A Multi-version Cache Replacement and Prefetching Policy for Hybrid Data Delivery Environments

André Seifert, Marc H. Scholl

University of Konstanz
P.O. Box D188
D-78457 Konstanz
Germany
{Andre.Seifert, Marc.Scholl}@uni-konstanz.de

Abstract

This paper introduces MICP, a novel multi-version integrated cache replacement and prefetching algorithm designed for efficient cache and transaction management in hybrid data delivery networks. MICP takes into account the dynamically and sporadically changing cost/benefit ratios of cached and/or disseminated object versions by making cache replacement and prefetching decisions sensitive to the objects' access probabilities, their position in the broadcast cycle, and their update frequency. Additionally, to eliminate the issue of a newly created or outdated, but re-cacheable, object version replacing a version that may not be re-acquired from the server, MICP logically divides the client cache into two variable-sized partitions, namely the REC and the NON-REC partitions for maintaining re-cacheable and non-re-cacheable object versions, respectively. Besides judiciously selecting replacement victims, MICP selectively prefetches popular object versions from the broadcast channel in order to further improve transaction response time. A simulation study compares MICP with one offline and two online cache replacement and prefetching algorithms. Performance results for the workloads and system settings considered demonstrate that MICP improves transaction throughput rates by about 18.9% compared to the best performing online algorithm and it performs only 40.8% worse than an adapted version of the offline algorithm P.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment

Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002

1. Introduction and Motivation

A mobile hybrid data delivery network is a communication infrastructure that allows bi-directional communication between a fixed host, also called Mobile Support Station, and mobile clients either through a low bandwidth point-to-point channel or to all active clients through a high bandwidth broadcast channel. As the name indicates, hybrid data delivery combines push- and pull-based data delivery in an efficient way by broadcasting the data items that are of interest to a large client population and unicasting less popular data items only when they are requested by the clients. While a combined push/pull data delivery mode has many advantages such as user scalability, bandwidth efficiency, support for disconnections, etc., it also suffers from two major disadvantages: First, the client data access latency depends on the length of the broadcast cycle for data items that are fetched from the broadcast channel. Second, since most of the data requests can either be satisfied by the clients themselves or the broadcast channel, the server lacks clear knowledge of the client access patterns. While the latter weakness can be diminished by regularly sending data usage profiles to the server or the technique proposed in [SRB97], the former can be relaxed by designing and deploying an efficient *cache replacement* and *prefetching policy* that is closely coupled with the transaction manager of the mobile client.

Due to the physical constraints immanent in any mobile communication environments, such as high communication latency, low network bandwidth in the uplink direction, intermittent connections, etc., the majority of applications executed at mobile clients are of the *read-only* type. Since there are typically many more reads than writes in production database systems [HSY99a, HSY99b], most of the data contentions among transactions result from read-write conflicts. An alternative approach to reduce, but not eliminate, data

contention caused by simultaneously running transactions is to maintain two or multiple versions of database items. *Multi-versioning*, also called transient versioning, is effective in relaxing read-write conflicts that may occur when read-write and read-only transactions are processed concurrently. Interference between read-write and read-only transactions can be diminished by forcing read-only transactions to *read obsolete database items*, thus increasing the level of concurrency. For example, forcing a read-only transaction T_i to read data granules that were up-to-date by the time T_i started or sometime before, allows the transaction scheduler to serialize T_i before all concurrently active read-write transactions, thereby improving the performance of the system. However, diminishing the data contention between read-write transactions by means of maintaining multiple data versions is not as effective as for read-only transactions since read-write transactions typically need to access up-to-date object versions to provide serializability guarantees. Therefore, in this paper we concentrate on multi-versioning as the means of improving the performance of mobile applications issuing read-only transactions.

1.1 Multi-version Client Caching

So far, we have indicated that multi-versioning is a valuable and practicable approach to process read-only and read-write transactions concurrently. In what follows, we highlight various issues a mobile cache and prefetch manager needs to take into account so that key performance metrics such as throughput is maximized and abort rate is minimized. In particular, we propose a new combined caching and prefetching algorithm ideally suited for mobile clients that use multi-version concurrency control (MVCC) protocols [SS01] to relax conflicts between concurrent transactions. Since data caching is an effective (if not the most effective) and therefore indispensable way of reducing transaction response times [CK89], cache replacement policies have been extensively studied in conventional database management systems [EH84, OOW93, JS94, JN98, LKN+99]. Since conventional caching techniques are inefficient for mobile networks where communication channels form an intermediate memory level between the client and the server and where communication quality varies over space and time, mobile caching policies [AAF+95, TS97, KL98, XHL+00] have been designed, that are tailored to the peculiarities and constraints of the mobile environment. However, to our knowledge, none of the proposed caching strategies designed either for the stationary or for the mobile client-server architecture tackles the problem of managing multi-version client buffer pools efficiently. Multi-version client caching differs from mono-version caching by at least two key observations. First, the *cost/benefit ratio* of dissimilar versions of a data item in the client cache may *vary over time* depending on the storage behavior of the server, i.e., if the server discards an object version useful for the client, this version's cost/benefit ratio increases since it cannot be re-acquired from the server. Second, versions of different data items may for the same reason have *dissimilar cost/benefit ratios* despite being equally likely to be referenced.

The following example illustrates the aforementioned peculiarities. Suppose a diskless mobile client executes a read-only transaction T_i with begin of transaction (BOT) serializability guarantees [SS01], i.e., T_i is always forced to observe the most recent object versions that existed by its starting point. Assume the start timestamp TS of T_i is 1 and the database consists of four objects {A, B, C, D}. The client cache size is small and may hold only two object versions. Further, it is up to the client how many versions of each object it maintains. For space and time efficiency reasons, the database server holds a restricted number of versions, namely the last two committed versions of each data item. Additionally, assume the client's access pattern is totally uniform, i.e., each object is equally likely to be accessed. At the logical time 5 (read-write transaction with commit timestamp 5 has just terminated) the client cache holds the set of objects $\{A_0, B_0\}$ and the server keeps objects $\{A_1, A_3, B_0, B_1, C_0, C_4, D_2, D_5\}$. Note that the subscripts assigned to object versions correspond to the commit timestamp of the transaction that created the respective version. Now, suppose the client needs to read a transaction-consistent version of object C. Since there is no cache-resident version of object C, the client fetches the missing object from the server. By the time the object arrives at the client, the local cache replacement policy needs to select a replacement victim to free some cache space. In this case, a judicious cache replacement strategy would evict B_0 since it is the only object version that can be re-acquired from the server, i.e., a cache replacement policy suitable for a multi-version cache needs to incorporate both *probabilistic information* on the likelihood of object references in the future and *data re-acquisition costs*.

