Efficient Data Allocation For Broadcast Disk Arrays

Wai Gen Yee
{waigen@cc.gatech.edu}

September 6, 2001

Abstract

In this paper, we study the problem of data allocation in a broadcast environment. We divide this work up into two parts. First, we tackle the task of allocating data to a given number of physical broadcast channels in order to minimize the average wait for a data item. We compare our methods to two popular alternatives called $FLAT$ and VF^K. Our methods turn out to result in better performance and may be cheaper to compute as well. Second, we apply this allocation work to the optimal design of broadcast disks–logical partitions of a set of data items which result in a skewed broadcast schedule. In this phase, we simultaneously discover a good allocation of data to the disks and the optimal number of disks to use. Experimental results show that our techniques result in near minimal average wait for a data item.

1 Introduction

Today, portable devices allow people to carry data and computation power to remote locations. People want to keep their data current, even when out of the office, and therefore use wireless means to maintain a connection to a server. However, as the user base grows, the server must be able to scale its performance in terms of the average expected delay for a data request.

Broadcast has long been recognized as a scalable way for servers to transmit their data to an unlimited number of clients [4, 8]. By cycling their data, the delay for a requested data item is bound by the size or period of the cycle. Canonical examples of the types of data that were originally conceived for this type of data dissemination architecture include stock quote [14] and telephone directory information [6], although more general use is possible. However, the period of each cycle may be long, and the popularity of each data item may be skewed. These factors increase the average expected delay by allocating as much bandwidth to less popular data items as to more popular ones.

Popularity variation is counteracted by skewing the bandwidth allocation so that more popular data items are broadcast more frequently. We consider two ways of doing this. First, if there are multiple physical broadcast channels available, we devise schemes for allocating data to be cycled on these channels. This is a solution when multiple low bandwidth channels are multiplexed from a single channel. Clients can then choose to listen to any subset of channels.

If only one broadcast channel is available, we consider using broadcast disks to generate a broadcast schedule. Broadcast disks are an abstraction and serve as a way of organizing data into a popularity *hierarchy* which results in a skewed transmission of data and quicker access to more popular data. Each level of this hierarchy is given equal amounts of bandwidth, so data items in smaller levels are cycled more frequently onto the broadcast channel–i.e., the abstract disks corresponding to the smaller levels "spin" faster [1]. A broadcast disk array is thus a set of abstract disks with varying recycling speeds.

The contributions of this paper include techniques to:

- allocate data to a given number of physical broadcast channels,
- generate single channel broadcast schedules using broadcast disks.

Organization of the paper We first state our architectural assumptions and formalize the multiple channel problem in Section 2.1. Work related to the topics covered in this paper is discussed in Section 3. Then, in Section 4, we discuss other methods of allocating data to multiple channels, and cite their limitations. In response, we offer our own methods in Section 5. In this section, we theoretically justify our approach for generating an optimal solution, then offer a cheaper greedy method. We compare the performance of our methods with those of others in Section 7. The single channel case is considered in Section 9. Here, we show how work from Section 5 is applied to find an optimal broadcast disk design. We wrap up the paper and discuss future work in Section 11.

2 Multi-Channel Case

Multiple physical channels have capabilities and applications that cannot be mapped onto single channels. As stated in [17], some example advantages include better fault tolerance, configurability, and scalability.

By having access to multiple physical channels, fault tolerance is improved. For example, if a server broadcasting on a certain frequency crashes, its work must be migrated to another server. If this server is already broadcasting on another frequency, it can only accept the additional work if it has the ability to access multiple channels.

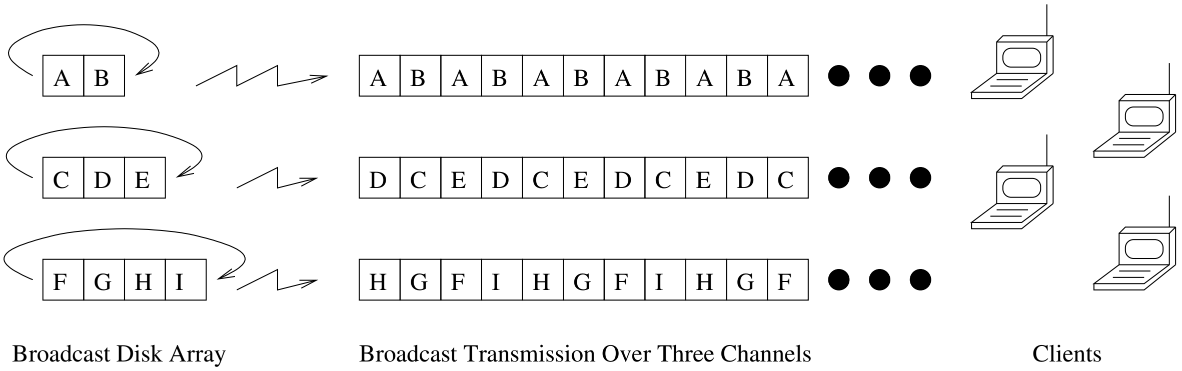

Figure 1: Architecture. An array of three broadcast disks ($K = 3$), each containing a variable number of data items ($N_1 = 2, N_2 = 3, N_3 = 4$), broadcasting its contents. Each "square" represents a unit-sized data item. The right side of the diagram shows the $K = 3$ data items broadcast at each tick.

More flexibility is allowed in configuring broadcast servers. Assume that there are two contiguous cells which contain broadcast servers that transmit at different channels. A single server that wishes to take over the responsibility of transmitting in both cells can only do so if it can transmit over multiple channels.

Finally, being able to transmit over multiple channels has scalability benefits. A broadcasting system must be able to handle both high powered and low powered clients. In order to do so, multiple channels can be used, and clients can monitor a number of channels commensurate to their capacities and data needs.

2.1 Architectural Assumptions

We assume that there are K channels in a broadcast area, denoted C_i, $1 \leq i \leq K$. A database is made up of N unit-sized data items, denoted d_j, $1 \leq j \leq N$. Each data item is broadcast on one of these channels, so channel i broadcasts N_i data items, $1 \leq i \leq K$, $\sum_{i=1}^{K} N_i = N$. Each channel cyclically broadcasts its data items. Time is slotted into units called ticks. A tick is defined as the amount of time necessary to transmit a unit-sized data item. Each data item d_i is assigned an access probability, p_i. Requests are assumed to be exponentially distributed, so p_i does not vary from tick to tick. See Figure 1.

Clients listen to any or all of these channels depending on their data needs. Clients can begin listening at any time, not necessarily at the beginning of a tick.

2.2 Problem Statement

Expected delay, w_i, is the number of ticks a client can expect to wait for the broadcast of a request for data item i. Average expected delay is the average number of ticks a client must wait for a typical request. This is computed as the expected delay of a data item times the probability that that data item is requested during a tick, summed over all data items:

$$\sum_{i=1}^{N} w_i p_i \tag{1}$$

When N_i data items are cyclically broadcast on channel i, the expected delay in receiving any particular data item j is $w_j = \frac{N_i}{2}$. Given K channels, the average expected delay is therefore:

$$\frac{1}{2}\sum_{i=1}^{K}\left(N_i \sum_{d_j \in C_i} p_j\right) \tag{2}$$

Our goal is to allocate the data items to each of the K channels in a way that minimizes the cost denoted by Equation 2. This is an optimization problem with a built-in tradeoff. More popular data items should be placed on channels with fewer data items, so that they are broadcast more frequently. However, allocating more data items to a channel reduces the frequency at which each is broadcast.

In the following example we show the effects of two general design strategies. One is referred to as *flat* design. The flat design allocates an even number of data items to each channel, regardless of popularity. The other is referred to as *hierarchical* design. In this design, the data items are divided into two levels of a popularity hierarchy, with the more popular data items allocated to the smaller level. This results in the more popular data being cycled more frequently. We defer discussion of particular hierarchical design strategies until later in this paper.

