Broadcast Data Organizations and Client Side Cache [*]

Oleg Shigiltchoff
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
oleg@cs.pitt.edu

Panos K. Chrysanthis
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
panos@cs.pitt.edu

Evaggelia Pitoura
Dept. 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 study different client side cache organizations for various types of 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.

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 partcular, in the context of wireless and mobile environments, *Broadcast push* [1] 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 appear on the broadcast channel. Such applications 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 tendering, and networks of sensors. Any number of clients can monitor the broadcast channel. If data is properly organized to cater to the needs of the client, such a scheme makes an effective use of the low wireless bandwidth. It is also ideal to achieve maximal scalability in regular web environments.

To reduce access time, clients may cache data items of interest locally. However when data are updated at the server, the problem arises of keeping the data in the client cache consistent with the updated data on the server [4, 7, 2]. Clearly, any invalidation method is prone to starvation of queries by update transactions. 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 propose maintaining multiple versions of data items on the broadcast as well as in the client cache [3]. Multiversion broadcast allows more client transactions to read consistent data values (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.

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, as in the case of data streams.

A close examination of the above mentioned broadcast applications let us to identify three kinds of queries: *Horizontal* (or "historical") queries in which a client accesses many versions of the same data item; *Vertical* (or "snapshot") queries in which a client accesses different data items of the same version; and *Random* (or hybrid) queries in which a client accesses data and versions randomly. This prompted us to ask the following question that motivated this work: Besides the size of a broadcast, what is the impact of the broadcast organization on performance (determined by the access time and power consumption)? We proposed several broadcast organizations that are variants of two basic multiversion organizations: *Horizontal* and *Vertical*. Also, we proposed a compression scheme along the lines of *Run Length Encoding* (RLE) [6], applicable to both of these 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 part of the

[*]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.

older versions. Due to the reduction of the commiunication traffic, the client can retrieve the needed version of a data item sooner if data items do not change very often, which reduces the time during which the client stays on as well as waits. Furthermore, our compression schemes incur no decompression overhead at the client.

Our results [5] showed that the average client performance is significantly affected by the broadcast organization, that is, where to place the data and the old versions. 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.

In this paper, we study the effect of client caching on a multiversion broadcast environment. We propose and evaluate two variants of the traditional *least recently used* (LRU) caching replacement policy that explicitly consider versions and apply our proposed compression scheme to maximize the number of hits, effectively increasing the cache size within specific space constraints. We also propose a compressed, multiversion *autoprefetch* caching scheme and show that it exhibits the best performance with respect to client's perceived latency. One of the most interesting results of our evaluation is that the biggest impact of caching is on ameliorating the negative effects due to any incompatibilities between a broadcast organization and a client data access behavior.

In the next section, we present the system model, and in Section 3 we present broadcast organization and our compression scheme. In Section 4, we describe client side caching. Section 5 presents our experimental platform whereas our experimental results are discussed in Section 6.

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 "pull", 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 bcast that 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 our model, a server constantly broadcasts a fixed number of versions for each data item. For each new cycle, the oldest version of the data is discarded and a new, the most recent, version is included in the bcast. The number k of older versions that are retained can be seen as a property of the server. In this sense, a k-multiversion server, i.e., a server that broadcasts the previous k values, is one that guarantees the consistency of all transactions with span k or smaller. *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). In this paper, we focus only on client side caching and broadcast organization and how to reduce its size without adopting any indexing scheme.

The logical unit of a broadcast is called *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 organization is how to organize the broadcast, that is where to place the data 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 broadcast.

3. BROADCAST ORGANIZATION

3.1 Basic Organization

A set of multiversion data can be represented as a two dimensional array, where dimensions correspond to version numbers (*Vno*) and data ids (*Did*), and the array elements are the data values (*Dval*). That is, *Dval[i, k]=v* means that k-version of data item i is equal to v.

A simple sequential broadcast can be generated by linearizing the two dimensional array in two different ways: *Horizontal* broadcast or *Vertical* broadcast. In the 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 the 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.

