performance of access latency and cache byte hit ratio for *RAND* is shown in Fig. 4. Although *Min-SAUD* attempts to optimize stretch by design, it performs much better than *LRU* and only slightly worse than *SAIU* in terms of these two metrics. Similar results are observed for *INCRT* and *DECRT*.

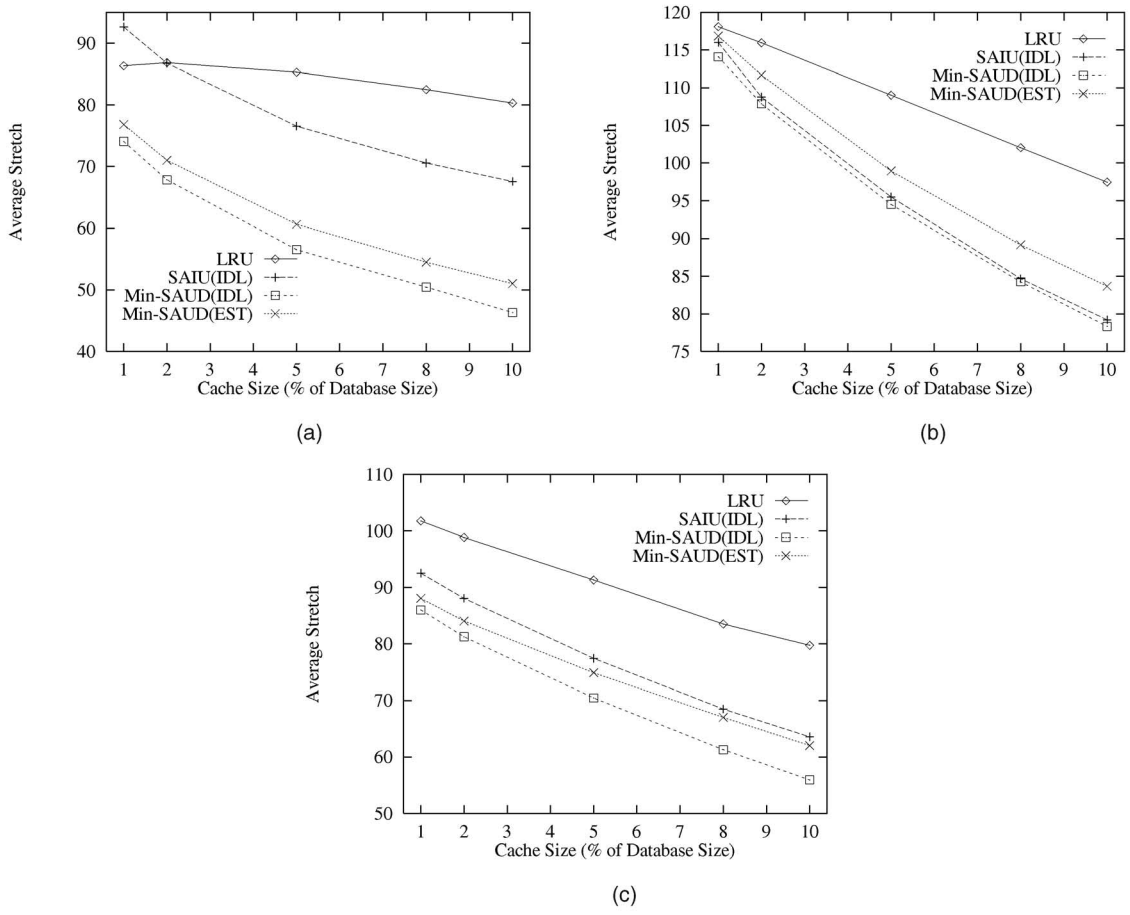


Fig. 3. Stretch performance under various cache sizes. (a) INCRT. (b) DECRT. (c) RAND.

6.2 Experiment #2: Impact of the Cache Validation Delay

As pointed out in the previous sections, cache validation cost can have a great impact on the performance of a cache replacement policy. Such influence is investigated by experiments in this section. Fig. 5 illustrates the performance when the *IR* broadcast interval varies from 1 second to 50 seconds. Note that the larger the *IR* broadcast interval, the longer the cache validation delay. In Fig. 5, we also include a *No Caching* scheme for comparison. In the *No Caching* scheme, the clients do not cache any data locally and, hence, its performance is not affected by the IR interval.

Let's first compare the *INCRT*, *RAND*, and *DECRT* size settings. The influence of the cache validation delay on stretch follows a decreasing order. The reason is as follows: When most of the queries are cache hits, the access latency is dominated by the cache validation delay. In contrast, when the cache hit ratio is low, the access latency is dominated by the data broadcast speed. In other words, the higher the cache hit ratio, the more dominant the cache validation delay in the performance. Furthermore, for a certain cache validation delay, the influence on stretch is more significant for smaller data items (with shorter service times). As a result, since *INCRT* has more (smaller) cached data items than the other two and *DECRT* has the least (larger) cached data items, the cache validation delay has the most impact on stretch for *INCRT* and the least impact for *DECRT*.

