

# Broadcast Data Organizations and Client Side Cache \*

Oleg Shigiltchhoff  
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 devices due to their resource limitations. 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* [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. 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 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]. The same problem 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 channel 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. Further, multiversion broadcast provides clients with the possibility of accessing multiple server states.

A close examination of broadcast applications such as stock trading and E-Commerce, led 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 browse) queries in which a client accesses data and versions randomly. This prompted us to ask the following question that motivated this work: *What is the impact of the broadcast organization on performance, determined by the access time and power consumption at the clients with different access patterns?*

We proposed several broadcast organizations that are variants of two basic multiversion organizations: *Horizontal* and *Vertical* [5]. Also, we proposed a compression scheme along the lines of *Run Length Encoding* (RLE) [6], applicable to both of these organizations.

\*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.

Our results showed that the average client performance is significantly affected by the broadcast organization, that is, where to place the data and the old versions [5]. Specifically, if the primary interest of clients is in “historical” queries, the best broadcast organization is Horizontal, whereas if it is in “snapshot” queries, the best organization is Vertical. In case of mixed environment it is possible to support an adaptive broadcast due to the flexibility of the broadcast format.

In this paper, we study the effect of client-side caching in 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 assumed system model. We discuss broadcast organization and our compression scheme in Section 3 and client side caching in Section 4. We present our experimental platform and our experimental results in Sections 5 and 6, respectively.

## 2 System Model

In a broadcast dissemination environment, a data server periodically broadcasts data to a large client population. Each period 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” or on-demand scheme). We assume that all updates are performed at the server and disseminated from there.

Without loss of generality, in this paper we consider a broadcast that disseminates a fixed number of data items. The values of the data items may or may not change between two consecutive bcycles. In multiversion broadcast, a server constantly broadcasts a fixed number of versions for each data item. At each bcycle, 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 a server 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). 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 to support selective tuning and reduce power consumption, often at the cost of access time. In this paper, we focus only on client-side caching and broadcast organization 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 a 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.

## 3 Broadcast Organization

**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 ( $Dval$ ) of the items. 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: horizontally or vertically. 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, but differ in the order in which they broadcast the data values [5].

**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 (or power for tuning) for processing time (or power for computing). We proposed a simple compression scheme that exhibits the above properties [5].

Our compression scheme is based on the observation that data values do not always change from one version to another:  $Dval[i, k] = Dval[i, k+1] = \dots = Dval[i, k+N] = v$ , where the value of the item having  $Did=i$  remains equal to  $v$  for  $N$  consequent versions. Clearly, there is no reason to broadcast all versions of a data item 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, any sequence of elements with repetitive data values is replaced with the first element of the sequence and the number of additional repetitions ( $Dval$ , *Repetition*) [5]. For instance a sequence of 1 1 1 1 is transformed in the compressed value (1, 3).

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

$[Did[Vno(Dval, Repetition)]]^* \text{ sequences,}$   
and the Vertical broadcast produces  
 $[Vno[Did(Dval, Repetition)]]^* \text{ sequences.}$

## 4 Client side cache

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.

### 4.1 LRU Variants

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* replacement method (which we will refer to as *LRU-standard*) to discard data 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:  $TimeStamp$ , which is the cycle during which the element was last read. Following the LRU policy, when the cache becomes full, the cache element with the oldest (i.e., smallest)  $TimeStamp$  is selected for replacement.

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 to the element having the oldest version, choosing for replacement the data item with the oldest  $Vno$  instead of the data item with the oldest  $TimeStamp$ . 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. We call this scheme *LRU-global* and has the same space requirements as LRU-standard<sup>1</sup>.

LRU-global is expected to behave effectively in the same way as LRU-standard for skewed data accesses, 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  $Did$  to be replaced, the LRU-global is used.

### 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, checking if the requested version is within the interval  $Vno$  and  $Vno + f$ . 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 pro-

<sup>1</sup>An alternative is to select the victim randomly. In such a case LRU-global needs only three fields:  $[Did, Vno, Dval]$ .

posed 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 broadcast, 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 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 been 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 and it can even increase the span of local transactions beyond the one supported by the server. This fact is also confirmed by our experimental results.