1.2 Multi-version Client Prefetching

Apart from demand-driven caching and judicious eviction of object versions from the cache, another technique that can be used to reduce on-demand fetches is data prefetching, by which the client optimistically fetches versions of data items from the server and/or broadcast channel into the cache in expectation of a later request. Since prefetching, especially if integrated with caching, strongly affects transaction response time, various combined caching and prefetching techniques have been studied in stationary computing [PZ91, CFK+95, TPG97, JN98]. Work on prefetching in mobile data broadcasting environments has been conducted by [AFZ96]. Again, as for caching, prefetching mechanisms proposed in the literature are inefficient for mobile data dissemination

applications utilizing MVCC schemes to manage read-only transactions. The reasons are twofold: First, algorithms, such as PIX and LIX, proposed for data prefetching in broadcast environments [AFZ96] are based on *simplified assumptions* such as no database updates and no use or availability of uplink communication facilities. Second, and more importantly, all previous prefetching strategies were designed for *mono-version database systems* and therefore lack the ability to make proper prefetching decisions in a multi-version environment. In contrast, we base our model on more realistic assumptions and develop a prefetching algorithm that is multi-version compliant. As prefetching may unfold its total strength if deeply integrated with data caching, our prefetching algorithm uses the same cost/benefit metric for evaluating prefetching candidates as the cache replacement algorithm. To ensure that the prefetching algorithm does not hurt, but rather improves performance, we allow prefetches of only those object versions that have been recently referenced and whose cost/benefit ratio exceeds the value of any cached object version.

1.3 Paper Structure

The paper is structured as follows: Section 2 describes the model underlying MICP (**M**ulti-version **I**ntegrated **C**aching and **P**refetching algorithm). Section 3 contains a detailed description of MICP and is concluded by the introduction of an implementable version of MICP, called MICP-L. Section 4 reports on detailed experimental results that show the superiority of our algorithm compared to previously proposed caching and prefetching policies and presents the performance gap of MICP compared to an offline algorithm having full knowledge of the client access pattern. The paper's conclusions and summary are to be found in Section 5.

2. System Design and Assumptions

The primary components of the data delivery architecture are the database server, the hybrid network, and the mobile clients. The following subsections depict the design of the hybrid data delivery network, the organization and structure of the client and server cache, and the client cache invalidation and synchronization scheme.

2.1 Hybrid Data Delivery Model

We have chosen a hybrid data delivery system as the underlying network architecture for MICP since a hybrid push/pull scheme has the ability to mask the disadvantages of one data delivery mode by exploiting the advantages of the other. Since broadcasting is especially effective when used for popular data, we assume that the server broadcasts only such data that is of interest to the majority of the client population. Our broadcast structure is logically divided into three segments of varying size: a) *index segment*, b) *data segment*, and c) *concurrency control information segment*. Each minor cycle is supplemented with an index to eliminate the need for the clients to listen to the broadcast continuously in order to locate the desired object version on the channel. We choose (1, m) indexing [IVB97] as the underlying index allocation method by which the whole index, containing, among other things, a mapping between the objects disseminated and the identifiers of the data pages in which the respective objects appear, is broadcast *m* times per major broadcast cycle. The data segment, on the other hand, solely contains hot-spot data pages. Note that we assume a flat broadcast disk approach for page scheduling, i.e., each and every hot data page is only broadcast once within a major cycle. For data consistency reasons, we model the broadcast program so that all data pages disseminated are a consistent snapshot as of the beginning of each major broadcast cycle. Thus, the modified or newly created object versions committed after the beginning of an ongoing major broadcast cycle will not be included in any data segment. To guarantee cache consistency despite server updates, each minor broadcast cycle is preceded with a concurrency control report as described in Section 2.2.2.

The second core component of the hybrid data delivery system is the point-to-point channel. A point-to-point channel may be utilized by the client to request locally missing or non-scheduled object versions from the server. Further, clients are allowed to use the back channel to the server when a required object version is scheduled for broadcasting, but its expected arrival time is above the uplink usage threshold [AFZ97] dynamically set up by the server. This optimization helps clients improve their response times.

2.2 Client and Server Cache Model

Conventional caching and prefetching strategies are typically page-based since the optimal unit of transfer between systems resources are pages with sizes ranging from 8 KB to 32 KB [GG97]. In mobile data delivery networks caching and prefetching data on a coarse granularity such as pages is inefficient due to the physical constraints and characteristics of the mobile environment. As mentioned before, the communication in client-server direction is handicapped by low bandwidth wireless channels. Choosing page-sized granules to be the unit of transfer for data uploads would be a waste of bandwidth compared to sending objects of much smaller size in case of a low degree of locality. Since a data broadcasting server typically serves hundreds of thousands of mobile clients and each client tends to have its own set of frequently accessed data items, it is not unrealistic to assume that the physical data organization of the server may not comply with the individual access pattern of the clients. Therefore, in order to increase the hit ratio of the

client cache and to save scarce uplink bandwidth resources, we deploy our caching and prefetching scheme on an object basis. However, to allow clients to cache pages as well, we opt for a *hybrid client cache* consisting of a *small-size page cache* and a *large-size object cache*. While the page cache is used as working storage memory to extract and copy requested or prefetched object versions into the object cache, the object cache's task is to efficiently maintain those object versions, i.e., it is used as data storage memory. Note that our intuition behind such a cache structure was experimentally confirmed by a performance study [DMF+90] demonstrating that an object-based caching architecture is superior to a page-based one when physical clustering is poor and the client's cache size is small relative to the size of the database, which is typically the case in mobile environments. We further assume that the broadcast server also manages its cache by a *hybrid of page* and *object caching*. The structure of the server cache is similar to the one described in [Ghe95] with the exception that multiple versions of objects may be maintained for concurrency control purposes. Again, the use of both cache types allows us to exploit the benefits of each. While the page cache is useful for efficiently serving broadcast requests, installation reads [OS94], etc., the object cache is attractive for recording object modifications.

2.2.1 Version Control Model

To implement version tracking, each object version is assigned a monotonically increasing timestamp that reflects the logical time when it was created. Whenever a read-write transaction issues a write operation on an object X and commits within the major broadcast cycle $MBC_{i,j}$, it creates a new version of X, denoted $X_{i,j}$, where the subscripts *i* and *j* symbolize the number of the major broadcast cycle (MBC) and minor broadcast cycle respectively, that existed by the transaction's commit time. Associating timestamps to object versions is required in order to distinguish between different versions of the same object and to synchronize read-only transactions with committed and/or currently active read-write transactions [SS01]. Since multi-versioning imposes additional memory and processor overheads on the clients and the server, we assume that the number of versions maintained in the involved memory levels is restricted. For clients it is sufficient to maintain at most two versions of each database object at any time since we assume that clients do not execute transactions in parallel. In contrast, the server may need to maintain every object version in order to guarantee that any read-only transaction can read from a transaction-consistent database snapshot. Since such an approach is impracticable, we assume that the server maintains a fixed number of versions (see Section 4.5 for a performance experiment on this issue).

