Energy-Efficient Indexing For Information Dissemination In Wireless Systems

Narayanan Shivakumar, Suresh Venkatasubramanian
Dept. of Computer Science
Stanford, CA 94305
{*shiva, suresh*}*@cs.stanford.edu*

Abstract

We consider the application of high volume information dissemination in broadcast based mobile environments. Since current mobile units accessing broadcast information have limited battery capacity, the problem of quick and energy-efficient *access to data becomes particularly relevant as the number and sizes of information units increases. We propose several randomized and Huffman-encoding based indexing schemes that are sensitive to data popularity patterns to structure data transmission on the wireless medium, so that the average energy consumption of mobile units is minimized while trying to access desired data. We then propose an algorithm for PCS units to tune into desired data independent of the actual transmission scheme being used. We also empirically study the proposed schemes and propose different transmission modes for the base station to dynamically adapt to changes in the number of data files to be broadcasted, the available bandwidth and the accuracy of data popularity patterns.*

1 Introduction

In a Personal Communications System (PCS), users send and receive data through a wireless medium. Information may be voice, data, text, facsimile, or video information [JLLM94]. PCS users are located in system-defined *cells*, which are bounded geographical areas. In current systems, PCS users communicate with *base stations* located in their cell on private channels to access desired data. This scheme requires significant amounts of wireless bandwidth, and the base stations become bottle-necks since they service queries on a request-by-request basis. To avoid multiple transmissions of commonly requested data, there have been recent proposals [Che92, IVB94] to broadcast popular data items such as stock information, sports updates and multi-media newsfeeds on a known set of channels. An unlimited number of PCS users can then tune into these common channels, and extract their desired data without tying up the limited number of wireless channels. Also, the base stations will not be overloaded with service requests.

We consider the problem of how to organize broadcast information so that PCS users can access desired data without wasting battery power on their power-starved PCS units in sifting through undesired data. This problem becomes particularly relevant in a high volume information dissemination system with several large information units such as multi-media data being transmitted constantly on the wireless link.

In this paper, we make the following assumptions about the structure of the system. All data items have unique keys known by PCS users, and all data items are indexed by their key. The base station transmits index information of broadcast data items that function as pointers to the data items. The PCS units sift through these small sized indices, spending less energy (and time) than in sifting through the large data items. The wireless bandwidth is split into two bands: the *data* band that transmits the actual data items, and the *index* band that transmits the index information along with the *<CNUM, TIME>* pointers, indicating the channel number and the offset in time when the relevant data item will be transmitted. We assume there is some "well known" channel called *ROOTCNUM* on the Index Band that every PCS unit will first tune into if it requires a data item. We also assume each base station has some knowledge of popularity patterns of data items in terms of how many PCS users are interested in the various data items within the cell. The base station may use this information in structuring the broadcasted data so that index information of more popular data items is transmitted more often, and PCS users can download popular data items while consuming minimal power. In this paper, we primarily consider the problem of how to construct such an efficient index structure, and how to transmit it on the Index Band. Since the popularity information may vary from time to time and from cell to cell, we also consider the problem of how to dynamically update our transmitted index structure. We do not consider the related problem of scheduling the transmissions of data items on the Data Band. We expect to use a scheduler used in Video-On-Demand (VOD) systems such as Pyramid [VI95], or the disk-striping scheduler in [AAFZ95] to schedule the actual transmission of data, while our indexer uses the generated transmission schedule to fill in the <CNUM, TIME> pointers in our index structure.

Since there may be several ways of organizing indices on air, we evaluate the different schemes by comparing the (1) *tuning time*, which is the time spent by the PCS unit in listening to wireless transmissions before it downloads desired information, (2) *index access time*, which is the time elapsed from the point a PCS unit starts sifting through the broadcasted data for desired data to the time the PCS unit gets a <CNUM, TIME> pointer to the desired data, and (3) the number of index channels used to transmit the index. Tuning time is a particularly important measure since PCS units consume valuable energy receiving wireless packets for that period of time, while at other times they consume minimal power by lapsing into the *doze mode* [SCB92]. Since the amount of power consumed while receiving packets is about 5000 times the power consumed while in the doze mode [SCB92], it is important to try and minimize the tuning time of the power-starved PCS. From now, we will refer to the time spent downloading index information as tuning time, since the tuning time to download the data file (once an index pointer is obtained) is constant and independent of the indexing scheme, and hence does not contribute in evaluating the different indexing schemes. The index access time is also important since it indicates the *quality of service* (QOS) provided by the system in terms of how long a user will have to wait from the time he desires a data item to when he obtains the index pointer to the data item; index access time is equal to the tuning time in addition to the time spent dozing while waiting for a desired index pointer.

In this paper, we propose an indexing scheme based on binary Alphabetic Huffman trees [HT71, Knu85] so that the average tuning time of PCS units is minimized. The overall idea is that the index information for broadcast data is organized in a tree at different levels (based on popularity patterns), and we transmit the nodes of the tree (with the index information) on the Index Band. We also present some alternate indexing schemes that are sensitive to popularity patterns, to evaluate our proposed index structure. Our primary contributions in this paper are as follows:

1. We propose a simple k-ary extension to Alphabetic Huffman trees, so that we can pack in more index information into a given wireless packet size (a property of the wireless system).
2. We propose an efficient way of embedding index trees into a two dimensional channel-time space in a practical implementation so that the access time for accessing desired leaves (index pointers) is minimized.
3. We present a non-CPU intensive algorithm to be executed by the PCS unit to find desired data independent of the indexing scheme (Alphabetic Huffman based and the other presented indexing schemes) being used by the base station.
4. We show how to dynamically update index structures while they are being transmitted on air, when popularity patterns of data items change.
5. We empirically study the performance of all proposed schemes and see that the Huffman based indexing dominates the other schemes for a large number of broadcasted files with significant energy savings (upto 13.5 times less energy than the energy consumed in the other schemes), while the other schemes tend to do better when there are very few files to be transmitted.
6. We propose a hybrid index transmission mode for the base station incorporating the best features of the different indexing schemes to dynamically adapt to changes in the number of data items to be broadcasted, the number of available index channels, and the accuracy of popularity patterns.

1.1 Related Work

Current broadcast based multi-media and VOD systems such as Pyramid [VI95] address a related problem in transmitting movies by assuming a "TV Guide" cached locally at the receiving unit. Since the TV Guide indicates the transmission time and channel number of various movies, finding the data item is straightforward. Such caching schemes are not useful in mobile environments when mobiles may visit different cells and the broadcasts may be time-varying and may be different across cells [IVB94].

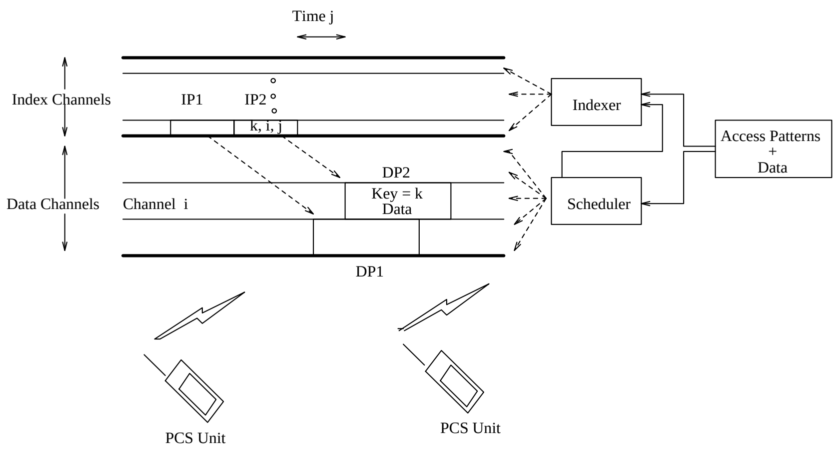

Figure 1: Architecture of a Broadcast System.

The notion of "indexing on air" was introduced in [IVB94] and two different indexing schemes were proposed. However, these schemes assume uniform popularity patterns for all data, while our schemes are sensitive to popularity patterns. Disk-Striping on air was introduced in [AAFZ95] where data items are classified into three tiers of data (for example: very popular, popular, not popular), and the items in the highest tier are transmitted more often than the others. We believe that finding thresholds dividing data items into these tiers is subjective, and may not lead to the best solution. [Chi94] propose a windowed randomized algorithm that is sensitive to data popularity patterns. We use a simplifed version of this scheme as an experimental comparison to our method.

