TECHNICAL RESEARCH REPORT

Disseminating Updates to Mobile Clients

by Konstantinos Stathatos, Nick Roussopoulos,
John S. Baras

CSHCN T.R. 98-16
(ISR T.R. 98-62)

The Center for Satellite and Hybrid Communication Networks is a NASA-sponsored Commercial Space Center also supported by the Department of Defense (DOD), industry, the State of Maryland, the University of Maryland and the Institute for Systems Research. This document is a technical report in the CSHCN series originating at the University of Maryland.

Web site http://www.isr.umd.edu/CSHCN/

Disseminating Updates to Mobile Clients

Konstantinos Stathatos Nick Roussopoulos John S. Baras
Center for Satellite and Hybrid Communication Networks
University of Maryland
College Park, MD 20742 *

Abstract

In this paper, we address the problem of propagating data updates to a large number of mobile clients. Typically, mobile clients operate autonomously, i.e., disconnected from data servers, for prolonged periods of time relying on locally replicated corporate data (e.g., database views, file systems). From time to time, they need to refresh their replicas with data changes registered at a central data repository. We propose a hybrid approach for delivering these updates to the clients. We use a broadcast channel to "cache on the air" fresh updates for as long as they are high on demand. At the same time, any requests for older updates are individually serviced by the server through a separate channel. The air-caching satisfies the bulk of clients' needs, increasing data throughput many-fold compared to traditional data delivery mechanisms. We describe a hierarchical air-cache structure, and analyze the performance of broadcasting a log of committed updates. Based on that, we propose a technique that dynamically modifies the contents and the structure of the air-cache according to the number and the (dis)connection habits of the clients. Through extensive simulations, we demonstrate the adaptiveness, efficiency, and practicality of the proposed system even for very large client populations.

*This work was supported by the Center for Satellite and Hybrid Communication Networks, under NASA cooperative agreement NCC3-528

1 Introduction

The remarkable sales increase of portable computers and the proliferation of wireless communication technologies are strong evidence that mobile computing is becoming ever more important. Today, rapid technological advances offer laptop computers comparable to desktop workstations with significant processing power and large disk capacity. Such machines enable a wide range of applications to be carried away from the office desk. As a result, many organizations today—and many more in the future—have a portion of their workforce accessing corporate information on the road, from their home, or from other remote locations. Most of the time they operate off-line, i.e., not connected to the corporate network, relying on local data replicated from a central repository. This replication masks the effects of disconnection but it also brings about the problem of staleness and the need to refresh data regularly and efficiently. In other words, any data updates occurring at the repository must sooner or later be propagated to the mobile clients.

This model of operation is important for any corporation in which business transactions may occur outside the office. For instance, sales agents visiting potential customers need information about new products or services, new pricing policies, special offers, product availability, etc. Money managers need the latest stock and bond indexes. Realtors accompanying potential buyers need new house listings, possibly together with photographs, directions, and other related information. The growing market for these mobile applications has already been recognized by the database industry. Several products are emerging that support off-line operation, offering data replication and update propagation between a central database and "lite" DBMSs running on mobile computers (e.g., Oracle Lite, Sybase SQL Anywhere Studio).

Update propagation techniques typically rely on the transaction log of the server which records changes committed in the database [RK86, GWD94, BDD+98]. Log entries are sent to the clients, where they are "replayed" to refresh the local copies. For such a refresh in the mobile environment, a client needs to reconnect to the network and receive all the updates that were committed while it was off-line. This requires reviewing the part of the log that was appended since its previous refresh, finding all relevant updates, and applying them to the local data.

In this paper, we address the problem of propagating logged updates to large, widely deployed mobile workforces. Our approach is based on the idea of *adaptive hybrid data delivery*, the dynamic combination of *broadcast* for massive data dissemination and *unicast* for upon-request data delivery [SRB96]. At the core of this technique lies the notion of *air-caching*, i.e., the temporary storage of popular data in a broadcast channel. The effect of caching on the air is created by repetitive broadcast of data. Clients that tune in the channel can download cached data from the air without having to contact the server. Only air-cache misses, i.e., requests for data not included in the broadcast, are directed to the server. As demonstrated in [SRB97], this technique can be the basis of highly scalable data dissemination systems.

Broadcasting is a very efficient way for disseminating data, especially as the number of potential recipients grows [Won88, HGLW87, IV94, AAFZ95]. What makes it even more appealing in this case is that updates exhibit very high locality of reference. All clients want the updates since they went off-line, making the recent part of the log an extremely hot spot [DR92]. Driven by this, we propose using the air-cache mechanism to disseminate the log to the clients. Each client individually takes over the task of filtering out the updates that affect its data. Client-side filtering of the updates has been shown to be preferable in large scale systems since it avoids contention at the server [DR98]. When combined with broadcasting, the benefits are multiplied since data transmission cost is amortized over many clients.

The key issue with the proposed idea is identifying what part of the log is popular enough to be air-cached. This cannot be a one time decision as the popularity of the updates is time-dependent and continuously changing. They start very hot, but as they age their popularity drops. Thus, we need an adaptive algorithm that manages the air-cache according to the clients' needs and the age of the data. The main complication we have to address is the unique property of the air-cache that it must be managed exclusively based on cache misses, since the server cannot have any information about cache hits [SRB96]. What makes the problem even more intricate in this case is that not all clients have the same habits, in terms of connecting to and disconnecting from network. Borrowing the terminology from [BI94], clients can range from *workaholics*, who stay connected most of the time, to *sleepers*, who only connect sporadically. As a result, upon reconnection, workaholics usually need a small part of the log, while sleepers tend to require much longer parts. In order to accommodate this diversity in the clients' needs, we employ a hierarchical air-cache which provides multiple levels of data caching, each with different performance characteristics. This gives us the flexibility to air-cache the log in a way that suits different client groups.

The rest of the paper is organized as follows. First, we review the concept of air-caching and describe the new hierarchical version of it. Next, in Section 3, we develop a performance model for broadcasting the log through the hierarchical air-cache. This model serves as the foundation for the proposed hybrid system presented in Section 4. Section 5 contains several experimental results obtained from a detailed simulation. Finally, we briefly present some related work in Section 6 and conclude in Section 7.

2 Hierarchical Air-Cache

Repetitive data broadcast has attracted a lot of attention as an efficient way of disseminating data to very large client populations (e.g., [Won88, HGLW87, IV94, AAFZ95]). The main motivation is its scalability potential; the number of clients concurrently accessing data can grow arbitrary large with no effect on the systems' performance. There is, however, an important obstacle to reaping this scalability potential: the lack of data usage feedback. Clients do not acknowledge the receipt of information, and the server cannot directly assess

the goodness of the data being broadcast. Often, that forces system designers to take decisions about what to broadcast based solely on a priori information (e.g., precompiled user profiles). While reasonable for some applications, this approach is not suitable for situations where the popularity of the data changes all the time.

As a remedy to this problem, we have proposed *adaptive hybrid data delivery* which dynamically combines broadcasting with individual upon-request data delivery [SRB96]. Under this approach, the server repetitively broadcasts only the data that it deems popular, expecting to satisfy most of the client requests. At the same time, it permits clients to send requests for data not being broadcast. Replies to these requests are unicast to the requesting client through a separate channel. In some sense, the server is "caching on the air" popular data where most clients can get it from, and services only requests for data not cached. In the same sense, accesses from the broadcast are *air-cache hits*, and requests directed to the server are *air-cache misses*. One of the main advantages of this technique is that, while the server does not get information about the cache hits, it does get valuable information from the misses. As we demonstrated in the past, these cache misses provide adequate feedback to compose a good picture of the actual workload [SRB97].

The air-cache exhibits the properties of repetitive broadcast schemes which distinguish it from other traditional caches. For example, its size is not fixed; it is defined by the volume of data being broadcast, and can vary according to the clients' demands. An important performance property is that it can be accessed only sequentially in the sense that clients need to wait for the data of interest to appear on the broadcast channel. A direct consequence is that data access latency depends on how often data get broadcast. Naturally, this is determined by the volume of the data being broadcast, i.e., the size of the air-cache. So, the larger the air-cache the bigger the expected data access latency.

A *hierarchical air-cache* is a flexible cache structure that supports different broadcast frequencies, and therefore, different access latencies. It adopts the "multi-disk" model of the broadcast disks architecture [AAFZ95] to create a memory hierarchy on the air. Generally, a hierarchical air-cache consists of C cache levels, named $AC_1, AC_2, \ldots, AC_C$. The average latency of each level AC_k is determined by the frequency f_k at which data cached in AC_k are repeated in the broadcast. More popular data are cached in the faster (lower latency) levels and are broadcast more often; less popular data are cached in the slower (higher latency) levels. Note that the repetition frequencies have only a relative meaning. This means that the value of f_k only suggests that an item cached in AC_k is being broadcast f_k/f_j times more (or less) often than an item in AC_j. Put another way, between two consecutive broadcasts of an item in AC_j there are on average f_k/f_j broadcasts of an item in AC_k. As a convention, we select AC_C to be the fastest cache and AC_1 to be the slowest, i.e., $f_C > f_{C-1} > \ldots > f_1$. These frequencies are not restricted to integer values, and since they are important only in a relative sense, we can always set them so that $f_1 = 1$.