2.2.2 Cache Synchronization Model

Hoarding, caching, or replicating data in the client cache is an important mechanism for improving data availability, response time, and reducing the power-consumption at mobile clients. However, data updates at the server make cache consistency a challenge. An effective cache synchronization and update strategy is needed to ensure consistency and freshness between the data cached at the client and the original data at the server. Although *invalidation messages* are space and time efficient compared to *propagation messages*, they lack the ability to update the cache with new object versions. Due to the inherent tradeoffs between propagation and invalidation, we employ a hybrid of the two techniques. On the one hand, the broadcast server periodically disseminates a concurrency control report, or CCR, which is a simple structure that contains, in addition to concurrency control information, identifiers and values of versions of those objects modified during the last minor broadcast cycle [SS01]. Based on those reports, mobile clients operating in connected mode can easily update their caches at low costs. However, since CCRs contain only concurrency control information wrt. the last minor broadcast cycle, those reports are useless for cache synchronization of recently reconnected clients that had missed one or more CCRs. To resolve this problem, we assume that the server maintains the update history of the last *w* MBCs as proposed in [BI94]. This history is used for client cache invalidation as follows: when a mobile client wakes up from a disconnection, it waits for the next CCR to appear and checks whether the following equation is valid: $t_{CCR,c} < t_{CCR,l} + w$, where $t_{CCR,c}$ denotes the timestamp of the current CCR and $t_{CCR,l}$ represents the timestamp of the latest CCR report received by the client. If so, a dedicated invalidation report (IR) can be requested by the client to invalidate its cache properly. An IR is implemented as a list of tuples that contains the same elements as a CCR, with the exception that only the identifiers of the modified objects are maintained. If the client was disconnected for more than *w* MBCs, the entire cache contents has to be discarded upon reconnection.

3. New Integrated Algorithm

The design of MICP consists of two complementary algorithms that behave synergistically. The first algorithm, responsible for selecting replacement victims, is called *PCC* (**P**robabilistic **C**ost-based **C**aching) and the second one dealing with data prefetching is denoted *PCP* (**P**robabilistic **C**ost-based **P**refetching). While PCC may be employed without PCP in order to save scarce CPU processing and battery power of mobile devices, PCP's potential can be exploited by coupling it with a cache replacement policy that uses the same or similar metric for decision making.

3.1 The Multiversion Cache Replacement Algorithm

The major goal of any cache replacement policy designed either for broadcasting or for unicasting environments is to minimize the *average response time* a user/process experiences when requesting data items. Traditional cache replacement policies try to achieve this goal by making use of two different approaches. The first category requires information from the database application. That information can be either obtained from the application directly or from the query optimizer that processes queries of the corresponding application. The second category of replacement algorithms bases its decisions on observations of past access behavior. The algorithm proposed in this paper belongs to the latter group, extends the LRFU policy [LKN+99] and borrows from the 2Q algorithm [JS94]. Like LRFU, PCC quantifies the probability of an object being re-referenced in the future by associating with each object a score value that reflects the effects of the ***frequency and recency of past references***. More precisely, PCC computes a combined recency and frequency factor for each object X whenever it is referenced by a transaction, according to the following formula:

$$CRF_{n+1}(X) = 1 + 2^{-(\lambda \times (t_c - t_l(X)))} \times CRF_n(X) \quad (1)$$

where $CRF_n(X)$ is the computed value of the combined recency and frequency factor of object X over the last *n* references, t_c denotes the reference number associated with the current time of object reference, $t_l(X)$ is the reference number assigned to object X when it was last accessed, and λ ($0 \le \lambda \le 1$) is a kind of "slide controller" that allows PCC to weigh the importance of recency and frequency information for the replacement selection. Note that if λ converges towards 0 PCC behaves more like an LFU policy and, contrarily, with λ approaching 1 it acts more like an LRU policy.

In contrast to the LRFU algorithm, PCC bases its replacement decisions not only on recency and frequency information of historical reference patterns, but additionally makes use of three further factors besides the future reference probability of objects as expressed by CRF. First, in order to reflect the situation that instantaneous access costs of data items scheduled for broadcasting are non-constant due to the serial nature of the broadcast medium, PCC's replacement decisions are sensitive to the actual state and contents of the broadcast cycle. More precisely, PCC accounts for the ***costs of re-acquiring object versions*** by evicting those versions that have low probabilities of access and low re-acquisition costs. To provide a common metric for comparing costs of ejecting object versions that can be re-cached from the broadcast channel and/or database server, we measure re-acquisition costs in terms of broadcast units. Since we assume that the content and organization of the broadcast program does not change significantly between consecutive MBCs and the clients are aware of the position of each object version in the MBC due to (1, m) indexing, determining the number of units till an object version re-appears on the channel is straightforward. Estimating the costs of re-fetching a requested version from the server is more difficult since that value depends on parameters such as the current network and server load and the effect of caching at the server. To keep our caching algorithm as simple as possible, we use the uplink usage threshold as a simple guideline for approximating data fetch costs. Since the uplink usage threshold provides a tuning knob to control the server and network utilization and, thus, affects data fetch costs, its dynamically fixed value correlates with the data fetch latency a client experiences when requesting data items from the server. If the threshold is high, the system is expected to operate under a high workload and therefore data retrieval costs are high as well. In what follows, we denote the re-acquisition costs of an object version $X_{i,j}$ at time *t* by $RC_t(X_{i,j})$.

A second parameter PCC uses to make replacement decisions is the ***frequency of object updates***. As noted before, multi-version database systems suffer from high processing and storage overheads if the number of versions maintained by the server is not restricted. However, limiting the number of versions negatively affects the likelihood of data requests from the clients being satisfied by the server. To account for the probability that an object is updated within a major broadcast cycle, the server (re-)computes the update frequency of an object X at the end of each MBC whenever a new version of X has been created in the course of the last MBC by the following formula:

$$UF_{n+1}(X) = (1-\alpha) \times UF_n(X) + \alpha \times \frac{(ID_c - ID_l(X))}{\overline{UR}} \quad (2)$$

where α is an aging factor to adapt to changes in access patterns by assigning higher weights to recent updates, ID_c is the monotonically increasing identifier of the current MBC, $ID_l(X)$ is the identifier of the MBC where object X was last updated, $UF_n(X)$ is the combined update frequency value of the previous *n* updates of X, and $\overline{UR}$ denotes the average number of updates (update rate) within an MBC.