The resulting bcasts have the same size for both organizations [5], but differ in the order in which they broadcast the data values.

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 no, impact on the client. That is, it should not require additional processing at the client, so it should not trade access time to processing time. We proposed a simple compression [5] 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, often, *Dval[i, k]= Dval[i, k+1]= ...= Dval[i, k+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 items if its *Dval* does not change. Instead, the compressed scheme broadcasts *Dval* only if it is different from *Dval* of the previous version. In order not to lose information 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 step, these values are broadcast based on the selected organization. In the first step, any sequence of elements with repetitive data values is replaced with the first element of the sequence and the length of the sequence [5]. For instance sequence of 1 1 1 1 is transformed to the compressed value 1x3.

In the second step, the Horizontal and the Vertical broadcast are produced by using horizontal linearization and vertical linearization of the array, respectively. Formally, the Horizontal broadcast produces

[Did[Vno(Repetition, Dval)]]**

sequences, and the Vertical broadcast produces

[Vno[Did(Repetitions, Dval)]]**

sequences. Obviously we do not include into the broadcast those versions, which already have been included "implicitly" with other versions.

4. CLIENT SIDE CACHE

To improve the 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 mutliversion broadcast.

4.1 LRU Variants

The most simple 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* replacement method (which we will refer to as *LRU-standard*), to discard data elements from the cache when the cache runs out of space. The LRU-standard cache for multiversion broadcast requires four fields: [*Did, Vno,Dval, TimeStamp*]. *TimeStamp* shows when the cache element was used last time. The requested element can be found in the cache by matching *Vno* and *Dval* values for the element and for cache entries. When the cache is full, the replacement mechanism uses *TimeStamp* to select the item to be replaced (victim). The cache element with the oldest *TimeStamp* is selected.

Given that a data element in a multiversion broadcast corresponds to a version of a data item, there are two reasonable modifications to the LRU-standard. The first one is to shift the priority from the data element which was used the least recently, to the element having the oldest version. The rationale for this modification is that the oldest version of a data item has the least possibility to be used in the future. If multiple data items are associated with the oldest version, the LRU-standard is used to select the one to be replaced. For this reason, we call this scheme *LRU-global*. Then, LRU-global cache has three fields: [*Did, Vno,Dval*]. The search mechanism is the same as in the case of LRU-standard, but the replacement strategy chooses as victim the item with the oldest *Vno* instead of the item with the oldest *TimeStamp*.

LRU-global is expected to behave effectively in the same way as LRU-standard for skewed accessed data, where the most frequently accessed data are those with most recent versions, while it does not prematurely discard more recent versions in favor of older ones.

The second modification is very similar to the one in LRU-global except that the element that is selected as victim has the same *Did* as the replacing element. We call this method *LRU-local* since it tries to exploit the local properties of the data. If there is no element in the cache with the same *Did* as the replacing data element, the client uses the LRU-global strategy.

4.2 Compressed LRU Variants

Given the performance gains of our compression scheme [5], one should expect that a compressed caching scheme should also have a significant impact on performance. Our *LRU-compressed* methods is a variant of LRU standard that operates on a cache with "compressed" data elements. LRU-compressed is compatible with our compressed broadcast since we use the same compression scheme for both. In the case of compressed broadcast, the data elements can be directly cached. In the case of uncompressed broadcast, data elements need to be combined with the corresponding already cached versions of the data.

Our compression scheme requires that each cache entry has one extra field f, which contains information of how many adjacent versions of this data element have the same *Dval*. The total number of the required fields is five: [*Did, Vno, Dval, TimeStamp, f*]. For example, assume some data elements [*a,b*], [*a,b+1*], and [*a,b+2*] have *Dval= x*, and the requested data element to be cached is [*a,b+1*]. In this case, the LRU-compressed will add in the cache the element [*a,b*] and write 2 into the f field, reflecting the fact that data elements [*a,b+1*] and [*a,b+2*] have the same *Dval= x* as [*a,b*]. Similarly, the requested element can be found in the cache by matching first the *Dval* value and then, if the version of the requested element is greater than *Vno* and less than *Vno + f*, the element is found in the cache. When the cache is full, the cache element with the oldest *TimeStamp* is selected as the victim.