Example 1.1 Consider a set of $N = 6$ data items, $\{A, B, C, D, E, F\}$, with the following skewed access probabilities:

p_A	p_B	p_C	p_D	p_E	p_F
.37	.2	.15	.11	.09	.08

Given $K = 2$ broadcast channels, consider the two data allocation schemes shown in Figure 2. The flat design (top of Figure 2) allocates an equal number of data items to each channel. The hierarchical design (bottom of Figure 2) allocates fewer data items to channels containing the

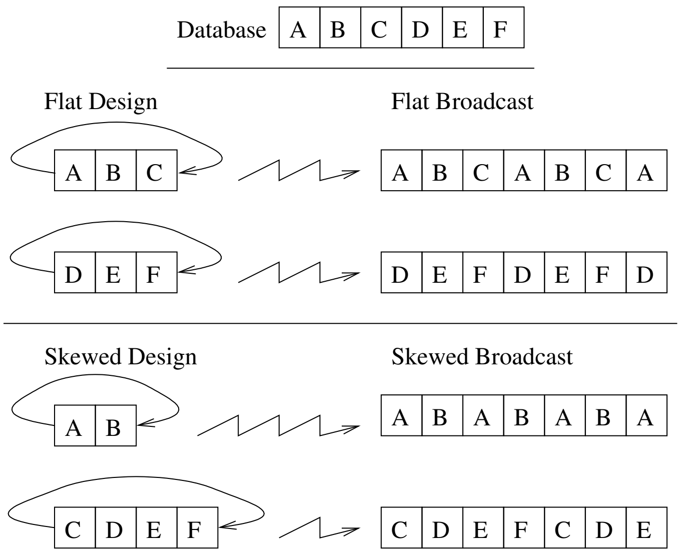

Figure 2: Flat versus Hierarchical Allocation of Data

most popular ones. We see that by skewing the broadcast schedule, the average expected dealy is lower.

flat design	$\frac{N_i}{2}\sum_{d_j \in C_i} p_j$
channel 1 ($i=1$)	(1.5)(.72)=1.08
channel 2 ($i=2$)	(1.5)(.28)=0.42
average expected delay	1.5

hierarchical design	$\frac{N_i}{2}\sum_{d_j \in C_i} p_j$
channel 1 ($i=1$)	(1)(.57)=.57
channel 2 ($i=2$)	(2)(.43)=.86
average expected delay	1.43

□

3 Related Work

In *push*-based broadcasting, a broadcast server schedules the transmission of its contents based on statistical assumptions of client access to data. It has long been recognized as a way of simultaneously reducing server load and catering to an unbound number of clients with a bound delay [4, 8]. Since the server does not handle individual requests, its performance is independent of the number of clients. Since schedules are heuristically designed, worst-case delay can typically be estimated for each data request.

Pull-based or on-demand broadcast is a different paradigm, and involves server interaction with actual client requests in order to generate a broadcast program. Pull-based techniques, such as $R \times W$ [3] are good at adapting to changing user interests. They also have finite, but unpredictable worst-case delay. Furthermore, their dynamic nature makes things such as optimization of tuning time (mentioned below) and client-side cache management very difficult [1].

Recent techniques of pushing data over a single broadcast channel have been developed which achieve near-optimal expected delay for data requests [19, 20]. These techniques and ours both try to minimize the same cost, but in different ways. The single channel techniques are *bottom-up*, because the transmission priorities of all data items are computed and considered at each tick in order to synthesize a schedule. Our techniques are *top-down*. The entire set of data items is mapped into a fixed number of channels, and data from each channel is transmitted independently. Furthermore, while the former may perform better for the single channel cost, little work has been done on performance with multiple channels.

Work has also been done on hybrid push/pull techniques given multiple channels. In [13], channels are allocated between push and pull-based data dissemination depending on the current data usage within a cell. [13] assumes that data items are flatly transmitted on the push channels, without regard to their relative popularities. It does not deal with other kinds of distribution of data over these channels.

Our work, on the other hand, focuses on the allocation of data to multiple push channels, which is not discussed in [1], nor addressed in the work mentioned above. Recently, [15] addresses this problem, and offers the VF^K algorithm which gives near-optimal results over a limited range of parameters in terms of expected delay, but, as we will show, this algorithm is expensive, and may break down in certain conditions. In this paper, we show how to partition data to achieve optimal average delay (in [15], optimal results are generated via computationally expensive linear programming and branch-and-bound techniques) and offer a good approximation algorithm that is both generated more quickly than the optimal solution, and results in better performance than VF^K.

As a side note, other work on broadcasting has focused on semantic issues of the data. For instance, consistency control is discussed in [18]. Other work focuses on tuning time [10], which is the time that a client must actively listen to a broadcast channel. Although these works are not within the scope of this paper, the techniques that we describe are immediately applicable to these other works.

4 Previous Methods For Broadcast Disk Design

In this section, we describe two other methods for the allocation partitions of data over various channels, and mention another way to look at the problem of allocating data over channels.

The first allocation method [13] is $FLAT$ where each of the K channels is allocated $\frac{N}{K}$ unique data items. $FLAT$'s greatest feature is its simplicity. Such an allocation is easy to design. In the case of changes in system configuration, e.g., changing the number of channels, $FLAT$ can be easily reconfigured. Another feature is that each channel has the same period, $\frac{N}{K}$. This simplifies consistency control and tuning time optimization, because each data item in a disk is transmitted exactly once in $\frac{N}{K}$ ticks [9].

However, $FLAT$ has performance problems, because it allocates the same amount of bandwidth to popular and unpopular data items. Each time an unpopular data item is transmitted over a shared channel, clients must wait longer for more popular ones. As shown in Example 1.1, this hurts performance when there is a skew in data popularity.

The VF^K algorithm [15] allocates data hierarchically. It greedily partitions a set of data items, ordered by popularity, where the decrease in average expected delay is greatest. It then heuristically picks the partition with the greatest delay, and repeats the partitioning step until K partitions are generated.

Assume that data items are arranged in a row, where the data items to the left have higher popularities than those to the right. In this case, splitting the row at some point creates new *left* and *right* partitions. VF^K restricts its search space by requiring that, during each split, the new right partition contains more data items than the new left one. Data items from each partition are then assigned to a broadcast disk. This algorithm is a bit more complex than $FLAT$, and as implemented in [15], has $O(K(K+N^2))$ complexity. However, as we shall see, the performance benefits in terms of average expected delay over $FLAT$ are significant for most cases.

In [19, 20], broadcasting data over multiple channels is briefly mentioned, but they address a different problem. They assume that each channel broadcasts the same set of data items in the same sequence. For example, in [20], the difference among the channels is that the sequence of data items in each channel is offset differently. Channel 1, with no offset, may be broadcasting data item j, while channel 2, with offset o, is broadcasting data item $(j+o) \bmod N$. The amount of offset is determined by the nature of user subscription to each channel. This work only covers the cases in which there are 1, 2 or 3 channels available.

5 Optimal Data Allocation: A Dynamic Programming Approach

In this section, we present a technique for allocating data to channels which minimizes the average expected delay of a data item. We start by pointing out important properties of the problem, and explain how they can be used to generate an optimal solution. To do this, we borrow from techniques described in [7, 12].

We are given a set of N data items. Our goal is to find ways of partitioning these data items into K disjoint subsets which minimizes Equation 2. There are two important properties of this problem. First, this problem has an optimal substructure. An optimal solution is made up of smaller optimal solutions.

Theorem 5.1: An optimal partitioning of the unit-sized data items of the channel allocation problem has optimal substructure.

Proof: Assume that in the optimal solution, there is a partition between data items i and $i+1$, $1 \leq i < N$. Then, data items 1 to i must also be optimally partitioned. If not, we can repartition these data items in order to get a better solution for data items 1 through N, contradicting our optimality assumption. The same argument applies for the partitioning of data items $i+1$ to N.$\square$

Second, this problem has overlapping subproblems. Each intermediate solution is composed of a finite number of solutions to smaller problems. Therefore, in the search for an optimal solution, a finite number of precomputed solutions to subproblems can be referenced to construct larger solutions, reducing computation time.

In our problem of partitioning data items, the values that are computed and reused are the costs of candidate partitions. In general, there are an exponential ($> 2^N$) number of possible partitions. However, as we will show, the structure of an optimal solution can be highly constrained.

Our claim is that there exists an optimal solution that partially orders the data items in terms of popularity. In other words, in the set of optimal solutions, there exists one made of partitions that can be ordered $C_1, C_2, ..., C_K$ so that all the data items in C_i are at least as popular as those in C_j, where $j > i$.