Last but not least, a replacement policy suitable for supporting MVCC protocols needs to take into account the ***server's storage policy***. Besides the update frequency of each data item, the version maintenance strategy of the server affects the likelihood that an obsolete object version can be re-acquired once evicted from the client cache. The more versions of an individual object are kept by the server, the higher the probability that the server can satisfy requests for that object. PCC incorporates the versioning policy of the server by means of two complementary methods: First, it computes re-acquisition costs of in-memory object versions based on their re-fetch

probabilities (see Equation 4 and 5) and second, it takes care of non-re-cacheable object versions by placing them into a dedicated partition of the client object cache, called ***NON-REC*** (non-re-cacheable), while re-cacheable object versions are maintained in the ***REC*** (re-cacheable) part of the client cache.

The reason for cache partitioning is to prevent undesirable replacement of non-re-cacheable versions by referenced or prefetched re-cacheable object versions. With regard to the size of the cache partitions we experimentally established that NON-REC should not exceed 50% of the overall client cache size and REC should hold at least 50% of the objects stored in the cache. The justification for those values is as follows: the majority of users issuing read-only transactions want to observe up-to-date object versions, i.e., they usually initiate queries with either BOT or strict forward BOT read guarantees [SS01]. The assumption that clients do not execute more than one read-only transaction at a time and transactions are issued with at least BOT data currency requirements implies that at their starting point only up-to-date object versions are useful, i.e., NON-REC is empty at this stage. As transactions progress, more and more useful object versions may become non-re-cacheable and need to be placed into NON-REC. Since the storage space needed to maintain non-re-cacheable object versions is not known in advance and depends on such factors as transaction length, data currency guarantees under which transactions run, data update frequency at the server, etc., PCC adapts to this situation by changing the size of NON-REC dynamically. That is, as demand for storage space in NON-REC arises, PCC extends NON-REC by re-allocating object slots from REC to NON-REC as long as its size does not exceed 50% of the overall cache size. Without this bound, the system performance could degrade due to insufficient cache space reserved for up-to-date or nearly up-to-date (re-cacheable) versions. It is important to note that the described cache structure is suitable for managing *read-write transactions* as well. In this case, the cache manager allocates all the available cache space to REC which then keeps only up-to-date object versions.

As all of the aforementioned parameters influence replacement decisions, it is obvious that there is a need for a single combined performance metric to enable comparison of those values that would be meaningful for the cache manager. To this end, we combine the estimates given above into a single performance metric, called probabilistic cost/benefit ratio (PCB), which is computed for each cache-resident object version $X_{i,j}$ at eviction time t as follows:

$$PCB_t(X_{i,j}) = CRF_t(X) \times (T_{Hit}(X_{i,j}) + T_{Miss}(X_{i,j})). \quad (3)$$

In the above formula, $CRF_t(X)$ denotes the re-reference probability of object X at time t, T_{Hit} is the weighted time in broadcast units it takes to re-fetch object version $X_{i,j}$ if evicted from the cache, and T_{Miss} represents the weighted time required to re-process all completed read operations of the active read-only transaction in case it needs to be aborted since $X_{i,j}$ is not any more system-resident and thus cannot be accessed. The time to service an object version request that hits either the broadcast channel or the server memory is the product of the following parameters:

$$T_{Hit}(X_{i,j}) = (1 - UF(X)^{N_{version}(X_{i,j})}) \times RC(X_{i,j}), \quad (4)$$

where $N_{version}(X_{i,j})$ denotes the number of versions of object X with commit timestamps equal to or older than $X_{i,j}$ currently kept by the server. Further on, we compute $T_{Miss}(X_{i,j})$ as a weighted approximation of the amount of time, denoted $T_{Rep}(X_{i,j})$, it would take the client to restore the current state of the active read-only transaction for which $X_{i,j}$ is useful in case it has to be aborted due to a fetch miss of $X_{i,j}$:

$$T_{Miss}(X_{i,j}) = UF(X)^{N_{version}(X_{i,j})} \times T_{Rep}(X_{i,j}), \quad (5)$$

where $T_{Rep}(X_{i,j})$ is the sum of the weighted retrieval and processing times for the object versions that have to be obtained from the broadcast cycle and for those found in the client cache (we assume the length of processing a client cache hit to be one broadcast tick), and is computed as follows:

$$T_{Rep}(X_{i,j}) = 0.5\overline{L} \times (1 - \overline{CHR}) \times N_{read} + \overline{CHR} \times N_{read}. \quad (6)$$

In formula 6, $\overline{L}$ represents the average length of the MBC, $\overline{CHR}$ denotes the average client cache hit rate, and the expression N_{read} symbolizes the total number of read operations executed so far by the active read-only transaction that is forced to read object version $X_{i,j}$ for data consistency reasons if object X is requested by the transaction. As formula 6 indicates, we assume that the average latency to fetch a non-cache resident object version into the client memory takes half a broadcast period independent of whether that object appears on the broadcast channel or has to be requested through the point-to-point channel. We opted for this simplification to refrain the algorithm from further complexity inflation. The complete PCC algorithm invoked upon a reference to an object version can be found in the full version of the paper [SS02].

3.2 Multi-version Prefetching Algorithm

While PCC achieves the goal of improving transaction response times by caching requested object versions close to the database application, PCP tries to further reduce fetch latency by pro-actively loading useful object versions with high access probability and/or high re-acquisition costs into the cache in anticipation of their future reference. As uncontrolled prefetching without reliable information might not improve, but rather harm the performance, the greatest challenge of PCP is to

decide when and which object version to prefetch and which cache slot to evict when the cache is full. PCP tackles those challenges as follows: in order to behave synergistically with PCC, PCP bases its prefetching decisions on the same performance metric, namely PCB. Since calculating PCB values for every object version that flows past the client is very expensive, if not unfeasible, PCP computes those values only for a small subset of the potential prefetching candidates, namely ***recently referenced objects***. The reason for choosing this heuristic is the assumption that reference sequences exhibit temporal locality [Den72]. Temporal locality indicates that once an object has been accessed, there is a strong probability that the same object (either the same or different version) will be accessed again in the near future. To decide whether an object has recently been referenced clients need to maintain historical information on past object references. As will be explained later, we assume that clients retain such information for the last *r* distinct object accesses where *r* depends on the actual client cache size. Based on this statistical data, PCP selects its prefetch candidates by a simple policy. In order for a disseminated object version $X_{i,j}$ to qualify for prefetching, there must exist any recent entry for X in the reference history. The exact decision how recent an object reference has to be in order for the object to qualify for prefetching is left up to the client since the prefetching decision process is computationally expensive and has to be aligned to the client's resources. If the object qualifies for prefetching, PCP computes $X_{i,j}$'s PCB ratio and compares the score with the corresponding values of all cached data items. If $X_{i,j}$'s ratio is greater than the least PCB value of all cached versions then $X_{i,j}$ is prefetched and replaces the lowest valuable version. As for the PCC algorithm, prefetch candidates compete for the available cache space only with those versions that belong to the same cache partition.