For our purposes, we assume that data are organized into equal size pages. Also, we follow the established convention for broadcast based systems of measuring time in terms of *broadcast units*, i.e., the time required to broadcast a page. Figure 1 presents an example

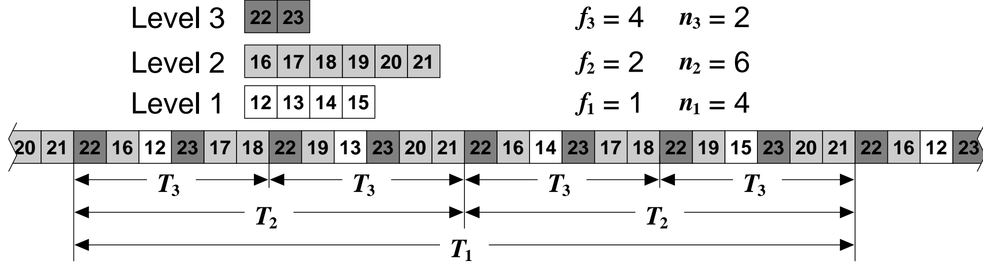


Figure 1: Hierarchical air-caching

of a 3-level hierarchical air-cache and a portion of the broadcast stream it generates. Each level AC_k is characterized by its frequency f_k and the number of data pages n_k it contains. In this example, AC_1, AC_2, and AC_3 contain 4, 6, 2 pages respectively. Furthermore, their frequencies are 1, 2, and 4. This means that pages in AC_3 get broadcast twice as often as pages in AC_2, which in turn get broadcast twice as often as those in AC_1.

The actual latency of AC_k is determined not only by its own size and frequency but also by the sizes and frequencies of all other levels. We define the period T of the air-cache to be the minimum time interval during which all cached pages are broadcast at least once. Because AC_1 is the slowest level, this period is equal to the interval between two consecutive broadcasts of any one page in AC_1. On average, within each period there must be $f_1 = 1$ broadcast of each page in AC_1, f_2 broadcasts of each page in AC_2, and so forth. Thus, the period of the air-cache lasts $T = \sum_{k=1}^{C} n_k f_k$ page broadcasts, or broadcast units. Similarly, we can define the period T_k of level AC_k as the average interval between two consecutive broadcasts of any one page in it. Since within every T there must be f_k broadcasts of the pages in AC_k, we can infer that $T_k = T/f_k$. This also means that $T_C < T_{C-1} < \ldots < T_1 = T$.

An interesting issue is how to actually realize the air-cache in the broadcast channel. In other words, we need an algorithm that, given the structure and the contents of the air-cache, can schedule the broadcast so that the correct caching effect is achieved. Our implementation is using a set of queues, and an adapted version of a *worst-case fair weighted fair queueing* algorithm [BZ96, HV97]. This approach is very efficient, and, more important, it can easily accommodate on-line changes in the structure of the air-cache. For brevity, we skip the details of the algorithm, and refer the interested reader to [Sta98].

3 Hierarchical Log Air-Caching

In this section, we present an analytical performance model for a system that uses a hierarchical air-cache to disseminate logged updates to mobile clients. Here, we are restricting the model to a broadcast-only case in order to show how the structure of the air-cache affects

the refresh time of the clients. Our ultimate goal is to define an optimization problem that relates the structure of the air-cache to a given log access pattern. That will be the base for the adaptive technique, presented later in the paper.

3.1 Definitions

Let us consider a set of mobile clients that operate on data replicated from some data server. This server is the central site that records all updates and enforces data consistency. Committed updates are recorded in a log. This log consists of equal size pages $\ell_1, \ell_2, \ldots$, where ℓ_1 is the oldest page, ℓ_2 the second oldest, and so forth. The subscript corresponds to the page's *log sequence number* (LSN).

Suppose that at some point the server keeps in a hierarchical air-cache log pages ℓ_c, ℓ_{c+1}, ..., ℓ_z, with ℓ_c being the oldest page in the cache, and ℓ_z is currently the most recent page. Because the popularity of log pages decreases with their age, more recent pages are cached in higher levels, and older pages in lower levels. We put the n_C most recent pages in the highest air-cache level AC_C, the next n_{C-1} pages in AC_{C-1}, and so forth. Within each level, log pages are broadcast in decreasing order of age (i.e., older pages first). Figure 1 is an example of log air-caching, where pages ℓ_{12} through ℓ_{23} are cached in three levels.

A mobile client is said to be in *sleep mode* (off-line) when it is neither listening to the broadcast channel nor connected to any network. At times, it "wakes up" and comes on-line, i.e., starts monitoring to the broadcast stream and possibly connects to the network, in order to refresh its data. This requires that it retrieves all the updates that occurred while it was sleeping. In other words, it has to download all log pages created after its previous refresh.

Definition 1 *A client requires an* (m)-refresh *if, in order to get up to date, it needs to download all the recent log pages starting with* ℓ_m. *If* z *is the LSN of the currently most recent log page, an* (m)*-refresh requires retrieving pages* $\ell_m, \ell_{m+1}, \ldots, \ell_z$.

For the sake of the analysis, let us assume that all the pages required for an (m)-refresh can be found in the air-cache. If page ℓ_m is cached in level AC_k, the client needs to download some pages from level AC_k (at least ℓ_m) plus all the pages cached in the higher levels $AC_{k+1}, \ldots, AC_C$. None of the pages cached in the lower levels $AC_1, \ldots, AC_{k-1}$ is of interest since they are older than ℓ_m.

Definition 2 *For a given air-cache structure, a client refresh results in a* (k, u)-hit *if it requires downloading the* u *most recent log pages cached in* AC_k*, as well as all the pages cached in the higher levels* $\mathrm{AC}_{k+1}, \ldots, \mathrm{AC}_C$. *The* u *pages of* AC_k *required by a* (k, u)*-hit are called the* useful segment *of the hit.*

Going back to the example in Figure 1, a (18)-refresh results in a $(2,4)$-hit as it requires 4 pages from AC_2, and all the pages from AC_1. Pages ℓ_{18}, ℓ_{19}, ℓ_{20}, and ℓ_{21} are the hit's useful segment.

From the description of the air-cache, we know that all the pages cached in AC_k are broadcast exactly once every T_k units, in decreasing order of age. According to its definition, the useful segment of a (k,u)-hit consists of the u most recent pages in AC_k. This means that the client discerns two separate parts within any interval of T_k units: the part during which the pages of the useful segment are broadcast, and the part during which the rest pages from AC_k are broadcast (both interleaved with pages from other levels). Because the client needs u contiguous pages out of the n_k pages cached in AC_k, we can infer that the first part lasts $T_u = \frac{u}{n_k} T_k$ units. The second obviously lasts $T_k - T_u$ units. As we will see in the next section, this observation is critical for the performance of a (k,u)-hit.

3.2 Performance Model

Here we compute the time required for an (m)-refresh to be satisfied by the air-cache. Let *refresh time* R_m be the time elapsed from the moment the client wakes up and starts monitoring the air-cache until it retrieves all the pages it needs. Let x be the number of pages required for the (m)-refresh. In terms of broadcast units, R_m is the total number of pages the client scans from the broadcast until every one of the x pages it needs is broadcast at least once. Usually R_m is greater than x for two reasons: First, older pages that the client does not need are broadcast, and second some of the pages it does need may be broadcast more often. From the client's perspective, the optimal performance, i.e., minimum refresh time, is $R_m = x$.

In the following, given the structure of the air-cache, we compute the expected refresh time $\mathrm{E}[R_m]$ for an (m)-refresh. This refresh will be satisfied by means of a (k,u)-hit for the proper values of k and u. Therefore, in order to compute the desired result, we generally examine the performance of a (k,u)-hit.

The first thing to note is that we can identify lower and upper bounds for the refresh time. On one hand, since the client needs all the pages from AC_{k+1}, it will take at least T_{k+1}, which is the minimum time to download all the pages of that level. At the other extreme, it cannot take more than T_k since in this time all the pages it needs (and probably more) must be broadcast at least once. The actual time the client will take to download the x pages depends on the broadcast time of the useful segment relatively to the arrival time of the client, i.e., the moment it starts monitoring the broadcast. Let X be a random variable that represents the time the first broadcast of the useful segment ends after the client wakes up. With the help of X, we can compute the expected refresh time $\mathrm{E}[R_m]$. Since the useful segment is broadcast once every T_k units and a client can wake up at any moment, the variable X is uniformly distributed over the interval $[0, T_k)$. It turns out that we need to

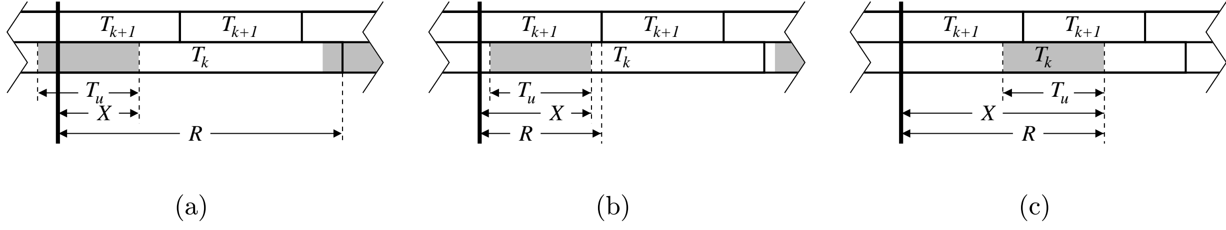


Figure 2: Effect of the useful segment on performance

consider three cases. These are depicted in Figure 2 where the thick vertical line corresponds to the moment the client wakes up, and the grey box represents the useful segment.