2 Broadcast Architecture

In Figure 1, we present the architecture of our proposed broadcast based dissemination mechanism. *Files* (data) are to be broadcasted to PCS users who are listening for their desired file. Files may be text files, multi-media data, stock information or any other piece of discrete information.

We propose the bandwidth available to the base-station be split into the *Index Band* and *Data Band*. The index band and the data band are further split into several channels. These channels may be *physical* channels (frequency multiplexed), or may be *virtual* channels that are essentially several channels that are time multiplexed onto the same physical channel. The *Scheduler* and the *Indexer* at the base-station control the transmissions on the data band and the index band respectively.

The *Scheduler* receives the files to be transmitted, and possibly more information such as access patterns, priority and size of the file. It then computes a "good" schedule for the file transmissions: for every file, the scheduler computes the <*CNUM, TIME*> vector that indicates the channel and time the file will be transmitted. In this paper, we do not address the problem of scheduling the files. We assume there is an abstract Scheduler (as in [VI95, AAFZ95]) functioning orthogonally to our Indexer, providing our indexer with the file broadcast time and channel numbers.

Our *Indexer* receives the access patterns of the files, and the scheduling information from the Scheduler. Based on the popularity patterns of the files, the indexer computes an index structure that provides fast access links to the files being transmitted in the data band. Intuitively, we would like the Indexer to provide faster access links to the more popular files, so that the average tuning energy consumed by the PCS units in "searching" for the desired file is minimized. In the classical database sense, the scheduler decides "where to store the data" (channel number, time on air), and some amount of "memory" (bandwidth) is wasted by using the index to obtain fast and energy-efficient access to files. In Figure 1, we see *index packets* (IPs) being broadcasted in the index channels containing time, channel pointers $< i, j >$ to the file with key $= k$ indicating that the corresponding file (data packets DPs) will be available on the i^{th} data channel, j time units later.

To illustrate the various indexing schemes we will subsequently develop, we use a running example of files to be broadcasted and their popularity patterns (in expected number of PCS units desiring a file) as shown in Table 1. Although the files in this example have single character keys, our indexing schemes (outlined later) work whenever keys have any form of ordered labelling.

Key	Frequency
A	23
B	4
C	12
D	10
E	17
F	31
G	15
H	21
I	29
J	19
K	7
L	12
M	16
N	14
O	20

Table 1: Files and their Popularity Patterns.

2.1 Structure of Index Packets

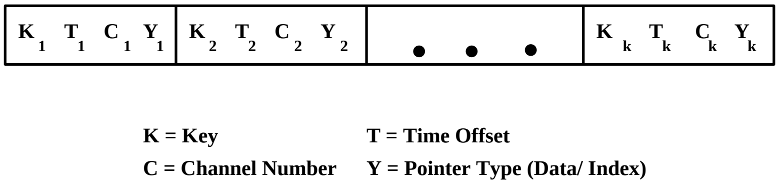

Figure 2: Structure of a packet

In this section, we present the schema of the index packets. Let the wireless packet be of size s (in bits). Each packet stores k instances of $< KEY, TYPE, CNUM, TIME >$ (Figure 2) which may mean one of the following.

1. If $TYPE = DATA$, file with key equal to KEY will be transmitted in channel $CNUM$, $TIME$ time units later.
2. If $TYPE = INDEX$, files with keys lesser or equal in value to KEY, have more index pointers in the next index packet on channel $CNUM$, $TIME$ time units away (useful for multi-level indices).

Since, we need one bit to encode the $TYPE$ ($DATA$ or $INDEX$) k can be determined by

$$k = \lfloor \frac{s}{y+c+t+1} \rfloor$$

where y, c, and t refer to the number of bits required to represent the maximum number of data files to be broadcasted, the maximum number of channels, and the maximum offset in time.

2.2 Some Simple Broadcasting Schemes

In this section, we outline some simple indexing schemes that could be used to broadcast indexing information for a given set of files, and show why such schemes may not be efficient for a large-scale dissemination system. Let n refer to the number of files to be broadcasted. We present examples to illustrate each of the schemes assuming that the fanout (k) of the wireless packet is three.

Round Robin Index: The simplest way of broadcasting index information is to send index information for each file one by one cyclically (Figure 3). The PCS unit then has to be constantly tuned in until its desired file pointer is transmitted, all the while consuming valuable power. This scheme will require the PCS unit to download an average of $\frac{n}{2*k}$ index packets, which is energy-inefficient for $n >> k$. Note that this

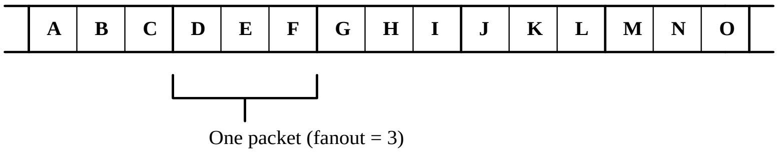

Figure 3: Round Robin Index

scheme treats all files identically, ignoring popularity patterns. This scheme does not require any ordering of keys.

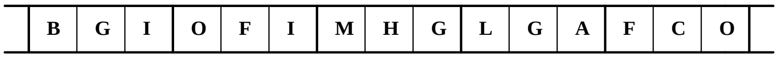

Figure 4: Random Index Broadcast

Random Index: In this scheme, we transmit index entries for a file with a probability proportional to the popularity of that file (Figure 4). The intuition is that popular files are transmitted more often probabilistically, and hence can be located quickly. However, the downside is that users tuning for unpopular files may have to wait a long time, possibly forever (starvation) for the index information of a file to be transmitted. The PCS unit has to be listening constantly until its required file pointer is available: the average tuning time in this case is still $O(n)$ (Appendix A). This scheme does not require ordering of keys either.

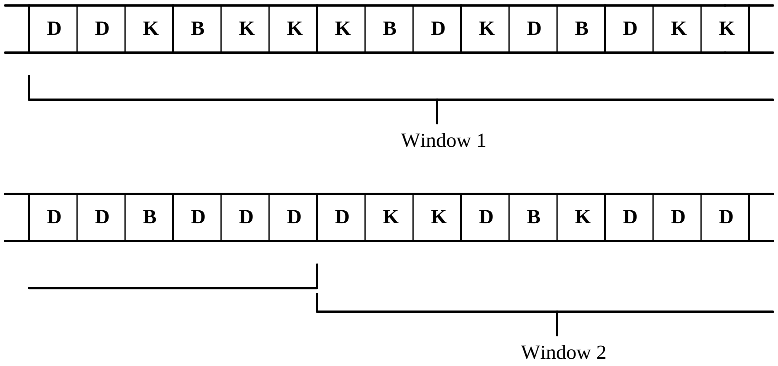

Figure 5: Randomized Index Broadcast with Bounded Wait

Windowed Randomization: [Chi94] propose a compromise on the previous scheme so that PCS units listening for unpopular information do not suffer too much. In their scheme, a fixed *window* size W is defined. Index entries are sent multiple times within one *epoch* of size W [1]. The problem is now to minimise the average tuning time for a packet, given the constraints on the number of packets per epoch. One way of approximating this is to allow each entry to occur a number of times proportional to its frequency within each epoch. The order of the entries is random.For example, in Figure 5 we see an example using keys **B**, **D** and **K** from Table 1. The window size is 21. The Windowed Randomization scheme ensures that the worst case tuning time is bounded by W. However, since W must be at least [2] $1/p_n$, where p_n is the probability of access of the least popular file, the size of a window should be at least n/k. Hence the average tuning time is $\omega(n)$ (i.e. lower bounded by c*n for some constant $c > 0$). Notice that this scheme does not require ordering of keys.

Since each of the above schemes have tuning times that vary linearly with the number of files to be broadcast, these schemes do not scale very well for a large-scale information dissemination system (also empirically shown in the Experimental section) since PCS units will waste too much energy in sifting through undesired data. In the next section, we present a multi-level indexing scheme that is sensitive to popularity patterns and provides tuning times logarithmic in the number of broadcast files.

3 Huffman Based Indexing

In this section, we present our primary index organization scheme based on *Alphabetic Huffman* Trees. We first draw a parallel between our problem and the well-known Huffman [Huf52] compression scheme. Since Huffman trees cannot function as search trees (we show an example indicating why), and thereby are impractical as indexing structures, we present a k-ary generalization of Alphabetic Huffman trees [HT71, Knu85], a sub class of Huffman trees that can function as search structures.

3.1 Generic Huffman Trees

In this section we develop a formal model for minimizing *tuning time*, which is the time spent listening for the desired index entry (proportional to the tuning energy), in an abstract search structure. In the next section, we will show how to use this model in a practical broadcast scheme.