Theorem 5.2: In an optimal solution, for any two partitions, C_i, C_j, if $|C_i| < |C_j|$, then $\forall d_k \in C_i, d_l \in C_j$, $p_k \geq p_l$.

Proof: Assume that we have an optimal solution consisting of $C_1, ..., C_K$. Pick any two partitions C_i, C_j where $|C_i| < |C_j|$. Assume that $\exists(d_k \in C_i, d_l \in C_j)$ where $p_k < p_l$. The contributions of C_i and C_j to the total cost are:

$$\frac{1}{2}[|C_i|(\sum_{d_a \in C_i, a \neq k} p_a + p_k) + |C_j|(\sum_{d_a \in C_j, a \neq l} p_a + p_l)]$$

Now, assume that we exchange d_k with d_l in the two partitions. The change in cost is:

$$\begin{aligned} & \frac{1}{2}[|C_i|(p_l - p_k) + |C_j|(p_k - p_l)] \\ = & \frac{1}{2}[(|C_i| - |C_j|)(p_l - p_k)] \\ < & 0 \end{aligned}$$

The exchange results in a better solution, contradicting the optimality assumption.□

Corollary 5.1: In an optimal solution, for any two partitions C_i, C_j, if $\exists d_k \in C_i, d_l \in C_j$ where $p_k > p_l$, then $|C_i| \leq |C_j|$.

Proof: The proof is similar to the one for Theorem 5.2.

Because of Theorem 5.2, we can conclude that there exist optimal solutions to the partitioning problem which order partitions by size, or, equivalently, by popularity of their data items. Therefore, the only partitions that need to be considered are those which contain data items with popularities which are "contiguous." In other words, if a partition C_m contains d_i and data items as popular as or less popular than d_i, then C_m must contain d_j, where $p_j \leq p_i$ and $\neg\exists d_k$, where $p_j < p_k \leq p_i$. Similar containment rules exist if C_m contains data items as or more popular than d_i. Therefore, a good starting point for algorithms which exhaustively search for an optimal solution, including ours, is a set of data items ordered by popularity. This justifies the overlapping subproblem property we claim above.

Note that the ordering result of Theorem 5.2 distinguishes our work from similar work done previously on partitioning. The histograms related work [12] assumes data to be naturally ordered based on semantics, such as date or age. In our work, however, no such semantics exist, as data items are solely defined by their popularities. In [15], VF^K uses a greedy approach which assumes that data items are ordered based on popularity, but fails to justify this assumption.

We can now construct a dynamic programming solution to our partitioning problem. Define C_{ij} as the average expected delay of a channel containing data items i through j. Just like Equation 2, C_{ij} is computed as the expected delay of a data item in a channel of size $j-i+1$ times the total probability that an item in that channel is accessed:

$$C_{ij} = \frac{j-i+1}{2} \sum_{q=i}^{j} p_q \tag{3}$$

$opt\_sol_{i,j}$ is the optimal solution (i.e., minimum delay) for allocating data items from i through N on j channels. Trivially, the optimal solution for data items i through N given one disk is $opt\_sol_{i,1} = C_{i,N}$. We can now express the optimal average expected delay with the following recurrence:

$$opt\_sol_{i,K} = \min_{l \in \{i,i+1,\ldots,N\}} C_{il} + opt\_sol_{l,K-1} \tag{4}$$

Using dynamic programming to solve this problem requires $O(KN^2)$ complexity and $O(KN)$ space to keep track of partial solutions. We refer to this algorithm as $OPTIMAL$.

5.1 A Cheaper Approximation

Although the two hierarchical algorithms described above yield optimal ($OPTIMAL$) and near-optimal (VF^K) solutions, their respective complexities may make them unusable with large databases. In this section, we propose a greedy algorithm that allocates data to channels more cheaply, and describe ways to implement it.

One way to reduce costs is to precompute the N partial sums $(b_1, b_2, \ldots, b_N)$ in the access probability distribution for the data items where:

$$b_i = p_i + b_{i-1}, \quad b_1 = p_1, 1 \le i \le N \tag{5}$$

Then, since we can transform the right term of C_{ij} ($\sum_{q=i}^{j} p_q$ from Equation 3) to $b_j - b_{i-1}$, we can compute C_{ij} in constant time.

$$C_{ij} = \begin{cases} \frac{1}{2}(j-i+1)*(b_j - b_{i-1}) & i > 1 \\ \frac{1}{2}(j-i+1)*b_j & i = 1 \end{cases} \tag{6}$$

A simplified form of our algorithm is called $GREEDY-SPLIT-SIMPLE$ and runs in $O(NK)$ time. It first finds the location at which splitting the data items causes the greatest cost decrease. With the resultant partitions, it then finds the next location at which to split the data items, and so on, until there are K partitions:

Algorithm 5.1

```
GREEDY-SPLIT-SIMPLE
input: set of $N$ unit sized data items ordered by popularity, $K$ partitions
begin
  $numPartitions := 1$;
  while $numPartitions < K$
          do for each partition $k$ with data items $i$ through $j$
                  do comment: Find the best point to split in partition $k$
                  for $s := i$ to $j$
                      comment: Initialize the best split point for this partition as the first data item.
                               If we find a better one subsequently, update the best split point.
                      do if $s := i \vee localChange > C_{ij}^s$
                            do
                               $localS = s$; $localChange = C_{ij}^s$;
                            od fi od
                  comment: Initialize the best solution as the one for the first partition.
                           If we find a better one subsequently, update the best solution.
                  if $k := 1 \vee globalChange > localChange$
                    do
                       $globalChange := localChange$; $globalS := localS$; $bestpart := k$;
                    od fi od
          split partition $bestpart$ at point $globalP$
          $numPartitions := numPartitions + 1$;
  od
end
```

The change in cost in partitioning a set of data items from i to j at point s (the partition or split point), C_{ij}^s is computed in constant time by the following equation:

$$C_{ij}^s = C_{is} + C_{s+1,j} - C_{ij}, \quad i \le s < j \tag{7}$$

Example 5.1 Consider the set of $N = 6$ data items from Example 1.1. Our goal is to allocate them to $K = 3$ physical channels in a way that minimizes average expected delay. We are given the data items, sorted by popularity. The first split occurs between data items 2 and 3. This is the point at which $C^s_{ij}, i = 1, j = 6, s = 2$ is minimized. The following split occurs between data items 4 and 5 for the same reason. Because of these two splits, the average expected delay is reduced from 3 ticks to 1 tick. See Figure 3.

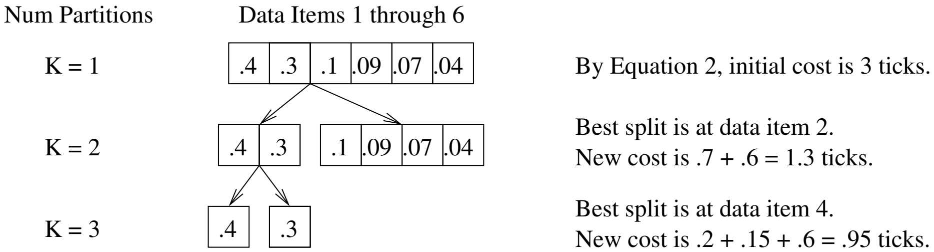
Num Partitions
Data Items 1 through 6
K = 1
.4
.3
.1
.09
.07
.04
By Equation 2, initial cost is 3 ticks.
K = 2
.4
.3
.1
.09
.07
.04
Best split is at data item 2.
New cost is .7 + .6 = 1.3 ticks.
K = 3
.4
.3
Best split is at data item 4.
New cost is .2 + .15 + .6 = .95 ticks.

Figure 3: Example of application of $GREEDY-SPLIT$, to generate $K = 3$ partitions. The numbers in the boxes are the popularities of the data items. The data items are numbered 1 to 6 from left to right.

□

5.1.1 Modification of the Greedy Algorithm

We show $GREEDY - SPLIT - SIMPLE$ above for illustrative purposes. We implement a slightly modified version of $GREEDY - SPLIT - SIMPLE$, called $GREEDY - SPLIT$. In each iteration of $GREEDY - SPLIT$, we save the best split points of the partitions that are not split for future iterations.

