Adaptive Multiversion Data Broadcast Organizations

Oleg Shigiltchoff[1], Panos K. Chrysanthis[1] and Evaggelia Pitoura[2]

[1] Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260, USA
{oleg,panos}@cs.pitt.edu

[2] Department of Computer Science
University of Ioannina
GR 45110 Ioannina, Greece
pitoura@cs.uoi.gr

Abstract. Broadcasting provides an efficient means for disseminating information in both wired and wireless setting. In this paper, we propose a suite of broadcast organization schemes for multiversion data broadcast, i.e., data broadcast in which more than one value is broadcast per data item. Besides increasing the concurrency of client transactions, multiversion broadcast provides clients with the possibility of accessing multiple server states. For example, such a functionality is essential to support applications that require access to data sequences and have limited local memory to store the previous versions, such as in the case of sensor networks. We identify two basic multiversion organizations, namely vertical and horizontal broadcasts and propose an efficient compression scheme applicable to both. We also consider a multiversion client data cache and introduce appropriate cache replacement techniques. Finally, we propose an adaptive scheme that dynamically selects the appropriate broadcast organization based on the client access pattern. We provide performance evaluation results for both flat and broadcast disk organizations and for a variety of client query patterns.

1 Introduction

A major problem on the Internet is the scalable dissemination of information. This problem is particularly acute with the presences of mobile users. The traditional unicast pull framework simply does not scale up and it is not suitable for mobile devices due to their inherent resource limitations including power of mobile devices and the capacity of the wireless links. A solution to this scalability problem is to use multicast communication for both wireline and wireless devices. In particular, in the context of wireless and mobile environments, *broadcast push* [2] takes both the communication and energy limitations into account, exploiting the asymmetry in wireless communication and the reduced energy

consumption in the receiving mode. Servers have both much larger bandwidth available than client devices and more power to transmit large amounts of data.

In broadcast push, the server repeatedly sends information to a client population without explicit client requests. Clients monitor the broadcast channel and retrieve the data items they need as they arrive on the broadcast channel. Applications of broadcast push typically involve a small number of servers and a much larger number of clients with similar interests. Examples include stock trading, electronic commerce applications, such as auctions and electronic trading, and traffic control information systems. Any number of clients can monitor the broadcast channel. If broadcast data is organized according to the clients' interests, such a scheme makes an effective use of the low wireless bandwidth. Besides wireless settings, where there is physical support for broadcast, broadcast push provides a scalable means to disseminate data in web environments as well.

To reduce access time, clients may cache data of interest locally. However, in the case of data updates, the problem arises of keeping the data cached in the client consistent with the updated data on the server [28, 37, 20]. The same problem also exists in the context of broadcast push, even without client caching. Broadcasting is a form of a cache "on the air".

In our previous work, we proposed maintaining multiple versions of data items on the broadcast channel as well as in the client cache for concurrency control purposes [26] (similarly to multiversion schemes in traditional unicast pull models, e.g., [23]). With multiple versions, more client transactions read consistent data (i.e., data values that belong to the same server database state) and complete their operation successfully. The time overhead induced by the multiple versions is smaller than the overall time lost for aborts and subsequent recoveries in the absence of multiple versions [26]. Furthermore, multiple versions increase client's tolerance to network disconnections that are common in wireless communications [24].

Besides increasing the concurrency of client transactions and their tolerance to disconnections, multiversion broadcast provides clients with the possibility of accessing multiple server states. For example, such a functionality is essential to support applications that require access to data sequences and have limited local memory to store the previous versions as is the case with data streams.

The performance (characterized by the client access time and power consumption) of multiversion broadcast is directly related to the way data are organized in the the broadcast. The main contributions of this paper are:

1. The introduction of two different basic broadcast organizations for multiversion broadcast, namely *Vertical* and *Horizontal*.
2. The development of a compression scheme along the lines of *Run Length Encoding* (RLE) [32], applicable to both of the proposed multiversion broadcast organizations. Our compression scheme exploits the fact that adjacent versions of an item may have the same value and reduces the broadcast size by not explicitly sending unchanged parts of older versions.

3. The proposal of different cache replacement policies for multiversion broadcast that consider versions and apply our compression scheme to maximize the number of hits.
4. The evaluation of the conditions under which each of the proposed broadcast organizations performs better, and the introduction of an adaptive broadcast organization scheme that improves performance in dynamically changing environments.

The remainder of this paper is structured as follows. In Section 2, we present the system model. Section 3 describes different multiversion broadcast organizations for both single and multiple disk organizations as well as our compression scheme. Section 4 evaluates client side cache. Adaptive broadcast is introduced in Section 5. Section 6 presents our experimental platform, while our experimental results are discussed in Section 7. Section 8 discusses related work, and Section 9 concludes the paper.

2 System Model

In a broadcast dissemination environment, a data server periodically broadcasts data items to a large client population. Each period of the broadcast is called a *broadcast cycle* or *bcycle*, while the content of the broadcast is called a *bcast*. Each client listens to the broadcast and fetches data as they arrive. In this way, data can be accessed concurrently by any number of clients without any performance degradation (compared to a "pull" or on-demand approach). However, access to data is strictly sequential, since clients need to wait for the data of interest to appear on the channel. We assume that all updates are performed at the server and disseminated from there.

Without loss of generality, in this paper, we consider a model in which the bcast disseminates a fixed number of data items. However, the data values (values of the data items) may or may not change between two consecutive bcycles. In multiversion broadcast, the server maintains multiple versions of each data item and broadcasts a fixed number of versions for each data item. At each cycle, the oldest version of the data is discarded and a new, the most recent, version is included in the broadcast. The number m of older versions that are retained can be seen as a property of the server: an m-multiversion server is a server that broadcasts m values of each item. In terms of client transaction consistency, an m-multiversion server guarantees the consistency of all transactions with span m or smaller [26]. The *span* of a client transaction T, is defined to be the maximum number of different bcycles from which T reads data.

The client listens to the broadcast and searches for data elements based on the pair of values (data id and version number). Clients do not need to listen to the broadcast continuously. Instead, they tune-in to read specific items. Such selective tuning is important especially in the case of portable mobile computers, since they most often rely for their operation on the finite energy provided by batteries and listening to the broadcast consumes energy. Indexing has been used

	Vno = 3	Vno = 2	Vno = 1	Vno = 0
Did = 3	Dval = 1	Dval = 1	Dval = 1	Dval = 1
Did = 5	Dval = 8	Dval = 8	Dval = 8	Dval = 5
Did = 8	Dval = 6	Dval = 1	Dval = 1	Dval = 2
Did = 9	Dval = 5	Dval = 4	Dval = 4	Dval = 4

Fig. 1. Example of an array representing a sequence of four versions of four data items.

to support selective tuning and reduce power consumption, often at the cost of access time. In this paper, we focus only on broadcast organization and how to reduce its size without adopting any indexing scheme.

The logical unit of a broadcast is called a *bucket*. Buckets are the analog to blocks for disks. Each bucket has a header that includes useful information. The exact content of the bucket header depends on the specific broadcast organization. Information in the header usually includes the position of the bucket in the bcast as an offset time step from the beginning of the broadcast as well as the offset to the beginning of the next broadcast.

The main question in multiversion broadcast is how to organize the broadcast, that is, where to place the new and the old versions. In the next section, we elaborate on this issue, considering in addition broadcast compression as a method to reduce the size of the broadcast.

3 Multiversion Broadcast Organization

3.1 Basic Organization

A set of multiversion data to be broadcast can be represented as a two dimensional array, where dimensions correspond to broadcast version numbers (*Vno*) and data ids (*Did*), and the array elements are the data values (*Dval*) of the items. That is, $Dval[i, k] = v$ means that the k-version of the i-data item is equal to v. Data items can appear in any order. Versions appear in descending order with the most recent version appearing in the left most column and the oldest version in the right most column. This data representation can be extended to any number of data items and versions.

A simple sequential broadcast can be generated by linearizing the two dimensional array in two different ways: horizontally or vertically. In *Horizontal broadcast*, a server broadcasts all versions (with different *Vno*) of a data item with a particular *Did*, then all versions (with different *Vno*) of the next data item with the next *Did* and so on. In *Vertical broadcast*, a server broadcasts all data items (with different *Did*) having a particular *Vno*, then all data items (with different *Did*) having the next *Vno* and so on. Formally, the Horizontal broadcast transmits $[Did[Vno,Dval]^*]^*$ sequences, whereas the Vertical broadcast transmits $[Vno[Did,Dval]^*]^*$ sequences.