Let n be the number of files to be broadcasted. Let us consider a model of transmission where the *popularity patterns* of files, which indicate the expected number of PCS accesses to the files, is known prior to the broadcast. Let the popularity distribution[3] of the files be $F = (f_1, f_2, \ldots, f_n)$. We need to minimize

[1] Their scheme is not an indexing scheme, but a data broadcast scheme

[2] This ensures that each packet occurs atleast once in an epoch

[3] We will show in a subsequent section how to compute and maintain popularity patterns.

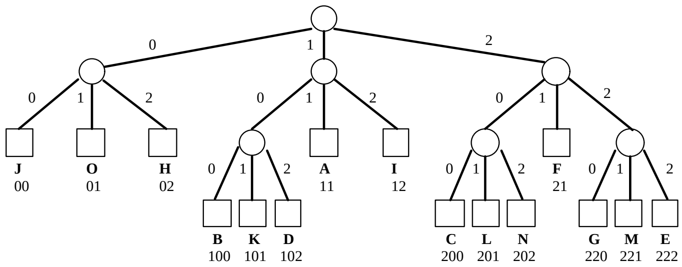

Figure 6: 3-ary Huffman Tree

the *average tuning time* for a PCS unit to locate the index entry for a file it desires. If the tuning time to locate index entry for file i is $T(i)$, we have to minimise the following expression by choosing T(i) for all $i = 1, 2, \ldots, n$.

$$\frac{\sum_{i=1}^{n} f_i \times T(i)}{\sum_{i=1}^{n} f_i} \tag{1}$$

This problem is analogous to the data compression problem [LH87] where codes are computed for different ASCII characters in a given text depending on the frequency of occurence of the characters so that the resulting encoded text is of minimum length. That is, the goal in compression is to compute an encoding $E(i)$ for all characters $i = 1, 2, \ldots, n$ so that the total encoded length

$$\sum_{i=1}^{n} f_i \times E(i) \tag{2}$$

is minimized. This problem was optimally solved using binary trees by Huffman[Huf52]. Since the denominator in (1) is a constant, we can minimize the average tuning time by considering our problem as a Huffman encoding problem in which we want to "encode" the index entries to be broadcasted. Intuitively, we organize the index pointers along the nodes of the Huffman tree so that index pointers of more popular data items are higher up in the tree than others. The Huffman tree can then be transmitted as our index structure, with the tree depth of the index pointer of a file denoting the number of packets that will be examined by a PCS unit before the corresponding file is reached (proportional to tuning time). For a Zipfian distribution [Zip49] on the frequencies, the average number of packets examined is $O(\log n)$ (See Appendix B).

However, the Huffman tree thus constructed is not a search tree since users will need to know the encoding of a file before they can traverse the tree for the given file. To illustrate, we show in Figure 6 a regular 3-ary Huffman tree constructed from Table 1. As we can see, even though the index pointers are at different levels of the tree based on the popularity information, there is no way of traversing the tree to find a desired pointer by knowing only its key. Hence the only way we can access a specific leaf is to know its Huffman *code*. For example, to access leaf B (index pointer to file B), PCS units must know the code 100 for them to be able to choose the right pointers from the root of the tree. This is not a problem in data compression,

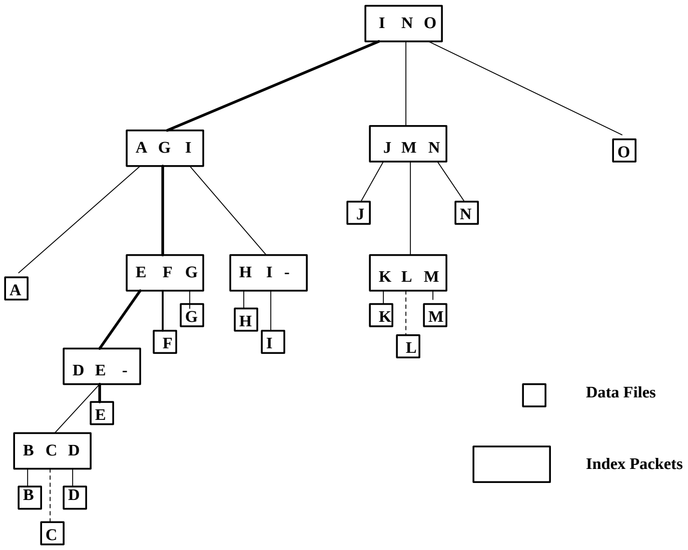

Figure 7: Alphabetic Huffman Based Index.

because both the compressor and the uncompressor have access to the tree used to construct the codes, or can compute the codes given the frequency counts of the different ASCII characters.

However, this will pose a problem for PCS units since they only have the key of the file they are looking for. PCS units cannot know the Huffman code of their desired file beforehand since the code depends on the popularity patterns of other files being broadcasted at that time, and may change over time and between cells. One approach would have PCS units request codes from the base station for its file, but this will again require private channels for communication thereby fragmenting bandwidth and rendering base stations as the bottle-neck. Another approach would be to broadcast the key to code mapping on another set of index channels, but this is again a *meta-search* problem since data will now have to be structured on those channels leading us back to the same index organization problem.

3.2 Alphabetic Huffman Trees

In this section, we present a Huffman-based indexing scheme that is sensitive to data popularity patterns, but requires PCS units to know only the key of their desired file.

There exist a special class of Huffman trees termed Alphabetic Huffman trees [HT71, Knu85] that can function as search trees. The principal feature of such trees is that they preserve leaf ordering on any input sequence used to construct them (similar to B+ Trees [Knu73]); that is a left-to-right scan of the leaves of each tree will show the leaves ordered by their keys. In such an ordered structure, a simple comparison-based tree traversal suffices to determine the location of a desired leaf. Additionally, being Huffman trees, they are sensitive to a frequency distribution on the input.

The Alphabetic Huffman tree constructed in [HT71] has binary fanout. In our application, given a wireless packet of any size (Section 2.1), we can pack in more tree pointers (as opposed to only 2 in [HT71]) into one wireless packet. For example, if the wireless packet size is 128 bytes and each KEY and index pointer takes 6 bytes, we can pack as many as 21 KEY, index pointers into one packet. In Appendix C, we briefly outline the Hu-Tucker [HT71] algorithm, and our k-ary extension where k is the fanout of the wireless packet size. Since the details of the construction are very involved, we have structured the rest of the paper so that the construction may be skipped at a first reading. It suffices to know from this point that Alphabetic Huffman tress are very similar to our earlier outlined Huffman indexing scheme, except that the leaves are ordered by their keys and hence can function as a search structure.

In Figure 7, we show the 3-ary Alphabetic Huffman tree for the example in Table 1. Index packets represented by rectangles, contain keys that are interpreted as specified in Section 2.1. Data packets represented by squares, are indicated in the figure only by their keys. Since the k-ary construction produces a variable fanout (between 2 and k), we use "-" to indicate dummy pointers that pad the residual fanout pointers. Observe that a left-to-right scan of the leaves of the tree shows the leaves to be ordered based on their keys. In Figure 7, we use a darkly outlined path to indicate how a PCS unit may access file E by knowing the key of the desired file to be E. Intuitively, the PCS units download index packets and compute the next index pointer to follow based on the values of the keys in the index packet. For instance, PCS units desiring file E choose the left-most child index pointer from index packet $< I, N, O >$ and access $< A, G, I >$ next since $E < I$ and so on. (We specify the exact algorithm to download packets in a subsequent section.)

The Alphabetic Huffman tree is not optimal over all possible search structures, but the construction outlined in [HT71] guarantees optimality over all Alphabetic trees. From [Knu73] we however see that Alphabetic trees produce a total encoding length (Equation (2)) no greater than 2 units more than the entropy of the data, which is a lower bound on any search tree. Hence our index based on Alphabetic Huffman trees is close to optimal, and for Zipfian frequency distributions, will yield tuning times logarithmic in the number of files (see Appendix B).

3.3 Mapping Index Trees To Channel-Time Space

In the last section, we saw how to compute an energy-efficient index structure based on Alphabetic Huffman trees. However in a practical implementation, we will need to broadcast the nodes of the tree as packets on wireless channels and at certain time slots. We now present a simple way of computing the transmission schedule in the channel-time space given a variable fanout index tree (such as in an Alphabetic Huffman tree). In our scheme, we assign one index channel to each level of the tree, and nodes of a given level are transmitted cyclically on the corresponding index channel. In Figure 8 we see an example of how to map the nodes of the index tree onto the channel-time space. For clarity, we represent the index packets in Figure 8 using numbers rather than the values of the keys as in Figure 7.