Clearly, LRU-compress requires more space to store a cache line, but by comparing LRU-standard and our proposed LRU-compressed, this small overhead can effectively increase the cache size and consequently, the number of hits. (The LRU-standard requires three cache entries to store the three versions of *Did=a*.) Of course, as in the case of a compressed boradcast, the LRU-compressed depends on a parameter related to the probability that the adjacent versions of the same data element have the same value. Formally, we define the *Randomness Degree* parameter as follows. Let *Randomness[k,i]=0* if *Dval[k, i] = Dval[k, i+1]* and *Randomness[k,i]=1* otherwise. Then average randomness *Randomness*[k] of k-data element over all versions represents how frequently the data value of this data element changes. Then it is natural to use this average randomness as a parameter (*Randomness Degree*) describing the probability that *Dval[k, i]* is not equal to *Dval[k, i+1]*. For instance, *Randomness Degree=0* means that *Dval[k, i]= Dval[k, i+1]* for any i.

4.3 Autoprefetch Compressed LRU Variants

All the caching schemes presented thus far are passive ones, in the sense that if a client does not refer to a data element, the data element is not cached. However, under certain circumstances and in particular when energy is not a major issue as in the case of stationary wireless devices or docked mobile ones, LRU-compressed can be made to behave pro-actively by prefetching newer versions of data that have already being cached. We call this *LRU-compressed autoprefetch*.

LRU-compressed autoprefetch is similar to LRU-compressed (the same search and replacement mechanisms), but it keeps updating the cache elements during a broadcast even if the client does not access that data element. When the client reads the broadcast and it appears that the newest version of a data element has the same *Dval* as the second newest version of the data element, the cache f field is incremented indicating that there is now a longer sequence of versions with the same *Dval*. For example, if the cache has $Dval[a,b,f=3]=x$ and the broadcast gives that $Dval[a,b+1]=x$. Then the element in the cache is updated to $Dval[a,b,f=4]=x$. It can be seen that the cache entry contains in fact more data items than before. This fact is also confirmed by our experimental results.

Other modifications can also possible. For example, LRU-compressed autoprefetch could use *Vno* instead of *TimeStamp* as the desicive factor for selecting a cache element for replacement.

5. 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 simulator runs the server in two modes, corresponding to the two broadcast organizations, namely Vertical Broadcast and Horizontal Broadcast (determined by the *Bcast Type* parameter). The broadcast could be either Compressed or basic Sequential (determined by the *Compression* parameter). The server generates broadcast data with different degree of randomness (from 0 to 1).

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. The client searches the data by using five different access types: Random with uniform distribution, Random with skewed distributions of data items, Random with skewed distribution of versions, Vertical and Horizontal (determined by *Access Type* parameter). For the Random with uniform distribution access the data items and their versions are determined randomly with uniform distribution to simulate the case when all versions of all data elements are equally important for a client. For Random with skewed distribution of data items *Did* are determined randomly with Zipf distribution, *Vno* are determined randomly with uniform distribution. For Random with skewed distribution of versions *Did* are determined randomly with uniform distribution, *Vno* are determined randomly with Zipf distribution. For the Vertical and Horizontal 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 Vertical access, the client searches for two versions (determined randomly with uniform distribution) of 5 consecutive data elements. For Horizontal access, the client tries to find 5 versions of 2 data items (determined randomly with uniform distribution).

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 to wake 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 access, in which the client knows all data it wants to access before the broadcast and dynamic access, in which the client determines the next data to access after it finds the previous data. Nothe that for the broadcast with the dynamic access, the performance is determined by how fast the data are found in the last broadcast cycle. Therefore, by simulating only the last broadcast, it is posssible 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 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 necessary for description of the data repetitiveness 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.