When different cache replacement policies are considered, *Min-SAUD* performs the best in all cases. Compared with *SAIU*, *Min-SAUD* adapts to different *IR* broadcast intervals much better. For example, in the *INCRT* size setting, the performance of *Min-SAUD* degrades 98 percent when the *IR* broadcast interval is increased from 1 second to 50 seconds, whereas the stretch of *SAIU* degrades 265 percent. This convinces us of the need to integrate the cost of cache validation in a cache replacement policy.

Another observation from Fig. 5 is that the performance of the cache schemes improves first and degrades again as the *IR* broadcast interval is decreased. This is particularly true for the *RAND* size setting. This can be explained using Fig. 6. As the *IR* broadcast interval is decreased, the cache validation delay becomes shorter. Hence, the stretch for a cache-hit query improves, leading to an overall performance improvement. However, when the *IR* broadcast interval becomes smaller than a certain value (e.g., 5 seconds for *RAND*), the overhead for *IR* broadcasts becomes very high. As a result, the data retrieval delay and, hence, the stretch for a cache-miss query or an invalid cache-hit query (i.e., a cache hit but an obsolete copy) turns out to be very long. Therefore, the overall performance begins to degrade.

6.3 Experiment #3: Influence of the Item Size Ratio

This section explores the influence of item size ratio, the ratio of the maximum item size s_{max} to the minimum item size s_{min}, on the system performance. In the experiments, s_{max} is fixed to 100 KB and s_{min} varies from 100 KB to 0.1 KB,

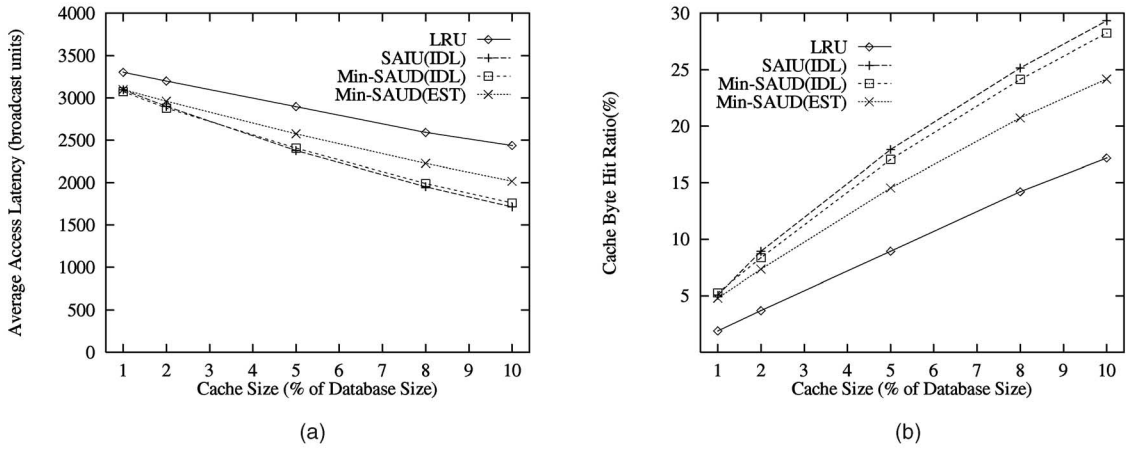


Fig. 4. Performance under various cache sizes (RAND). (a) Access latency. (b) Cache byte hit ratio.