Theorem 5.1: The complexity of $GREEDY - SPLIT$ is $O(N \log N)$.

Proof: In each iteration i of $GREEDY-SPLIT$ $(1 \leq i \leq K-1)$, we perform $O(\frac{N}{i})$ computations to find the best split point of last two partitions generated, and perform $O(\log i)$ computations to look up the best split. For K partitions, we therefore must perform on the order of $N \sum_{i=1}^{K} \frac{1}{i} + \sum_{i=1}^{K} \log i$ computations. The first term in this expression can be expressed as NH_K, where $H_K = \ln K + O(1)$ is the K^{th} harmonic number. The second term is bound by $\log K! < K \log K$ [7]. The total complexity of $GREEDY - SPLIT$ is therefore $O((N+K)\log K) = O(N \log N)$ since $KleqN$.

In our implementation, we save the costs changes of the best splits of the partitions that are not split in each iteration. In this way, we save the recomputation costs. We present our algorithm as shown above for the sake of simplicity.

5.2 Adding and Removing Partitions

From time to time, it may be necessary to add and delete channels. For example, in [13], channels are dynamically (de)allocated to broadcast transmission depending on data usage and workload. With $OPTIMAL$, adding and deleting channels requires maintaining the table of partial results to the inital solutions. Adding a channel is done by continuing the dynamic programming process for channel $K+1$, which amounts to computing another column in the table of partial results in $O(N^2)$ time. Deleting a channel is much quicker, and requires a lookup of the table in $O(N)$ time.

With VF^K, adding a channel is done in $O(K+N^2)$ time, whereas removing one is done in $O(K)$ time.

With $GREEDY-SPLIT$, adding a channel requires another iteration of the algorithm, which takes $O(\frac{N}{K}+\log K)$ time, as we mention above. Deleting a channel is done by finding the partition that increases cost the least and removing it. This process takes $O(\log K)$ time.

6 The FLAT Algorithm

The $FLAT$ algorithm allocates an equal number of data items to each channel. In this section, we explain the performance of such an algorithm.

Lemma 6.1 *Allocating N data items using $FLAT$ over K channels reduces average expected delay by a factor of $\frac{1}{K}$.*

Proof: From Equation 2, when $K=1$, the cost of an allocation scheme for N data items is $\frac{N}{2}$

Furthermore, when $K>1$, the cost is

$$\frac{1}{2}\sum_{i=1}^{K}\left(N_i\sum_{d_j\in C_i}p_j\right)$$

Using $FLAT$, N_i is fixed at $\frac{N}{K}$, so average expected delay is $\frac{N}{2K}$.

Corollary 6.2 *The performance of FLAT is determined by solely by K, the number of channels available. In other words, the particular data items in each channel is irrelevant.*

Proof: Obvious from Lemma 6.1.

7 Performance Results

In this section, we show the results of tests performed on VF^K (vfk), $FLAT$ (flat), $OPTIMAL$ (dp) and $GREEDY - SPLIT$ (greedy). The bases of comparison are time to compute the allocation solution and average expected delay. These two sets of tests are conducted to justify the use $GREEDY - SPLIT$ in allocating data.

7.1 Running Time

Besides variation in the number of channels available, variation can also happen in the popularities of the data items. Popularities do not change independently (because $\sum p_i = 1$), and changing popularities may affect the partial ordering requirement suggested by Theorem 5.2. Therefore, the performance estimates of the solution may become invalid, and restoring them may require the recomputation of all the partitions. It is therefore important that a partitioning algorithm run quickly.

All algorithms are implemented using GNU g++ version 2.95.2. VF^K is implemented as described in [15]. $OPTIMAL$'s implementation follows that of the examples in [7]. $FLAT$'s implementation is trivial. $GREEDY - SPLIT$ is composed of one outer and one inner for loop, and uses the equations of Section 5.1. No other data structures or temporary results are kept for $GREEDY - SPLIT$. We run two timing experiments. One varies K, and the other varies N.

Experiments are run on a Intel 2 × 400 MHz Pentium II, with 256 MB of memory, running Redhat Linux 6.0. This machine is shared by other applications during the test, so timing results should be considered approximations. In other words, the experimental results may vary from platform to platform, but their relationships to each other should be consistent.

As expected, the $OPTIMAL$ and VF^K are the most computationally expensive. The performances of all algorithms, except for $FLAT$, are directly related to increasing database size and number of channels. These algorithms are more sensitive to an increasing number of channels because this parameter causes more split operations, resulting in more computation. In these tests, $GREEDY - SPLIT$ far outperforms VF^K and $OPTIMAL$. See Tables 1 and 2.

7.2 Average Expected Delay

In this section, we measure average expected delay, while varying the number of channels, the size of the database, and popularity skew. The ranges of parameter values we use are similar to those used

# Data Items	vfk	flat	dp	greedy
10	0	0	0	0
100	1	0	14	0.25
200	3	0	99	0.25
300	6	0	321	0.5
400	11	0	751	0.25
500	17	0	1446	0.75
600	33	0	2482	1
700	34	0	3911	1
800	47	0	5799	1
900	56	0	8222	1
1000	69	0	11253	1.25

Table 1: Number of Data Items vs. Compute Time (ms). Number of Channels = 4. Skew = 80/20.

# Channels	vfk	flat	dp	greedy
1	6	0	4	0
50	29	0	23479	2
100	51	1	47372	2.375
150	77	0	71555	3
200	106	0	95603	4
250	134	0	120140	6
300	164	1	144179	7.75
350	195	0	168673	9
400	227	0	193119	10.75
450	253	0	217766	12.125
500	276	0	242350	13.75

Table 2: Number of Channels vs. Compute Time (ms). Number of Data Items = 500. Skew = 80/20.

Parameter	Value Range	Control Value	Step
# of Data Items	1...1000	500	100
Θ Zipfian Skew [11], ($\Theta = 0$ for unif, 1 for high skew.)	0...1	(80/20 dist.)	.1
# Channels	1...500	4	50

Table 3: Table of Parameters. *Value Range* gives the high/low values of the parameters in the experiments. *Control Value* is the value of the parameter when that parameter is not varied. *Step* indicates how much we change the particular parameter for each trial when it acts as the independent variable.

in the performance evaluation of other work on broadcast disks [2, 16, 18]. See Table 3 for parameter values.

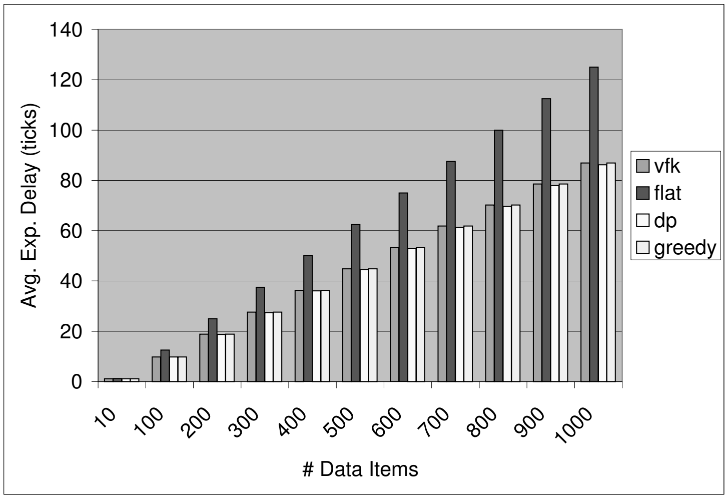

Figure 4: Number of Data Items vs. Average Delay. Number of Channels = 4. Skew = 80/20.

Varying The Number Of Data Items In this test, we see that the average delay is consistently better with the three non-$FLAT$ algorithms. VF^K and $GREEDY$ are close to optimal. Interestingly, as the number of data items increases from 100 to 1000, the ratio between $OPTIMAL$ time and $FLAT$ time ranges between .68 and .78. This means that the *percentage* improvement does not vary much with increasing database size. See Figure 4.