To evaluate the effect of the client side cache the client reads up to 40 different broadcasts cycles (parameter *Bcast Number*). The cache size was determined by *Cache Size* parameter.

Parameter	**Parameter description**
Compression	Basic Sequential broadcast Compressed broadcast
Bcast Number	Number of broadcasts
Bcast Type	Vertical broadcast Horizontal broadcast
Size	Number of data items
Versions	Number of versions
Randomness Degree	0–1, (0: all versions have the same value, 1: versions are completely independent)
Length	Size of a data element (size of an auxiliary symbol is 1)
Elements	Number of the requested data items
Access Type	Random uniform distribution access Random Zipf distribution of data Random Zipf distribution of versions Vertical access Horizontal access
StrideN	Number of the strides for Vertical/Horizontal accesses
StrideL	Length of the strides for Vertical/Horizontal accesses
Cache Size	The size of cache
Tries	Number of the same experiments to reduce deviations

Table 1: Simulation Parameters

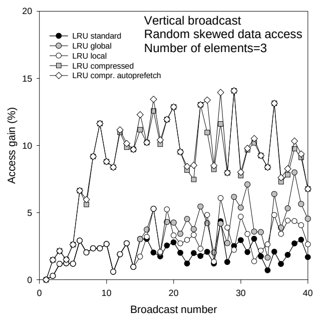


Figure 1: Access gain for different cache replacement strategies

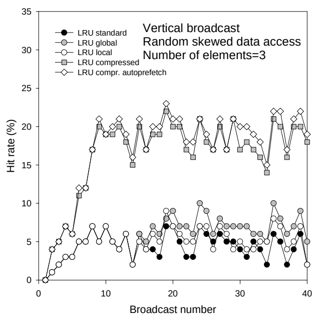


Figure 2: Hit rate for different cache replacement strategies

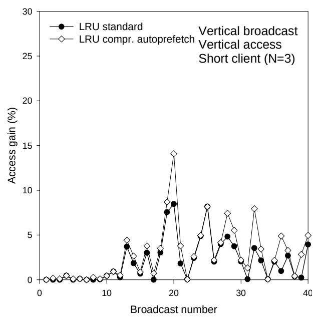


Figure 3: Access gain for vertical broadcast, vertical access, short client

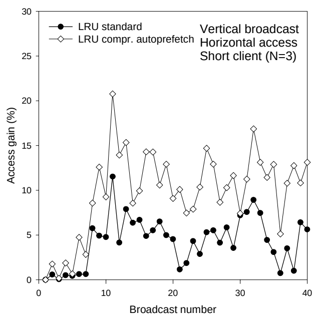


Figure 4: Access gain for vertical broadcast, horizontal access, short client

In order to estimate confidence intervals we performed the measurements 80 times (parameter *Tries*). Then we calculate the average access time and the corresponding standard deviation which are shown in our graphs. The discussed parameters are summarized in Table 1.

6. PERFORMANCE RESULTS

We have evaluated all the client side cache replacement strategies proposed above: LRU standard, LRU global, LRU local, LRU compressed, LRU compressed autoprefetch.

To find which of these replacement strategies performs the best, we compared them with respect to both average access time reduction gain and hit rate. The results are presented in Figure 1 and Figure 2 (both *Size=25, Versions=25, Tries=80, Elements=3, Randomness Degree=0.5, Length=16, Vertical Broadcast, Random skewed data access, Bcast Number=40, Cache Size=30*). The highest access gain and hit rate were shown by LRU compressed and LRU compressed autoprefetch (the latter one has slightly better performance): about 10% of access gain and about 20% of hit rate. The other replacement strategies have shown smaller access gains (about 2–5%) and hit rates (about 4–6%). The leadership of the two compressed replacement strategies can be explained by the fact that the effective number of data elements stored in these caches cache is 3–4 times greater than those stored in the cache with the other replacement strategies. This causes the corresponding 3–4 times performance improvement. Hence, by adding only one extra field (repetition number) to existing fields of the cache (in other words, reduce the number of cache slots by 20% by using 5 fileds for LRU-compressed instead of 4 for LRU-standard) we can obtain about 200% of performance improvement, as can be seen from Figure 1).