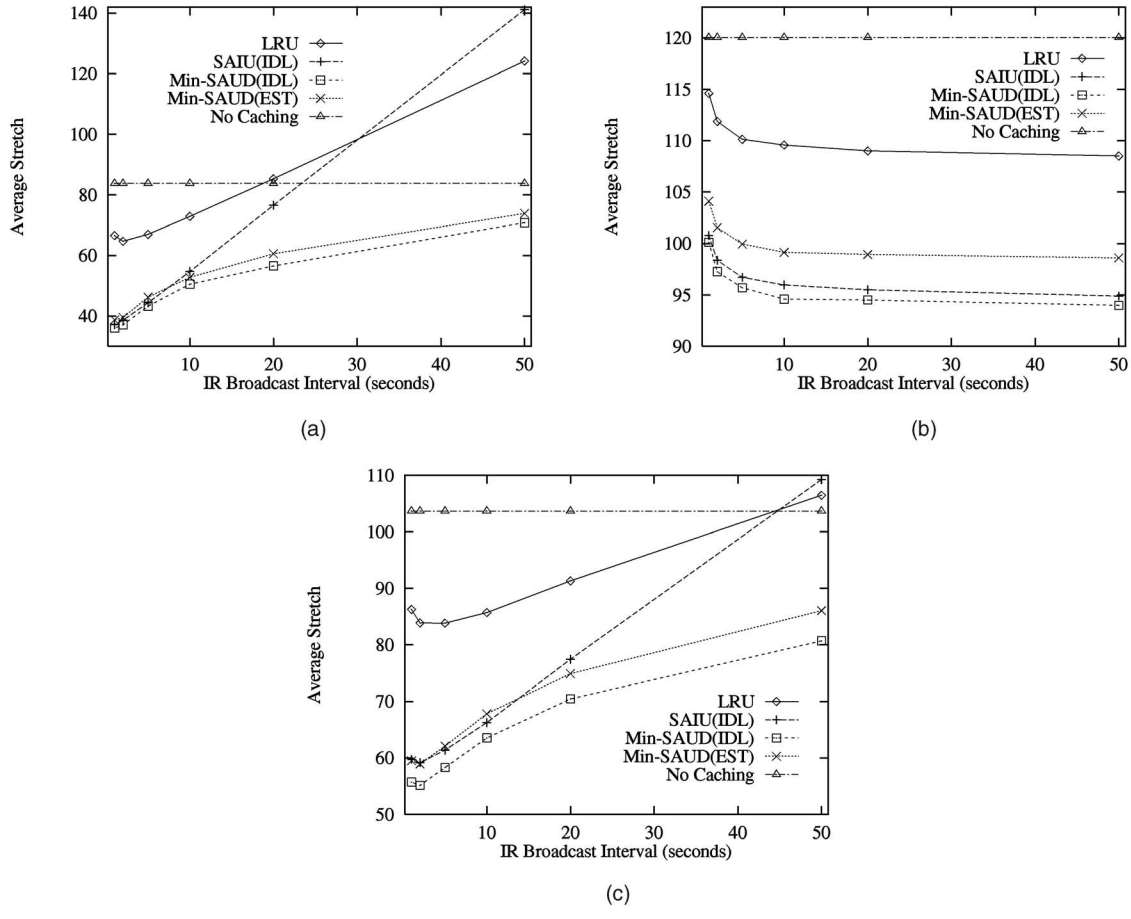


Fig. 5. Stretch performance under different *IR* broadcast intervals. (a) INCRT. (b) DECRT. (c) RAND.

i.e., the size ratio varies from 1 to 1,000. In order to make a fair comparison, when s_{min} is decreased, the cache size is reduced according to the following formula:

$$CacheSize = 0.5 \times (s_{max} + s_{min} - 1) \times DatabaseSize \times CacheSizeRatio.$$

The experimental results are shown in Fig. 7. The best performance is achieved by *Min-SAUD*. When the size settings are *INCRT* and *RAND*, the performance improvement of *Min-SAUD* over *LRU* and *SAIU* becomes prominent as the size ratio is increased. The reason is two-fold. First, in these two settings, for a larger size ratio, the frequently accessed items have smaller relative sizes to the cache size and more data items can be cached. Thus, *Min-SAUD*, as shown in Section 6.1, can more effectively make use of the cache space than the other schemes and, hence, has a greater improvement over *LRU* and *SAIU* for a larger size ratio. Second, as the size ratio is increased, since more (smaller) items are cached, the cache validation delay becomes a significant factor. As a result, a much better performance is

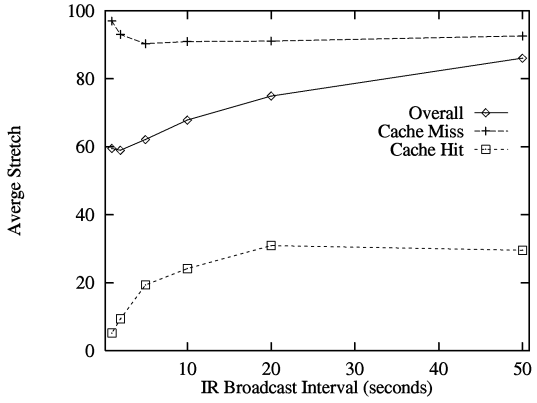


Fig. 6. Individual stretch for cache hit and cache miss (RAND, Min-SAUD(EST)).

obtained for *Min-SAUD*. However, the above phenomena cannot be observed in the *DECRT* size setting since the clients access the largest items more frequently. Consequently, its performance is almost the same for different item size ratios.

It is also observed in Fig. 7 that, for *INCRT* and *RAND*, the performance curve forms a "U" shape when the size ratio is varied. This is explained as follows: As shown in Fig. 8, when the size ratio is increased from 1 (i.e., uniform sizes) to 10 (i.e., nonuniform sizes), the cache hit ratio improves greatly since, for the case of nonuniform sizes, some smaller items are accessed more frequently. As a result, a better overall stretch is observed. However, with further increasing of the size ratio, because the sizes of the cold items become relatively larger to those of the hot items, caching a cold item will replace more hot items. This results in a worse cache hit ratio and also a worse overall stretch.