Consider for example, the two dimensional array of Figure 1 representing four data items selected to be broadcast with *Did*={3, 5, 8, 9}, each having four

versions with Vno=3 being the most recent one. For the Horizontal broadcast, the data values appear on the bcast in the following order (The complete bcast will include also the data ids and version numbers as indicated above):

1 1 1 1 8 8 8 5 6 1 1 2 5 4 4 4

while for the Vertical broadcast the data values on the bcast are placed in the following order:

1 8 6 5 1 8 1 4 1 8 1 4 1 5 2 4

The resulting bcasts have the same size for both organizations, but differ in the order in which they broadcast the data values.

The broadcast order affects the client access time. For instance, say a client that tunes in at beginning of our example bcast wants to access Vno=3 of data element with Did=9. In the Vertical broadcast, the client waits time equal to the time for broadcasting 3 data items before the item of interest appears in the bcast, while in the Horizontal broadcast, it waits time equal to the time for broadcasting 12 data items. We expect that different strategies are more appropriate for different applications. If users require different versions of a particular data (for example, the history of a stock index change), Horizontal broadcast is preferable. If users need the most recent data (for example, the current stock indexes), Vertical broadcast is expected to be more efficient. Our experimental results in Section 8 verify this intuition.

3.2 Compressed Organization

In both cases, Horizontal and Vertical, the broadcast size and consequently the access time can be reduced by using some compression scheme. A good compression scheme should reduce the broadcast as much as possible with minimal, if any, impact on the client. That is, it should not require additional processing at the client, so it should not trade access time for processing time. The following is a simple compression scheme that exhibits the above properties.

Our compression scheme is based on the observation that data values do not always change from one version to another. In other words, $Dval[i, k] = Dval[i, k\text{-}1] = \ldots = Dval[i, k\text{-}N]$=$v$, where the value of the item having $Did = i$ remains equal to v for N consequent versions. Then, when broadcasting data, there is no reason to broadcast all versions of a data item if its $Dval$ does not change. Instead, the compressed scheme broadcasts a $Dval$ only if it is different from the $Dval$ of the previous version. In order not to lose information (as well as to support selective tuning), it also broadcasts the number of versions having the same $Dval$.

The generation of a compressed broadcast proceeds in two steps. In the first step, the compressed values of the data elements are produced and in the second phase, these values are broadcast based on the selected organization. In the first step, any sequence of elements with repetitive data values is replaced by the first element of the sequence and the length of the sequence.

Did = 3	1 x 3			
Did = 5	8 x 2			5
Did = 8	6	1 x 1		2
Did = 9	5	4 x 2		

Fig. 2. The example array after the first step of the compression.

The array of Figure 1 is redrawn in Figure 2 to capture the first step of the compression algorithm. For example, 1 1 1 1 becomes 1x3 and 8 8 8 becomes 8x2.

In the second step of the compression, the Horizontal and Vertical broadcasts are produced by using horizontal and vertical linearization of the array respectively. Formally, the Horizontal broadcast produces [*Did*[*Vno(Dval x Repetition)*]*]* sequences, while the Vertical broadcast creates [*Vno*[*Did(Dval x Repetition)*]*]* sequences. Obviously we do not include into the broadcast those versions, which already have been included "implicitly" with other versions.

In the example of Figure 2, the compressed Horizontal broadcast produced in the second step is:

1x3 8x2 5 6 1x1 2 5 4x2

and the Vertical:

1x3 8x2 6 5 1x1 4x2 5 2.

In the case of the Vertical broadcast, it also makes sense to rearrange the sequence of broadcast data elements within a single-version sweep and make them dependent not on *Did* but on the number of implicitly broadcast elements. Applying this reordering to our example, the resulting Vertical broadcast is:

1x3 8x2 6 5 4x2 1x1 5 2

In this example, 4x2 and 1x1 switch their positions, because we broadcast implicitly two 4s and only one 1. The idea is that we broadcast first as "dense" data as possible, because when a client begins to read the broadcast, the client has higher chances to find the necessary data elements in "more dense" data. Of course, this optimization works under the assumption that client access data uniformly.

For providing a rough estimation of the reduction of the broadcast size (and, consequently, of the duration of the bcycle) due to our compression scheme, let us introduce *Randomness* and *Randomness Degree* to represent the repetitiveness of data from one version to the other. Let $Randomness[i,k] = 0$ if $Dval[i,k] = Dval[i,k-1]$ and $Randomness[i,k] = 1$ otherwise. Then, the average randomness *Randomness Degree[i]* of the data element i over all its versions represents how frequently the value of this element changes. We can use this average randomness as a parameter describing the probability that $Dval[i,m]$ is not equal to

$Dval[i, m-1]$. For instance, $RandomnessDegree[i] = 0$ means that $Dval[i, m] = Dval[i, m-1]$, for any m. The smaller the degree of randomness, the higher the gain of our compression scheme. Hence, we can expect that the broadcast of data having many "static" elements (for example, a cartoon clip with one-color background or a stock index of infrequently traded companies, etc.) improves the "density" of broadcast data.

As an example, consider the broadcast of data item j assuming *Randomness Degree*$[j] = 0.1$. Then, in average out of 100 versions we have 10 versions with values different from the values of the previous versions and 90 versions repeating their values. This means that instead of broadcasting 100 data values we broadcast only 10. We can roughly estimate that overhead created by the auxiliary symbols will not exceed 1 symbol per "saved" data item from the broadcast. Assuming, one data item consumes 16 bytes and one auxiliary symbol consumes 1 byte, the gain is 100*16/(10*16+90*1) = 6.4, which corresponds to a 84% reduction of the broadcast length. Similarly, the broadcast shrinks about 45%, for *Randomness Degree*=0.5 and about 9%, for *Randomness Degree*=0.9. This reduction holds for both Vertical and Horizontal broadcast.

3.3 Broadcast Encoding

In order to make our broadcast fully self-descriptive, we add all necessary information about version numbers and data items. One of our design principles has been to make the system flexible, allowing a client to understand the content of a broadcast without requiring the client explicitly to be told of the organization of the broadcast. For this purpose, we use four auxiliary symbols:

(*Did*), **V** (*Vno*), **=** (Assignment to *Dval*), ^ (Number of repetitions)

Using these symbols, the *uncompressed beast* for Horizontal broadcast in our example is fully encoded as:

V3#3=1V2#3=1V1#3=1V0#3=1V3#5=8V2#5=8V1#5=8V0#5=5
V3#8=6V2#8=1V1#8=1V0#8=2V3#9=5V2#9=4V1#9=4V0#9=4

and for Vertical broadcast

V3#3=1#5=8#8=6#9=5V2#3=1#5=8#8=1#9=4V1#3=1#5=8#8
=1#9=4V0#3=1#5=5#8=2#9=4

V3,V2,V1 and V0 are the version numbers. They determine *Vno* of the data elements which follows it in the broadcast. #3=1 means the element having *Did*=3 of the corresponding version (broadcast before) is equal to 1. So, V3#3=1#5=8 means $Dval[Did{=}3,\ Vno{=}3\ ]{=}1$ and $Dval[Did\ {=}5, Vno{=}3]{=}8$. Note that for Vertical broadcast we do not need to include the version number in the broadcast before each data element, but for Horizontal broadcast we have to do this. Because of this need of some extra auxiliary symbols, a Horizontal broadcast is

usually longer than its corresponding Vertical broadcast. However, given that the size of an auxiliary symbol is much smaller (which is typically the case) than the size of a data element, this difference in length becomes very small.

In the case of *compressed bcast*, the symbol ^ is used to specify that the following versions of a data item have the same value. The other auxiliary symbols are also used to give a client the complete information about *Did*, *Vno*, and *Dval* in a uniform format for both the compressed and uncompressed multiversion broadcast organizations. Returning to our example broadcasts, the compressed Horizontal broadcast is encoded as:

V3^3#3=1V2V1V0V3^2#5=8V2V1V0^0#8=5V3^0#8=6V2^1#8=1V1
V0^0#8=2V3^0#9=5V2^2#9=4V1V0

whereas the compressed Vertical broadcast as:

V3^3#3=1^2#5=8^0#8=6#9=5V2^2#9=4^1#8=1V1V0^0#5=5#8=2

Considering the Vertical bcast as an example, let us clarify some details of the broadcast. It starts from the version 0. First, it broadcasts the data elements with the most repetitive versions. V3^3#3=1^2#5=8^0#8=6#9=5 means that versions 3, 2, 1, 0 of data element 3 are 1, versions 3, 2, 1 of data element 5 are 8, version 3 of data element 8 is 6, version 3 of data element 9 is 5. V2^2#9=4^1#8=1 means that versions 2, 1, 0 of data element 9 are 4 and versions 2, 1 of data element 8 are 1. We do not broadcast versions 2 of data elements 3 and 5 because we broadcast them together with versions 3.