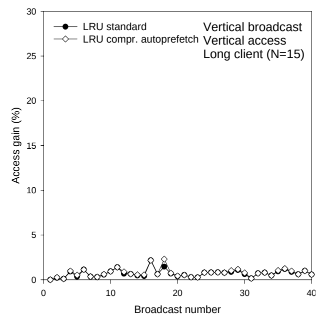
LRU-standard, LRU-global, LRU-local showed similar performance, which indicates that replacement strategies based on the oldest versions choose for replacement effectively the same data elements as the standard strategy based on the least recently used data element.

In all of our experiments we have considered all five replacement strategies but for clarity of the next 8 figures we will present data only for LRU standard and LRU-compressed autoprefetch. The behavior of the other three schemes is similar.

The next experiments were performed to see the cache behavior in the case of different combinations (vertical-vertical, vertical-horizontal, etc.) of the broadcast organization and client access. Figure 3 (*Size=25, Versions=25, Tries=80, Elements=3, Randomness Degree=0.5, Length=16, Vertical Broadcast, Vertical access, Bcast Number=40*) and Figure 4 (*Size=25, Versions=25, Tries=80, Elements=3, Randomness Degree=0.5, Length=16, Vertical Broadcast, Horizontal access, Bcast Number=40*) show that for short clients (*Elements= 3*) the cache performs better if access strategy is different from the broadcast organization. In such a way the cache reduces the gap between vertical broadcast-vertical access and vertical broadcast-horizontal access performances. For vertical broadcast-vertical access the LRU compressed autoprefetch cache reduces the access time from 2000 to 1900 time units (about 5%), and for vertical broadcast-horizontal access it reduces the access time from 3600 to 3200 time units (about 11%). Even more profound difference is shown in Figure 5 (*Size=25, Versions=25, Tries=80, Elements=15, Randomness Degree=0.5, Length=16, Vertical Broadcast, Vertical access, Bcast Number=40*) and 6 (*Size=25, Versions=25, Tries=80, Elements=15, Randomness Degree=0.5, Length=16, Vertical Broadcast, Horizontal access, Bcast Number=40*). In case of longer client (*Elements= 15*) the cache almost has no impact on the performance for vertical broadcast-vertical access case, but significantly (6% for LRU standard and 10% for LRU compressed autoprefetch) improves vertical broadcast-horizontal access performance.

Similar effect can be seen for Horizontal broadcast (Figure 7 – Figure 10 (*Size=25, Versions=25, Tries=80, Elements=3/15, Randomness Degree=0.5, Length=16, Vertical Broadcast, Vertical/Horizontal access, Bcast Number=40, Cache Size=50*). We again see smaller difference between horizontal broadcast-horizontal access and horizontal broadcast-vertical access for short client than for long client. The reason of this behavior is that due to its small size the short client behave similar to a size-one data element. Therefore both horizontal access and vertical access do not make much difference. This is not true for long client that cause significant differences in their behavior for different access organizations.

The last results shown in Figure 11 and Figure 12 (both *Size=25, Versions=25, Tries=80, Elements=15, Randomness Degree=0.5, Length=16, Vertical Broadcast, Horizontal access, Bcast Number=40*) indicate that increasing the cache size over a particular limit does not improve performance. For randomly uniformly accesses 15 data elements we eval-