Varying Skew We vary Θ in this test. Clearly, if there is no skew ($\Theta = 1$), all algorithms result in the same delay. However, as the popularity of data items becomes more skewed, the performance of $FLAT$ is worse relative to those of the the others. Again, both VF^K and $GREEDY$ keep up with the performance of $OPTIMAL$. See Figure 5.

Varying The Number Of Channels In our final test, we vary the number of channels from 1 to 500, where the number of data items, N, equals 500. We do not show the results for the cases when

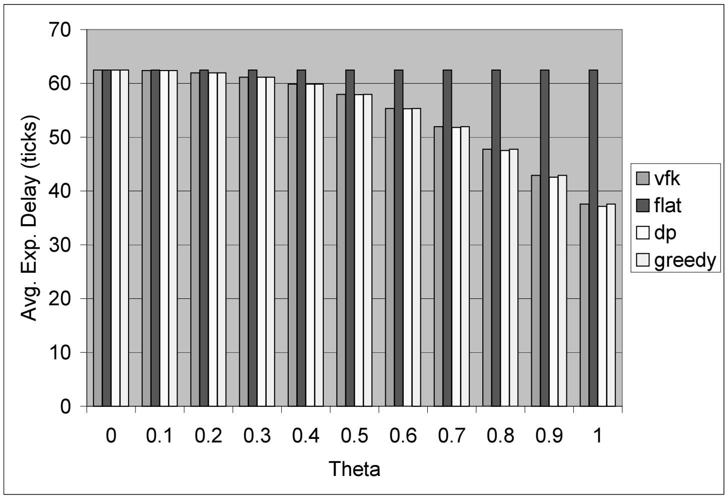

Figure 5: Access Probability (Popularity) Skew vs. Average Delay. Number of Data Items = 500. Number of Channels = 4.

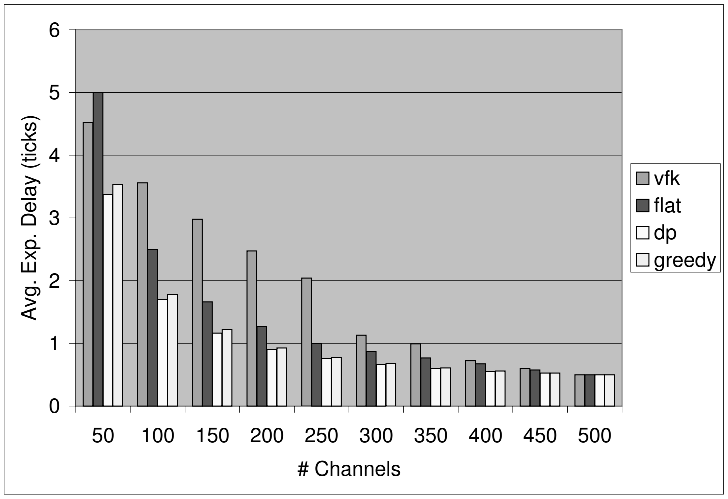

Figure 6: Number of Channels vs. Average Delay. Number of Data Items = 500. Skew = 80/20.

the number of channels is less than 50, because they would offset the scale of our chart, and hurt the readability of the rest of the results. The relationship among the results for each of the algorithms in the less than 50 channels tests is similar to the one for the 50 channel test. The difference is that the absolute values for the average expected delay for these tests ranges from about 9 ticks to about 43 ticks.

One interesting thing to note is the effect of having a high number of channels on the performance of VF^K. When the number of channels is ≥ 100, VF^K actually performs *worse* than $FLAT$. This happens because the heuristics used in VF^K limit the search space too aggressively (see Section 4) in order to simplify the algorithm. On the other hand, the performance of $GREEDY-SPLIT$ remains close to $OPTIMAL$.

8 Different Sizes

When data items are different sizes, $s_k, 1 \leq k \leq N$, the cost function becomes

$$\frac{1}{2}\sum_{i=1}^{K}\left(\sum_{d_k \in C_i} s_k \sum_{d_j \in C_i} p_j\right) \tag{8}$$

The minimum sum of squares problem (MSS) is NP-complete. MSS is stated as a partitioning of A into K subsets so as to minimize

$$\sum_{i=1}^{K}\left(\sum_{a \in A_i} s(a)\right)^2 \tag{9}$$

where $s(a) \in Z^+$ and $K \geq 2$.

Theorem 8.1 *The problem of optimally allocating data items of variable sizes NP-complete.*

Proof: Let VARSIZE be the decision problem of finding a set of C_i $(1 \leq i \leq K)$ that results in Equation 8 evaluating to a given c.

We must show that $VARSIZE \in NP$ and $L \leq_p VARSIZE$, where $L \in NPC$. Clearly $VARSIZE \in NP$. Given a certificate set of C_i, we can simply solve Equation 8 in polynomial time to verify if it evaluates to c.

We now prove that $VARSIZE$ is NP-hard by proving that $MSS \leq_p VARSIZE$. Given an instance of MSS, we can easily formulate an equivalent version of $VARSIZE$ where $K = |A|$ and

$p_i = s_i = s(a_i)$, $a_i \in A$ and $1 \leq i \leq K$. It is now obvious that $VARSIZE$ evaluates to c if and only if MSS evaluates to $2c$, for any given c.

9 The Design of Broadcast Disks

We now return to the concept of a channel as a medium for a set of broadcast disk. We begin this section by describing broadcast disks and thereby framing its design problem. We then describe our design techniques, and show its performance through extensive experiments.

9.1 Overview of Broadcast Disks

In this section, we describe and explain the characteristics of broadcast disks originally stated in [1]. For each of the characteristics, we describe how they have been handled in recent work on data broadcast.

Broadcast disk features include:

- **Fixed interarrival times between consecutive arrivals of the same data item.**

There are many reasons for desiring fixed interarrival times. One is a simple result from probability theory. If data requests are exponentially distributed, then lower interarrival variance results in lower expected waiting times.

Knowing interarrival times also helps client-side performance in terms of client-side caching [1, 21]. Predictable arrivals of data items increase the effectiveness of these protocols. Organizing the broadcast data items of the database into self contained cycles of known period simplify consistency control protocols. Again, the algorithms given in [3, 19, 20] do not result in predictable data arrivals. Furthermore, since they are generated on-the-fly, they have no notion of period.

- **A well-defined broadcast interval after which the broadcast schedule repeats.**

Periodic broadcasts aid in consistency control [6, 8, 16, 18], as there is a known unit to which a consistent set of updates can be applied. Predictible broadcasts in terms of interarrival times and periods also aid in the indexing of data, which can lead to power conservation from reduced tuning time for low-capacity clients [10].

- **An efficient use of bandwidth.**

The broadcast schedule should efficiently use the available bandwidth. To this end, various disks in the broadcast disk array are sized in a way as to allow more frequent transmission of more popular data.

Database researchers have concentrated on the first two points, but have done little work on the last. In [1], a rudimentary framework for designing broadcast schedules is proposed, but leaves out specific techniques. In [15], initial solutions to the channel problem that we formalize and solve in Section 5 is presented, but that work is never applied to broadcast disks.

On the other hand, network researchers have concentrated on the last, without regard to the first two. They search for algorithms that maximize bandwidth utilization and come up with near-optimal schedules [4, 3, 19, 20].

In this part of the paper, we explore how to simultaneously solve all three broadcast disk criteria.

9.2 Optimal K

In this case, the results of the data allocation to channels are mapped onto a single broadcast channel. This *simulates* the simultaneous transmission of the contents of multiple spinning disks of various periods. The contents of the disks with smaller periods would be broadcast more frequently on the single channel than the contents of those with larger ones. In this way, more popular data could be broadcast more frequently [1].

The design of broadcast disks requires three things: the determination of the number of disks to use; the allocation of data onto each disk (referred to as the *allocation problem*); the mapping of each disk onto the shared broadcast channel (referred to as the *mapping* problem). We have already investigated the allocation problem in the first part of the paper, so in this section, we discuss the mapping problem.

The problems described above are interrelated. First, as we see in Section 5, the number of disks, K, determines the allocation of data to disks. Therefore, by determining K, we can find a solution to the allocation problem. We will handle this problem later.