3.4 Broadcast Disk Organization

In this section, before we describe how horizontal and vertical organizations can be used with broadcast disks, we briefly review the basic idea of broadcast disks using an example; for a complete definition of broadcast disks refer to [5]. In a broadcast disk organization, the items of the broadcast are divided in ranges of similar access probabilities. Each of these ranges is placed on a separate disk. In the example of Figure 3, the buckets of the first disk $Disk_1$ are broadcast three times as often as those in the second disk $Disk_2$. To achieve these relative frequencies, the disks are split into smaller equal sized units called chunks; the number of chunks per disk is inversely proportional to the relative frequencies of the disks. In the example, the number of chunks is 1 (chunk 1) and 3 (chunks 2a, 2b and 2c) for $Disk_1$ and $Disk_2$ respectively. Each bcast is generated by broadcasting one chunk from each disk and cycling through all the chunks sequentially over all disks. A minor cycle is a sub-bcycle that consists of one chunk from each disk. In the example of Figure 3, there are three minor cycles.

A direct application of the broadcast disks on multiversion broadcast is to base the distribution of each $Dval[i,k]$ on its access probability. Then, the $Dval[i,k]$ sequences inside each bucket or inside each minor cycle can follow either a Horizontal or a Vertical organization. However, this approach requires

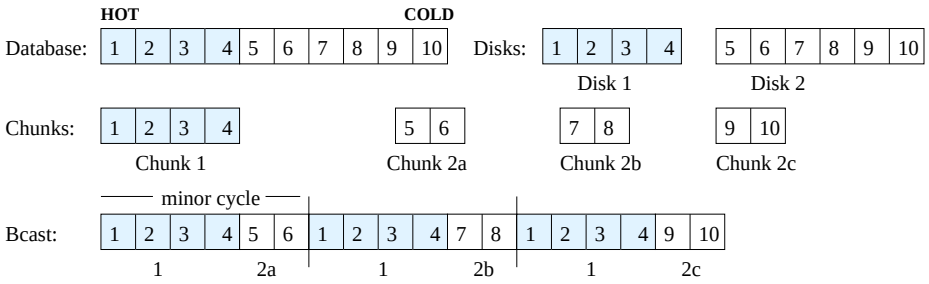

Fig. 3. Broadcast disk organization.

knowing the access probabilities of each version of each data item. This assumption is rather unrealistic in many settings. We describe next three "heuristic" approaches that differ on the amount of information about the client access pattern that is required at the server side.

Assume for simplicity that we have only two disks: a "fast" and a "slow" disk. One approach is to determine the *Did*s of the frequently accessed data elements and place all versions of these data items on the "fast" disk. All remaining data items are placed on the "slow" disk. Within each disk, data are organized either horizontally or vertically. This first approach is easily extended to multiple disks.

The second approach builds on the assumption that the most recent versions of items are usually the most frequently requested ones and places the most recent version of all data items on the "fast" disk. Clearly, this means that the data elements on the "fast" disk are vertically organized since they all belong to the same version. All previous versions of the data items are placed in the "slow" disk and can be organized either horizontally or vertically. This second approach can also be generalized for multiple disks, by associating different "temperatures" (degrees of popularity) to different versions. For example, in the case of a 7-multiversion server and three disks, "fast", "medium-speed" and "slow", the most recent versions of all data items are placed on the "fast" disk, the previous two most recent versions of all the data items are placed on the "medium-speed" disk and the remaining four oldest versions of all the data items are placed on the "slow" disk.

A third approach, that in some sense combines the above two ones, again uses a third "medium-speed" disk having speed between the "fast" and the "slow" disks. The approach distinguishes the *Did* of the data items based on their popularity. It places the most recent version of the hot (i.e., most frequently accessed) items on the "fast" disk, using a vertical organization. The other versions of the most frequently accessed items are placed on the "medium-speed" disk along with the most recent version of the remaining cold data items. Finally, the remaining versions of the cold data items are placed on the "slow" disk. Again this approach can be easily generalized for multiple disks and both horizontal and vertical organization can be adopted for each broadcast disk.

4 Client Side Cache

To improve performance, clients may cache items of interest locally. Caching reduces the latency in answering queries and the need to access the broadcast channel for every data item. In this section, we describe how caching can be used in conjunction with multiversion broadcast.

The simplest way to integrate caching with multiversion broadcast is to assume that data elements are the units of caching and adopt the traditional *Least Recently Used (LRU)* replacement method to discard elements from the cache when the cache runs out of space. Each cache entry has three basic fields for each multiversion data element: *Did*, *Vno*, and *Dval* and a fourth field: t, which shows at which broadcast cycle the element was used the last time. Following the LRU policy, when the cache becomes full, the cache element with the oldest (i.e., smallest) t is selected for replacement.

Compression can be applied to the caching content as well. Our compression scheme requires that each cache entry has one additional field f, which contains information of how many adjacent versions of this data element have the same *Dval*. Thus, each entry in the compressed LRU scheme is of the form $[Did, Vno, Dval, t, f]$. For example, a cache entry $[i, k, u, t, 2]$ indicates that the k, $k-1$, and $k-2$ versions of item i have the same value u. In the case of compressed broadcast, data elements can be directly cached. In the case of uncompressed broadcast, data elements need to be combined with the corresponding already cached elements.

In [31], we considered a number of variations of compressed-LRU. Our results showed that the compressed-LRU variant with autoprefetch exhibits the best performance with respect to client's perceived latency. With autoprefetching, a client acts pro-actively by prefetching newest version of data that have already being cached. This is a reasonable choice, when energy is not a major issue, for instance, in the case of stationary wireless or docked mobile devices. With autoprefetching, when the client reads the broadcast and it appears that the newest version of a data element has the same *Dval* as the newest cached value for this element, besides the Vno field, the f field of the corresponding cache entry is incremented as well indicating that there is now a longer sequence of versions with the same *Dval*. Autoprefetching increases the effective number of elements stored in the cache.

In this paper, we assume compressed-LRU with autoprefetch but consider different rules for cache replacement. There are a number of different criteria on which to decide which cache entry should be replaced. We suggest using the values of the following cache fields as the basis for replacement:

1. t: This corresponds to the standard LRU replacement strategy.
2. *Vno*: This approach flushes the oldest cache elements on the premise that they will not be of interest to the client.
3. f: This approach tries to keep in the cache the entries with the most compressed data, effectively, storing in the cache as much data as possible.

We call the replacement strategy which uses the t (100)-replacement (this is equivalent to LRU replacement), the strategy which uses the *Vno* field (010)-replacement, and the strategy which uses the f field (001)-replacement. It is also possible to combine the three strategies. To this end, we define *Score* as:

$$Score = A * t + B * Vno + C * f$$

The lower the score the higher the probability that that cache entry will be replaced. Coefficients A, B, and C show which of the three parameter is more significant and define its weight in selecting the item to be replaced. The (ABC) abbreviation shows the weight of each replacement mechanism. In this paper, we consider (100)-replacement, (010)-replacement, (001)-replacement as well as the "aggregated" (111)-replacement.

5 Dynamic Model of the Adaptive Broadcast

5.1 Client Access Behavior

Clients have different interests. We consider three types of access behavior. In *historical* access, clients are interested in the "history" of a particular data item, that is, they are interested in reading various versions of this item. For instance, a client may wish to monitor the progress of the values of some data item over a period of time (e.g., a specific stock price, the weather condition at a given city, the output of a particular sensor). In *snapshot* access, clients are interested in a specific version of a set of data items, for instance, they are interested in the current versions of the items or at their values at a specific time instance. For example, clients may be interested in weather conditions in a set of cities in some earlier time, the oldest available value for a number of stock prices, or the current values of a set of sensors. Finally, in the *browse* type of access, a client is interested in a randomly chosen version of a randomly chosen data item.