Case a: $[0 \leq X \leq T_u]$ The client starts monitoring within a broadcast of the useful segment (Figure 2(a)). This means that it just missed a portion of the useful segment and it has to wait for its next broadcast. The probability of that happening is $p_1 = \Pr[0 \leq X \leq T_u] = \frac{T_u}{T_k}$. Because the client has to wait until the next broadcast of the useful segment, the refresh time will be equal to a full period of AC_k, i.e., $R_m = T_k$. □

Case b: $[X > T_u$ **and** $X \leq T_{k+1}]$ The client starts listening outside the useful segment which however completes in time less than T_{k+1} (Figure 2(b)). Note, this case is possible only if $T_u < T_{k+1}$. Also, this case is not possible either if $k = C$, simply because there is no level AC_{C+1} (by convention $T_{C+1} = 0$ and $f_{C+1} = \infty$). With this in mind, the probability of the second case occurring is $p_2 = \Pr[T_u < X \leq T_{k+1}] = \max\left(0, \frac{T_{k+1} - T_u}{T_k}\right)$. Here, the pages cached in AC_{k+1} delay more than the useful segment of AC_k, and therefore the refresh time is equal to the period of level $k + 1$, i.e., $R_m = T_{k+1}$. □

Case c: $[X > T_u$ **and** $T_{k+1} < X < T_k]$ For the last case, the client wakes up outside the useful segment, which completes after level $k + 1$ (Figure 2(c)). The probability of this third case occurring depends on the relative sizes of T_u and T_{k+1}. More specifically, $p_3 = \Pr[\max\{T_u, T_{k+1}\} < X < T_k] = \frac{T_k - \max\{T_u, T_{k+1}\}}{T_k}$. Now, the refresh time is determined by the end of the useful segment. Therefore $R_m = X$, where X is uniformly distributed over $(\max\{T_u, T_{k+1}\}, T_k)$. □

From the above model, we can compute the expected refresh time $\mathrm{E}[R_m]$. The formula is given in Appendix A along with some other related results.

3.3 Cache Optimization

In the previous section we computed the expected time for an (m)-refresh to be satisfied, given the structure of the air-cache. The air-cache, however, is created to serve a large number of clients with different needs in terms of number of log pages. In order to optimize it for the whole client population we need a performance metric that normalizes over the size of clients' demands. A natural choice for such a metric is the *refresh factor* $F_m = \frac{R_m}{x}$. This intuitive metric gives the number of pages a client examines for every page it actually needs. More important, it gives a better indication as to how good the air-cache is for any client irrespectively of its disconnection time and the volume of updates it needs. Obviously, in the best case $F_m = 1$, which means that a client examines only the pages it needs, no more than once each.

Using this metric we can now formulate the air-cache optimization problem. Our goal is to structure the air-cache in a way that minimizes the expected refresh factor over all clients. The inputs to the problem are the range of pages to be cached (ℓ_c to ℓ_z), the maximum number of cache levels to be created C, and the log access pattern. The last is expressed in terms of the a *probability vector* $\boldsymbol{P} = (p_c, \ldots, p_z)$, where p_m is the probability that a reconnecting client needs a (m)-refresh. Formally, we have to solve the following optimization problem:

$$\begin{array}{rc} \text{Given} & c, z, C, \boldsymbol{P} \\ \text{find} & f_2, \ldots, f_C, \text{ and } n_1, n_2, \ldots, n_C \\ \text{that minimize} & \sum_{i=c}^{z} p_i \, \mathrm{E}\left[F_i\right] \\ \text{under the constraints} & f_C > f_{C-1} > \ldots > f_1 = 1, \\ & n_1 \geq 1, \\ & n_k \geq 0 \text{ for } 2 \leq k \leq C, \\ \text{and} & \sum_{k=1}^{C} n_k = z - c + 1 \end{array} \tag{1}$$

Note that we do not allow AC_1 to be empty ($n_1 \geq 1$). The reason behind this constraint is that for any optimal solution that AC_1 is empty we can get an equivalent solution where it is not empty, by removing all lower levels that are empty, and properly adjusting the relative frequencies. Finally, in practice, we also need to decide what range of the log should be air-cached, i.e., determine the parameter c. However, this is an orthogonal problem that we address in the next section of the paper.

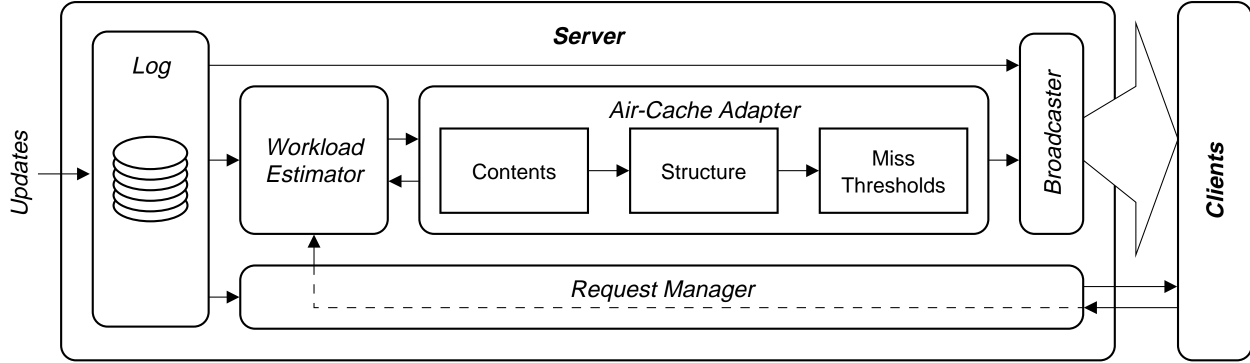


Figure 3: System overview

4 Hybrid Log Dissemination

In the previous section, developed a performance model for hierarchical air-caching of logged updates. Here, we propose a hybrid system that builds on this model to efficiently disseminate updates to large populations of mobile/disconnecting clients. The term "hybrid" reflects the fact that we use a broadcast channel to air-cache some recent part of the log, but also allow clients to directly connect to the server, and pull data in case of air-cache misses.

Basically, the proposed system has three major objectives: efficiency, scalability, and adaptiveness. In our context, efficiency translates to small refresh factors for reconnecting clients. When serving many clients with different needs, this calls for a solution to the optimization problem we just described. Scalability requires that the system performs equally well for a very large number of clients. As it was demonstrated in our previous work, such a hybrid system achieves scalability with a careful balance of broadcast and unicast data delivery. On one hand, the goal is to air-cache enough data to serve the bulk of clients needs, and let the server handle only a tolerable volume of cache misses. This prevents the server from becoming a performance bottleneck. On the other hand, we do not want to cache more than we have to, since that would unnecessarily increase the broadcast size. Last, adaptiveness requires that the system is efficient and scalable under any (possibly changing) workload. This requirement emphasizes the pivotal role of air-cache misses. As they are the only indication of the clients' activity, the server relies on them to assess the system's workload, and adapt accordingly.

The overall system architecture is shown in Figure 3. Clients can connect and disconnect at any time. A reconnecting client first tunes into the broadcast channel to determine how many log pages have been created since its last connection, and whether it should get them all from the air-cache or not. If yes, it does not contact the server; otherwise it sends a request for one or more log pages.

The server consists of five modules:

1. The *Log* records all the data updates, grouped in equal size pages. When new log pages are created, it notifies the other modules as necessary.
2. The *Broadcaster* is creates the air-cache by broadcasting log pages in the proper sequence.
3. The *Request Manager* handles client requests, and collects the necessary statistics on the misses.
4. The *Workload Estimator* uses the miss statistics to assess the clients' activity and log access pattern.
5. The *Air-cache Adapter* controls the contents and structure of the air-cache, based on the output of the workload estimator.

Next, we present the key components of this hybrid system. First, we introduce a new twist to the idea of air-cache misses, then we show how the estimator can estimate the workload from those misses, and finally, describe how the adapter dynamically modifies the air-cache.

4.1 Soft Cache Misses

Naturally, a client is expected to generate a cache miss when (some of) the updates it requires are not air-cached, i.e., when it needs more log pages than it can get from the air-cache. In this case, it will get all the pages it can from the broadcast, and request the remaining from the server.[1] Note that the *size* of the miss, i.e., the number of pages the client requests from the server, is variable.

We can, however, relax the notion of a cache miss and, sometimes, allow clients to generate misses even for air-cached data. The rationale behind this idea is that such misses may yield significant savings in terms of the refresh time. Consider, for example, the scenario in presented in Figure 4(a). This particular refresh translates into a (k, u)-hit with a small useful segment T_u (grey box) that will start being broadcast S units later. As we have seen, the refresh time R for the client is determined by the end of the useful segment. For the first T_{k+1} units after it wakes up, the client downloads pages from $AC_{k+1}, \ldots, AC_C$. After that, it has to wait for another D units until it can download the useful segment; no other page that it needs or it does not already have is broadcast in that period. Therefore, within the refresh time R there is a lengthy "dead interval" D which the client spends just waiting. It is not hard to see that if it did not have to wait for the delayed useful segment, the refresh time R' would be only T_{k+1}. But of course, this would be possible only if the client could get the useful segment in another way, i.e., directly from the server.

[1] Alternatively, for big requests, the client could scrap its local replica and rebuild it from scratch. Although not considered in this paper, such an option could be easily supported by the system.