Apart from prefetching current and non-current versions of recently referenced objects, PCP downloads from the broadcast channel all useful versions of data items that will be discarded from the server by the end of the MBC. The intuition behind this heuristic is to minimize the number of transaction aborts caused by fetch requests that cannot be satisfied by the server. A viable approach to reducing the number of fetch misses is to cache those versions at the client before they are garbage-collected by the server. To implement this approach, mobile clients need information as to whether a particular object version is disseminated for the last time on the broadcast channel. There are basically two ways how clients could receive such information. First and most conveniently, the server indicates whether an object version is about to be garbage-collected. That information could be provided by adding a respective field for each object version in the disseminated data pages. On the other hand, clients could determine whether an object version becomes non-re-cacheable by keeping track of the object version history. As the latter approach requires clients to have knowledge of the version management policy at the server, we opt for the first approach. Again, the complete pseudo-code of PCP can be found in [SS02].

3.3 Maintaining Historical Reference Information

It has been noted that MICP takes into account both recency and frequency information on data accesses in order to select replacement victims. Similar to LRFU, MICP maintains CRF values on a per-object basis that capture information on both recency and frequency of accesses. However, in order for MICP to be effective, such values need to be retained in client memory not only for cache-resident objects but also for evicted data items. The necessity to keep historical information of a referenced object even after all versions of this object have been evicted from cache was first recognized by [OOW93] and was termed "reference retained information problem". This problem arises from the fact that in order to gather both recency and frequency information, clients need to keep history information on recently referenced objects for some time. This is in particular required for determining the frequency of object references. If CRF values are maintained only for cached data items and the size of the client cache is relatively small compared to the database size, then there exists a danger that MICP might overestimate the recency information since frequency information is rarely available. On the other hand, storing reference information consumes valuable memory space that could otherwise be used for storing data objects.

To limit the memory size allocated for historical reference information, [OOW93] suggests storing that information only for a limited period of time after the reference had been recorded. As reasonable rule of thumb for the length of this period they use the *Five Minute Rule* [GG97]. However, applying it in a mobile environment may be inappropriate for the following reason: a time-based approach for keeping reference information ignores the available cache size and reference behavior of the client. For example, if a client operates in disconnected mode due to a lack of network coverage, its processing may be interrupted because a data request cannot be satisfied by the local cache. In such a situation the client needs to wait till reconnection for transaction processing to continue. Since disconnections might exceed 5 minutes, all the reference information will be lost during such a period. On the other hand, if the client cache size is small, the reference information must be discarded even sooner than 5 minutes after the last reference. To resolve the problem of determining a reasonable guideline for maintaining CRF values, we conducted a series of experiments. We figured out that clients with a cache size in the range of 1 to 10% of the database size should maintain reference information on all recently referenced objects that would fit into a cache if it were about *3 to 5*

times as large as its actual size (see Figure 3). Clearly, due to its time-independence such a rule avoids the aforementioned problem of discarding reference information during periods when clients are idle. Second, it limits the amount of memory required for storing historical information by coupling the retained information period to the client cache size.

3.4 Implementation and Performance Issues

Due to space restrictions, only some selected topics of implementing MICP are discussed. For more relevant implementation issues such as storage organization, garbage collection, etc., we refer the interested reader to the relevant literature [BC92, SS02]. The previous section has shown that MICP bases its replacement and prefetching decisions on a number of factors combined into the PCB ratio. However, this metric is dynamic since it changes at every tick of the broadcast. Although in theory one could obtain the required values while a page is being transmitted, such an approach would be much too expensive. To reduce overhead, we propose that the estimate of PCB for each cached data item is updated either only when a replacement victim is selected or at fixed times such as the beginning or the end of a minor broadcast cycle. While experimenting with our simulator, we noticed that both approaches are capable of remarkably reducing processing overhead while providing good performance results. However, we favor the latter technique since it may allow MICP to compute PCB values less frequently. In what follows, we refer to the version of MICP that calculates PCB values periodically as *MICP-L* where L stands for "light".

4. Performance Evaluation

We studied and compared the performance of MICP with other online and offline caching and prefetching algorithms via simulation and not analytically because the effects of such parameters as transaction length, client cache size or number of versions maintained by the server depend on a number of internal and external system parameters that cannot be precisely estimated by mathematical analysis. The simulator and the workloads are based on the model designed for evaluating the performance of various isolation levels for read-only transactions [SS01], having been extended by MICP as well as some other popular caching and prefetching algorithms. In the following description of the simulator, some details are omitted due to space constraints and can be found in the full version of the paper [SS02].

4.1 System Model

The simulation model consists of the following core components: a) *server*, b) *client*, c) *broadcast disk*, and d) *network*, which are briefly described below.

Both multiple mobile clients and a single broadcast *server* are modeled as consisting of a number of subcomponents including a processor, volatile (cache) and, in case of the server, stable memory (disks), i.e., we assume diskless mobile clients. Data is stored on 4 disks modeled as a FIFO queue. The unit of data transfer between the server and disks is a page of 4 Kbytes and the server keeps a total of 250 pages in its stable memory. The size of an object is 100 bytes and the database consists of a set of 10000 objects. To reflect the characteristics of a modern disk drive we experimented with the parameters from the Quantum Atlas 10K III disk. The client CPU speed is set to 100 MIPS and the server CPU speed is 1200 MIPS, which are typical processor speeds of today's mobile and stationary computers. A single FIFO input queue is used for processing events such as disk I/O or sending a message. All requests are charged in terms of (fractions of) broadcast units. The client cache size is set to 2% of the database size and the server cache size to 20% of the database size. As described in Section 2, we model the client cache as a hybrid cache. The page-based segment is managed by an LRU replacement policy and the object-based segment by various online and offline cache replacement strategies including MICP-L. Similarly, the server cache is partitioned into a page cache and a modified object cache (MOB). The page cache is managed using an LRU policy and the MOB is managed in a FIFO order. The MOB is initially modeled as a single version cache. This restriction is later removed to study the effects of maintaining multiple versions of objects in the server cache. Client cache synchronization and freshness are accomplished by inspecting the CCR at the beginning of each minor broadcast cycle and by downloading newly created object versions whose PCB values are larger than those of currently cached object versions.

The *broadcast program* has a flat structure. To account for the high degree of skewness in data access patterns [HSY99a, HSY99b] and to exploit the advantages of hybrid data delivery only the latest versions of the most popular 20% of the database objects are broadcast. Note that we assume that clients regularly register at the server to provide their access profiles so that the server can generate the clients' global access pattern. Every MBC is subdivided into 5 minor cycles each consisting of a data segment with 10 pages, a (1, m) index [IVB97], and a concurrency control report.