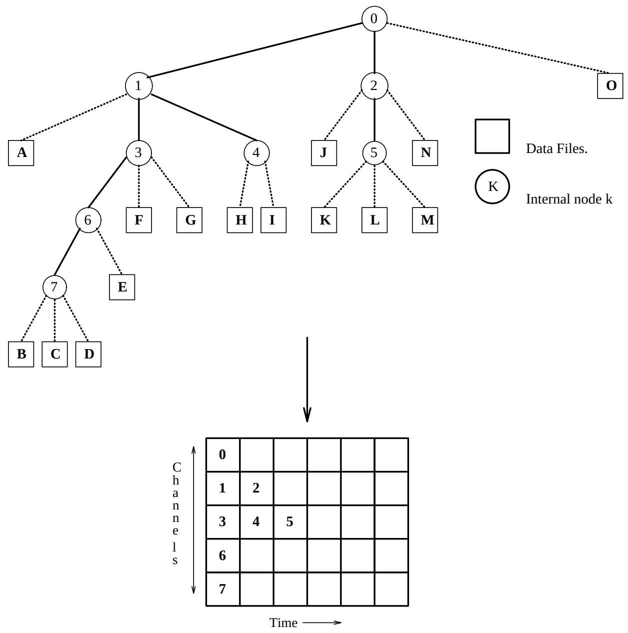

Figure 8: Mapping from the Tree to the Grid.

Each index packet needs to know the time offset when its child index packets are to be transmitted (to fill in the *CNUM, TIME* pointers). For instance in Figure 8, the index packet corresponding to internal node 1 will need to know when and where data file A, and index packets 3 and 4 are to be transmitted so that it may store logical pointers to them. We perform this "pointer filling" as follows.

Initially, we generate a two dimensional grid G with

$$G(i,j) = j^{th} \text{ internal node (in a left-to-right scan) at level } i \text{ in the tree.}$$

Let n_i denote the number of index packets at the i^{th} level. The index packet at position $G(i,j)$ will then be broadcasted at periodically at times

$$T = T_s^i + m * n_i + j, m \geq 0 \tag{3}$$

where T_s^i is the time at which broadcast begins on channel i (initially T_s^i is 0 for all channels). We can compute the earliest time at which the index packet corresponding to $G(i,j)$ will be broadcast using the following formula.

$$T' = T_s^i + \lceil(\frac{T - T_s^i - j}{n_i}\rceil) * n_i + j \tag{4}$$

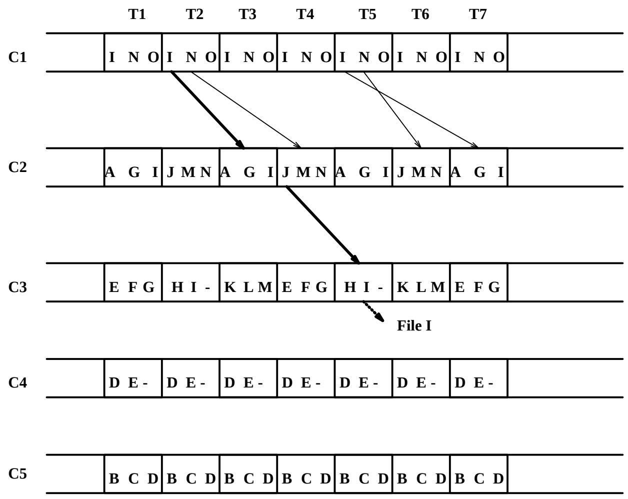

Figure 9: Channel Traversal

where T is the current time[4]. With this information, all nodes that point to a next level index for a given key, can fill in the $< CNUM, TIME >$ pointers. The leaf nodes of the index that point to a file obtain the $TIME$ and $CNUM$ of the file from the Scheduler, and set their pointers to $< CNUM, TIME >$.

In Figure 9, we show the final transmission schedule of the grid (in Figure 8) on the Index Band. In this figure, we assume that $ROOTCNUM = C1$. We have not shown any data pointers, and only a few index pointers in the diagram to retain clarity. The path followed by a PCS user that wishes to access (say) file I, is shown with a dark line. Notice that in our diagram, the pointers from an index packet that points to child packets are time varying. For instance, note the different pointers in the index packets transmitted on $< C1, T2 >$ versus the pointers in $< C1, T5 >$. This is because in our mapping, we always find the earliest occurence of a required child packet while filling index pointers as specified by Equation 4.

4 Scheme-Independent Pointer Following Algorithm

We now specify the algorithm[5] the PCS unit uses to reach its desired file. This algorithm is executed as soon as a desired file is chosen. Recall $ROOTCNUM$ to be the well-known root channel. We use two variables $NEXTTIME$ and $NEXTCNUM$ to keep track of when and from which channel should the next packet (data or index packet) be down loaded. Until then, the PCS unit will power down to its *doze* mode [IVB94] and

[4] If $T - T_s^i - j$ is a multiple of n_i, we merely add n_i to the final value.

[5] This algorithm can clearly be optimized by restructuring the **If** statements, but we present the algorithm in this form since it is easier to understand the flow.

consume minimal power. Let *doze(TIME)* make the PCS unit power down to minimal power until a timer wakes up the PCS unit after time *TIME* has elapsed.

1. *NEXTCNUM* = *ROOTCNUM*, *NEXTTIME* = 0
2. **Loop** forever
   (a) Download Index Packet from channel *NEXTCNUM*
   (b) **For** i = 1 **To** k
      i. **If** (KEY[i] = KEY) **and** (TYPE[i] = DATA) // Data file pointer
         - *NEXTCNUM = CNUM[i]*
         - *NEXTTIME = TIME[i]*
         - *doze*(NEXTTIME)
         - **GOTO** Step 3
      ii. **If** (KEY[i] $\geq$ KEY) **and** (TYPE[i] = INDEX) // Index packet pointer
         - *NEXTCNUM = CNUM[i]*
         - *NEXTTIME = TIME[i]*
         - *doze*(NEXTTIME)
         - **GOTO** Step 2.a
3. Download file at *NEXTCNUM, NEXTTIME*

5 Maintaining the Indexing Schemes

In the previous section, we showed how to compute index transmissions for data assuming that we are given popularity patterns of files. In this section, we first propose how to obtain popularity patterns of files. Also in any dynamic system, since the base station will need to adapt to changes in popularity patterns of files, we show how the indexing schemes can be updated on air.

5.1 Maintaining Popularity Information

One way of maintaining the expected popularity patterns of files, is to have *profiles* associated with PCS users that specify files the users are interested in. When a PCS user enters a zone or a *service-area* [JK94], he *registers* with the base station [JLLM94]. The base station can then access the PCS user's profile from some known database through the wireline network and update its local popularity information. This model may be useful in the case of Universal PCS units that will need to have registration and deregistration capabilities [JLLM94] to send and receive personal data.

An alternate approach that we explore in some of our experiments is to avoid having to maintain precise information by way of interaction with the PCS unit through the regular registration/deregistration process. We assume the existence of a *relative popularity ordering* specifying only which file is most popular, next most popular and so on rather than precise numbers of how many PCS users in a given zone are interested in the broadcasted files. Such information may be obtained through other sources (say through wireline information dissemination sources), and may be used to model local patterns. Such a scheme would function even for PCS units that have only receivers but no transmitters. Since transmitters comprise the largest area and consume the largest chunk of power and make PCS units large [Cox91], such a scheme that avoids using a transmitter may be useful.

5.2 Dynamically Updating Indices

In a dynamic system where the popularities of files may change, we need to periodically update the broadcasted index structure. The new index can be computed off-line at the base station, but will have to be introduced with care so that the correctness of the system is maintained. For instance, if a PCS unit is following pointers in the various index channels, we should ensure that the PCS unit follows the correct pointers belonging to the original index structure, and does not follow incorrect index pointers in the new structure and end up accessing the wrong file. This problem is similar to the *reader-writer* problem in Databases [GR92] since we should ensure that while a PCS unit that is waiting to read the next index pointer on air at a certain time slot and channel, the Indexer should not modify that part of the index. Normally, databases have *locks* associated with items so that a writer cannot write to a location that is currently being read. In our application, we cannot perform locking on air. Also, since the base station has no interaction with the PCS units, it is not aware which PCS units are at which levels in the index structure. Dynamic updating is not a problem for single-level index structures (as in the first three schemes) since the base station can merely lapse into the new index structure at any point in time, and the PCS unit will see no loss in continuity. Similarly, dynamic updating is not a problem when the index structure changes from a single-level to a multi-level index structure, or vice versa. We will now look into some possible schemes for dynamically evolving from one multi-level tree index structure to another.