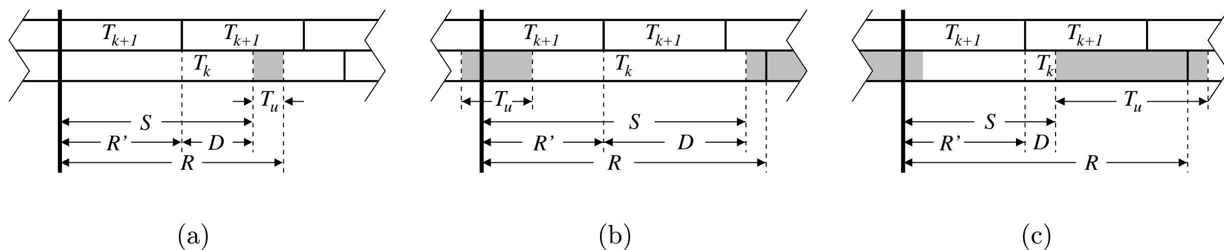


Figure 4: Examples of cache misses

In cases like the above, and from the client's perspective, it pays off to actually generate a miss even for pages that can be found in the air-cache. Since such a miss is only a performance enhancement and not a functional requirement we call it a *soft miss*. In this example, the soft miss turns the (k, u)-hit into a faster $(k+1, n_{k+1})$-hit. Another similar scenario where a soft miss can make significant difference is shown in Figure 4(b). This time, the client wakes up within a broadcast of the useful segment, which means that it will wait for $R = T_k$ to get the pages that were just missed. That includes a long dead interval D. If it could get those few pages by means of a soft miss, the refresh time would be cut down to $R' = T_{k+1}$.

But there is also a downside to soft misses; they add up to the server's load. However, exactly because they are "soft", it can be left to the server's discretion which of them (if any) to serve. For example, consider the example of Figure 4(c). Again, the client wakes up within a broadcast of the useful segment, but towards the end of it. A soft miss in this case would indeed reduce the refresh time substantially. However, its size would be quite big, and, depending on its load, the server might be reluctant to serve it.

For the server to be able to decide which soft misses it can accommodate and inform clients when to send one or not, we need to quantify the "importance" of a miss. The above examples suggest that we must give preference to small size misses with big potential to reduce the clients' refresh time. Based on that, we define the *merit* of a (k, u)-hit to create a soft miss to be $M = \dfrac{S - T_{k+1}}{T_u}$, where S is the time when the useful segment starts being broadcast after the client tunes in the channel. Intuitively, this metric favors situations with a small useful segment that starts being broadcast long after the client wakes up, causing lengthy dead intervals. If the useful segment starts before T_{k+1}, there is no dead interval and M is negative. Note, that this definition applies for the top level AC_C where $T_{C+1} = 0$.

The server establishes and broadcasts a *merit threshold* θ_k to instruct clients to send soft misses only if their merits exceed it. This way it can explicitly control the volume of such misses it receives. When it is fairly busy it should select a high threshold in order to limit the "performance misses" to a minimum. On the contrary, when not loaded, it can lower the threshold and offer more performance improvement chances to clients. In Section 4.3 we

For the Air-Cache
C : Number of air-cache levels
T : Period of air-cache
old : The oldest page in the air-cache

For every level AC_k
f_k : Frequency (if not fixed)
θ_k : merit threshold
new_k : most recent page
$next_k$: page to be broadcast next

Table 1: Index information for log air-caching

explain how exactly the server regulates that. However, the less obvious but more important advantage of this technique is that, based on the analysis of Section 3.2, we can compute both the probability $b(k, u)$ that a (k, u)-hit will cause a soft miss as well as the expected size $g(k, u)$ of that miss. For the interested reader, the formulas are again given in Appendix A. As we will see later, this provides the grounds for accurate estimation of the workload.

The last piece of the picture is a suitable indexing scheme for the broadcast channel. In other words, along with the log pages, we need to broadcast information about the cache contents and structure so the clients can figure out how many pages they need, estimate how long it will take to download them, and compute their merit to send a miss to the server. In Table 1, we present the information that the clients need to do that. Assuming a small number of levels, the volume of this data is quite small. Thus, we choose to broadcast it along with every log page. An enhancement over this simple scheme would be to extend index entries with more detailed information about the updates being broadcast. For example, bit-vectors could be used to indicate the data that were updated by the log entries in each page [JEHA97]. These would allow clients to detect which of the log pages affect their data, and possibly, save time and power by downloading pages selectively. Depending on the size of the additional information, such enhancement would require a more sophisticated indexing technique [IVB94a].

4.2 Workload Estimation

In this section we describe how we can assess the actual workload of the system from the relatively small number of misses that reach the server. For our purposes, the workload is expressed as the rate $\boldsymbol{L} = (\lambda_1, \ldots, \lambda_z)$, where λ_m is the rate at which clients require (m)-refreshes. This is the output of the workload estimator that is passed on to the air-cache adapter. Note that the size of $\boldsymbol{L}$ grows as new log pages are created. But, at the same time more and more of the early log pages stop being accessed, zeroing the respective elements of $\boldsymbol{L}$. Therefore, in practice we only need to keep a reduced version of $\boldsymbol{L}$ starting with the oldest log page that got accessed over the last adaptation period.

The way λ_m is estimated depends on the whether page ℓ_m is air-cached or not. If it is not, then all (m)-refreshes yield hard misses since at least one of the required pages (ℓ_m) cannot be found in the air-cache. Therefore, these hard misses are the actual number of

clients that required an (m)-refresh.

If, however, ℓ_m is air-cached the problem is a little more complicated. In this case, a client request for an (m)-refresh will result in some (k, u)-hit with the proper values of k and u. The server does not get any information about this hit, unless a soft miss is created. This means that it receives feedback for only a fraction of the actual (m)-refreshes. But, here is where the knowledge of the soft miss probabilities can be of service. If (m)-refreshes, resulting in a (k, u)-hit, create soft misses with probability $b(k, u)$, then the actual number of (m)-refreshes can be computed by dividing the number of soft misses by this probability.

Still, not all elements of $\boldsymbol{L}$ can be computed in this way. A small difficulty arises for computing the value of an λ_m when the corresponding miss probability $b(k, u)$ is zero or close to zero. In such a case, clients do not send any soft misses, and thus, the server has absolutely no information about those (m)-refreshes. To overcome this problem, we estimate such missing values by interpolating on the ones that are available. Obviously, there is no way of knowing whether these estimates reflect the real workload. However, our experiments confirm that this procedure yields a quite accurate estimation of the workload.

4.3 Air-cache Adapter

The air-cache adapter is the core of the system which makes the critical operating decisions. It has to provide answers to three questions: how many log pages should be air-cached, what is the best way to structure the air-cache for the given log access pattern, and when clients are allowed to send soft misses. Naturally, the answers to these questions are based on the output of the estimator. The adapter is invoked periodically at predefined intervals. As it is shown in Figure 3, the adapter consists of three separate modules that operate in sequence, each deciding on one of abovementioned issues.

In between adaptations, new log pages may be created. These are placed in a separate level of their own. In other words, if the last adaptation phase created an air-cache of C levels, all new pages created before the next adaptation phase will be placed in AC_{C+1}. This way, new pages do not affect the relative structure of the other levels, as this was last determined by the adapter.

Next, we elaborate on the three modules of the air-cache adapter.

4.3.1 Air-Cache Contents

The first decision the system has to take is the extent of the log that needs to be air-cached. The issue here is that we need to satisfy two contradicting goals. On one hand, we would prefer to select as few log pages as possible so that we end up with a small broadcast period

and, therefore, small air-cache latency. But, on the other hand, the less pages we select the more refreshes will span beyond the air-cache, and cause hard misses to be sent to the server. So, in order to prevent the server from overflowing, we have to make sure that it does not receive more misses than it can handle.

We define the capacity (or throughput) μ of the server to be the maximum rate at which it can unicast log pages. This is determined by the server's processing power and/or the available network bandwidth. Our goal is to limit the workload imposed by the misses below that capacity. An important aspect of the system is that server must handle two types of misses. For this reason, we divide the server's capacity into two parts, μ_h and μ_s and allocate them to hard and soft misses respectively ($\mu_h + \mu_s = \mu$).

The number of log pages in the air-cache affects only hard misses. Therefore, the goal of the first module of the adapter is to find what is the minimum number of log pages that should be air-cached so that the workload of hard misses does not exceed the allocated capacity μ_h. Bear in mind that misses can be of different sizes, i.e., different misses request different number of log pages. As a consequence, besides the estimated rate of the misses, we need to take into account the load that each miss generates in terms of the number of pages it requires. Formally, the problem is to find the maximum c for which

$$\sum_{i<c} (c-i)\lambda_i < \mu_h$$

This is rather easy problem; the proper value of c can be found with a single scan of the vector $\boldsymbol{L}$. This value determines the oldest page ℓ_c to be put in the air-cache.

4.3.2 Air-Cache Structure

Having selected the range of log pages to broadcast, we need to decide how to structure the air-cache so that we minimize the refresh time for the clients. As we discussed earlier, this calls for a solution of the optimization problem presented in Section 3.3. The problem was formulated in its most general form. However, the adapter is required to make fast on-line decisions for the structure of the air-cache. For practical solutions to the problem, we limit the number of decision variables. Specifically, we preselect the broadcast frequencies to be $f_k = 2^{k-1}$. This means that each cache level is twice as fast as the next lower level, i.e., $f_{k+1} = 2f_k$ and $T_{k+1} = \dfrac{T_k}{2}$. Under these assumptions, our problem is reduced to finding the n_k's to distribute the cached pages in the C levels so that the objective function is minimized.

But even this is a hard combinatorial problem that we cannot afford to solve optimally online. Instead, we have developed a greedy algorithm that finds a good solution by minimizing an approximation of the objective function. The inputs to the algorithm are the

rate vector $\boldsymbol{L}$ provided by the workload estimator, and the range of log pages to be cached as determined by the first module of the adapter. As the experimental results reveal, this algorithm yields near optimal results. The algorithm is briefly presented in Appendix B.