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

## 5 Experimental Testbed

Our simulator 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

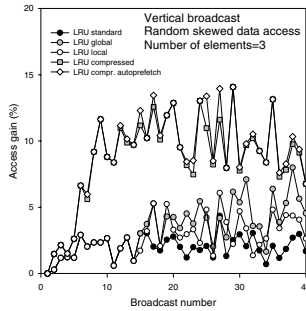
Parameter	Parameter description
<i>Compression</i>	Basic Sequential broadcast Compressed broadcast
<i>Broadcast Number</i>	Number of broadcast cycles
<i>Bcast Type</i>	Vertical broadcast (VB) Horizontal broadcast (HB)
<i>Size</i>	Number of data items
<i>Versions</i>	Number of versions
<i>Randomness Degree</i>	0–1, (0: all versions have the same value, 1: versions are completely independent)
<i>Length</i>	Size of a data element; default = 16 (size of an auxiliary symbol is 1)
<i>Elements</i>	Number of the requested data items
<i>Access Type</i>	Random uniform distribution Random Zipf distribution of data Random Zipf distribution of versions Vertical access (VA) Horizontal access (HA)
<i>StrideN</i>	Number of the strides for Vertical/Horizontal accesses
<i>StrideL</i>	Length of the strides for Vertical/Horizontal accesses
<i>Cache Size</i>	The size of cache
<i>Tries</i>	Number of the same experiments to reduce deviations

Table 1. Simulation Parameters

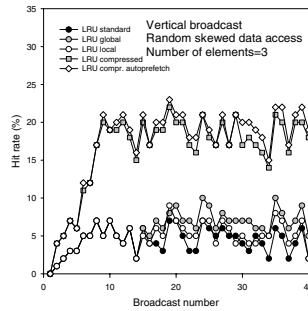
data items in the set is determined by the *Size* parameter and the number of versions by the *Versions* parameter. The communication is based on the client-server mechanism via sockets. For simplicity the data values are integer numbers from 0 to 9.

The simulator runs the server in two modes, corresponding to the two broadcast organizations, namely Vertical Broadcast (**VB**) and Horizontal Broadcast (**HB**) (determined 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*.

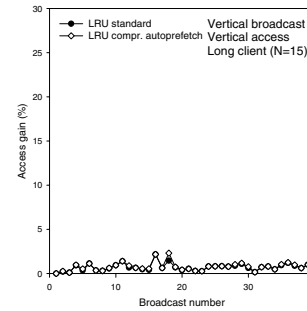
The client generates the data elements (various versions of data items) to access before tuning into the broadcast. The parameter *Elements* determines the number of the data elements to be accessed. The client searches the data by using five different access types (determined by *Access Type* parameter): In *Random with uniform distribution*, *Did*'s and their *Vno*'s are determined randomly with uniform distribution to simulate the case when all versions of all data elements are equally important for a client. In *Random with skewed distribution*, *Did*'s are determined randomly with Zipf distribution, *Vno*'s are determined randomly with uniform distribution. In *Random with skewed distribution of versions*, *Did*'s are determined randomly with uniform distribution, *Vno*'s are determined randomly with Zipf distribution. In *Vertical access (VA)* and *Horizontal access (HA)*, the requested data elements are grouped into a number of strides (determined by *StrideN*), each containing *l* el-



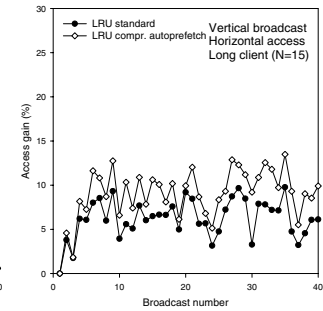
**Figure 1. Access gain with Caching**



**Figure 2. Hit rate for cache schemes**



**Figure 3. Access gain for VB-VA**



**Figure 4. Access gain for VB-HA**

elements (determined by *StrideL*). (Clearly,  $StrideL * StrideN = Elements$ .) For example, if  $StrideN=2$  and  $StrideL=5$ , for VA, the client searches for two versions (determined randomly with uniform distribution) of 5 consecutive data elements. For HA, 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. If a client tunes in the middle of a broadcast, it sleeps and 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 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. Note that for the broadcast with the dynamic access, the performance is determined by how fast the data is found in the last cycle. Thus, by simulating only the last broadcast, it is possible to estimate the performance for both cases.

While the client is reading from the broadcast, 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.