Our modeled *network infrastructure* consists of three communication paths: a) broadcast channel, b) uplink channel from the client to the server, and c) downlink channel from the server to the client. The network parameters of those communication paths are modeled after a real system such as Hughes Network System's DirecPC [Hug01]. We set the default broadcast bandwidth to 12 Mbps and the point-to-point bandwidth to 400 Kbps downstream and to 19.2 Kbps upstream. The point-to-point network is modeled as a FIFO queue and each point-to-point channel is dedicated to 5 mobile clients. Charged network costs consist of CPU costs for message

processing at the client and server, queuing delay, and transfer time. Processor costs include a fixed and a variable cost component while the latter depends on the message size. With respect to message latency we experimented with a fixed RTT end-to-end latency of 300 ms. To exclude problems arising when clients operate in disconnected mode (e.g. consistency checking), we assume that clients are always tuned to the broadcast stream and do not suffer from intermittent connectivity.

4.2 Workload Model

To produce data contention in our simulator, we periodically modify data objects at the server by a workload generator that simulates the effects of read-write transactions being executed at the server. In our selected system configuration 100 objects are modified by 20 transactions during a MBC. Objects read and written by read-write transactions are modeled using a Zipf distribution with parameter θ=0.80. The ratio of the number of write operations versus the number of read operations is fixed at 0.2, i.e., only every fifth operation issued by the server results in an object modification. Read-only transactions are modeled as a sequence of 10 to 50 read operations and are managed by the MVCC-SFBS protocol [SS01]. The access probabilities of client read operations follow a Zipf distribution with parameter θ=0.95 and θ=0.80. While the θ=0.95 setting is intended to stress the system by directing about 90% of all object accesses to 10% of the database, the θ=0.80 setting models a more realistic medium-contention workload (about 75/25). To account for the impact on shared resources (network, server) when clients send fetch requests to the server, we model our hybrid data delivery network in a multi-user environment that services 10 mobile clients. As noted before, clients run at most one read-only transaction at a time and in order to account for a transaction think time between consecutive operations and transactions, we add a delay between those events. Further note that clients are not allowed to request object versions from the server if they are scheduled for broadcasting. To control the data access behavior of read-only transactions that were aborted, we use an abort variance of 100 percent which means that the restarted transaction reads from a different set of objects.

4.3 Other Replacement Policies Studied

In order to prove MICP's superiority, we need to compare it with state-of-the-art online cache replacement and prefetching policies. We experimented with LRFU since it is known to be the best performing page-based cache replacement policy [LKN+99]. However, since LRFU does not use any form of prefetching, comparing MICP to LRFU directly would be unfair. Therefore, we decided to incorporate prefetching into the LRFU and denote the resulting algorithm as *LRFU-P*. In order to treat LRFU-P as fair as possible with respect to data prefetching, we adopt the prefetching heuristic from MICP. That is, we select all newly created object versions along with the versions of those objects that have been referenced within the last 1000 data accesses as prefetch candidates. Out of those candidates, LRFU-P prefetches those versions whose CRF values are larger than the smallest CRF value of all cached object versions. The rest of the algorithm works as described in [LKN+99].

In addition to comparing MICP to LRFU-P, we experimented with the W^2R algorithm [JN98]. We selected the W^2R scheme for comparison mainly because it is an integrated caching and prefetching algorithm similarly to MICP and performance results have shown [JN98] that W^2R outperforms caching and/or prefetching policies such as LRU, 2Q [JS94], and LRU-OBL [Smi85]. However, since W^2R was designed for conventional page-based database systems, it has to be adapted to the characteristics of a mobile hybrid data delivery environment in order to be competitive with MICP. Our goal was to re-design W^2R in such a way that its original design targets and structure are still maintained. In the following we refer to the amended version of W^2R by *W^2R-B* where B stands for broadcast. Like W^2R, W^2R-B partitions the client cache into two rooms called the Weighing Room and the Waiting Room. While the Weighing Room is managed as an LRU queue, the Waiting Room is managed as a FIFO queue. In contrast to W^2R, W^2R-B admits both referenced and prefetched object versions into the Weighing Room. However, W^2R-B grants admission to the Weighing Room only to newly created object versions, i.e., those listed in the CCR, and whose underlying objects have been referenced within the last 1000 data accesses. The other modified objects contained in CCR and all the prefetch candidates from the broadcast channel are kept in the Waiting Room. As before, an object version $X_{i,j}$ becomes a prefetch candidate if some version of object X has been recently referenced. With regard to the room size we experimentally found out that the following settings work well for W^2R-B: the Weighing Room should be 80% of the total cache size and the remaining 20% should be dedicated to the Waiting Room.

Last but not least, we experimented with an offline cache replacement algorithm, called P [AAF+95], to present the theoretical bounds on the performance of MICP. We have chosen P as an offline policy due to its straightforward implementation as P determines its replacement victims by selecting the object with the lowest access probability. Since the client access pattern follows a Zipf distribution, the access probability of each object is known at any point in time. Like LRFU, P is a "pure" caching algorithm. Therefore, we had to extend P by incorporating prefetching. To ensure that clients cache all useful versions of objects with the highest likelihood of access, we added an aggressive prefetching strategy to P and called the new algorithm *P-P*. P-P's relatively simple prefetching strategy is as follows: a newly created

or disseminated object version $X_{i,j}$ is prefetched from the broadcast channel, if $X_{i,j}$'s access probability is higher than the lowest probability of all cached object versions. It should be intuitively clear, that such a policy is suboptimal since it neither considers the update and caching behavior of the server nor the serial nature of the broadcast channel.

4.4 Basic Experimental Results

In this section we compare the performance of MICP-L to that of the above introduced online and offline cache replacement and prefetching policies under the baseline setting of our simulator. We later vary those parameters in order to observe the changes in the relative performance differences between the policies under different system settings and workload conditions. We point out that all subsequently presented performance results lie within a 90% confidence level with a relative error of ±5%. We now study the impact of the read-only transaction length on the performance metrics when MICP and other policies are used for client cache management. Figure 1 shows experimental results of increasing the read-only transaction length from 10 to 50 observed objects. An exponential decrease in the throughput rate is observed when transaction length is increased. More importantly, Figure 1(a) shows that the MICP-L outperforms LRFU-P, on average, by *18.9%* and W^2R-B by *80.4%*. Further, but not shown in the graph, the throughput degradation caused by computing PCB values periodically (20 times per MBC) rather than every time when replacement victims are selected is insignificant since MICP outperforms MICP-L by only *2.8%* on average. When comparing the relative performance differences between MICP-L and other online policies under the 0.95 workload to those under the 0.80 workload, it is interesting to note that the performance advantage of MICP-L declines when the client access pattern becomes less skewed. The reason is related to the degradation in the client cache effectiveness experienced when client accesses are more uniform in nature and due to a weakening in the predictability of the future reference patterns by inspecting the past reference string. In this situation the impact of the client caching policy on the overall system performance is smaller, and therefore the throughput gap between the investigated online policies narrows. With regard to the client cache hit rate we also estimate MICP-L's superiority compared to other online policies. For example, MICP-L's cache hit rate is, on average, *5.8%* higher than that of LRFU-P. At the first glance, a relatively big performance gap might be surprising since both policies select replacement victims (at least partially) on CRF value basis. Thus, one would expect client cache hit rates of both policies to be fairly close to each other. But since MICP-L tries to minimize broadcast retrieval latencies by replacing popular object versions that soon re-appear on the broadcast channel with other less popular versions, which, if not cached, incur high re-acquisition costs when requested, MICP's hit rate is expected to be slightly lower than LRFU-P. However, as both MICP-L and LRFU-P incorporate prefetching, the performance gain from preloading objects into the cache is higher for MICP-L since it keeps more object versions in the client cache that are of potential utility for the active read-only transaction, while LRFU-P, on the contrary, maintains more up-to-date versions potentially useful for future transactions.