4.3.3 Defining Soft Miss Thresholds

The last part of the adaptation process is the selection of merit thresholds for soft misses. The trade-off here is similar to that of hard misses. We want to allow as many soft misses as possible without, however, swamping the server. The limiting factor here is μ_s, the server capacity allocated to soft misses.

We control the volume of soft misses on a per-level basis. We establish a merit threshold θ_k for each level AC_k, limiting the number of soft misses for pages in this particular level. Also, AC_k is allocated a capacity μ_k, a portion of μ_s proportional to the number of pages cached in it.

Given the structure of the air-cache, we can compute for any value of the merit threshold θ_k the probability $b(k,u)$ that a (k,u)-hit will generate a soft miss. In addition, we can compute the expected size $g(k,u)$ of such a miss. Therefore, as we also have an estimate for the rate of (k,u)-hits, we can compute the expected soft misses load for level AC_k. This is our basis for selecting the merit thresholds. Specifically, if q is the LSN of the most recent log page cached in AC_k, then θ_k is assigned the minimum value for which

$$\sum_{u=1}^{n_k} g(k,u) b(k,u) \lambda_{q-u+1} < \mu_k$$

5 Experiments

In this section we present the most important results that we obtained from a detailed simulation of the proposed system. In the presentation of the experiments, time measurements as well as simulation parameters are expressed in terms of broadcast units.

The simulation model consists of a server, a variable number of mobile clients, and the network interconnecting them. This network is hybrid in the sense that there are three separate communication paths:

1. The broadcast channel with a fixed bandwidth capable of delivering 1 page per time unit.

2. The unicast downlink from the server to the clients which is a shared resource used for all server replies. We have assume that this link has similar characteristics with the broadcast channel, i.e., it too can transfer 1 page per unit.
3. The uplink(s) from the clients to the server. Because of the small size of requests from the clients, do not consider the possibility of congestion in the uplink channels.

The server model implements the architecture shown in Figure 3. We assume that the processing power of the server is sufficient to utilize the full bandwidth of the downlink. This means that the server can unicast log pages at a maximum rate of $\mu = 1$ page per unit. The generation of updates is simulated through a separate module running at the server. Its function is to create new log pages, and notify the air-cache adapter every time it does. This process is governed by the "inter-arrival" time distribution of log pages, i.e., the distribution of the interval between the generation of successive pages. For all the experiments presented herein, we used exponential inter-arrival time with mean 1000 time units. The adaptation period of the server was also set to 1000 units.

The client model we used is quite simple. Basically, the only characteristic of the clients is the distribution of their sleep time. At the end of a sleep period, a client wakes up, retrieves all log pages created during this period, and then goes back to sleep. We assume that clients do not remain awake after they receive the updates they need. For this kind of operation, the only state information for each client is the most recent log page it received the last time it woke up.

5.1 Fixed Size Air-Cache, No Hard Misses

For the first set of experiments we consider a (probably unrealistic) scenario where the server maintains a fixed number of log pages in the air-cache. Essentially, we relieve the server from the first part of the adaptation procedure, i.e., from having decide how many pages to air-cache. In addition, clients that wake up require log pages only within the range of these pages (following some given distribution), regardless the last log page they received before they went to sleep. This means that no hard misses are sent to the server because clients never require refreshes beyond what is air-cached. Consequently, all the server capacity is allocated to soft misses. The reason we used this hypothetical scenario is that it constitutes some sort of a static case for which we can compute the theoretically optimal air-cache performance. This provides a solid baseline to compare our system against.

For the results presented here, the server always air-caches the last 200 pages. Clients always require refreshes that start with any of these pages. We tested the system for three different distributions for the range of the refreshes, which are depicted in Figure 5:

Normal The clients' refresh size follows a normal distribution with mean 100 and standard

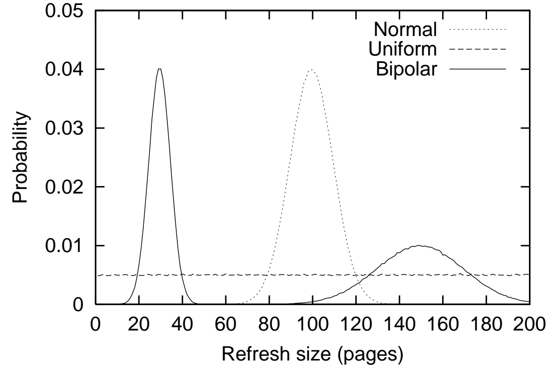


Figure 5: Distributions of client refreshes

deviation 10. This is close to a best case situation where all clients need approximately the same number of pages.

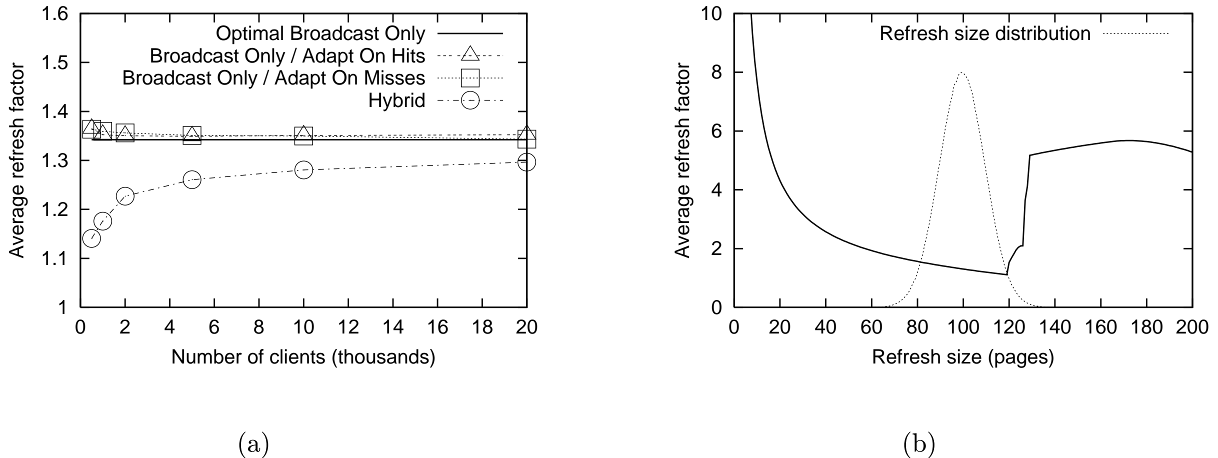
Uniform The sizes of refreshes are uniformly distributed over all 200 pages. For the server, this is a worst case scenario, since it has to equally satisfy a very wide range of client needs.

Bipolar This reflects the situation where clients are partitioned into two equal size groups: workaholics who sleep only a little and usually need few log pages (normally distributed with mean 30 and standard deviation 5), and sleepers who tend to sleep more and require many more updates (normally distributed with mean 150 and standard deviation 20).

Our goal is to show that the system can efficiently disseminate updates even in very large scale, adapting to the (dis)connection habits of the client population. Efficiency is measured in terms of the clients' refresh factors. In the rest of this section, we present the results we obtained for the above three types of workload at a variable scale, i.e., a variable number of clients.

For these experiments we plot up to four different curves to emphasize different aspects of the system's performance. In particular, we want to demonstrate the effectiveness of the cache optimization algorithm, the accuracy of the workload estimation procedure, and the performance benefit of soft misses:

Optimal Broadcast-Only The first curve corresponds to the theoretically optimal performance of the air-cache, when only broadcast delivery is used (i.e., there are no misses). Basically, it is the solution to the optimization problem of Section 3.3 for the given frequencies, and serves as our comparison baseline. These results were obtained with exhaustive search over all the possible air-cache configurations. Keep in mind that the



(a) (b)

Figure 6: Fixed size air-cache - Normal distribution

performance for all "broadcast-only" scenarios depends only on the log access pattern and not on the number of clients. Therefore, it is the same at any scale.

Broadcast-Only / Adapt On Hits This and the next two curves were obtained from the simulation, each under a different setup. In this one, log pages are delivered only through the air-cache, and clients do not generate any misses. Instead we (magically) provide the server with all the information about the clients air-cache hits. This way, the server has a complete and perfect picture about the activity and the needs of the clients. Its only task is to structure the air-cache to according this picture. Compared to the "optimal broadcast-only" curve, this case demonstrates the effectiveness of the air-cache optimization algorithm, without any possible side effects of the miss-based workload estimation.

Broadcast-Only / Adapt On Misses For this curve, we allow clients to send soft misses to the server. The server uses the misses to estimate the workload, but it does not reply to the clients; clients still get the data from the broadcast. Compared to the previous curve, this time the server has the additional task to compose a picture of the workload just from the misses. Therefore, this curve shows the ability of the server to estimate the workload.

Hybrid The last curve corresponds to the performance of the system under normal operation. Clients send soft misses which the server does service, helping them download the required log pages faster. This result shows the performance improvement from soft misses over the broadcast-only cases.

Normal Distribution Figure 6 presents the results for the normal distribution of client refreshes. First, in Figure 6(a) we show the refresh factor for different sizes of the client

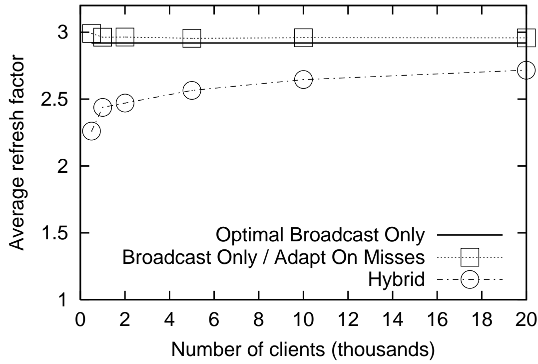
population. The sizes vary from 500 clients to 20000 clients. The clients sleep time is uniformly distributed with mean 15000 units, and the mean size of each refresh is 100 pages. With simple arithmetic, we can compute that at the low end of the scale the clients request, on average, about 3 pages per time unit, while at the high end about 130 pages per unit. Notice that, considering both the unicast and the broadcast channel, the server can transmit only two pages per unit. This means that, for these experiments, the rate at which clients request data is 65 times larger than the available network bandwidth.