We also distinguish between *dynamic* and *static* access. In the case of static access, the client knows all data it wants to access before the broadcast. In contrast, in dynamic access, the client determines the next data to access after it finds the previous item.

Our experimental results show that the Horizontal organization is more appropriate for the historical type of access, whereas the Vertical Broadcast organization is more appropriate for the snapshot one. Thus, the choice between Horizontal and Vertical broadcast should be based on the access type of the majority of the clients. However, this may change over time. To this end, we introduce adaptive broadcast.

5.2 Adaptive Broadcast

In adaptive broadcast, the type of broadcast organization (i.e., Vertical or Horizontal) is dynamically determined by the access behavior of the majority of the clients. For adaptive broadcast to be supported, information about the type

	Broadcast cycle											
	Linear profile				Binary profile				Random profile			
	1	2	3	4	1	2	3	4	1	2	3	4
Client 1	1	1	1	1	1	0	1	0	1	0	0	1
Client 2	1	1	1	0	1	0	1	0	0	1	0	0
Client 3	1	1	0	0	1	0	1	0	1	1	0	1
Client 4	1	0	0	0	1	0	1	0	0	0	1	1
Client 5	0	0	0	0	1	0	1	0	1	0	1	0

Table 1. Clients profile.

of client behavior must be available at the server. This information is minimal, just the client's access type: historical, or snapshot. The browse type of access is not affected by the type of the broadcast organization, thus clients with such behavior are not considered in determining the broadcast organization type. To convey their access type, clients send messages via the uplink channel to the server specifying their access type when this changes. The server keeps information reflecting the interest of the majority of its clients. At the beginning of each broadcast cycle, if the server notices a switch in its clients' interests, it moves from one type of broadcast organization to the other.

There are various reasons for changes of the client access type. New mobile clients may enter the area of coverage of the server, or previously supported mobile clients may leave the server's area. Client interests may also change over time. A set of client access types constitutes the clients' accumulative profile, or simply profile. We consider three different profile types depending on how the profile changes over time:

1. *linear profile*
2. *binary profile*
3. *random profile*

The *linear profile* simulates the case when the overall client interest migrates gradually from one access type to the other. It can be considered as a synthetic profile. In the *binary profile*, all clients synchronously change their interest. This is a synthetic profile. The *random profile* models the case in which clients independently change their access interests with some probability. The main parameter of a random profile is the probability that a client changes its interests between two consecutive broadcast cycles, broadcast cycles i and $i+1$. We call this parameter *Instability*. Note that Instability is different from *Randomness Degree*.

As an example, consider Table 1 that depicts the profiles of a set of five clients. The table also shows how this profile changes over four broadcast cycles. Assume that 1 denotes snapshot access and 0 denotes historical access. For instance, in the linear profile, client 1 uses snapshot access for all broadcast cycles. Client 2 uses snapshot access for broadcast cycles 1, 2, and 3, and historical access for broadcast cycle 4.

Parameter	Description	Default Value
Compression	Basic Sequential broadcast Compressed broadcast	Compressed
Disk	0 - Single disk 1 - Multiple disks	0
HDSize	Fast disk size	
Frequency	Relative rotation frequency of fast disk relative to slow disk	
Bcast Cycle	Broadcast cycle	
Bcast Type	Vertical broadcast Horizontal broadcast	Vertical
Size	Number of data items	10
Versions	Number of versions	
Randomness Degree	0–1, (0: all versions have the same value, 1: versions are completely independent)	0.5
Length	Size of a data element (size of an auxiliary symbol is 1)	16
Elements	Number of the requested data items	
Access Type	Browse access Snapshot access Historical access	Browse
StrideN	Number of the strides for Snapshot/ Historical access	
StrideL	Length of the strides for Snapshot/ Historical access	
Tries	Repetition of each experiment to reduce deviations	80
Instability	Probability that a client changes its access pattern	

Table 2. Simulation parameters.

Table 1 depicts a linear profile where the access type changes from snapshot to historical. In particular, for the first two broadcast cycles the majority of the clients follow a snapshot access pattern, so the server uses the Vertical broadcast organization, whereas, for the last two broadcast cycles the majority changes, so the server switches to Horizontal broadcast. In the binary profile in Table 1, the access behavior of all five clients changes at each broadcast cycle. Finally, in the random profile, changes of access type occur randomly.

6 Experimental Testbed

The simulation system consists of a broadcast server which broadcasts a specified number of versions of a set of data items, and a client which receives the data. The number of data items in the set is determined by the *Size* parameter and the number of versions by the *Versions* parameter. The communication is based on the client-server mechanism via sockets. For simplicity the data values are integer numbers from 0 to 9. The server repeatedly broadcasts data, each broadcast cycle has a number, determined by the *Bcast Cycle* parameter.

The simulator runs the server in two modes, corresponding to the two broadcast organizations, namely Vertical Broadcast and Horizontal Broadcast (deter-

mined by the *Bcast Type* parameter). The broadcast is either Compressed or basic Sequential (determined by the *Compression* parameter). The server generates broadcast data with different degree of randomness (from 0 to 1), which is determined by the parameter *Randomness Degree*.

The client searches the data by using three different access types: Browse, Snapshot and Historical (determined by the *Access Type* parameter). The client generates the data elements it needs to access (various versions of data items) before tuning into the broadcast. The parameter *Elements* determines the number of the data elements to be requested by the client. For the Browse type of access, the data items and their versions are determined randomly to simulate the case when all versions of all data items are equally important for a client. For the Snapshot and the Historical accesses, the requested data elements are grouped into a number of strides (determined by *StrideN*), each containing l elements (determined by *StrideL*). (Clearly, *StrideL*StrideN* = *Elements*.) For example, if *StrideN*=2 and *StrideL*=5, for Snapshot access, the client searches for 2 versions (determined randomly with Zipf distribution) of 5 consecutive data elements. For Historical access, the client tries to find 5 versions of 2 data items (determined randomly with Zipf distribution). In both cases, this pattern can be written as *Elements* = 5x2.

The client may tune in at any point in the broadcast, but it starts its search for data elements at the beginning of the next broadcast. Thus, if a client does not tune in at the beginning of a broadcast, it sleeps until it wakes up at the beginning of the next broadcast which is determined by the next broadcast pointer in the header of each bucket. A client reads a broadcast until all the desired data elements are found. In this way, it is guaranteed that the desired data elements are found within a single broadcast. This scheme is applicable for both static and dynamic access. Note that for the broadcast with dynamic access, the performance is determined by how fast the data is found in the last broadcast cycle. Therefore, by simulating the last broadcast, it is possible to estimate the performance for both cases. While the client is reading, it counts the number and type of characters it reads. This can be converted into *Access Time* – the time elapsed between the time the client starts its search and until it reads its last requested data element, given a specific data transmission rate. In our study, access time is the measure of performance for both response time and power consumption (recall that we do not consider selective tuning in this paper, hence a client stays in active mode throughout its search). The smaller the access time, the higher the performance and the smaller the consumption of energy. We assume that the auxiliary characters (e.g., #, =, ^ , V) consume one time unit and the data elements may consume 4, 16, 64 etc. time units, depending on complexity of the data. The *Length* parameter is used to specify the size of data element. In the experiments reported in this paper, we have chosen *Length* to be 16, which may correspond to 16 bytes.

For the broadcast disk case, we implemented two different speed disks ("fast" for all versions of frequently accessed data and "slow" for all versions of infrequently accessed data). *HDSize*, is the size of the fast disk, which corresponds

to the number of values, which are "hot". *Frequency* is the relative frequency of how faster (more frequently) the "hot" data are broadcast, compared to the "cold" data. In this case, the client generates requests to data not uniformly, but following the Zipf-distribution. We varied the *HDSize* parameter from 1 to 20 and the *Frequency* parameter from 1 to 6. The disks were implemented for both Vertical and Horizontal broadcasts. In real world, the clients are expected to let the server know about their distributions, and on the base of that knowledge the server is expected to place the most frequently accessed data to the faster rotating disk. In our experiments, we show that, besides single disks, compression improves the performance in the case of broadcast disks as well.

In order to estimate confidence intervals, we performed the measurements 80 times (parameter *Tries*). Then, we calculated the average access time and the corresponding standard deviation which are shown in our graphs. The discussed parameters and their default values are summarized in Table 2.