5.2.1 Dual Index Bands

One approach would be to have two sets of index bands, with the base station alternating between the two index bands when it wants to transmit an index. That is, when an index transmission is going on in one index band, the base station starts using the second index band for the new index structure and keeps alternating between the two bands when it needs to change the index. The problem now is that PCS units will need to know which band to tune into. Also, this scheme will require twice the number of index channels which may be expensive.

5.2.2 Flushing Indices After Timeout Period

Another possibility that uses a single index band, would have the base station stop transmitting index packets in the root channel and continue index transmissions on the other channels when it chooses to change the index. In such a scenario, PCS units that are in the lower levels of the tree will eventually get access to their

Time

		T1	T2	T3	T4	T5	T6	T7		T82	T83	T84	T85	T86
	C1	0								10	10	10	10	10
	C2	1	2							11	12	11	12	11
Channels	C3	3	4	5						13	14	15	16	13
	C4	6								17	18	17	18	17
	C5	7								19	20	19	20	19

Figure 10: Modifying An Index Tree After Timeout Period.

desired file and will "exit" the index structure. Also, the index structure is effectively "locked" when the base station stops transmitting the root level index packets, and no new PCS unit can enter the lower levels of the index tree. The base station can start transmitting the root channel packets, and the lower levels of the new tree like before after some minimum time-out period which will guarantee that all PCS units have exited the index tree. If there are l number of levels in the index tree, the minimum time-out period required will be k^l since there can be at most k children per index packet (at each level). This simple scheme has the disadvantage that the average access time of the PCS units will increase, since the PCS units that are trying to enter the "locked" root channel will need to wait until the root channel transmits the new index structure. (We assume that the root channel will have packets indicating the *residual* time-out period so that PCS units can tune in again after the indicated time period for the new root level packets thereby not wasting energy.) In Figure 10, we show the case when the index tree in Figure 8 has to be flushed at T_1 to transmit some new index tree (with nodes numbered greater than 10). Since $k = 3$ and $l = 4$, the base station will need to wait for 81 (3^4) time units before it can be sure that all PCS units have exited the old index structure.

5.2.3 Flushing Indices Using Frontiers

We now present an optimization of the approach in Section 5.2.2 based on computing the *frontier* of each level, so that the new index transmissions may commence much earlier. The intuition is that the base station can compute the maximum amount of time a PCS unit can possibly spend in a given level (channel) assuming the worst-case scenario when a PCS unit is looking for the last index pointer on a given channel before the packets start to cycle again. For example in Figure 11, we show how the index transmission will evolve from one index tree to another using the frontier scheme (outlined in the next paragraph). Dark lines indicate the frontiers of the different channels.

After the base station decides to flush the current index tree and start transmitting a new index tree, the two principal questions that need to be answered are:

Time

		T1	T2	T3	T4	T5	T6	T7		T82	T83	T84	T85	T86
	C1	0	10	10	10	10	10	10						
	C2	1	2	11	12	11	12	11						
Channels	C3	3	4	5	13	14	15	16						
	C4	6	6	17	18	17	18	17						
	C5	7	7	19	20	19	20	19						

Figure 11: Modifying An Index Tree By Computing Frontiers.

1. When can the base station flush a given channel, and start transmitting index packets of the new tree?
2. How can pointers in the index packets be filled for the new index tree since the index packets in the different levels are offset depending on the size of each level frontier in the old index tree?

We first extend the pointer computation algorithm as follows. Consider channel i at time t. Each pointer emanating from a packet on this channel points to a packet on channel $i+1$ at time $t' > t$. Let F_i^t be the maximum of all such t'. We shall refer to F_i^t as the *frontier* of channel i at time t. We also define F_0^t to be 0 for all t. We now specify how the TIME pointers of parent index packets to children index packets should be initialized. Given a grid G which represents the index tree, the time T' at which the node in $G(i,j)$ is broadcast was specified by Equation 4. Now, let the time at which the new index is created be T_n. Define $a \ominus b$ to be the *monus* operator

$$a \ominus b = \begin{cases} 0 & \text{if } a < b \\ \text{a-b} & \text{otherwise} \end{cases}$$

Now we can rewrite Equation (4) as

$$T' = \lceil \frac{(T \ominus T_n^i) \ominus j}{n_i} \rceil * n_i + j + T_n^i \tag{5}$$

where T_n^i =max$(T_n, F_i^t), t = T_n$

We now specify the algorithm to be followed by the base station in flushing the various index channels once it decides to flush an index tree. The intuition is that no index packets of the old tree will be used once the frontier for each level is past. The algorithm is activated when $t = T_n$, and runs at each level of the tree until all channel frontiers have been crossed. Whenever a level of the tree crosses its frontier, we know that all preceding levels have already crossed their frontiers, therefore by keeping a count of the number of channels (starting from the root) that have crossed their frontier, we can determine exactly when the old index has been completely flushed from the broadcast.

Assume that the current channel is i, and the counter is *level-count*

1. T_n^i =max(T_n, F_i^t)
2. **If** $t < T_n^i$
   (a) Use Equation 4 to compute T'
   (b) $F_{i+1}^t = max(T', F_{i+1}^t)$
3. **Else If** $t = T_n^i$
   (a) $T_s^i = T_n^i$
   (b) *level-count*=*level-count*+1
   (c) **If** t <max(T_n, F_{i+1}^t)
      - Use Equation 5 to compute T'
   (d) **else** Use Equation 4 to compute T'

The above algorithm may be used by the base station to compute the index tree to grid mapping even when indices are not being updated by setting T_n to 0 and F_i^0 to 1 for all $i > 0$. Observe now that the conditional in Step 2 is always true, and Equation 4 will be used.

6 Experiments

The experiments reported in this section were designed to answer the following questions: (1) How do the different indexing schemes presented scale as we increase the number of files broadcasted; (2) Are the schemes "stable" as we vary the fanout of the data packets; (3) Do popularity patterns really help as opposed to assuming uniform distribution? (4) How do the schemes perform in the case of imprecise knowledge of access patterns?

We simulated a broadcast based file dissemination system with 1000 PCS units tuning for a set of files for a period of two hours. Each PCS unit is active 20% of the time and chooses its next desired file while active. Once a file is chosen, the PCS unit listens to the root channel and follows pointers that lead to its desired file. After the desired file is found, the PCS unit goes back to sleep for some time, before it becomes active again. In this simulation model, each PCS unit may access several files over its lifetime, with the restriction that only one file is tuned for by a PCS unit at any point in time.

In our initial experiments, we assume that the popularity distribution of files is Zipfian [Zip49] as suggested[6] in [Knu73]. If the files are relatively ranked in non-increasing order of popularity, then the base station expects the probability that a file f of rank r will be accessed by some PCS is

$$P(f) = \frac{1}{r * \sum_{v=1}^{N} 1/v}$$

[6] We also validate this assumption using some "real" data in the third experiment.

We also assume that the PCSs follow a memoryless Zipfian access pattern so that PCSs make their choice of the next required file independent of their previous choices. We ran each of the simulations 20 times and report the following graphs that are averaged over the simulations (each data point in the plots corresponds to the average of the corresponding point in 20 different executions of the simulation). In our graphs, we also report the results of *U-Alpha* for comparison, where U-Alpha is a Alphabetic Huffman tree that assumes a uniform distribution of access patterns. We report U-Alpha since it has logarithmic access properties as in traditional B+ Tree kinds of search structures, and helps us understand how useful popularity patterns are.

In the first experiment, we tested the scalability of the five indexing schemes as the number of files to be broadcasted increases. We fixed the fanout of the data to 20 (corresponding to 128 bytes/packet as in [IVB94]). In Figure 12, we present for each of the five different schemes the average number of index packets, while varying the number of files, that were examined by a PCS unit before it reaches its chosen required file. We see that as the number of files being broadcasted increases, the k-ary Alphabetic Huffman scheme and the U-Alpha indexing scheme dominate the other schemes by having the PCS unit tune into significantly lesser number of packets. This is an indication of the expected logarithmic performance of the Huffman and the U-Alpha schemes as opposed to the near linear trends in the other schemes. In Figure 13, we plot the average access time for the different schemes as the number of files increase. Here we see that our packing of the sparse Alphabetic Huffman tree to the time-channel space dominates the other schemes by having a significantly lower response time. The sudden drops in the plot for U-Alpha show the points where the number of levels in the tree change, while we do not have such spikes for the Alpha-Huffman plot since there are no marked transitions in the sparse and unbalanced tree. The downside of the Huffman based schemes are that they require more index channels than the other schemes as shown in Figure 14. We also plotted the cumulative distributions of the number of packets and the elapsed time (as opposed to the averages alone) and we again noted that our k-ary index structure and the subsequent 2-d mapping dominates the other schemes. We have not reported those plots since they do not convey any new information in addition to what can be inferred from the shown average plots.