Second, the mapping solution we decide to use affects the effectiveness of the allocation solution described in Section 5 in terms of its effectiveness for broadcast disks. The allocation solution assumes that there exist multiple physical disks, each of which are broadcast onto their own broadcast channel. In this case, as the number of disks increases, the average expected delay decreases, until there is an exclusive broadcast channel for each data item.

Assume we use a simple mapping algorithm which cycles through each disk in a round-robin fashion, and broadcasts the *next* data item on that disk.:

Algorithm 9.1

```
SINGLE-CHANNEL
input: $K$ disks of data, each of size $N_i$
begin
  while $TRUE$
        $x := 0$;
        do
        for $i := 1$ to $K$
            do
            broadcast the $((x \bmod N_i) + 1)^{th}$ data item from disk $i$
        od
        $x := x + 1$;
  od
end
```

Let us consider how Algorithm 9.1 interacts with our allocation solution. The inputs to the allocation algorithm, $GREEDY - SPLIT$ in this case, include K, the number of disks. Given a single disk, $K = 1$, $GREEDY - SPLIT$ does nothing, and Algorithm 9.1 simply cycles each data item onto the single broadcast channel. At the other extreme, the lowest cost attainable by $GREEDY - SPLIT$ occurs when $K = N$. In this case, a single data item is allocated to each disk. However, Algorithm 9.1 cycles through each disk, transmitting that disk's *next* data item. Again, the result is that each data item is cycled through the broadcast channel, as it is when $K = 1$. Intuitively the optimal mapping solution requires some value of K between 1 and N. The goal of this section is to determine the optimal K.

Example 9.1 In Figure 7, we compute the average expected delay when the contents of a set of broadcast disks is mapped to a single shared medium using Algorithm 9.1.

> The single channel average expected delay varies irregularly depending on the number of channels used, but reaches a minimum when the number of disks used is about 60. For comparison, we also plot the optimal single channel average expected delay. We describe more details of these costs below.□

9.3 Justification of the Mapping Algorithm

We choose Algorithm 9.1 as our mapping algorithm because it is simple and intuitive. It cycles though each disk, and each data item contained on that disk. This makes the algorithm simple to reason

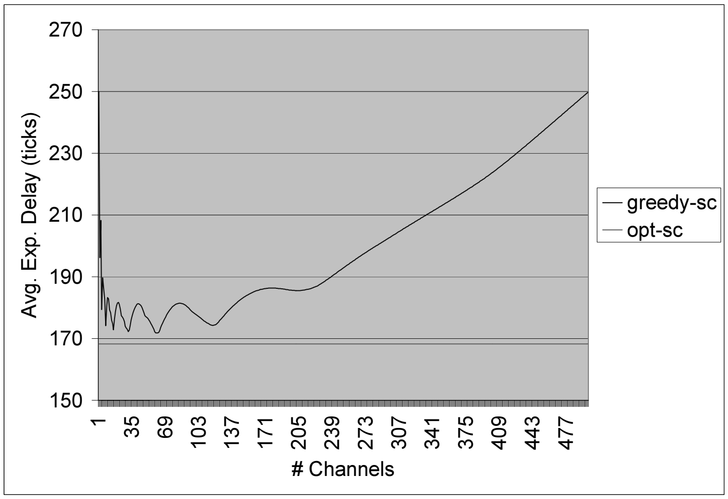

Figure 7: Average Expected Delay vs. Number of Broadcast Disks. The average expected delay using GREEDY (greedy-sc) is compared to the optimal average expected delay over a single channel (opt-sc) as the number of channels increases. Number of Data Items = 500. Skew = 80/20.

about. Furthermore, the resultant broadcast has *intuitive* properties.

Algorithm 9.1 allocates bandwidth to each data item in an intuitive way: the amount of bandwidth allocated to a data item is inversely proportional to the size of its disk. A data item from each disk is broadcast every $\frac{1}{K}$ ticks. A particular data item on a disk i is broadcast every $\frac{1}{KN_i}$ ticks. Given two data items on two broadcast disks, i and j, the relative rate at which each is broadcast is $\frac{\frac{1}{KN_i}}{\frac{1}{KN_j}} = \frac{N_j}{N_i}$. In other words, the relative frequency of broadcast of data items from two disks is inversely proportional to the ration of their sizes, which is what one would expect when data is allocated to each disk.

Finally, this algorithm is simple to implement and has low overhead. It can be implemented by maintaining a circular list for each disk. At each tick, all that must be done is the updating of a pointer to the next data item.

9.4 Problem Statement

We are given a set of unit sized data items, sorted by popularity. Our goal is to allocate them to the number of broadcast disks which minimizes the average expected delay.

In Section 5, we are given the number of broadcast disks, K, as a parameter. Given K, the cost function given by Equation 2 is $\frac{1}{2}\sum_{i=1}^{K} N_i \sum_{d_j \in c_i} p_j$. However, in our current problem, the cost function changes because K is a variable. Using Algorithm 9.1 in the single channel case, each disk is accessed once every K ticks, and each data item on disk i is accessed once every N_i times that disk is accessed. Therefore, the expected delay of a data item on disk i is $\frac{KN_i}{2}$. The average expected delay of a data item on this disk is $(\frac{KN_i}{2})\sum_{d_j \in c_i} p_j$. Our total cost over all disks is therefore:

$$\frac{K}{2}\sum_{i=1}^{K} N_i \sum_{d_j \in c_i} p_j \tag{10}$$

9.5 Simple Algorithm For Optimal K

In this section, we introduce *Safok*, the Simple Algorithm For Optimal K. Using the results from Section 5, we can generate an optimal broadcast disk design. $OPTIMAL$ minimizes the right term of Equation 10 for a given K. Therefore, in order to find a minimal total cost, we simply find $OPTIMAL$ costs for all $K, 1 \le K \le N$, and multiply this value by K. The K which results in the lowest product is our solution.

The simplicity of $GREEDY - SPLIT$ are evident here. Since, in effect, we are searching for data-to-channel allocation solutions for $K = N$, $OPTIMAL$ and VF^K are prohibitively expensive. We therefore use $GREEDY - SPLIT$ to approximate the optimal.

Algorithm 9.2

```
SAFOK
input: set of $N$ unit sized data items ordered by popularity
begin
  $initial\_cost := C_{1N}$
  $current\_cost := initial\_cost$
  for $i := 1$ to $K$
      do
      $cost_i = current\_cost;$
      $kCost_i = \frac{k}{2} * cost_i;$
      if $i = 1 \vee kCost_i < bestKCost$
        do
            $bestKCost = kCost_i;$
            $bestK = i;$
        od
        Search all partitions for the best split point, $s$, between partition boundaries $l$ and $r$;
        Make a split at $s$.
        $current\_cost = current\_cost - C^s_{lr}.$
      od
  end
```

Safok basically runs $GREEDY - SPLIT$ for $K = N$, but records the single channel and multiple channel costs. The final result is contained in the variables *bestKCost* and *bestK*. Because the overhead of recording these values requires $O(1)$ time per iteration, and $GREEDY - SPLIT$ runs in $O(N \log K + K \log K)$ time, Safok runs in $O(N \log N)$ time.

10 Performance

In this section, we compare the performance of $Safok$ (denoted by *greedy-sc* in the figures) against a popular push-based bottom up single channel scheduling algorithm and the optimal average expected delay. The bottom up algorithm we use is representative of work done by the networking community [3, 4, 5, 19, 20]. It achieves near-optimal performance in terms of average expected delay by assigning a priority to each data item i at each tick:

$$\text{priority}_i = Q_i^2 p_i$$

In the equation above, Q_i is the number of ticks since the last time data item i was transmitted and p_i is its popularity. The data item with the highest priority at each tick is transmitted. We will refer to this algorithm as $Vaidya - c$.

The metrics by which we compare $Safok$ and $Vaidya - c$ are based on the broadcast disk characteristics. We will compare how the two do in each category.

- **Fixed interarrival times between consecutive arrivals of the same data item.**

In order to measure the fixedness of interarrival times of a broadcast schedule, we measure interarrival time variability. We only measure the variability of the $Vaidya - c$ schedule, because there is naturally no variability in a $Safok$ schedule.