7 Performance Results

7.1 Single Disk Broadcast

In this section, we report on the results of our experiments that demonstrate the applicability of our proposed multiversion broadcast organizations and the advantages of our compression scheme.

Performance of the Compression Scheme For this set of experiments, we consider the Vertical Broadcast organization and the Browse access for the clients. Our results are also applicable for the other broadcast organizations and access types. *Size* is set to 90 and for the first two sets of experiments *Elements* is set to 5.

The effectiveness of the Compressed Broadcast depends on the size of the data elements on the broadcast (represented by *Length* parameter) as opposed to the size of the auxiliary symbols (note that we assume that such symbols consume one time unit). Figure 4 shows the dependence of the access time on the size of the data items for the Compressed and the Sequential (i.e., uncompressed) server broadcasts. Compression reduces the client's access time for any size of data (about 2 times for Randomness Degree=0.5). The greatest gain in terms of absolute access time occurs for the largest data sizes as expected.

The effectiveness of our compression depends on the frequency of updates. The smaller the update frequency, the larger the benefits of compression. To model this, we use the average randomness *Randomness Degree[i]* of the data element i to represents how frequently the value of this element changes, in particular to express the probability that any two consequent versions of k, $Dval[k, j]$ and $Dval[k, j-1]$ are not equal. Figure 5 depicts the dependence of the performance of compression on the *Randomness Degree*. For *Randomness Degree*=0.0, we observe about 10 times improvement. When the versions become

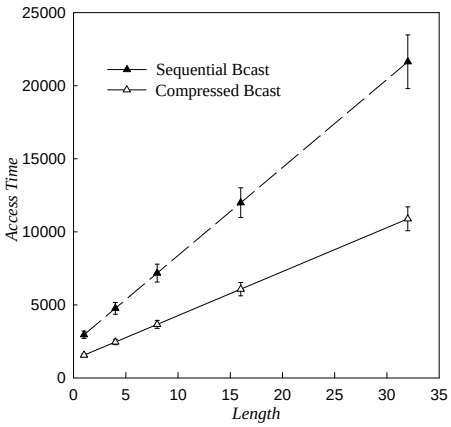

Fig. 4. Dependence of the access time on the size of the data items.

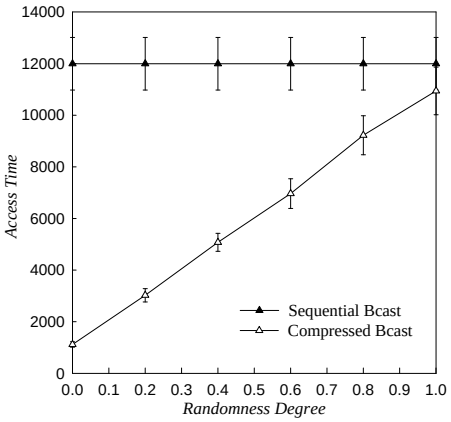

Fig. 5. Compression performance for different Randomness Degree.

more different, the performance of the compressed broadcast worsens, getting close to that of the sequential broadcast as *Randomness Degree* approaches 1.

The experiment shows that the proposed compressed scheme works best if data do not change from one version to another *every* time interval. However, even if they do change, the compressed broadcast just converts to a simple sequential broadcast. The auxiliary symbols overhead is so small that the fact that even at *Randomness Degree*=1 there exist some data items for which the data values are the same for adjacent version numbers (and so, we still have some minimal compression) is enough to have some minor performance improvement. This is a situation to be expected in reality.

The dependence of the *Access Time* on the number of elements requested (given by *Elements* parameter) is shown in Figure 6. We see that at the beginning the increase of the number of elements requested leads to a significant increase of the access time, but later (for *Elements* higher than 4) the access time increases more slowly. This behavior does not look very surprising if we consider the access time as the time needed to search from the beginning of a bcast to "the furthest data element". The other requested data elements are "in between" and are "picked up" on the way. As the number of *Elements* increases, the place where the last searched element was "picked up" shifts toward the end of the broadcast, making the *Access Time* "saturated". The important feature is that the absolute difference between the access time for the compressed broadcast and the sequential broadcast is the largest for *Elements* higher than 4. However, the relative difference stays approximately the same (slightly greater than 2 times).

The results presented in Figure 4 to Figure 6 are obtained for the Vertical broadcast and the Browse access type, but qualitatively the tendencies observed are valid for all other broadcast organizations and access schemes.

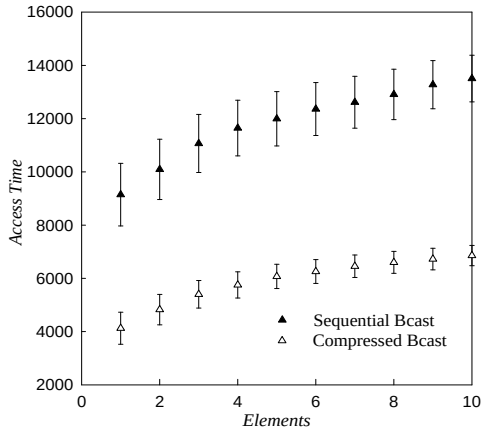

Fig. 6. Performance for different number of searched elements.

Broadcast Organization and Client Access Type Figure 7 shows the dependence of the *Access Time* on *Randomness Degree* when the server uses the Vertical Broadcast, whereas Figure 8 when the server uses the Horizontal Broadcast. *Elements* is set to 20. The main conclusion is that for the Snapshot access type, the most efficient broadcast organization is the Vertical Organization, whereas for Historical access type, the most efficient broadcast organization is the Horizontal one.

In particular, Figure 7 shows that for the Vertical broadcast the most efficient access scheme is the Snapshot access and the worst is the Historical access (about 1.5 times worse than the Snapshot access). The Browse access is somewhere in between (about 1.4 times worse than the Snapshot Access) closer to the Historical Access. Figure 8 shows the opposite results. The best scheme for the Horizontal Broadcast is the Historical Access, and the worst is the Snapshot Access (about 1.4 times worse than the Historical Access).

These results are valid for both Compressed and Sequential Broadcasts. The interesting feature is that for small values of *Randomness Degree*, it is more important for the performance whether the broadcast is Compressed or Sequential than whether the access scheme "corresponds" to the broadcast. We can see on the figures that for *Randomness Degree* less than 0.7 the *Access Time* for any access type is smaller in the case of the Compressed Broadcast. But for *Randomness Degree* higher than 0.7, there are cases when the Sequential Broadcast with the "right" access scheme can beat the Compressed Broadcast with the "wrong" access scheme. Hence, in order to have the best performance, the broadcast organization and access scheme should have "similar patterns", either Vertical Broadcast organization and Snapshot Access, or Horizontal Broadcast organization and Historical Access.

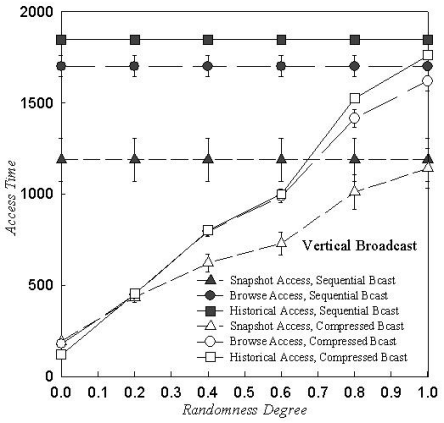

Fig. 7. Vertical broadcast at different Randomness Degree.

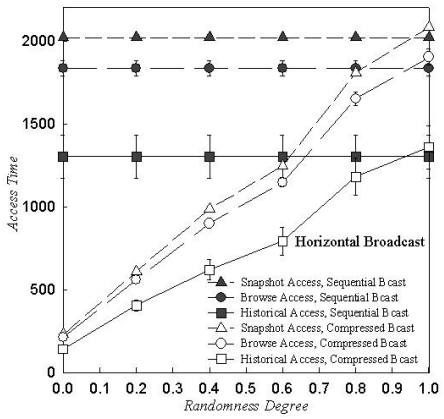

Fig. 8. Horizontal broadcast at different Randomness Degree.