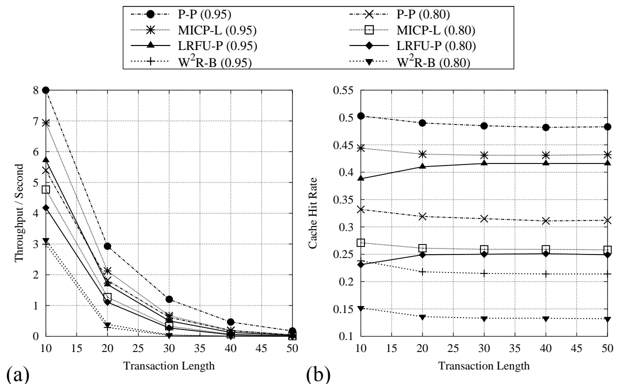

Figure 1: Performance of MICP-L compared to LRFU-P, P-P, and W^2R-P under various read-only transaction sizes

4.5 Additional Experiments

This section discusses the results of some other experiments conducted to determine how MICP-L and its counterparts perform under varying number of historical reference information and number of versions maintained by the server. As before, we report the results for both the 0.95 and the 0.80 workload.

Effects of the Version Management Policy of the Server on the Performance of MICP-L

To study the effect of keeping multiple versions per modified object at the server, we experimented with varying the version storage strategy of the MOB. As noted before, in order to save installation reads the server maintains modified objects in the MOB. In the baseline setting of the simulator, the MOB was organized as a mono-version object cache, i.e., only the most up-to-date versions of recently modified objects are maintained. We now remove that restriction by allowing the server to maintain up to 10 versions of each object. However, as the MOB is organized as a FIFO queue and limited to 20% of the database size, such a high number of versions will only be maintained for a small portion of the frequently updated database objects. As intuitively expected, the system performance increases with growing number of non-current object versions maintained at the server. However, it is interesting to note, that the gain in throughput performance levels off when the server maintains *more than 4* non-current versions of a recently modified object. Beyond this point, no significant

performance improvement can be achieved by further increasing the version retain boundary. As shown in Figure 2, the performance gap of MICP-L relative to LRFU-P and W^2R-B narrows when the maximum number of versions maintained by the server increases. For example, for the 0.95 workload the throughput performance degrades between MICP-L and LRFU-P *from 21.5% to only 2.5%* as the maximum number of versions maintained by the server increases. The reason is that under a multi-version storage strategy potentially more non-current object version requests from long-running read-only transactions can be satisfied by the server. As a result, fewer read-only transactions have to be aborted because the versions they request had been garbage-collected.

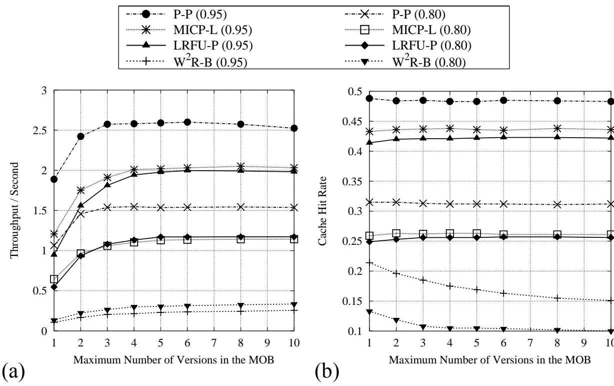

Figure 2: Performance of MICP-L compared to its competitors with varying number of non-current versions maintained by the server

Further, it is worth noticing, that the LRFU-P performs slightly better than the MICP-L if the following two conditions are satisfied: First, the server does not start overwriting obsolete object versions until at least 2 versions of each particular object are stored in the MOB. Second, the client access pattern is not very skewed in nature (80/>20). The reason why LRFU-P outperforms MICP-L under such a system setting is the inaccuracy of the PCB values on which MICP-L bases its caching and prefetching decisions.

Effects of the History Size on the Performance of MICP-L

MICP-L requires historical information on the past reference behavior of the client in order to make precise predictions about its future access pattern. In order to collect this information, objects' reference history needs to be maintained in the client memory even after their eviction from the cache. Since keeping superfluous history information in form of CRF values wastes scarce client cache, we wanted to determine a rule of thumb for estimating the amount of reference information clients need to maintain in order to achieve good throughput performance. To this end, we use the history size/cache size ratio (HCR) defined as:

$$\mathrm{HCR} = \frac{\mathrm{HS}}{\mathrm{CS}} \tag{7}$$

where HS denotes the total number of CRF values maintained at the client and CS is the client cache size available for storing data items. As shown in Figure 3, we measured MICP-L's performance for various HCR and cache size values. The results show, that MICP-L reaches its performance peak if clients maintain CRF values of all those recently referenced objects that would fit into the client cache if it were *3 to 5 times* larger than its actual size. Beyond that point, i.e., when HCR is larger than 5, MICP-L's throughput slightly degrades. The reason for this degradation is related to an increase in the number of prefetches caused by MICP-L's prefetching heuristic that allows clients to download useful object versions into their local memory if their corresponding object has been referenced within the retained information period. As a result of those additional prefetches, object versions useful for the active read-only transaction may be replaced by up-to-date object versions which are of potential use for future transactions. This slightly hurts the cache hit rate and hence the throughput performance.