The second experiment was designed to explore the stability of the indexing schemes as the size (fanout) of the data packet was varied. We varied the fanout for two different scenarios: when there are a small number of files, and when there are a large number of files. We used the same distribution model for the base station and the PCS units as in the first experiment. In Figures 15 and 16, we report the results of the average number of packets examined and the access time for the file pointer for the different schemes when the number of files was 50, as the fanout of the data packet was varied from 10 through 60. We see that the Alphabetic Huffman and the U-Alpha perform very poorly when the number of files is small. This is due to the extra overhead of the multiple levels of pointers in the tree structure. We however see in Figures 17 and 18 that the Alphabetic Huffman and U-Alpha dominate when the number of files is high (1000).

In the final experiment, we relaxed the assumption of the base-station having complete knowledge of the expected file access patterns. For this phase, we assumed that the files to be transmitted on the wireless medium were 1000 public files (such as home pages, gifs and papers) available on our research group's Web

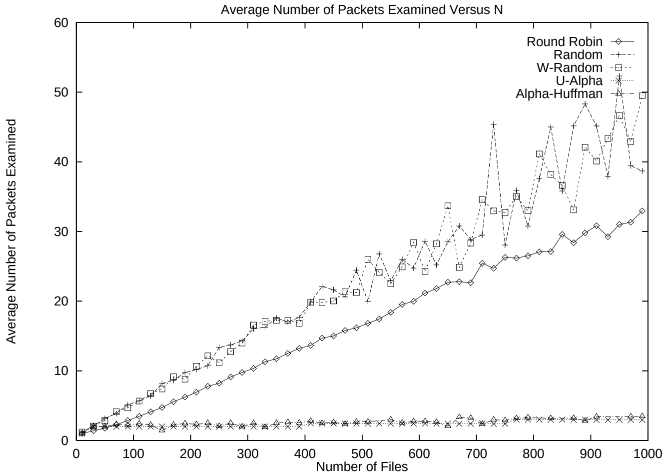

Figure 12: Average Number of Packets Examined with Change in Number of Files.

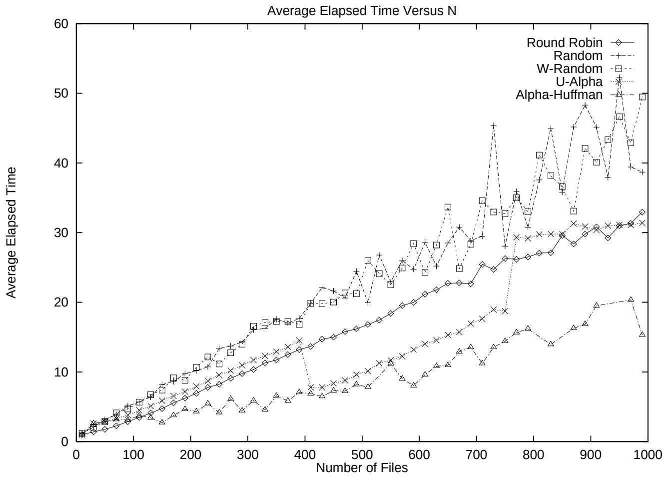

Figure 13: Average Time Elapsed with Change in Number of Files.

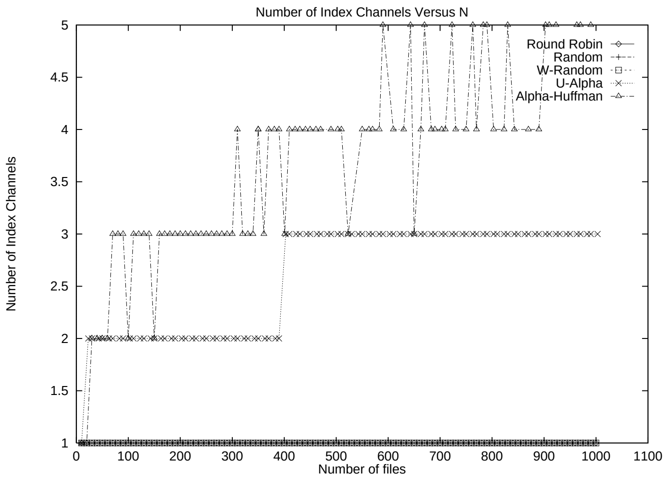

Figure 14: Number of Index Channels with Change in Number of Files.

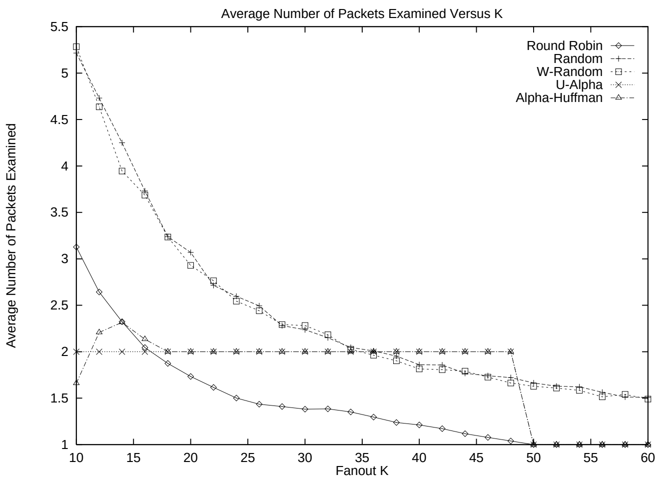

Figure 15: Average Number of Packets Examined with Change in Fanout for Small Number of Files.

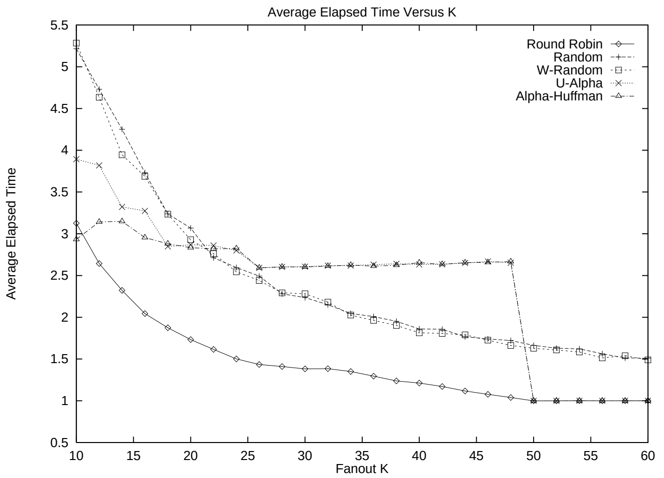

Figure 16: Average Time Elapsed with Change in Fanout for Small Number of Files.

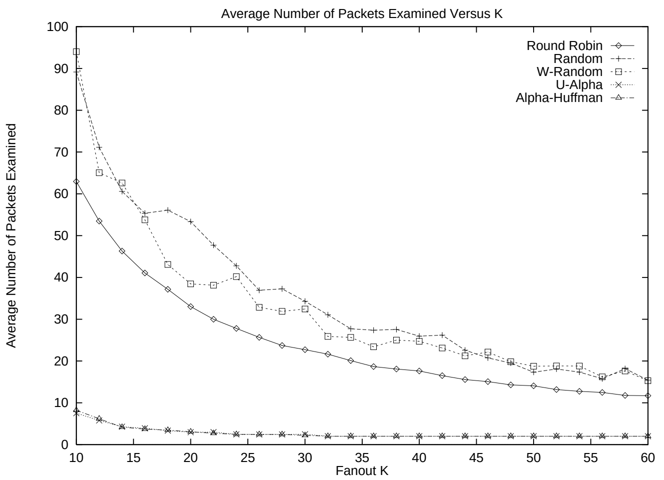

Figure 17: Average Number of Packets Examined with Change in Fanout for Large Number of Files.