In the following experiments, we vary database size (from 1 to 1000 data items) and theta (from 0 to 1) as we did in previous experients, but increase the simulation time to 1000000 ticks to approach steady state values. We record the standard deviation s in the interarrival times for each data item i as the positive square root of:

$$s_i^2 = \frac{\sum_{j=1}^{O_i - 1}(I_{ij} - \overline{I_i})^2}{O_i - 1} \tag{11}$$

In the equation above, O_i is the number of occurences of data item i in the broadcast schedule, I_{ij} is the number of ticks between the j^{th} and the $(j+1)^{th}$ occurence of data item i, and $\overline{I_i}$ is the mean interarrival time of data item i. The average standard deviation is therefore:

$$\sum_{i}^{N} p_i s_i \tag{12}$$

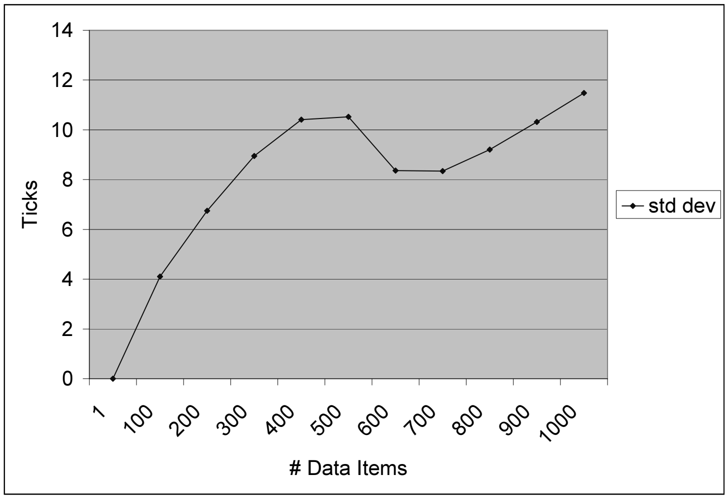

Figure 8: Number of Data Items vs. Std. Dev. of Interarrival Time. Skew = 80/20.

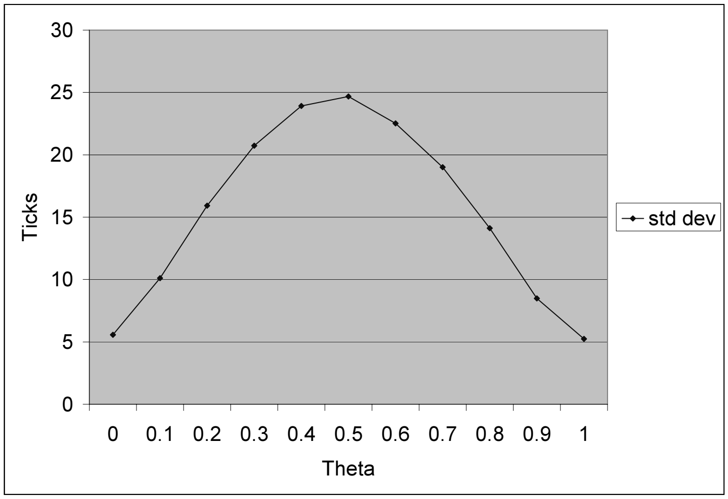

Figure 9: Access Probability (Popularity) Skew vs. Std. Dev. of Interarrival Time. Number of Data Items = 500.

As we see in the test results, the average variability in interarrival time of data items using $Vaidya - c$ is quite high. From Figure 8, if the database size is 500 data items, the average standard deviation is about 10.5 ticks. If the arrivals times are normally distributed, a typical client would have to listen (4)(10.5) = 42 ticks in order to have a 98% chance of receiving a requested data item. If we are uncertain about the distribution of arrival times, then using Chebyshev's Inequality, to get a 98% probability of receiving data item i, we would have to listen for $(2)\sqrt{50}(10.5) \sim 148$ ticks. Considering that the database only contains 500 data items, this is a huge cost. In contrast, using broadcast disks, the exact arrival time of each data item can be calculated potentially allowing a listening time of 1 tick for each data item.

Figure 9 gives even worse results. It shows that, regardless of popularity skew, there is at least a standard deviation of about 5 ticks for the interarrival time of a typical data item. So, if these interarrival times are normally distributed, a client must typically wait $(4)(5) = 20$ ticks for a data item. Since the tuning time for a broadcast disk schedule is potentially 1 tick, we can conclude that a client must tune in possibly $20\times$ longer if it is not listening to broadcast disks, which is a huge waste of energy.

- **A well-defined broadcast interval after which the broadcast schedule repeats.**

Of the work cited above done by the community, only [5] contains a repeating (periodic) broadcast schedule. In that case, in order to avoid the complexity of computing a priority for each data item at each tick, a schedule is precomputed and repeatedly broadcast. Although broadcast periodicity is achieved, there is no high level semantic meaning associated with the length of the repeated broadcast schedule sample.

With $Safok$, we have notions of minor and major cycles, and can determine the expected delay of data items contained in each.

Given:

N - number of data items

N_i - size of disk i, and $C_i \succ C_i$ iff $i < j$.

p_i - popularity of data item i, and $p_i \geq p_j$ iff $i \leq j$.

- Minor period - The number of ticks in the sequence of ticks which begins with a transmission from the first disk and ends with a transmission from the last disk.

$$A = K \tag{13}$$

- Major period - The number of ticks in the sequence of ticks which begins with a minor period consisting of the first elements of each broadcast disk and ends with a minor period consisting of the last elements of each broadcast disk. In other words, this is the minimal number of ticks necessary to broadcast all disks an integral number of times.

$$B = K\text{lcm}\{N_i, i = 1, ..., K\} \tag{14}$$

- Period of disk i - The number of ticks required to transmit all of disk i.

$$D_i = KN_i \tag{15}$$

- Number of times disk i is transmitted during each major period.

$$E_i = \frac{B}{D_i} \tag{16}$$

- Expected delay for a data item on disk i.

$$F_i = \frac{D_i}{2} \tag{17}$$

- **An efficient use of bandwidth.**

$Vaidya - c$ has some benefits and drawbacks in comparison to broadcast disk scheduling. Its benefits include near-optimal average expected, and, because priorities are computed at each tick, it can adapt changing values of p_i.

However, in practice, this adaptive nature also hurts $Vaidya - c$. The overhead of maintaining priorities for every data item at every tick is high. This problem exists in all bottom-up algorithms. Some sub-optimal solutions include repeated broadcast of a precomputed schedule [5] and heuristic pruning of search space [3]. In [20], a scheme which aggregates data items into buckets is suggested. In this case, at each tick, computations are done for each bucket instead of each data item. These schemes hurt overall performance. For instance, test results reported in [20] show that using 5 and 10 buckets in a 1000 data item database doubles and triples the average expected delay, respectively, in some cases.

Furthermore, with bucketing, $Vaidya-c$ is no longer adaptive to changes in data-item popularities. When popularities change the buckets must be regenerated. In any case, in these experiments, we consider the ideal case in which bucketing is not used.

Another drawback in using $Vaidya-c$ is that the schedules are difficult to analyze. The main reason for this is consecutive transmissions of a given data item are not equally spaced apart. It is therefore difficult to precisely measure the expected delay of each data item. We therefore attain experimental results by generating a broadcast schedule using $Vaidya-c$, and explicitly measuring the average expected delay of each item. The simulations we use to generate the following experimental results for $Vaidya-c$ are 500,000 ticks long.

For comparison, we also compute the optimal average expected delay (denoted by *opt-sc*), which is given in [20] as:

$$\frac{1}{2}\left(\sum_{i=1}^{N}\sqrt{p_i}\right)^2$$

In our tests, we vary the popularity skew and the size of the database just as we did in the tests results shown in Section 7. However, since the number of disks K is no longer a parameter, we do not include it as an independent variable.

Parameter	Value Range	Control Value	Step
# of Data Items	1...1000	500	100
Θ Zipfian Skew, ($\Theta = 0$ for unif, 1 for high skew.)	0...1	(80/20 dist.)	.1

Table 4: Table of Parameters for the Single Channels Tests.

Varying The Number Of Data Items In these tests, as the database size increases, the average expected delay of all schedules increases linearly. All performances are similar.