To evaluate the effect of the client side cache the client

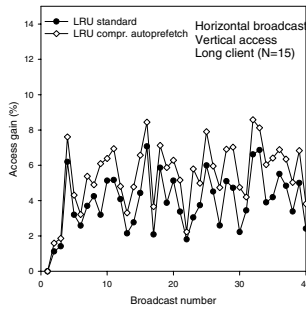
reads up to 40 different cycles (parameter *Broadcast Number*). The cache size was determined by *Cache Size* parameter. In order to estimate confidence intervals we performed the measurements 80 times (parameter *Tries*). Then we calculate the average access time and the corresponding standard deviation which are shown in our graphs. The discussed parameters are summarized in Table 1.

## 6 Performance Results

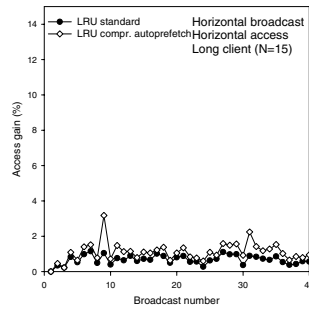
We have evaluated all the cache replacement strategies proposed above: LRU-standard, LRU-global, LRU-local, LRU-compressed, LRU-compressed-autoprefetch. In all the experiments reported below, the parameters settings are: *Size=25*, *Versions=25*, *Randomness Degree=0.5*, *Broadcast number = 40* and *Length=16* which may correspond to 16 bytes.

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 *Elements=3*, *Vertical Broadcast*, *Random skewed access*, *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, addition of 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 fields for LRU-compressed instead of 4 for LRU-standard) improves the performance about 200% (see Figure 1).

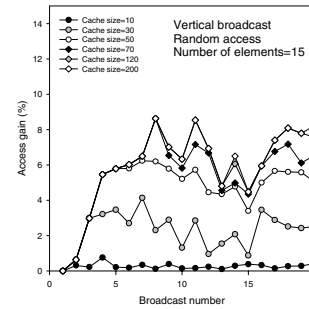
LRU-standard, LRU-global, and LRU-local showed similar performance, which indicates that replacement strate-



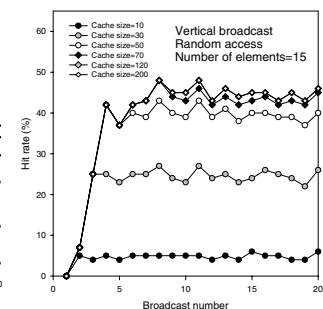
**Figure 5. Access gain for HB-VA**



**Figure 6. Access gain for HB-HA**



**Figure 7. Access gain vs. cache size**



**Figure 8. Hit rate vs. cache size**

gies 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 for the next 4 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 with *Elements*=15 to see the cache behavior in the case of different combinations (vertical-vertical (VB-VA), vertical-horizontal (VB-HA)) of the broadcast organization and client access. Figure 3 and Figure 4 show that 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 almost has no impact on the performance, and for vertical broadcast-horizontal access it reduces the access time about 6% for LRU standard and 11% for LRU compressed autoprefetch).

Similar effect can be seen for HB and VA/HA. In Figures 5 and 6, we again see that if broadcast organization and access method are different, HB and VA in this case, the access gain is greater than if they are the same.

The last results, shown in Figures 7 and 8, indicate that increasing the cache size over a particular limit does not improve performance. For randomly uniformly accesses and 15 data elements, we evaluated 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 after the cache size reached 50, the performance improvement slowed down and leveled for cache sizes greater than 70. This means that 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 schemes based on *Least 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.

## References

- [1] S. Acharya, M. Franklin, S. Zdonik, "Balancing Push and Pull for Data Broadcast", *ACM SIGMOD Conf.*, pp. 183–194, 1997.
- [2] J. Jing, A. H. Elmargamid, S. Helal, R. Alonso, "Bit-Sequences: An adaptive Cache Invalidation Method in Mobile Client/Server Environment", *ACM/Baltzer MONET*, 2(2):115–127, 1997.
- [3] E. Pitoura and P. K. Chrysanthis, "Exploiting Versions for Handling Updates in Broadcast Disks", *25th VLDB Conf.*, pp. 114–125, 1999.
- [4] J. Shanmugasundaram, A. Nithrakashyap, R. Sivasankaran and K. Ramamritham, "Efficient Concurrency Control for Broadcast Environments", *ACM SIGMOD Conf.*, pp. 85–96, 1999.
- [5] O. Shigiltchhoff, P. K. Chrysanthis and E. Pitoura, "Multiversion Data Broadcast Organizations", *6th ADBIS Conf.*, pp. 135–148, 2002.
- [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(4):351–372, 1998.