6.4 Experiment #4: Influence of the Data Access Skewness

The influence of the access skewness is evaluated in this section. Fig. 9 presents the experimental results as the θ parameter of the Zipf distribution varies from 0 to 0.95. When θ is 0, the access pattern is uniform. The larger the θ value, the more skewed the access pattern.

It is shown that *Min-SAUD* has the best stretch performance in all cases. In particular, due to reasons similar to those described in Section 6.1, *Min-SAUD* improves the performance over *LRU* and *SAIU* most greatly when θ is set to 0.95 for *INCRT*, where the smallest items are accessed frequently.

With increasing skewness, the stretch performance becomes better in most cases. This is mainly due to a higher cache hit ratio. However, there is an exception for *DECRT*. The performance degrades when θ is increased from 0 to 0.2 (see Fig. 9b). This is because, when the access

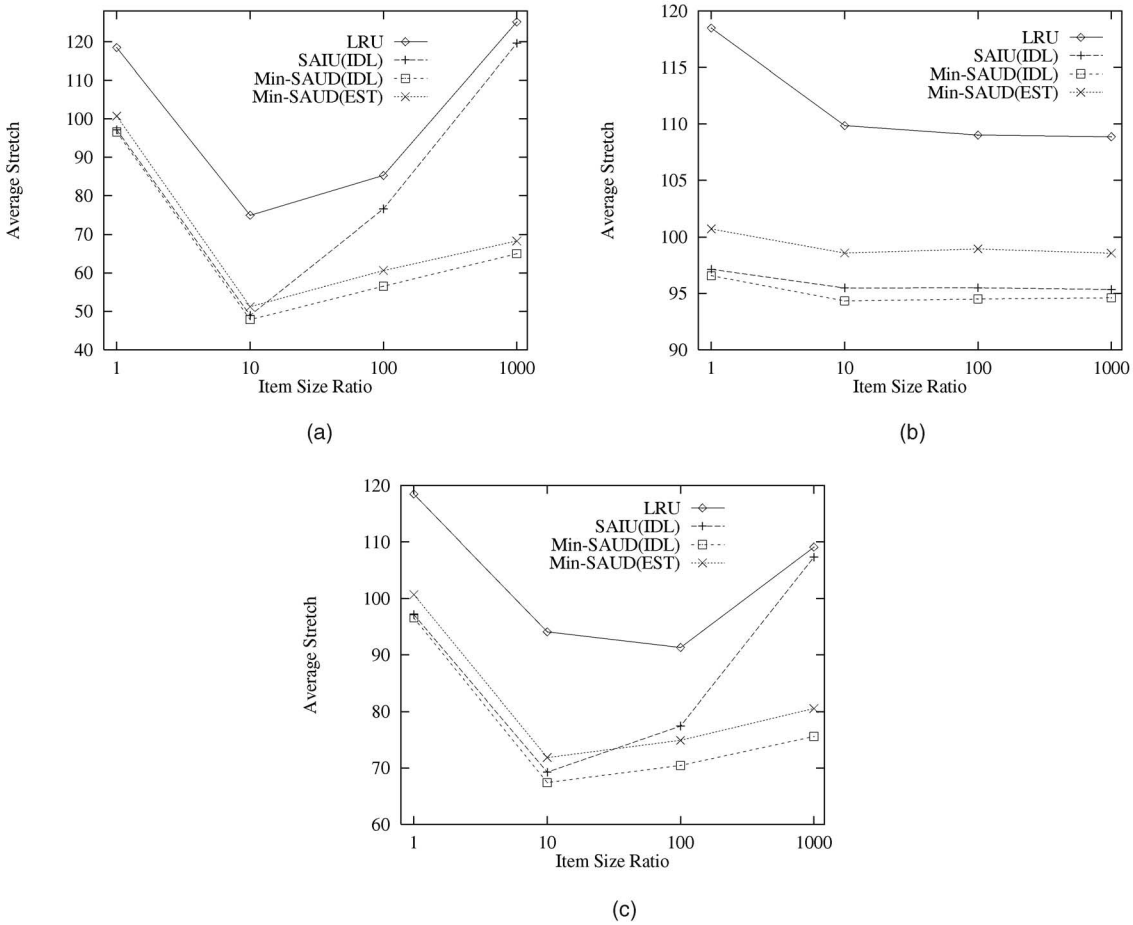


Fig. 7. Stretch performance for different item size ratios. (a) INCRT. (b) DECRT. (c) RAND.