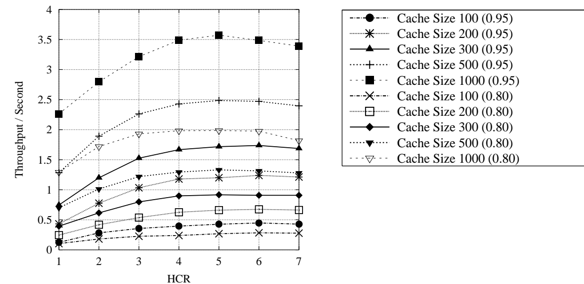

Figure 3: Performance of MICP-L under various cache sizes when HCR is varied

5. Conclusion

We have presented the design and implementation of a new integrated cache replacement and prefetching algorithm called MICP. MICP has been evolved to efficiently support the data requirements of *read-only transactions* in a mobile hybrid data delivery environment. In contrast to many other cache replacement and prefetching policies, MICP not only relies on *future reference probabilities* when selecting replacement victims and for prefetching data objects, but additionally uses information about the *contents and* the *structure of the broadcast channel*, the *data update frequency*, and the *server storage management*. MICP combines those statistical factors into a single metric, called PCB, in order to provide a common basis for decision making and to achieve the goal of maximizing the transactional throughput. Further, in order to reduce the number of transaction aborts caused by the eviction of useful, but obsolete, object versions from the server, MICP divides the client cache into *two variable-sized cache partitions* and maintains non-re-cacheable object versions in a

dedicated part of the cache called NON-REC. We evaluated the performance of MICP experimentally using simulation configurations and workloads observed in a real system and compared it with the performance of other state-of-the-art online and offline cache replacement and prefetching algorithms. The obtained results show that the implementable approximation of MICP, called MICP-L, improves the throughput rate, on average, by *18.9%* when compared to LRFU-P, which is the second best performing online algorithm after MICP-L. Further, our experiments revealed that the performance degradation of MICP-L relative to MICP is modest *2.8%* (not graphically shown).

References

[AAF+95] S. Acharya, R. Alonso, M. Franklin, S. Zdonik. Broadcast Disks: Data Management for Asymmetric Communications Environments. SIGMOD Conference 1995, pages 199-210.

[AFZ96] S. Acharya, M. Franklin, S. Zdonik. Prefetching from a broadcast disk. ICDE 1996, pages 276-285.

[AFZ97] S. Acharya, M. Franklin, S. Zdonik. Balancing Push and Pull for data broadcast. SIGMOD Conference 1997, pages 183-194.

[BC92] P. M. Bober, M. J. Carey. On Mixing Queries and Transactions via Multiversion Locking. ICDE 1992.

[BI94] D. Barbará, T. Imielinski. Sleepers and Workaholics: Caching Strategies in Mobile Environments. SIGMOD Conference 1994, pages 1-12.

[CFK+95] P. Cao, E. W. Felten, A. Karlin, K. Li. A Study of Integrated Prefetching and Caching Strategies. ACM SIGMETRICS 1995, pages 188-197.

[CK89] E. Chang, R. Katz. Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS. SIGMOD Conference 1989, pages 348-357.

[Den72] P. J. Denning. On Modeling Program Behaviour. In Proceedings Spring Joint Computer Conference, pages 937-944, Arlington, VA., 1972.

[DMF+90] D. J. DeWitt, D. Maier, P. Futtersack, F. Velez. A study of three alternative workstation-server architectures for object-oriented database systems. VLDB 1990, pages 107-121.

[EH84] W. Effelsberg, T. Haerder. Principles of database buffer management. ACM TODS 9(4), pages 560-595, Dec. 1984.

[Ghe95] S. Ghemawat. The Modified Object Buffer: A Storage Management Technique for Object-Oriented Databases. Tech. Report MIT/LCS/TR-666, Sep. 1995.

[GG97] J. Gray, G. Graefe. The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb. SIGMOD Record 26(4), pages 63-68, 1997.

[HSY99a] W. W. Hsu, A. J. Smith, H. C. Young. Analysis of the Characteristics of Production Database Workloads and Comparison with the TPC Benchmarks, Tech. Report, CSD-99-1070, UC Berkeley, 1999.

[HSY99b] W. W. Hsu, A. J. Smith, H. C. Young. I/O Reference Behavior of Production Database Workloads and the TPC Benchmarks – An Analysis at the Logical Level, Tech. Report, CSD-99-1071, UC Berkeley, 1999.

[Hug01] Hughes Network Systems. DirecPC Home Page. http://www.direcpc.com, Jan, 2001.

[IVB97] T. Imielinski, S. Viswanathan, B. R. Badrinanth. Data on Air: Organization and Access. IEEE TKDE, 9(3): pages 353-372, May/June 1997.

[JN98] H. S. Jeon, S. H. Noh. A Database Disk Buffer Management Algorithm Based on Prefetching. Proceedings of ACM CIKM, Bethesda, Maryland, USA, pages 167-174, Nov. 3-7, 1998.

[JS94] T. Johnson, D. Shasha. 2Q. A low overhead high performance buffer management replacement algorithm. VLDB 1994, pages 439-450.

[KL98] S. Khanna, V. Liberatore. On broadcast disk paging. ACM STOCS 1998, pages 634-643.

[LKN+99] D. Lee, J.-H. Kim, S. H. Noh, S. L. Min, J. Choi, Y. Cho, C. S. Kim. On the Existence of a Spectrum of Policies that subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies. ACM SIGMETRICS 1999, pages 134-143.

[OOW93] E. J. O'Neil, P. E. O'Neil, G. Weikum. The LRU-K page replacement algorithm for database disk buffering. SIGMOD Conference 1993, pages 297-306.

[OS94] J. O'Toole, L. Shrira. Opportunistic Log: Efficient Installation Reads in a Reliable Object Server. OSDI, Nov. 1994.

[PZ91] M. Palmer, S.B. Zdonik. Fido: A Cache That Learns to Fetch. VLDB 1991, pages 255-264.

[Smi85] A. J. Smith. Disk Cache-Miss Ratio Analysis Design Considerations. ACM TOCS, 3(2), pages 161-203, August 1985.

[SRB97] K. Stathatos, N. Roussopoulos, J. S. Baras: Adaptive Data Broadcast in Hybrid Networks. VLDB 1997, pages 326-335.

[SS01] A. Seifert, M. H. Scholl. Processing Read-only Transactions in Hybrid Data Delivery Environments with Consistency and Currency Guarantees, Tech. Report No. 163, University of Konstanz, Dec. 2001.

[SS02] A. Seifert, M. H. Scholl. A Transaction-Conscious Multi-version Cache Replacement and Prefetching Policy for Hybrid Data Delivery Environments, Tech. Report No. 165, University of Konstanz, Feb. 2002.

[TPG97] A. Tomkins, R.H. Patterson, G. Gibson. Informed multiprocess prefetching and caching. ACM SIGMETRICS 1997.

[TS97] L. Tassiulas, C. J. Su. Optimal Memory Management Strategies For a Mobile User in a Broadcast Data Delivery System. IEEE J-SAC, 15(7), pages 1226-1238, Sep. 1997.

[XHL+00] J. Xu, Q. Hu, D. L. Lee, W.-C. Lee. SAIU: An Efficient Cache Replacement Policy for Wireless On-demand Broadcasts. ACM CIKM 2000, pages 46-53.