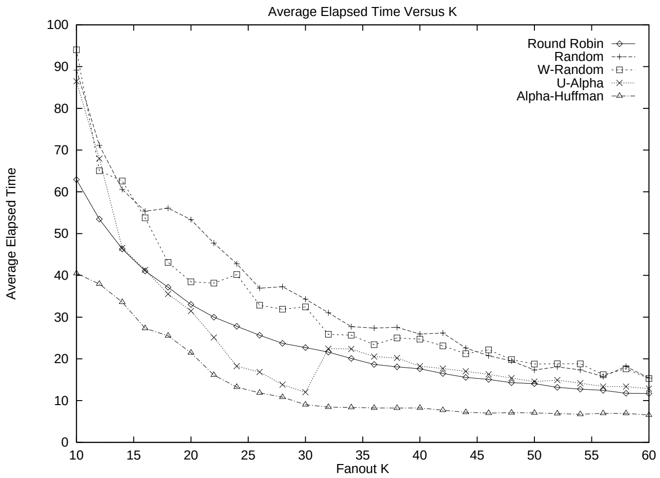

Figure 18: Average Time Elapsed with Change in Fanout for Large Number of Files.

Server [htt]. We extracted monthly log information of incoming Web page accesses to these files, and used them to generate the access patterns of the PCS users. As earlier, the PCS user is memoryless and tosses a coin to decide which file it wishes to tune to based on the extracted log information. On the other hand, we only gave the relative popularity ordering of the files to the base station. The indexer at the base station then generates its index structure assuming a Zipfian distribution of access patterns. This decoupling of file popularity patterns and user access patterns allows the base station to maintain minimal information in generating its index structure, and also (as discussed in Section 5.1) possibly nullifies the requirement of a transmitter on the PCS.

In Figures 19, 20 and 21, we report the results of the final experiment with the fanout of the data packet set to 20 (as earlier). In Figures 19 and 20, we only report results for the Round Robin, U-Alpha and the Alphabetic Huffman scheme since the other schemes performed very poorly. We also plotted *Opt Alpha*, which is Alphabetic Huffman with the actual user access patterns (as opposed to only the relative ordering of files) for comparison purposes. In Figure 19, we report the cumulative distribution of the number of packets examined before the pointer to the data file was obtained. We see that the Alphabetic Huffman, U-Alpha and Opt Alpha rise steeply and all data files are found within 5 packets (hops) or less, while (as expected) in the Round Robin a PCS unit may have to wait for an entire cycle of 50 packets (1000/20) before it finds its desired file pointer. In Figure 20, we report the cumulative distribution of the file pointer access time for the different schemes. Again we see that Alphabetic Huffman has the best trend in access time among the Round Robin and U-Alpha indexing schemes. It is encouraging to note that the Alphabetic Huffman scheme performs fairly closely to the Opt Alpha scheme in spite of the incomplete information available to the base station indicating that (as in Section 5.1) our index structure can tolerate inaccuracies in maintaining popularity patterns. In Figure 21, we report the average values for the number of packets

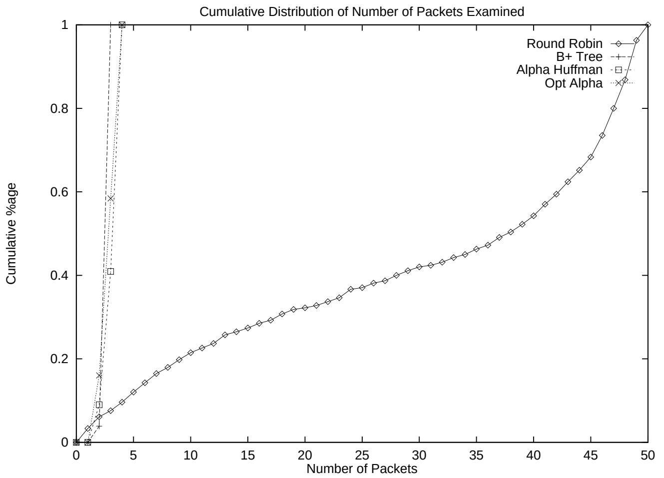

Figure 19: Cumulative Distribution of Number of Packets Examined.

examined, and the access time along with the number of index channels employed for the different indexing schemes. We can see the relative tradeoffs in the different schemes in terms of the access time, number of packets examined and the number of index channels required.

From the experiments, it is interesting to note that despite the lack of popularity patterns in U-Alpha, it performs consistently well in terms of the number of packets examined (equivalent to the height of the tree which is logarithmic). It does however suffer in terms of the access time compared to the regular Alphabetic Huffman tree with popularity patterns, since in the grid structure files at higher levels (more popular files) have less chance of getting bumped across multiple index channels, and hence do not have the time lag at each step waiting for the next index pointer. However, in U-Alpha since the leaves are all very close to the bottom of the tree, they all have to spend time at index channels corresponding to the higher levels waiting for the next index pointer to arrive in time. We expect that other logarithmic index structures (such as B+ Trees) that are balanced in terms of height will have similar problems for the outlined reason. Another counter-intuitive result is that Random and Windowed Randomized schemes do worse than a naive Round Robin scheme, since we would expect encoding popularity information into the transmission would help the PCS units. But as we see in Appendix B, the average tuning time is still linear, and it seems PCS units accessing unpopular files more than offset the gain in power obtained by scheduling popular files more often.

6.1 Dynamically Adjusting Index Transmission

From our observations based on our experimental results in the previous section, we propose the following modes of operation for the base station.

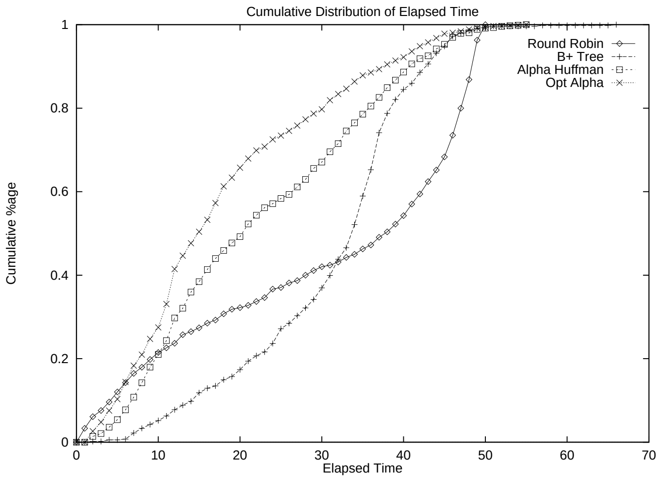

Figure 20: Cumulative Distribution of Time Elapsed.

Indexing Scheme	Num Packets	Normalized	Elapsed Time	Normalized	Num Channels
Round Robin	30.95	8.84	30.95	1.36	1
Random	44.65	12.76	44.65	1.95	1
W-Random	47.32	13.52	47.32	2.07	1
Alpha-Huffman	3.50	1.00	22.84	1.00	4
U-Alpha	2.96	0.85	31.22	1.37	3
Opt Alpha	3.27	0.93	18.62	0.82	4

Figure 21: Average Values for Minimal Knowledge Experiment.

1. **Alphabetic Huffman Mode:** If the number of files is large and popularity information is available (exact or at least relative ordering), and if there are sufficient number of index channels available (logarithmic in number of files), transmit the index using the Alphabetic Huffman scheme.

2. **U-Alpha Mode:** When the number of files is large and popularity information of files is not available, and if there are sufficient number of index channels available (logarithmic in number of files), the base station should transmit the index using the Alphabetic Huffman scheme assuming uniform distribution patterns for files.

3. **Round Robin Mode:** When the number of files to be broadcast is some small constant c ($c \leq 2$) within the fanout of the data packet, or if no popularity information is available, or if the number of available index channels is limited, the base station should operate in the Round Robin scheme. This mode can also be used if the keys of data files are unordered.

One approach would be to assign one of these three modes of operation to the base station at design time. In a more dynamic scheme, the base station could actually switch between these 3 modes of operation without the user knowing about it (as specified in Section 4) depending on the instantaneous availability of index channels, popularity patterns, and on the number of files to be broadcasted.

7 Conclusion

We considered the problem of organizing index information in a broadcast oriented information dissemination mechanism in wireless systems. We proposed several index organization techniques sensitive to file popularity patterns to save on battery power and minimize access time to files using Huffman based-encoding schemes and randomized techniques. We proposed an algorithm for the PCS to download its desired file independent of the indexing technique used by the base station. We showed how to dynamically update indices transmitted on air. We present experimental results of simulations employing the different indexing schemes, and observe that energy consumption by a PCS unit, and the access time for a file can be improved significantly (upto 13.5 times and 2.07 times respectively) by choosing the right index structure. Based on the results, we propose an adaptive index transmission mechanism for the base station to dynamically adjust to changes in the number of files, available number of index channels and the precision of popularity information.