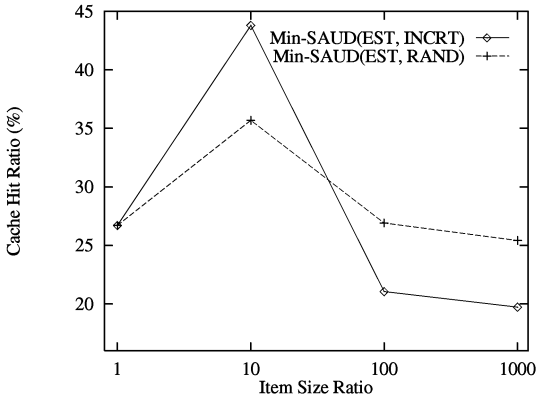


Fig. 8. Cache hit ratio under various item size ratios (Min-SAUD(EST)).

pattern changes from a uniform pattern ($\theta = 0$) to a lightly skewed pattern ($\theta = 0.2$), under the *DECRT* scheme more large items are accessed and a worse data retrieval delay is obtained. On the other hand, for a lightly skewed access pattern, the cache hit ratio cannot be improved very much. Thus, the overall access latency worsens a little bit and, hence, a slightly worse stretch performance is observed.

6.5 Experiment #5: Algorithm Complexity

We have shown in the previous few sections that *Min-SAUD*, in most cases, demonstrates a much better stretch performance than *LRU* and *SAIU*. In this section, we study the time complexity of replacement operations for the *Min-SAUD* policy. The *LRU* policy is included as a yardstick. Recall that a heap structure is used to implement *Min-SAUD* (Section 4.3). *LRU* is also implemented with a heap structure in the simulation. As can be seen, the time complexity for replacement consists of two parts: the removal of victims and the insertion of the incoming item. Thus, we approximate the time complexity by the number of item nodes that are visited in the heap for every replacement.

Fig. 10 shows the results obtained for the default system setting. The time complexity for both the average case and the worst case is measured. The worst case occurs when a very large item is to be cached (in this case, many cached items need to be removed for making room). From Fig. 10, we can see that *Min-SAUD* has only a little bit worse average complexity than *LRU*. The time complexity for *INCRT* is higher than those for *DECRT* and *RAND* in both *LRU* and *Min-SAUD*. This is because, in the *INCRT* size setting, more small data items are preferentially kept in the cache, thus the heap size is much larger than those for *DECRT* and *RAND*, which leads to worse complexity.

7 Conclusion

In this paper, we have investigated the cache replacement issue in a realistic wireless data dissemination environment

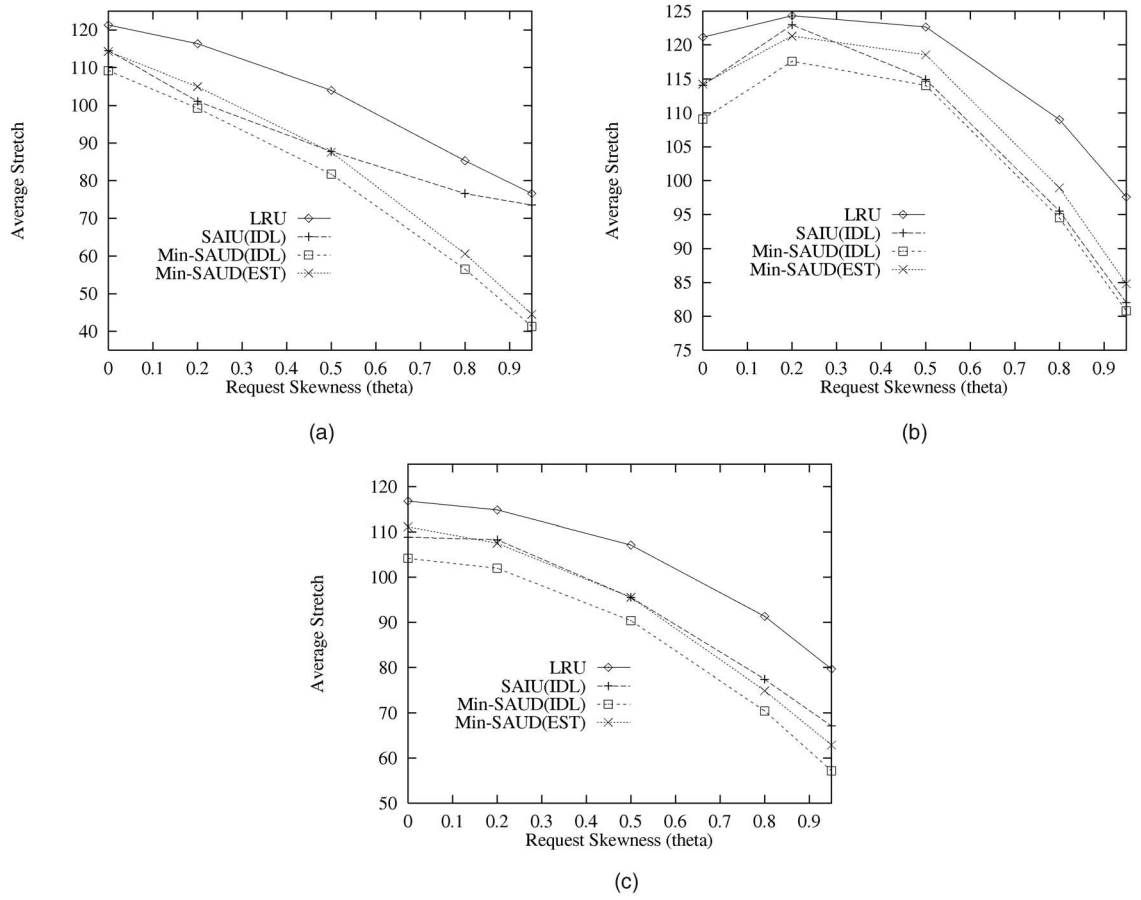


Fig. 9. Stretch performance under various levels of access skewness. (a) INCRT. (b) DECRT. (c) RAND.