The first thing to note from this figure is that, for the most part, the three "broadcast-only" curves are virtually indistinguishable. They all yield the same refresh factor. This result suggests two things: first, the proposed greedy air-cache optimization algorithm generates a near optimal air-cache structure, and second, the workload estimator can accurately assess the clients needs relying only on the misses. An interesting observation is that there seems to be a slight performance degradation at the left end of the graph, i.e., under a light workload. The reason behind this surprising behavior is that under very small request rates, it is harder for the server to detect a pattern in the in the clients' demands. Note that this happens even when the server adapts on the hits, which means that it should not be attributed to the workload estimation.

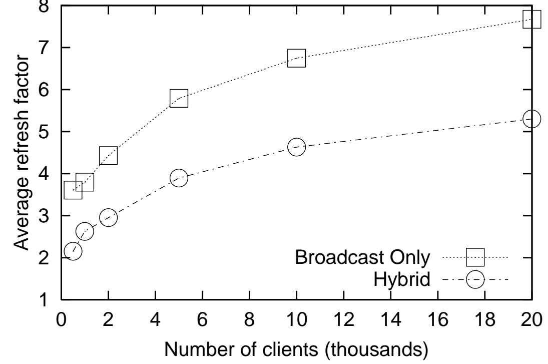
Probably more interesting is the fact that the performance of the full fledged hybrid system is better than the broadcast-only setups. Obviously, this is a reflection of the performance advantage soft misses bring to the system. We defer the discussion on this issue for the next set of experiments where this performance difference is more meaningful.

As we mentioned earlier, this normal distribution is almost a best case scenario for disseminating the log. The reason is that all clients need approximately the same number of pages. Thus, the system can structure the air-cache to match these types of requests really well, yielding an average refresh factor (1.34) close to 1. To demonstrate how this is actually achieved, in Figure 6(b) we present how this factor is broken down for different refresh sizes. The solid line gives the average refresh factors. To put things into perspective, we have also superimposed the log access pattern, at no particular vertical scale. Clearly, the air-cache is optimized to yield minimum factors where the bulk of refreshes are (for sizes between 80 and 120) at the expense of very rare refresh sizes outside this range.

Uniform Distribution Under a uniform distribution of refresh sizes the system has the hardest possible task: structuring the air-cache to serve a very wide range of client demands without giving preference to any one group in particular. This means that it cannot undermine the performance for one type of refreshes in order to improve performance for another, as it did in the previous case. Instead, it tries to level off the performance of everyone as much as possible. The result is a higher average refresh factor. This shown in Figure 7(a) where we plot the average refresh factors for this type of workload. The other parameters of the experiment are the same as in the previous one. This time, the optimal performance for a broadcast-only delivery is 2.92, more than double the factor obtained under the nor-



(a)



(b)

Figure 7: Fixed size air-cache - Uniform distribution

mal distribution. Nonetheless, even in this case, the broadcast-only version of our system performs very close to the optimal.

Here, because of the higher refresh factors, there is also a higher margin for improvement with soft misses. Indeed, in Figure 7(a), the gap between the broadcast-only curve and the hybrid is wider. The reason for that is that soft misses alleviate the delays of lengthy broadcast refreshes. The extend of the performance improvement depends on the volume and size of soft misses the server can accommodate. The lighter the workload the more and bigger soft misses it can service, and thus, the bigger the refresh time savings.

These performance savings are better illustrated Figure 7(b), where we present the average refresh factors for only those refreshes that actually generated a soft miss. The line labeled broadcast-only reflects their performance when the misses were not serviced by the server, and the clients eventually received the data from the broadcast. The other line shows the performance of the same clients when the soft misses were served by the server. The improvement from soft misses is clear; the average refresh factor drops between 30% and 40%. This big difference also implies that the generated soft misses indeed exhibited high merits. It is also interesting to observe the behavior of these two lines with respect to the scale of the experiment. As the workload to the system increases, soft misses affect higher refresh factors. This is an artifact of the dynamic selection of merit thresholds. When the workload increases, the server has to be more selective as to what soft misses it is willing to serve. Thus, it raises the merit thresholds, limiting the ability to create soft misses to refreshes with high factors.

Bipolar This distribution reflects an in-between scenario where the system opts to satisfy two groups of clients with very different needs for updates. As it was expected, in this

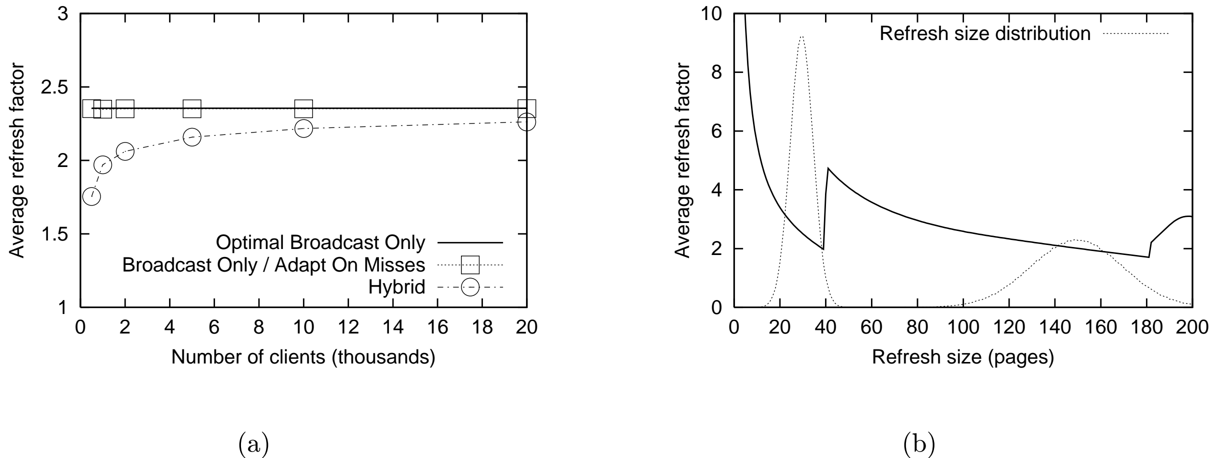


Figure 8: Fixed size air-cache - Bipolar distribution

case the air-cache can do a better job delivering the log pages than under the uniform case, but not as good as the normal case. This time the optimal average refresh factor for broadcast-only delivery is 2.35 (Figure 8(a)). As far as our system is concerned, once again we achieve optimal performance for the broadcast-only versions, and similar improvement with the hybrid version. Figure 8(b) plots the performance of the system for the different refresh sizes. Again, with the help of the superimposed log access pattern, we can see that the algorithm allocated the log pages into cache levels so that the smallest refresh factors fall under the two bells of this bipolar distribution. In other words, it structured the air-cache to satisfy both sleepers and workaholics alike.

Finally, this experiment revealed another significant aspect of the system. Even though the broadcast-only version of our system matches the theoretically optimal performance, it does so using an apparently different air-cache structure. The exhaustive search indicated that the optimal structure is to allocate the 200 pages in four levels so that $(n_1, n_2, n_3, n_4) = (19, 141, 1, 39)$. On the other hand, with the adaptive algorithm the average number of pages in each level were $(159.6, 2.5, 37.9, 0)$; yet they both yield the same performance. In order to ensure that this was not an error in our algorithm or the simulation model, for this particular workload, we computed and compared the theoretical expected refresh factors for all the possible allotments of the 200 pages in up to four cache levels. What we found was that there is a considerable number of combinations that yield performance very close to the optimal, including one similar to that produced by our algorithm. The explanation for that is that even though they may appear quite different, in practice they produce very similar broadcast sequences. In this case, for example, the optimal structure puts the 39 most recent pages in AC_4 where they get broadcast 4 times more often than the bulk of the rest pages (141) cached in AC_2. Our system created a similar effect in a different way: it placed almost the same number (37.9) of the most recent pages in AC_3 where again they get broadcast 4 times more often than the bulk of the rest pages (159.6) placed, in this case, in AC_1.

5.2 Variable Size Air-Cache

For the second set of experiments we used a more realistic scenario, where the size of the air-cache is not fixed. This time the decision of how many log pages to air-cache rests with the server, as it would in a actual deployment of the system.

Clients also operate in a more natural way. In other words, every time they wake up, they ask for all the log pages generated while they were sleeping. After they get all these pages, they go back to sleep for a random period. Sleep times are chosen to create a mixed sleepers/workaholics client population. Half of the clients are characterized as workaholics with sleep time normally distributed with mean 30000 units and standard deviation 5000 units. The other half are sleepers with sleep time normally distributed with mean 150000 units and standard deviation 20000 units. These parameters were chosen so that we obtain a workload similar to the bipolar distribution of the previous section; on average a workaholic requires 30 pages, and a sleeper 150.

Contrary to the previous case, now clients may also generate hard misses, if they happen to need old pages that have been dropped out of the air-cache. As it was described in Section 4.3, in this case the capacity μ of the server must be split to μ_h for hard misses, and μ_s for soft misses. The actual split is specified by the parameter *SoftMissesShare* which corresponds to the portion of μ allocated to soft misses. For example, a value of 0.2 for this parameter means that $\mu_s = 0.2\,\mu$ and $\mu_h = 0.8\,\mu$.