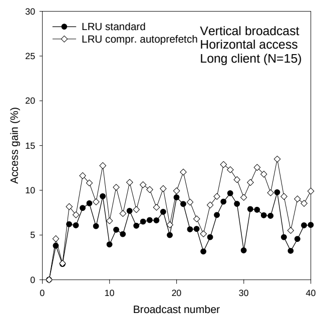
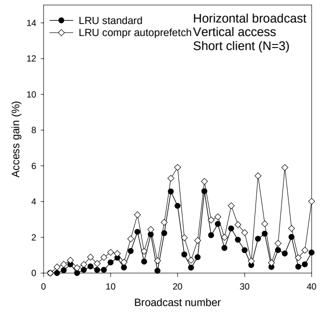


Figure 5: Access gain for vertical broadcast, vertical access, long client

Figure 6: Access gain for vertical broadcast, horizontal access, long client



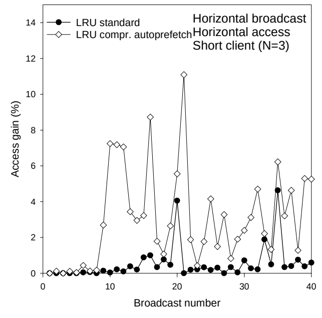


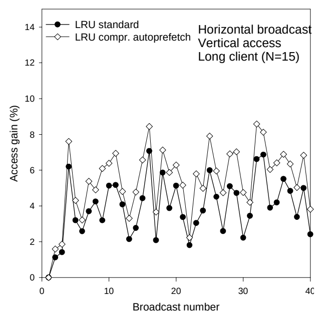
Figure 7: Access gain for horizontal broadcast-vertical access, short client

Figure 8: Access gain for horizontal broadcast-horizontal access, short client

uated access gains and hit rates for different cache sizes by using LRU compressed cache. In the beginning, when cache size increased from zero, the performance improved quickly, but since the cache size reached 50 the performance improvement shown down and has saturated for cache sizes greater than 70. In such a way, for 625 (25 data items times 25 versions) data elements the optimal cache size is about 50–70, which is 8–11% of the amount of data. Increasing of the data items to 50 and versions to 50 lead to increasing the optimal size of the cache to about 200–220 that also is 8–9% of the amount of data.

7. CONCLUSION

In this paper, we studied the effect of client caching in a multiversion push environment. Different cache scemes based on *Leaset Recently Used* replacement method have been evaluated. The use of our compression technique 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.



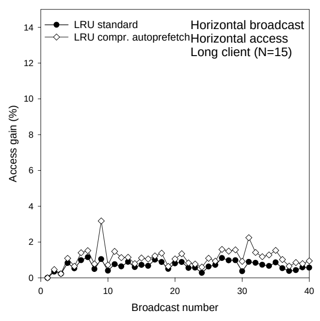
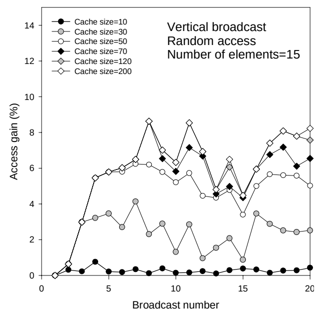


Figure 9: Access gain for horizontal broadcast-vertical access, long client

Figure 10: Access gain for horizontal broadcast-horizontal access, long client



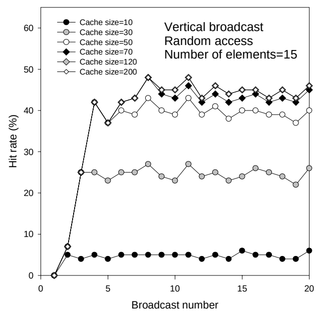


Figure 11: Access gain for different cache size, LRU-compressed cache

Figure 12: Hit rate for different cache sizes, LRU-compressed cache

8. REFERENCES

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

[2] 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

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

[4] J. Shanmugasundaram, A. Nithrakashyap, R. Sivasankaran and K. Ramamrithamt. Efficient Concurrency Control for Broadcast Environments. *ACM SIGMOD Conference* (1999) 85–96

[5] O. Shigiltchoff, P K. Chrysanthis and E. Pitoura. Multiversion Data Broadcast Organizations. *6th ABDIS Conf.*, (2002) 135-148

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

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