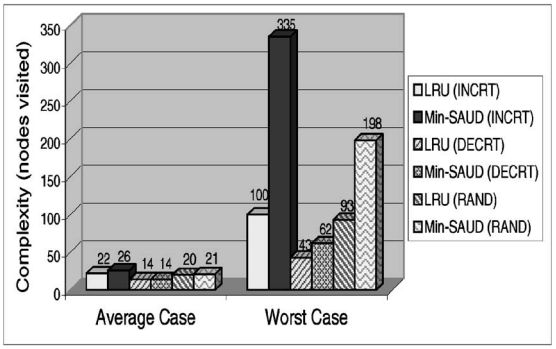


Fig. 10. Comparison of algorithm complexity.

where restrictions on data size, data update, and client disconnection imposed by most of the previous work are relieved. Moreover, unlike the existing work, we took into account the cost of cache validation in the design of a cache replacement policy under cache consistency. An optimal gain-based cache replacement policy, *Min-SAUD*, which incorporates various factors, namely, data item size, retrieval delay, access probability, update frequency, and cache validation delay, was proposed.

We showed by analysis that the stretch of *Min-SAUD* is optimal when the independent reference model and the Poisson processes of data accesses and updates are assumed. A series of simulation experiments were also conducted to evaluate the performance of *Min-SAUD*. The results demonstrated that, in most cases, *Min-SAUD* performs substantially better than the well-known *LRU* policy and the *SAIU* policy under various workloads, especially when the cache validation delay is an important concern. *Min-SAUD(EST)*, a practical realization of the *Min-SAUD* policy, showed a close performance to *Min-SAUD(IDL)*, which has perfect knowledge of access and update frequencies. Through analysis in Section 4.3.1 and Section 6.5, it is not difficult to see that the time complexity of *Min-SAUD*, $O(MlogN)$, is reasonable.

To the best of our knowledge, this is the first study that analytically studied how the various factors, such as access latency and cache validation delay, affect cache performance. The analysis serves as the basic guideline for the design of cache management strategies. In this paper, we have employed the stretch as the major performance measure. On the other hand, we can see that the proposed technique can be easily extended to optimize *Min-SAUD* under other metrics such as access latency and cache hit ratio. For example, to optimize access latency, we can simply revise the gain function as

$$gain(i) = p_i\left(\frac{b_i}{1+x_i} - v\right);$$

to optimize cache hit ratio and byte hit ratio, the gain function is revised as $gain(i) = \frac{p_i}{1+x_i}$ and $\frac{p_i s_i}{1+x_i}$, respectively. Performance evaluation of these policies can be done in a similar manner.

While this study was performed in the context of wireless data dissemination, it is obvious that the analytical study can be applied to data caching on remote clients under the cache consistency requirement. Along with studies on cache consistency for the Web [14], this study can be applied to Web client caching or Web proxy caching.

As part of future work, we also plan to extend the cache replacement policy to a cache admission policy for client data caching. As shown in the simulation results, there is still room for improving the parameter estimate methods. If better estimation methods can be proposed, the performance of the practical *Min-SAUD* policy will be further improved toward that of the ideal policy. It would be an interesting topic to combine the prefetching technique into the current scheme.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions that improved the quality of this paper. The research was supported by the Research Grant Council, Hong Kong SAR, China, under grant numbers HKUST-6241/00E and HKUST6079/01E.

REFERENCES

[1] M. Abrams, C.R. Standridge, G. Abdulla, S. Williams, and E. Fox, "Caching Proxies: Limitations and Potentials," *Proc. Fourth Int'l World Wide Web Conf. (WWW4)*, pp. 119-133, Dec. 1995.

[2] S. Acharya, "Broadcast Disks: Dissemination-Based Data Management for Asymmetric Communication Environments," PhD dissertation, Brown Univ., May 1998.

[3] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik, "Broadcast Disks: Data Management for Asymmetric Communications Environments," *Proc. ACM SIGMOD Conf. Management of Data*, pp. 199-210, May 1995.

[4] S. Acharya, M. Franklin, and S. Zdonik, "Prefetching from a Broadcast Disk," *Proc. 12th Int'l Conf. Data Eng. (ICDE '96)*, pp. 276-285, Feb. 1996.