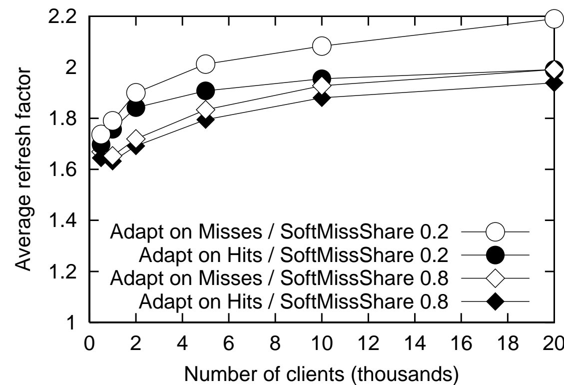


Figure 9: Variable size air-cache

The results of this experiment are shown in Figure 9. Again, the number of client ranges from 500 to 20000. In order to show the effects of the capacities allocated to each part of the system, we plot the results for two different values of the parameter *SoftMissesShare*. For *SoftMissesShare*=0.2, most of the capacity of the server is allocated to hard misses; the opposite holds for *SoftMissesShare*=0.8. For each value of this parameter, we also include the results of a test where the system uses the hits to adapt. These contrast the performance of the system to the ideal, but unrealistic, scenario where the server has perfect knowledge about the workload.

The first conclusion from this graph is that the system exhibits the same significant scalability. Under all configurations, it can efficiently service at least up to 20000 clients, yielding small refresh factors. In fact, the resulting factors are even smaller than those in the previous experiment (Figure 8(a)). This is because in the previous experiment the server was forced to always air-cache 200 pages, even though it was not necessary. This time the server could decide for itself, and indeed chose to have a smaller air-cache. It is important to point out that the system not only performs and scales very well, but also exceeds its nominal capabilities by many times. At the highest scale, it delivered data at a rate 65 times the available network capacity. In other words, by exploiting the commonality between multiple clients, we achieve a manyfold increase of the effective bandwidth.

Furthermore, by comparing the four curves in the figure, we see that the system performs better for the large value of the parameter *SoftMissesShare*, i.e., when it allocates more resources to soft misses. There are two reasons for that: First, the server can accommodate more soft misses, and thus, help more clients improve their refresh times. Second, more misses provide a better picture of the clients' needs and help the server make more informed decisions. This is evident by the fact that, for *SoftMissesShare*=0.8, the the performance curve of the system when adapting on the misses follows very closely the ideal curve of hits-based adaptation. On the contrary, the gap between hits-based and misses-based adaptation is wider for *SoftMissesShare*=0.2, especially in large scale when the server cannot afford to serve many misses.

These results suggest that most of the server capacity should be allocated to soft misses. In order to see whether there is more benefit allocating even more than 0.8 of the server's capacity, we also ran experiments with $\mu_s = 0.9\,\mu$, and $\mu_s = 1\,\mu$. In the first case the results almost matched those for $\mu_s = 0.8\,\mu$. However, when all the server was allocated to soft misses, the refresh times of the clients more than doubled. But, this was an expected result. When we allocate all capacity to soft misses, we effectively prohibit hard misses. This means that the server has to make sure that it receives no hard misses. The only way this can happen is by air-caching all the log, or at least a very big part of it. Naturally, the result is high broadcast periods and, consequently, high refresh factors. Given these observations, a value of 0.8 appears to be a good and safe choice for the *SoftMissesShare* parameter.

6 Related Work

The idea of broadcasting data from some information source to a large number of receivers is being explored for more than a decade. Early work was done in the context of teletext and videotex systems [AW85, Won88], community information services [GBBL85, Gif90], as well as database machines [HGLW87, BGH$^+$92]. More recently, with the proliferation of wireless communication and mobile computing, it regained much more research attention [IB94, FZ96]. Issues that are been addressed include optimized broadcast schedules [Won88, IV94, AAFZ95, ST97, HV97], client prefetching and caching with respect to

the broadcast program [AAFZ95, AFZ96b], power efficiency considerations for battery powered computers [IVB94b, IVB94a], as well as combination of data delivery methods [IV94, AFZ97, DCK+97].

Broadcast of updates was first employed by the Boston Community Information System [Gif90]. The system was using an FM channel to periodically broadcast news articles to an entire metropolitan area. [AFZ96a] investigates the dissemination of updates in the broadcast disks environment. Specifically, they explore several alternatives for propagating updated data (pages) within the broadcast disk program, and study their effects to client caching and prefetching. A basic assumption in this study is that clients are always active, monitoring the broadcast channel. In the same context, they have also proposed augmenting their architecture by allowing clients to explicitly request expeditious delivery of data that are delayed in the regular broadcast program [AFZ97]. In essence, this is technique serves the same purpose soft misses serve in our system.

The issues are somehow different when considering disconnecting clients. In this context, the problem of efficient (in)validation of data cached in mobile clients has been studied by several researchers[BI94, WYC96, JEHA97]. The common thread in all these projects is the use of a broadcast channel to broadcast some form of an invalidation report. Reconnecting clients examine these reports and detect possible stale data in their caches. However, when necessary, the actual updated data are provided on demand.

The presence of updates raises also the problem of finding the most cost effective data replication policy for mobile computers. In [HSW94], the authors analyze the performance of several policies considering the possible pricing schemes for wireless connectivity. Their study considered only point-to-point connections. A interesting avenue of future work is to study the implications of incorporating a mechanism for broadcasting of updates, which amortizes the communication cost over multiple clients.

Finally, in [CTO97], the authors investigate the trade-offs involved in incrementally updating views cached in mobile clients, for different types of update reports. Issues concerning view maintenance for the mobile environment are also discussed in [WSD+95].

7 Conclusions

In this paper, we addressed the problem of propagating updates from a data server to a large number of mobile clients. Such clients typically operate off-line on data replicated from the central database, but occasionally they need to refresh their data with changes committed at the server. We proposed a system that employs adaptive hybrid data delivery, i.e., dynamic combination of broadcasting and unicasting, to disseminate the log of updates to the clients. This technique is realized through air-caching. The server repetitively broadcasts the hot part of the log expecting to satisfy most of the clients' needs, and services only a limited

number of cache misses, i.e., explicit requests sent to the server. These misses provide the necessary feedback for the server to adapt on.

First, we described a hierarchical form of air-caching that supports multiple cache levels. each with different average access latency. We analyzed the performance for broadcasting a log of updates using the hierarchical air-cache, and formulated the optimization problem of structuring the air-cache according to the clients' access pattern. Then, we described the proposed adaptive hybrid system, and elaborated on its key components. We also introduced the notion of soft air-cache misses, i.e., misses for cached data, that allow clients to improve performance over broadcast delivery.

The experimental results confirmed our performance expectations. The system can detect the clients request patterns, and adapt the structure of the air-cache almost optimally to match the sleeping habits of the clients. Also, the dissemination of updates was very efficient. The refresh times for clients was almost constant across a wide scale (up to 20000 clients). Moreover, the results demonstrated the important double role of soft misses for the system: they provide information on the clients sleep time habits, and in some cases help improve performance over broadcast only delivery, especially under not very heavy workloads.

Our plans for future work include an extension of this system to support multiple log files, and application of this technique to incremental view maintenance. Also, we will explore the implementation of publish/subscribe services, again in the context of mobile computing. Last, we are planning to integrate the dissemination of updates with a similar system offering querying capabilities.

References

[AAFZ95] Swarup Acharya, Rafael Alonso, Michael J. Franklin, and Stanley B. Zdonik. Broadcast Disks: Data Management for Asymmetric Communications Environment. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 199–210, San Jose, California, May 1995.

[AFZ96a] Swarup Acharya, Michael J. Franklin, and Stanley B. Zdonik. Disseminating Updates on Broadcast Disks. In *Proceedings of the 22th International Conference on Very Large Data Bases*, pages 354–365, Mumbai (Bombay), India, September 1996.

[AFZ96b] Swarup Acharya, Michael J. Franklin, and Stanley B. Zdonik. Prefetching from Broadcast Disks. In ICDE [ICD96], pages 276–285.

[AFZ97] Swarup Acharya, Michael J. Franklin, and Stanley B. Zdonik. Balancing Push and Pull for Data Broadcast. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 183–194, Tucson, Arizona, May 1997.

[AW85] Mostafa H. Ammar and John W. Wong. The Design of Teletext Broadcast Cycles. *Perfomance Evaluation*, 5(4):235–242, December 1985.

[BDD+98] Randall G. Bello, Karl Dias, Alan Downing, James Feenan, William D. Norcott, Harry Sun, Andrew Witkowski, and Mohamed Ziauddin. Materialized Views in Oracle. In *Proceedings of the 24th International Conference on Very Large Data Bases*, pages 659–664, New York, NY, USA, August 1998.

[BGH+92] Thomas F. Bowen, Gita Gopal, Gary E. Herman, Takako M. Hickey, K. C. Lee, William H. Mansfield, John Raitz, and Abel Weinrib. The Datacycle Architecture. *Communications of the ACM*, 35(12):71–81, December 1992.

[BI94] Daniel Barbará and Tomasz Imielinski. Sleepers and Workaholics: Caching Strategies in Mobile Environments. In SIGMOD [SIG94], pages 1–12.

[BZ96] Jon C. R. Bennett and Hui Zhang. WF^2Q: Worst-case Fair Weighted Fair Queueing. In *Proceedings of INFOCOMM*, San Francisco, CA, March 1996.