7.2 Broadcast Disks

In this section, we restrict our discussion to the first approach for distributing data between the "fast" and the "slow" disk presented in Section 3.4. Similar results apply to the other approaches as well. The goal of our experiments are two fold. First, we test whether the popular technique of broadcast disks is "compatible" with our compression scheme. Second, we want to get a better insight of the use of the broadcast disks for disseminating multiple versions of data. Specifically, two important parameters in this setting are the size of the fast disk and the relative speed of the two disks. We performed two experiments to see how these parameters affect performance of multiversion broadcast.

We set the relative speed of the disks (*Frequency*) as a constant equal to 2 and varied the size of the fast disk (*HDSize*). The results are presented in Figure 9. We see that for both *Elements* values 2 and 5, the Compressed broadcast outperforms the Sequential broadcast. However for *Elements*=2 the curve has the minimum at *HDSize*=10 and for *Elements*=5 the curve has the minimum at *HDSize*=1. This means that for smaller *Elements* values, broadcast disks may improve the performance, but for larger values of *Elements* the broadcast disks scheme creates more overhead (due to the increase of the overall broadcast size created by the repetitive broadcast of the values on the fast disk) than gains. Higher *Elements* values lead to higher probability that the necessary data are located on "slow" disks, so the overhead of repetitive fast disks overwhelms the gains produced by the different speed of the disk rotation. This observation is valid for both the Compressed and the Sequential broadcast.

In the second set of experiments, we set *HDSize* as a constant (equal to 5, 10 and 15) and varied *Frequency*. The results are shown on Figure 10. We intentionally set *Elements* parameter to 2, because the previous experiment showed that

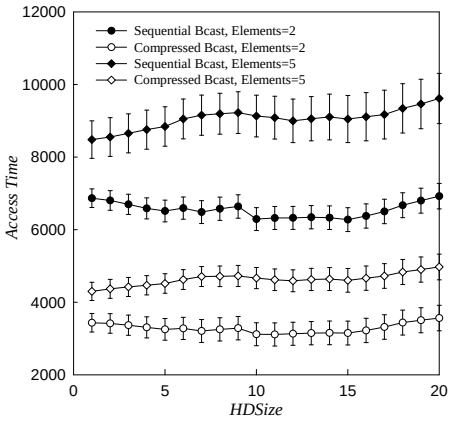

Fig. 9. Performance of different absolute sizes of fast and slow disk for broadcast disks.

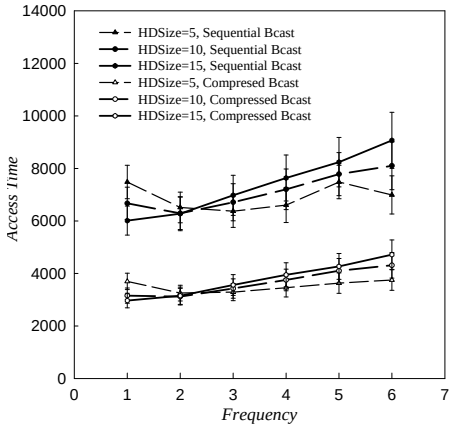

Fig. 10. Performance of different relative frequencies (fast, slow disks) for broadcast disks.

higher values are not appropriate for the broadcast disks scheme. The results from Figure 10 show that the optimal relative frequency of the disks does exist for certain values of *HDSize*. In this experiment, it is 3 for *HDSize*=5, 2 for *HDSize*=10, and the disks scheme does not improve performance (*Frequency*=1 in fact means that the broadcast is flat) for *HDSize*=15. Hence, we found another parameter responsible for the decision if disks scheme should be used or should not. This result is valid for both the Sequential and the Compressed Broadcasts.

7.3 Client Side Cache and Adaptive Broadcast

In this subsection, we consider two techniques which can reduce the effect of the "incompatibility" of the broadcast organization and the client access pattern. One technique is the client side cache and the other is the server adaptive broadcast. In all the following experiments, $Size$ and $Versions$ is each set to 25.

We consider an adaptive compressed broadcast organization. Figures 11–12 ($Elements = 10x1$, $CacheSize = 30$) show the Access time gains and the Hits rates for the Linear profile of 20 clients for 20 broadcast cycles. Four different cache organizations are considered. The results for the other two profiles, Binary and Random, of 20 clients for 20 broadcast cycles are shown in Figures 13–14 ($Elements = 15x2$, $CacheSize = 90$) and 15–16 ($Elements = 4x2$, $CacheSize = 60$) respectively. For all profiles, the highest hit rate was obtained for (100) and (001) cache organizations (15-20%) and the lowest for (010) cache organization (around zero). The "aggregate" (111) cache organization behaved a little worse than (100) and (001). Similar results were obtained for the access time gain, which was 2-4% for (100) and (001) cache organizations and practically zero for (010). The (111) cache organization again was a little worse than (001) and (100) types.

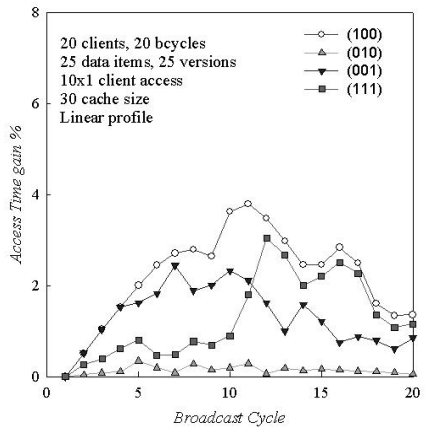

Fig. 11. Access time gain for different cache organizations in the case of the Linear profile.

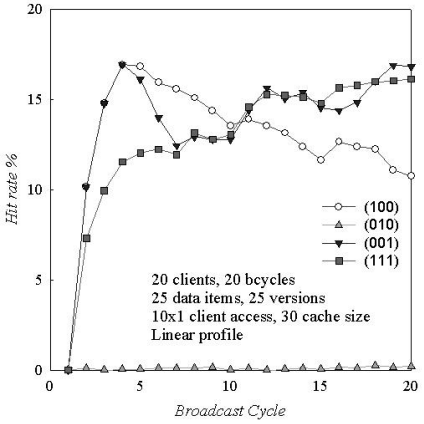

Fig. 12. Hit rate for different cache organizations in the case of the Linear profile.

Similar experiments were performed with non adaptive server broadcast. The behavior of the cache was the same: (100) and (001) caches had the access time gain about 2-4%, (111) cache had it about 1-3%, and (010) had no gain in the access time at all.

We describe next experiment with adaptive and non adaptive (vertical) broadcast; these experiments are performed for all four cache organizations as well as without a cache. To make the figures more clear, we include the data for (111) cache only. The other cache organization showed very similar behavior.

Figure 17 ($Elements = 10x1$) shows the access time vs. Broadcast cycle for the adaptive and non-adaptive broadcast with (111) type client cache in case of the Linear profile of 20 clients for 20 broadcast cycles. Initially, (broadcast cycle = 1), most of the clients have the Snapshot access pattern. In case of non-adaptive broadcasting server, which is chosen by default to be the Vertical, the access time is low because the broadcast organization (Vertical) and the client access pattern of the majority of clients (Snapshot) match. Later (higher broadcast cycles) the access time steadily goes up and reaches the maximum at the end of broadcast (broadcast cycle = 20). At this point all clients use the Historical access, which does not match the broadcast organization. In case of adaptive broadcast, the broadcast organization in the beginning (broadcast cycle 1) is Vertical that satisfies most of the clients. Therefore the access time is small. Then for broadcast cycles close to 10, the number of clients with the Snapshot and the Historical access types is the same, so about a half of the clients are satisfied. As a result, the access time goes up. When broadcast cycle reaches 10, the adaptive broadcast switches from the Vertical to the Horizontal and the access time goes down.

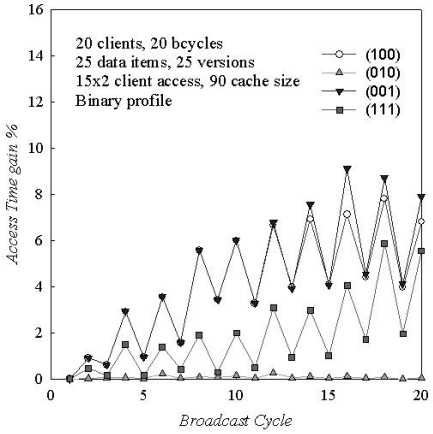

Fig. 13. Access time gain for different cache organizations in the case of the Binary profile.