[5] S. Acharya and S. Muthukrishnan, "Scheduling On-Demand Broadcasts: New Metrics and Algorithms," *Proc. Fourth Ann. ACM/IEEE Int'l Conf. Mobile Computing and Networking (MobiCom '98)*, pp. 43-54, Oct. 1998.

[6] C. Aggarwal, J.L. Wolf, and P.S. Yu, "Caching on the World Wide Web," *IEEE Trans. Knowledge and Data Eng.*, vol. 11, no. 1, pp. 94-107, Jan./Feb. 1999.

[7] D. Aksoy and M.J. Franklin, "RxW: A Scheduling Approach for Large-Scale On-Demand Data Broadcast," *ACM/IEEE Trans. Networking*, vol. 7, no. 6, pp. 846-860, 1999.

[8] D. Aksoy, M.J. Franklin, and S. Zdonik, "Data Staging for On-Demand Broadcast," *Proc. 27th VLDB Conf. (VLDB '01)*, Sept. 2001.

[9] D. Barbara and T. Imielinksi, "Sleepers and Workaholics: Caching Strategies for Mobile Environments," *Proc. ACM SIGMOD Conf. Management of Data*, pp. 1-12, May 1994. An extended version appeared in *VLDB J.*, vol. 4, no. 4, pp. 567-602, 1995.

[10] J. Bolot and P. Hoschka, "Performance Engineering of the World Wide Web: Application to Dimensioning and Cache Design," *Proc. Fifth Int'l World Wide Web Conf. (WWW5)*, May 1996.

[11] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web Caching and Zipf-Like Distributions: Evidence and Implications," *Proc. IEEE INFOCOM '99*, pp. 126-134, Mar. 1999.

[12] J. Cai and K.-L. Tan, "Energy-Efficient Selective Cache Invalidation," *ACM/Baltzer J. Wireless Networks (WINET)*, vol. 5, no. 6, pp. 489-502, 1999.

[13] G. Cao, "A Scalable Low-Latency Cache Invalidation Strategy for Mobile Environments," *IEEE Trans. Knowledge and Data Eng.*, vol. 15, no. 5, Sept./Oct. 2003.

[14] P. Cao and C. Liu, "Maintaining Strong Cache Consistency in the World-Wide Web," *IEEE Trans. Computers*, vol. 47, no. 4, pp. 445-457, Apr. 1998.

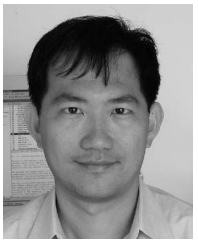
[15] B.Y.L. Chan, A. Si, and H.V. Leong, "Cache Management for Mobile Databases: Design and Evaluation," *Proc. 14th Int'l Conf. Data Eng. (ICDE '98)*, pp. 54-63, Feb. 1998.

[16] C.-Y. Chang and M.-S. Chen, "Exploring Aggregate Effect with Weighted Transcoding Graphs for Efficient Cache Replacement in Transcoding Proxies," *Proc. 18th IEEE Int'l Conf. Data Eng. (ICDE '02)*, pp. 383-392, 2002.