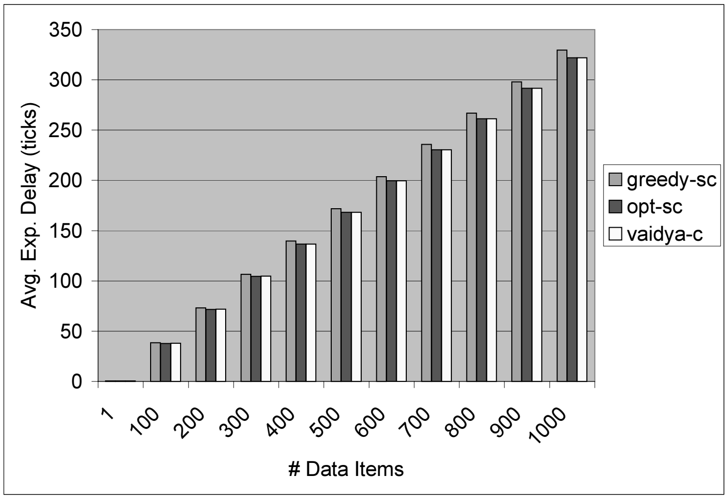

Figure 10: Number of Data Items vs. Average Delay. Skew = 80/20.

Varying Skew As the skew in the popularities of data items increases, the average expected delays for all schedules decreases as well. Again, all performances are similar. In both tests performed in this section, all performances are near optimal.

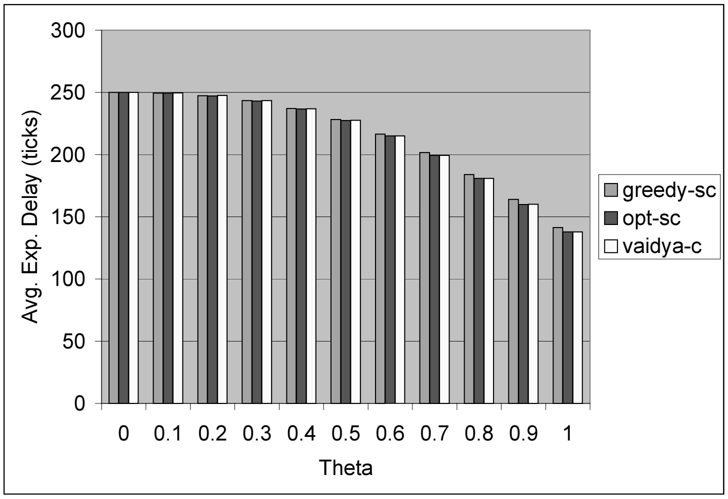

Figure 11: Access Probability (Popularity) Skew vs. Average Delay, Single Channel Case. Number of Data Items = 500.

11 Conclusion and Areas of Future Work

In this paper, we tackle the problem of designing broadcast disks in order to minimize average expected delay. We start by first optimally solving the problem of allocating data to physical channels in the presence of popularity skew. We prove properties of this problem which allows us to use dynamic programming. These properties also allow us to design a greedy algorithm called $GREEDY-SPLIT$ that gives near-optimal results.

In [15], the authors describe VF^K to solve the same problem, but their work has three problems: it lacks a theoretical foundation; it is unnecessarily complex; its results are never mapped to a single channel case. The first problem results in unexpected performance figures (see Section 7) and hurts is adaptability to other applications. The second problem inhibits its use on a large scale. The third problem means that the broadcast disk issue was never addressed–meaning that broadcast disks are only abstract objects used in single channel broadcast schedule design.

We use $GREEDY - SPLIT$ as the basis of $Safok$, which is an $O(N \log N)$ algorithm that both decides how many broadcast disks to generate and how to allocate data to those disks. We show how to map the results of $Safok$ onto a single broadcast channel, preserving the broadcast disk properties, without necessitating the "holes" that result when using the algorithm given in [1].

The resultant broadcast disk based broadcast schedule results in near-optimal average expected delay, and is comparable in performance to broadcast schedules generated by the networking community which do not consider the broadcast disk constraint. Moreover, broadcasting from broadcast disks does not require the server to make priority computations for each data item at each tick, which could increase server performance in practice.

The major improvements in enforcing the broadcast disk constraint is in predictible arrival times of data and use of disk semantics for consistency control. In fact, in our experiments, the tuning time using broadcast disks can be at least 20× lower than without.

Much work is left to be done. We are currently focusing on algorithms which handle data items of varying sizes. Different sized data items cannot be ordered the way the Theorem 5.2 suggests unit sized data be ordered. Without this ordering constraint, our solution techniques fail. In [20], priorities are assigned to each data item as a funciton the ratio of popularity and size is discussed, but that work simply tried to optimize bandwidth usage, but our broadcast disk semantics–the notion the data are cycled–makes such a scheme inapplicable. We are currently studying alternatives.

References

[1] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast disks: Data management for asymmetric communication environments. *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 1995.

[2] S. Acharya, M. Franklin, and S. Zdonik. Disseminating updates on broadcast disks. *International Conference on Very Large Databases*, September 1996.

[3] D. Aksoy and M. Franklin. Scheduling for large-scale on-demand data broadcasting. *Joint Conference Of The IEEE Computer and Communications Societies*, 1998.

[4] M. H. Ammar and J. W. Wong. The design of teletext broadcast cycles. *Performance Evaluation*, 5(4):235–242, December 1985.

[5] A. Bar-Noy, B. Patt-Shamir, and I. Ziper. Broadcast disks with polynomial cost functions. *Joint Conference Of The IEEE Computer and Communications Societies*, March 2000.

[6] T. F. Bowen, G. Gopal, G. Herman, T. Hickey, K. C. Lee, W. H. Mansfield, J. Raitz, and A. Weinrib. The datacycle architecture. *Communications of the ACM*, 35(12), December 1992.

[7] T. H. Cormen, C. E. Leiserson, and R. Rivest. *Introduction To Algorithms.* Mc Graw Hill, nineteeth edition, 1997.

[8] G. Herman, G. Gopal, K. C. Lee, and A. Weinrib. The datacycle architecture for very high throughput database systems. *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 97–103, 1987.

[9] T. Imielinski and B. R. Badrinath. Wireless computing: Challenges in data management. *Communications of the ACM*, October 1994.

[10] T. Imielinski, S. Viswanathan, and B.R. Badrinath. Energy efficient indexing on air. *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 1994.

[11] D. E. Knuth. *The Art Of Computer Programming*, volume 3:Sorting and Searching. Addison-Wesley Publishing Company, second edition, 1975.

[12] A. C. Konig and G. Weikum. Combining histograms and parametric curve fitting for feedback-driven query result-size estimation. *International Conference on Very Large Databases*, 1999.

[13] W. C. Lee, Q. Hu, and D. L. Lee. A study on channel allocation for data dissemination in mobile computing environments. *Mobile Networks and Applications*, 4:117–129, 1999.

[14] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus - an architecture for extensible distributed systems. *Proc. 14th SOSP*, December 1993.

[15] W. C. Peng and M. S. Chen. Dynamic generation of data broadcasting programs for broadcast disk array in a mobile computing environment. *Conference On Information And Knowledge Management*, pages 38–45, 2000.

[16] E. Pitoura and P. Chrysanthis. Exploiting versions for handling updates in broadcast disks. *International Conference on Very Large Databases*, 1999.

[17] K. Prabhakara, K. A. Hua, and J. Oh. Multi-level multi-channel air cache designs for broadcasting in a mobile environment. *Proceedings of the IEEE International Conference on Data Engineering*, 2000.

[18] J. Shanmugasundaram, A. Nithrakashyap, R. Sivasankaran, and K. Ramamritham. Efficient concurrency control for broadcast environments. *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 85–96, 1999.

[19] C. J. Su, L. Tassiulas, and V. J. Tsortras. Broadcast scheduling for information distribution. *Wireless Networks*, 5:137–147, 1999.

[20] N. H. Vaidya and S. Hameed. Scheduling data broadcast in asymmetric communication environments. *Wireless Networks*, 5:171–182, 1999.

[21] S. Zdonik, M. Franklin, R. Alonso, and S. Acharya. Are ‘disks in the air’ just pie in the sky? *IEEE Workshop on Mobile Computing Systems and Applications*, December 1994.