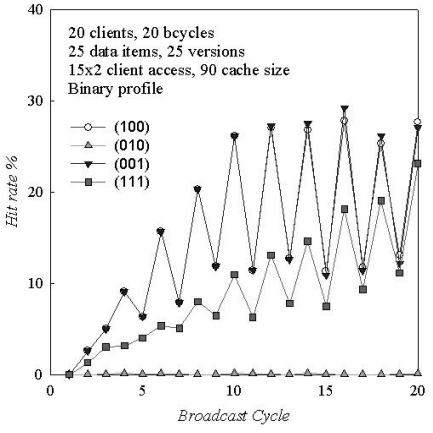

Fig. 14. Hit rate for different cache organizations in the case of the Binary profile.

Figure 18 ($Elements = 15x2$) shows the Access time vs. Broadcast cycle in the case of the Binary profile of 20 clients for 20 broadcast cycles. As can be seen, non-adaptive broadcast produces bursts of low and high access time, in accordance with the matching and non matching the client access pattern to the server broadcast organization. As expected, the adaptive broadcast removes those bursts and makes the performance much smoother and closer to the best possible one (i.e., minimum access time).

Figure 19 ($Elements = 4x2$) shows the Access Time vs. Broadcast Cycle in case of the Random profile of 20 clients for 20 broadcast cycles. In the average, the access time gain due the to adaptive broadcast is about 15%. The gain depends on the ratio of the clients with different interests. The further from unity this ratio is, the greater is the gain. We also performed experiments with different *Instability* parameters and found that the access time reduction is insensitive to that. However, the high values of *Instability* incurs higher overhead due to the frequent messages from the client to the server on the uplink channel.

The simulation data show that both adaptive broadcast and client side cache are good tools for the multiversion broadcast performance improvement in case of "incoherent" (i.e., Vertical-Historical or Horizontal-Snapshot) server broadcast organization and client access pattern. Caching is a local method and relies only on local resources whereas adaptive broadcast is a global method and involves extra communication. Both techniques are compatible and hence can be applied at the same time, creating an even greater cumulative positive effect.

Finally, it should be noted that feedback messages are not uncommon in a typical broadcast push environment in which the server relies on such feedback messages to decide on the content of the broadcast. Our adaptive broadcast proposal can be implemented in conjunction with a document selection method

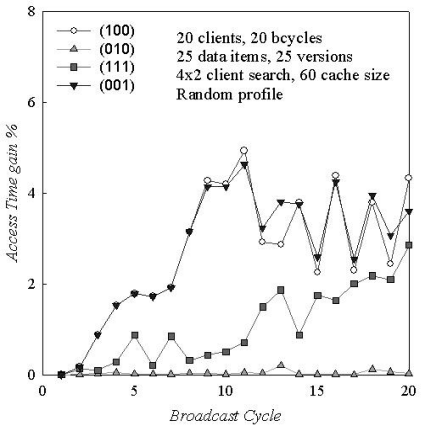

Fig. 15. Access time gain for different cache organizations in the case of the Random profile.

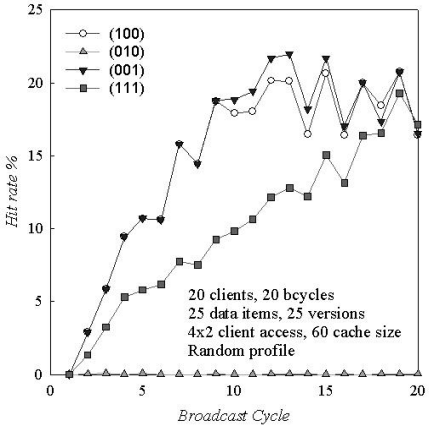

Fig. 16. Hit rate for different cache organizations in the case of the Random profile.

so that the data items and their organization on the broadcast cater better to the needs of the majority of the clients, hence achieving even greater scalability.

8 Related Work

The work in this paper falls within the solutions that use multicast/broadcast communication to support scalable dissemination of information. These solutions efficiently disseminate data to a large number of users by any combination of the following two schemes: *broadcast pull* and *broadcast push*. In broadcast pull, the clients make explicit requests for data. If multiple clients request the same data at approximately the same time, the server may aggregate these requests, and only broadcast the data once. Such a scheme makes an effective use of the low wireless bandwidth and clearly improves user perceived performance. Several preemptive and non-preemptive scheduling algorithms have been proposed that attempt to achieve maximum aggregation both based on the exact match of the requests [2, 15, 35, 36] and based on derivation dependency among requests [14, 29, 30].

In broadcast push, which is the context of our work, techniques for data organization, indexing for selective tuning and caching have been investigated. Among the first data organizations was the broadcast disks organization studied in Section 3.4 [5], while a scheme to generate non-uniform broadcasts that support range queries is described in [34]. Power conservative indexing methods for single-attribute and multi-attribute based queries appeared in [19, 16, 7].

The most commonly used caching policy LRU may lead to suboptimal performance in multicast environments and several algorithm has been investigated

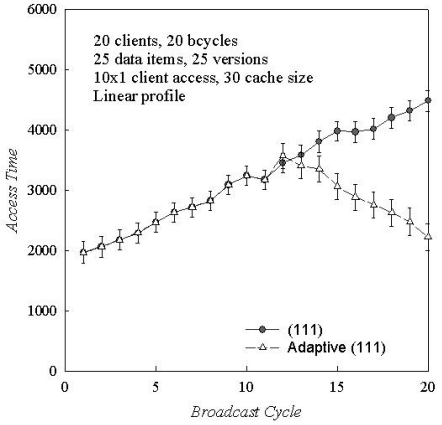

Fig. 17. Effect of the adaptive broadcast in case of the Linear profile

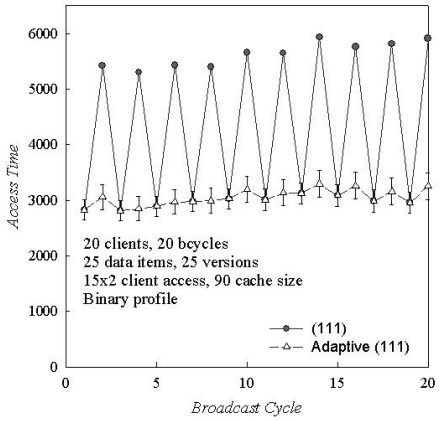

Fig. 18. Effect of the adaptive broadcast in case of the Binary profile

to improve it. For example, Landlord/Greedy-Dual-Size [11, 38] attempts to improve LRU by taking into account the cost of downloading a page whereas the PT [4] and Gray [21] algorithms takes in addition the popularity of a page. As part of our future work, we plan to evaluate such methods in the context of multiversion broadcast environments.

The concept of autoprefetching has been proposed in the context of mobile computing systems in an effort to improve cache performance to reduce access time without increasing power consumption [12, 17]. These schemes dynamically predict what data to be prefetched based on the data access frequency and update frequency. These approaches share similar objective with our proposed multiversion autoprefetch scheme but our scheme uses a simple prediction which autoprefetches only new versions of already cached data items. That is, in some sense, our autoprefetch behaves similar to a data recharging mechanism [13].

As mentioned in the introduction, a major goal of our work on multiversion broadcasts has been to achieve consistency and currency in broadcast environments. In [3], the authors discuss the tradeoffs between currency of data and performance issues when some of the broadcast data items are updated by processes running on the server. However, the updates do not have transactional semantics associated with them either at the server or at the clients. Realizing that serializability as the correctness criterion may be expensive, and perhaps unnecessary in such environments, and hence various protocols [28, 8, 25, 26, 22] attempt to cater to less demanding correctness requirements. All these protocols have been formally analyzed within a unified framework [27].

Finally, research on balancing the broadcast push of information and broadcast pull data delivery methods appeared in [2, 33, 10].

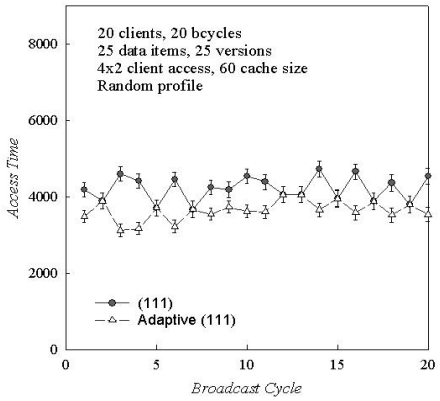

Fig. 19. Effect of the adaptive broadcast in case of the Random profile

9 Conclusions

In this paper, we identify two basic multiversion organizations, namely vertical and horizontal broadcasts and propose an efficient compression scheme applicable to both. We also examine the applicability of these schemes in the context of both single disk and multiple disk broadcasts. We consider multiversion client data caching and introduce appropriate cache replacement techniques.