[17] E.G. Coffman Jr. and P.J. Denning, *Operating Systems Theory.* Prentice Hall, 1973.
[18] C.C.F. Fong, J.C.S. Lui, and M.H. Wong, "Quantifying Complexity and Performance Gains of Distributed Caching in a Wireless Network Environment," *Proc. 13th Int'l Conf. Data Eng. (ICDE '97),* pp. 104-113, Oct. 1997.
[19] S. Hosseini-Khayat, "On Optimal Replacement of Nonuniform Cache Objects," *IEEE Trans. Computers,* vol. 49, no. 8, pp. 769-778, Aug. 2000.
[20] Q.L. Hu and D.L. Lee, "Cache Algorithms Based on Adaptive Invalidation Reports for Mobile Environments," *Cluster Computing,* vol. 1, no. 1, pp. 39-48, Feb. 1998.
[21] T. Imielinski and B.R. Badrinath, "Wireless Mobile Computing: Challenges in Data Management," *Comm. ACM,* vol. 37, no. 10, pp. 18-28, 1994.
[22] R. Jain, *The Art of Computer Systems Performance Analysis.* New York: John Wiley & Sons, 1991.
[23] J. Jing, A.K. Elmagarmid, A. Helal, and R. Alonso, "Bit-Sequences: A New Cache Invalidation Method in Mobile Environments," *ACM/Baltzer J. Mobile Networks and Applications,* vol. 2, no. 2, pp. 115-127, 1997.
[24] A. Kahol, S. Khurana, S.K.S. Gupta, and P.K. Srimani, "A Strategy to Manage Cache Consistency in a Distributed Mobile Wireless Environment," *IEEE Trans. Parallel and Distributed Systems,* vol. 12, no. 7, pp. 686-700, July 2001.
[25] S. Khanna and V. Liberatore, "On Broadcast Disk Paging," *SIAM J. Computing,* vol. 29, no. 5, pp. 1683-1702, 2000.
[26] P. Scheuermann, J. Shim, and R. Vingralek, "WATCHMAN: A Data Warehouse Intelligent Cache Manager," *Proc. 22nd VLDB Conf.,* pp. 51-62, Sept. 1996.
[27] H. Schwetman, *CSIM User's Guide (version 18).* MCC Corporation, http://www.mesquite.com, 1998.
[28] J. Shim, P. Scheuermann, and R. Vingralek, "Proxy Cache Design: Algorithms, Implementation and Performance," *IEEE Trans. Knowledge and Data Eng.,* vol. 11, no. 4, pp. 549-562, July/Aug. 1999.
[29] K.L. Tan, J. Cai, and B.C. Ooi, "An Evaluation of Cache Invalidation Strategies in Wireless Environments," *IEEE Trans. Parallel and Distributed Systems,* vol. 12, no. 8, pp. 789-807, Aug. 2001.
[30] L. Tassiulas and C.J. Su, "Optimal Memory Management Strategies for a Mobile User in a Broadcast Data Delivery System," *IEEE J. Selected Areas in Comm.,* vol. 15, no. 7, pp. 1226-1238, Sept. 1997.
[31] J. Wang, "A Survey of Web Caching Schemes for the Internet," *ACM Computer Comm. Rev.,* vol. 29, no. 5, pp. 36-46, Oct. 1999.
[32] K.-L. Wu, P.S. Yu, and M.-S. Chen, " Energy-Efficient Caching for Wireless Mobile Computing," *Proc. 12th Int'l Conf. Data Eng.,* pp. 336-343, Feb. 1996.
[33] J. Xu, Q.L. Hu, and D.L. Lee, W.-C. Lee, "SAIU: An Efficient Cache Replacement Policy for Wireless On-Demand Broadcasts," *Proc. Ninth ACM Int'l Conf. Information and Knowledge Management,* pp. 46-53, Nov. 2000.
[34] J. Xu, X. Tang, and D.L. Lee, "Performance Analysis of Location-Dependent Cache Invalidation Schemes for Mobile Environments," *IEEE Trans. Knowledge and Data Eng.,* vol. 15, no. 2, pp. 474-488, Mar./Apr. 2003.
[35] J. Yuen, E. Chan, K.Y. Lam, and H.W. Leung, "Cache Invalidation Scheme for Mobile Computing Systems with Real-Time Data," *ACM SIGMOD Record,* vol. 29, no. 4, pp. 34-39, 2000.
[36] G.K. Zipf, *Human Behaviour and the Principle of Least Effort.* Addison-Wesley, 1949.

Jianliang Xu received the BEng degree in computer science and engineering from Zhejiang University, Hangzhou, China, in 1998, and the PhD degree in computer science from Hong Kong University of Science and Technology in 2002. He is an assistant professor in the Department of Computer Science at Hong Kong Baptist University. His research interests include mobile/pervasive computing, location-aware computing, Internet technologies, and wireless networks. He has served as a program committee member and an executive committee member for several international conferences, and is currently serving on the program committees of the IEEE INFOCOM '04, IEEE MDM '04, and ACM SAC '04. He is also serving as an executive committe member of the ACM Hong Kong chapter. He is a member of the IEEE.

Qinglong Hu received the BS and the MS degrees in computer science from East China Normal University, Shanghai, China, in 1986 and 1989, respectively, and the PhD degree in computer science from the Hong Kong University of Science and Technology in January 1999. He is an advisory software engineer at the Database Technology Institute of IBM Silicon Valley Laboratories, San Jose, California. Currently, his research interests include mobile and pervasive computing, wireless networks, and distributed systems.



Wang-Chien Lee received the PhD degree in computer and information science from Ohio State University. He is an associate professor in the Computer Science and Engineering Department at Pennsylvania State University. His primary research interests lie in the areas of mobile and pervasive computing, data management, and Internet technologies. He has guest-edited special issues on mobile database related topics for several journals, including *IEEE Transactions on Computer*, *IEEE Personal Communications Magazine*, *ACM WINET*, and *ACM MONET*. He was the program committee cochair for the First International Conference on Mobile Data Access (MDA '99) and the International Workshop on Pervasive Computing (PC 2000). He has also been a panelist, session chair, industry chair, and program committee member for various symposia, workshops, and conferences. He is a member of the IEEE and IEEE Communications Society and also the ACM.

Dik Lun Lee received the MS and PhD degrees in computer science from the University of Toronto in 1981 and 1985, respectively. He is a professor in the Department of Computer Science at the Hong Kong University of Science and Technology and was an associate professor in the Department of Computer and Information Science at Ohio State University, Columbus. He has served as a guest editor for several special issues on database-related topics, and as a program committee member and chair for numerous international conferences. He was the founding conference chair for the International Conference on Mobile Data Management. His research interests include document retrieval and management, discovery, management and integration of information resources on the Internet, and mobile and pervasive computing. He was the chairman of the ACM Hong Kong Chapter.