8 Acknowledgements

We wish to thank Prof. H. Garcia-Molina, Dr. N. Krishnakumar and Prof. J. Widom for helpful discussions and comments.

References

[AAFZ95] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast disks: Data management for asymmetric communication environments. In *Proceedings of 1995 ACM SIGMOD*, volume 24, pages 199 – 210, May 1995.

[Che92] D.R. Cheriton. Dissemination-oriented communication systems. In *Stanford University, Technical Report*, 1992.

[Chi94] Tzicker Chiueh. Scheduling for broadcast-based file systems. In *NSF MOBIDATA workshop, Rutgers University*, 1994.

[Cox91] D.C. Cox. Digital portable radio communications. In *Bellcore Special Report*, volume SR-ARH-001023, June 1991.

[GR92] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Francisco, California, 1992.

[HT71] T. C. Hu and A.C. Tucker. Optimal computer search trees and variable-length alphabetic codes. *SIAM J. Appl. Math.*, 21(4):514–532, 1971.

[htt] http://db.stanford.edu.

[Huf52] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. IRE.*, 40(9):1098–1101, 1952.

[IVB94] T. Imielinski, S. Vishwanathan, and B.R. Badrinath. Energy efficient indexing on air. In *Proceedings of 1994 ACM SIGMOD*, pages 25–36, May 1994.

[JK94] R. Jain and N. Krishnakumar. Network support for personal information services to PCS users. In *Proceedings of the IEEE Conf. on Networks for Pers. Comm. (NPC)*, March 1994.

[JLLM94] R. Jain, Y. Lin, C. Lo, and S. Mohan. A caching strategy to reduce network impacts of PCS. *IEEE Journal on Selected Areas in Communications*, 12(8), October 1994.

[Knu73] D. E. Knuth. *The Art of Computer Programming, Vol. 3.* Addison-Wesley, Reading, Massachusetts, 1973.

[Knu85] D. E. Knuth. Dynamic huffman encoding. *J. Algorithms*, 6(2):163–180, 1985.

[LH87] D.A. Lelewer and D.S. Hirschberg. Data compression. *ACM Computing Surveys*, 19(3):261–295, 1987.

[Li92] W. Li. Random texts exhibit Zipf's law-like word frequency distribution. *IEEE Trans. Inf. Theory*, 38(6):1842, 1992.

[Mit70] D. S. Mitrinović. *Analytic Inequalities.* Springer-Verlag, New York, 1970.

[SCB92] S. Sheng, A. Chandrakasan, and R.W. Broderson. A portable multimedia terminal for personal communications. In *IEEE Communications Magazine*, pages 64 – 75, 1992.

[VI95] S. Vishwanathan and T. Imielinski. Pyramid broadcasting for video on demand service. In *IEEE Multimedia Computing and Networks Conference*, February 1995.

[Zip49] G. K. Zipf. *Human Behaviour and the Principle of Least Effort.* Addison-Wesley Press, Cambridge, Massachusetts, 1949.

A Random broadcast yields $O(n)$ average access time

We consider the random broadcast scheme, where index packets are transmitted on the channel with probabilities proportional to the access frequencies of the respective data channels, and show that the average tuning time is $O(n)$, where n is the number of data files that are available.

Let the frequency distribution on the files be $F = f_1, f_2, \ldots, f_n$. An index packet for file i will be broadcast with probability

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j} \tag{6}$$

Let T_i be the time at which an index packet for file i is encountered. We wish to determine the average value of T_i, given by $\bar{T_i}$.

$$\begin{aligned} \bar{T_i} &= 1 * p_i + 2 * (1 - p_i) * p_i + \ldots \\ &= \sum_{i=1}^{\infty} i * p_i * (1 - p_i)^{i-1} \end{aligned}$$

This yields the solution

$$\bar{T_i} = \frac{1}{p_i} \tag{7}$$

The average access time $\bar{T}$ (over all users), is given by

$$\begin{aligned} \bar{T} &= \sum_{i=1}^{n} p'_i * \bar{T_i} \\ &= \sum_{i=1}^{n} \frac{p'_i}{p_i} \end{aligned}$$

where p'_i is the actual user access probability for file i. Note that in general, we would like to have $p_i = p'_i, \forall i$. In this case, we obtain

$$\begin{aligned} \bar{T} &= \sum_{i=1}^{n} 1 \\ &= n \end{aligned}$$

If we do not have perfect knowledge of user access patterns, we can approximate the p_i's by a uniform distribution, where $p_i = \frac{1}{n}, \forall i$. In this case, we obtain, as before,

$$\begin{aligned} \bar{T} &= n * \sum_{i=1}^{n} p'_i \\ &= n \end{aligned}$$

B Achieving a $\log n$ depth

Zipf's frequency distribution scheme [Zip49],[Li92] uses *ranks* of the n data points to estimate probabilities of their occurence. If element i is assigned a rank $r(i), 1 \leq r(i) \leq n$, its probability of occurence is

$$p_i = \frac{1}{r(i) * \sum_{j=1}^{n} \frac{1}{j}}$$

which is the same as

$$p_i = \frac{1}{r(i) * H_n} \tag{8}$$

where $H_n = \sum_{j=1}^{n} \frac{1}{j}$

Since $H_n = \log_e n + O(1)$ [Mit70], we can rewrite p_i as

$$p_i \approx \frac{1}{r(i) * \log_e n}$$

Consider the lowest such p_i. This i has rank $r(i) = n$, and so

$$p_i = \frac{1}{n * \log_e n}$$

In a Huffman tree, the depth of an element is $O(-\log p_i)$, where p_i is the probability of the element. Therefore, since the elements of lowest frequency are at the bottom of the tree, and we have a k-ary tree, the depth of the Huffman tree constructed using this frequency distribution is

$$D = O(-\log_k \frac{1}{n * \log_e n})$$

which is

$$D = O(\log_k n)$$

C K-Ary Construction of Alphabetic Huffman Trees

C.1 The Hu-Tucker Algorithm

We are given an ordered list of index entries where the ith entry contains a pointer d_i to a data file, and a frequency count f_i. This forms the input to the Hu-Tucker algorithm which proceeds in two stages.

Stage 1: This stage operates in several passes. The input to each pass is a *construction sequence* which is an ordered list of nodes, each of which can be a leaf (one of the index entries) or a subtree whose leaves are index entries. The frequency count for a subtree is the sum of the frequencies of all the index entries present in it. At each pass, two of the nodes (not necessarily consecutive) are coalesced and replaced by a single node. This process repeats till there is only one node (corresponding to a tree on ALL the index entries).

The initial construction sequence is the original list of entries. At each pass, choose nodes i, j as candidates to be merged if the following conditions hold:

1. There are no leaves between i and j.
2. $f_i + f_j$ is minimum over all such pairs i, j if there exist several pairs.
3. In case of a tie in (2), i is the leftmost node among all such pairs.
4. In case of a tie in (3), j is the leftmost node among all such pairs.

If these four conditions are satisfied, create a new node i' with $f_{i'}$ equal to $f_i + f_j$, and make nodes i and j its children. Insert i' into the construction sequence at node i's previous position, and delete nodes i, j from the sequence. This construction produces a tree structure, but the left-to-right ordering of the nodes is not yet preserved.

Stage 2: The next stage of the algorithm produces the required ordering of the leaves. This is achieved as follows. Determine the level of each leaf in the tree (by doing a tree traversal). Then, at each level of the tree, starting at the lowest, start from the left most leaf and rearrange parent pointers such that the two leftmost leaves have the same parent, and so on. Once this is done for all levels, we have an alphabetic search tree.

It is not necessary that the rearrangement in the second stage of the construction may even be possible. Using the conditions for combining nodes as described above, [HT71] shows that it is always possible to do this rearrangement. It should also be noted that this rearrangement does not change the cost of the tree, as the level of a node does not change.

C.2 Our k-ary modification

In our modification, we allow at most k nodes to be combined into a single super-node during the passes in stage 1, instead of two nodes in [HT71]. We also allow combining k leaf nodes if they are consecutive in the construction sequence, while the other conditions remain the same. The second stage remains the same except that we allow upto k nodes to be coalesced to have the same parent. Since the conditions on combining nodes in the first step are modified minimally, we can still perform the reordering phase similar to that in [HT71].

One interesting feature in our extension for k-ary trees is that we may end up with a tree with smaller depth than the one we started with, unlike in the binary tree construction in [HT71]. It is important to note that in our construction, it may not be always possible (or optimal) to combine k nodes together. Therefore our k-ary Alphabetic Huffman tree will have a fanout that varies between 2 and k.