Our performance evaluation results showed that besides the size of a broadcast, the organization of the broadcast has an impact on performance, as different kinds of clients need different types of data. We recognized three kinds of clients applications based on their access behavior: "Historical" that access many versions of the same data, "snapshot" that access different data of the same version and "browsing" that access data and versions randomly.

Specifically, if the primary interest of clients is "historical" applications, the best way to broadcast is the Horizontal Broadcast. If the primary interest of clients is "snapshot" applications, the best way to broadcast is the Vertical Broadcast. In case of mixed environment it is possible to create adaptive broadcast with no extra cost due to flexibility of the broadcast format.

The suggested compression technique does not require extra time for client side decompression and works for both Vertical and Horizontal broadcasts. The auxiliary symbols overhead is small if the size of one data element significantly exceeds a few bits. The effectiveness of a compressed broadcast depends on the repetitiveness of the data. The less frequently data change, the better the gains are. But even in the worst case (completely random data), the Compressed broadcast does not exhibit worse performance than the Sequential broadcast.

The use of our compression scheme in the client cache exhibited similar advantages as in the case of the compressed broadcast, effectively increasing the size of the cache and consequently, the number of hits. However, the most interesting property exhibited by the client caching is that it can be used as a tool to

ameliorate the negative effects due to any incompatibilities between a broadcast organization and a client data access behavior.

Acknowledgments: This work was supported in part by the National Science Foundation award ANI-0123705 and in part by the European Union through grant IST-2001-32645.

References

[1] S. Acharya and S. Muthukrishnan Scheduling on-demand broadcasts: New metrics and algorithms. *Fourth Annual ACM/IEEE Conference MobiCom* (1998) 43–54

[2] S. Acharya, M.Franklin, and S.Zdonik Balancing Push and Pull for Data Broadcast. *ACM SIGMOD Conferences*, (1997) 183–194

[3] S. Acharya, M. Franklin, and S. Zdonik Disseminating Updates on Broadcast Disks. *22nd VLDB Conference*, (1996) 354–365

[4] S. Acharya, M. Franklin, and S. Zdonik Prefetching from a broadcast disk. *The Int'l Conference on Data Engineering*, (1996) 276–285

[5] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik Broadcast Disks : Data Management for Asymmetric Communication Environments. *ACM SIGMOD Conference*, (1995) 199–210

[6] S. Acharya, M. Franklin, S. Zdonik Dissemination-based Data Delivery Using Broadcast Disks. *IEEE Personal Communications*, **2(6)** (1995) 50–61

[7] R. Agrawal and P. K. Chrysanthis Efficient data dissemination to mobile clients in e-commerce applications. *3rd Int'l Workshop on E-Commerce and Web Information Systems*, (2001) 58–65

[8] D. Barbará Certification Reports: Supporting Transactions in Wireless Systems *The IEEE International Conference on Distributed Computing Systems*, (1997) 466–473

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

[10] M. Bhide, P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, P. J. Shenoy Adaptive Push-Pull: Disseminating Dynamic Web Data. *IEEE Transactions on Computers*, **51(6)** (2002) 652–668

[11] P. Cao and S. Irani Cost-aware WWW proxy caching algorithms. *The USENIX Symposium on Internet Technologies and Systems*, (1997) 193–206

[12] G. Cao Proactive Power-Aware Cache Management for Mobile Computing Systems. *IEEE Transactions on Computers*, **51**, No. 6, (2002) 608–621

[13] M. Cherniack, M. J. Franklin, and S. Zdonik Expressing User Profiles for Data Recharging. *IEEE Personal Communications: Special Issue on Pervasive Computing*, 2001) 6–13

[14] A. Crespo, O. Buyukkokten, and H. G. Molina Efficient query subscription processing in a multicast environment (extended abstract). *The 16th ICDE Conference*, (2000) 83

[15] H. D. Dykeman, M. Ammar, and J. W. Wong Scheduling algorithms for videotex systems under broadcast delivery. *The 1986 International Conference on Communications*, (1986) 1847–1851

[16] Q. Hu, W.-C. Lee, and D. L. Lee. Power conservative multi-attribute queries on data broadcast. *The 16th ICDE Conference*, (2000) 157–166

[17] H. Hu, J. Xu, D. L. Lee. Adaptive Power-Aware Prefetching Schemes for Mobile Broadcast Environments. *Mobile Data Management*, (2003) 374–380

[18] T. Imielinski et al. Data on Air : Organization and Access. *IEEE Transactions on Knowledge and Data Engineering*, **9**, No. 3, (1997) 353–372

[19] T. Imielinski, S. Viswanathan, and B. R. Badrinath Energy efficient indexing on air. *ACM SIGMOD Conference*, (1994) 25–36

[20] J. Jing, A. H. Elmargarmid, S. Helal, R. Alonso Bit-Sequences: An adaptive Cache Invalidation Method in Mobile Client/Server Environment. *ACM/Baltzer Mobile Networks and Applications*, **2(2)** (1997) 115–127

[21] S. Khanna and V. Liberatore On broadcast disk paging. *SIAM Journal on Computing*, **29(5)** (2000) 1683–1702

[22] V. C. S. Lee, S. H. Son, and K. Lam On the Performance of Transaction Processing in Broadcast Environments *The International Conference on Mobile Data Access (MDA'99)*, (1999)

[23] C. Mohan, H. Pirahesh, and R. Lorie Efficient and Flexible Methods for Transient Versioning to Avoid Locking by Read-Only Transactions. *ACM SIGMOD Conference*, (1992) 124–133

[24] E. Pitoura and P. Chrysanthis Multiversion Data Broadcast. *IEEE Transactions on Computers*, **51(10)** (2002) 1224–1230

[25] E. Pitoura and P. K. Chrysanthis Scalable Processing of Read-Only Transactions in Broadcast Push. *19th IEEE Int'l Conf. on Distributed Computing Systems*, (1999) 432–439

[26] E. Pitoura and P. K. Chrysanthis Exploiting Versions for Handling Updates in Broadcast Disks. *25th VLDB Conference*, (1999) 114–125

[27] E. Pitoura, P. K. Chrysanthis and K. Ramamritham Characterizing the Temporal and Semantic Coherency of Broadcast-based Data Dissemination. *The International Conference on Database Theory*, (2003) 410–424

[28] J. Shanmugasundaram et al. Efficient Concurrency Control for Broadcast Environments. *ACM SIGMOD Conference*, (1999) 85–96

[29] M. A. Sharaf and P. K. Chrysanthis Facilitating Mobile Decision Making. *2nd ACM Mobicom International Workshop on Mobile Commerce*, (2002) 45–53

[30] M. A. Sharaf and P. K. Chrysanthis Semantic-Based Delivery of OLAP Summary Tables in Wireless Environments. *10th ACM International Conference on Information and Knowledge Management*, (2002) 84–92

[31] O. Shigiltchoff, P. K. Chrysanthis and E. Pitoura Broadcast Data Organizations and Client Side cache. (To appear in) *ICDCS 2003 Workshops*, (2003)

[32] S.W. Smith *The Scientist and Engineer's Guide to Digital Signal Processing*, California Technical Publishing (1997)

[33] K. Stathatos, N. Roussopoulos, and J.S. Baras Adaptive Data Broadcast in Hybrid Networks. *The 23rd VLDB Conference*, (1997) 326–335

[34] K.-L. Tan and J.X. Yu Generating Broadcast Programs that Support Range Queries. *IEEE TKDE*, **10(4)** (1998) 668-672

[35] N. H. Vaidya and S. Hameed Scheduling data broadcast in asymmetric communication environments. *ACM/Baltzer Wireless Networks*, **5(3)** (1999) 171–182

[36] J. W. Wong Broadcast delivery. *Proc. of the IEEE*, **76** (1988) 1566–1577

[37] K.-L. Wu, P. S. Yu, M.-S. Chen Energy-Efficient Mobile Cache Invalidation. *Distributed and Parallel Databases*, **6** (1998) 351–372

[38] N. Young On-line file caching. *The Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, (1998) 82–91