[CTO97] Jun Cai, Kian-Lee Tan, and Beng Chin Ooi. On Incremental Cache Coherency Schemes in Mobile Computing Environments. In ICDE [ICD97], pages 114–123.

[DCK+97] Anindya Datta, Aslihan Celik, Jeong G. Kim, Debra E. VanderMeer, and Vijay Kumar. Adaptive Broadcast Protocols to Support Efficient and Energy Conserving Retrieval from Databases in Mobile Computing Environments. In ICDE [ICD97], pages 124–134.

[DR92] Alexis Delis and Nick Roussopoulos. Performance and Scalability of Client-Server Database Architecture. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 610–623, Vancouver, Canada, August 1992.

[DR98] Alex Delis and Nick Roussopoulos. Techniques for Update Handling in the Enhanced Client-Server DBMS. *IEEE Transactions on Knowledge and Data Engineering*, 10(3):458–476, May/June 1998.

[FZ96] Michael J. Franklin and Stanley B. Zdonik. Dissemination-Based Information Systems. *IEEE Bulletin of the Technical Committee on Data Engineering*, 19(3):20–30, September 1996.

[GBBL85] David K. Gifford, Robert W. Baldwin, Stephen T. Berlin, and John M. Lucassen. An Architecture for Large Scale Information Systems. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 161–170, Orcas Island, Washington, December 1985.

[Gif90] David K. Gifford. Polychannel Systems for Mass Digital Communications. *Communications of the ACM*, 33(2):141–151, February 1990.

[GWD94] Alex Gorelik, Yongdong Wang, and Mark Deppe. Sybase replication server. In SIGMOD [SIG94], page 469.

[HGLW87] Gary E. Herman, Gita Gopal, K. C. Lee, and Abel Weinrib. The Datacycle Architecture for Very High Throughput Database Systems. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 97–103, San Francisco, California, May 1987.

[HSW94] Yixiu Huang, A. Prasad Sistla, and Ouri Wolfson. Data Replication for Mobile Computers. In SIGMOD [SIG94], pages 13–24.

[HV97] Sohail Hameed and Nitin H. Vaidya. Log-time Algorithms for Scheduling Single and Multiple Channel Data Broadcast. In *The 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking*, Budapest, Hungary, September 1997.

[IB94] Tomasz Imielinski and B. R. Badrinath. Wireless Mobile Computing: Challenges in Data Management. *Communications of the ACM*, 37(10):18–28, October 1994.

[ICD96] *Proceedings of the 12th International Conference on Data Engineering*, New Orleans, Louisiana, February 1996.

[ICD97] *Proceedings of the 13th International Conference on Data Engineering*, Birmingham, U.K., April 1997.

[IV94] Tomasz Imielinski and S. Vishwanathan. Adaptive Wireless Information Systems. In *Proceedings of SIGDBS (Special Interest Group in DataBase Systems) Conference*, Tokyo, Japan, October 1994.

[IVB94a] Tomasz Imielinski, S. Viswanathan, and B. R. Badrinath. Energy Efficient Indexing on Air. In SIGMOD [SIG94], pages 25–36.

[IVB94b] Tomasz Imielinski, S. Viswanathan, and B. R. Badrinath. Power Efficient Filtering of Data on Air. In *4th International Conference on Extending Database Technology*, pages 245–258, Cambridge, United Kingdom, March 1994.

[JEHA97] Jin Jing, Ahmed Elmagarmid, Abdelsalam (Sumi) Helal, and Rafael Alonso. Bit-Sequences: An adaptive cache invalidation method in mobile client/server environments. *Mobile Networks and Applications*, 2(2):115–127, October 1997.

[RK86] Nick Roussopoulos and Hyunchul Kang. Principles and techniques in the design of adms+/-. *IEEE Computer*, 19(12):19–25, 1986.

[SIG94] *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, Minneapolis, Minnesota, May 1994.

[SRB96] Konstantinos Stathatos, Nick Roussopoulos, and John S. Baras. Adaptive Data Broadcasting Using Air-Cache. In *1st International Workshop on Satellite-based Information Services*, pages 30–37, Rye, New York, November 1996.

[SRB97] Konstantinos Stathatos, Nick Roussopoulos, and John S. Baras. Adaptive Data Broadcast in Hybrid Networks. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 326–335, Athens, Greece, August 1997.

[ST97] C.-J. Su and Leandros Tassiulas. Broadcast Scheduling for Information Distribution. In *Proceedings of IEEE INFOCOM'97*, Kobe, Japan, April 1997.

[Sta98] Konstantinos Stathatos. *Air-Caching: Adaptive Hybrid Data Delivery.* PhD thesis, Department of Computer Science, University of Maryland, 1998.

[Won88] John W. Wong. Broadcast Delivery. *Proceedings of the IEEE*, 76(12):1566–1577, December 1988.

[WSD+95] Ouri Wolfson, Prasad Sistla, Son Dao, Kailash Narayanan, and Ramya Raj. View Maintenance in Mobile Computing. *SIGMOD Record*, 24(4):22–27, December 1995.

[WYC96] Kun-Lung Wu, Philip S. Yu, and Ming-Syan Chen. Energy-Efficient Caching for Wireless Mobile Computing. In ICDE [ICD96], pages 336–343.

A Formulas

Expected refresh time Given the structure of the air-cache, the expected refresh time for a (k,u)-hit is $\mathrm{E}[R_m] = T_k\,\Phi(k,u)$ where

$$\Phi(k,u) = \begin{cases} \dfrac{1}{2} + \dfrac{u}{n_k} - \dfrac{2u}{n_k}\dfrac{f_k}{f_{k+1}} + \left(\dfrac{f_k}{f_{k+1}}\right)^2 & \text{if} \quad u < \dfrac{n_k f_k}{f_{k+1}} \\ \dfrac{1}{2} + \dfrac{u}{n_k} - \left(\dfrac{u}{n_k}\right)^2 & \text{if} \quad u \geq \dfrac{n_k f_k}{f_{k+1}} \end{cases}$$

Soft miss probability and expected size Here we give the formulas for the probability that (k,u)-hit will create a soft miss, as well as the expected size of that miss. For clarity, we define the function

$$h(k,u) = 1 - \frac{\theta_k u}{n_k} - \frac{f_k}{f_{k+1}}$$

For a given threshold θ_k, the probability that a (k,u)-hit will create a soft miss, i.e., its merit will be higher than the threshold, is

$$b(k,u) = \begin{cases} h(k,u) & \text{if} \quad u < \dfrac{n_k}{\theta_k}\left(1 - \dfrac{f_k}{f_{k+1}}\right) \\ 0 & \text{if} \quad u \geq \dfrac{n_k}{\theta_k}\left(1 - \dfrac{f_k}{f_{k+1}}\right) \end{cases}$$

The expected size of such a miss is

$$g(k,u) = u - \frac{u^2}{2n_k h(k,u)}$$

B Air-cache optimization algorithm

Here we describe the algorithm used by the adapter to optimize the air-cache. The goal is to distribute the cached log pages into different levels in a way that minimizes the clients' expected refresh factor. Formally, given that pages ℓ_c through ℓ_z should be cached, we have find the n_k's that minimize the objective function

$$\sum_{i=c}^{z} p_i \, \mathrm{E}\left[F_i\right]$$

where p_i is the probability that a client that wakes up requires a (i)-refresh. Note that this probability can be computed from the rate vector $\boldsymbol{L}$ since $p_i = \lambda_i / \sum_{j=c}^{z} \lambda_j$. Our algorithm is based on the following approximation of the objective function:

$$\begin{array}{rrcl}
 & \sum_{i=c}^{z} p_i \mathrm{E}\left[F_i\right] & = & \sum_{i=c}^{z} p_i \frac{\mathrm{E}\left[R_i\right]}{z+1-i} \\
\left(i = c + \sum_{q=1}^{k-1} n_q + j - 1\right) & & = & \sum_{k=1}^{C} \sum_{j=1}^{n_k} p_i \frac{T_k \Phi(k,j)}{z+1-i} \\
\left(\frac{\Phi(k,j)}{z+1-1} \approx \frac{1}{z+1-i}\right) & & \approx & \sum_{k=1}^{C} \sum_{j=1}^{n_k} p_i \frac{T_k}{z+1-i} \;=\; \sum_{k=1}^{C} T_k \sum_{j=1}^{n_k} \frac{p_i}{z+1-i} \\
\left(\pi_i \equiv \frac{p_i}{z+1-i}\right) & & = & \sum_{k=1}^{C} T_k \sum_{j=1}^{n_k} \pi_i \\
\left(\Pi_k \equiv \sum_{j=1}^{n_k} \pi_i\right) & & = & \sum_{k=1}^{C} T_k \Pi_k \;=\; T \sum_{k=1}^{C} \frac{\Pi_k}{f_k} \\
\left(f_k = 2^{k-1}\right) & & = & 2T \sum_{k=1}^{C} \frac{\Pi_k}{2^k}
\end{array}$$

We have developed a greedy algorithm that tries to minimize this new objective function. The algorithm starts by assigning all the pages into the lower level, and computes an initial value for the objective function. Then, it examines if the pages can be split into two parts so that if the most recent part is moved to the higher level, the value of the function is decreased. If there is not such a split then it stops. If there is, it splits the pages in the way that yields the minimum value for the objective function, and assigns the most recent part to the higher level. Then, it recursively applies the same check for the next level, i.e., it checks whether some of the pages that were moved in that level can be raised to even higher levels. The recursion stops when there is no split that can further reduce the value of the objective function.