Int. Computer Symposium, Dec. 15-17, 2004, Taipei, Taiwan.

An Efficient Protocol for Disseminating Data with Multi-dimensional Index on Multiple Broadcasting Channels*

Chuan-Ming Liu†, Kun-Feng Lin†, Susanne E. Hambrusch‡, and Chien-Hung Liu†
†Comp. Sci. and Info. Eng., National Taipei University of Tech., Taipei, Taiwan
‡Department of Computer Sciences, Purdue University, West Lafayette, IN 47907, USA
†{cmliu,s2598019,cliu}@ntut.edu.tw, ‡seh@cs.purdue.edu

Abstract- *In this paper we consider scheduling the broadcast of data with a multi-dimensional index tree on multiple channels. Poorly designed multi-channel broadcast schedule and client query algorithms can result in an increase in the* latency *(time elapsed from requesting to receiving data) compared to a single channel environment. We provide an algorithm for scheduling the broadcast and the resulting schedule has an optimal cycle length and can reduce the latency by a factor of $\frac{1}{c}$ for a simple query where c is the number of channels. We also provide an efficient protocol for the range-like query which results in a partial traversal of the index tree. We finally conclude this paper with our simulation work.*

keywords: broadcasting, multi-dimensional index trees, query process, multiple channels, latency.

1 Introduction

The continuous broadcast of data together with an index structure is an effective way of disseminating data in a wireless, mobile environment. Such a client-server model is a server-push and client-pull model. In the broadcasting system, each periodic broadcast constitutes a broadcast cycle. The availability of an index allows a reduction in the *tuning time* (the amount of time spent listening to the broadcast) and thus leads to lower power consumption for a client. In this paper we consider scheduling the broadcast of data with a multi-dimensional index tree, such as R^*-tree [2], in multiple channel environments where a mobile client can tune into any specified channel at one time instance.

Poorly designed multiple channel broadcast schedules and client query algorithms can result in an increase in the *latency* (time elapsed from requesting to receiving data) compared to a single channel environment. Suppose there are c channels. For a given n-node multi-dimensional index tree, we provide a client-server protocol where the cycle length of the generated broadcast is optimized and the latency and tuning time for a client to execute query are minimized. We point out that, if there are c channels, the latency for any query should not be more than c cycles; otherwise, using one channel to broadcast is better enough. We also compare our protocol with the one in a one-channel environment by simulation.

*Work supported by the National Science Council under the grant numbers NSC-92-2213-E-027-010.

2 Related Work

Most of the previous work has focused on the problem of minimizing the latency in the model where the server broadcasts only the data. The resulting broadcast schedule is generated by considering the given data access frequencies [1, 11]. [9] and [10] provide broadcast schedules for supporting range queries on one dimensional data.

Broadcasting data with index was first formalized in [5]. The problem is how to mix the data with index in order to reduce the latency and tuning time. Different index techniques such as hashing and distributed index were studied in [5, 7, 8]. When each data item has different access frequency, [3] proposes a method to construct an index tree based on the data access frequency to minimize the average tuning time. Protocols for scheduling the broadcast of a multi-dimensional index tree and supporting range-link queries in one channel can be found in [4].

Broadcasting data in multiple channels has been studied in [4, 7, 8, 11]. Different strategies for allocating data over multiple channels by clustering or partitioning dependent data are discussed in [11]. [7] and [8] both consider broadcasting index trees on multiple channels for non-uniform data. The proposed schedule in [8] lacks of flexibility and requires a number of channels equal to the height of the index tree. The schedule in [7] is only applicable to a problem of small size. When the size of broadcast data is large, they provide heuristics to schedule the broadcast. In our work, we consider broadcasting a multi-dimensional index tree in a multiple channel environment for uniform data. The generated broadcasting schedule fully supports the range-like queries and the resulting latency is shorter than the broadcasting in one channel.

3 Problems

Suppose that the number of channels used for broadcasting is $c > 1$. In the broadcast channels, we assume that each packet in the channels corresponds to each node in the index tree. Therefore, each packet will carry its corresponding node's information including the children list of that node. For each child in the child list, we also store the child's position and channel assigned as well as the index of that child. Therefore, by examining an internal node, we can decide which children should be received. We refer examining a node as *exploring* a node.

The execution of a query we consider on a multi-dimensional index tree results in not only a single path traversal from the root to a leaf (*one-path search*) but also partially traversing the index tree (*multi-path search*). Multi-path search is more general than one-path search and more practical for real world applications. We assume that all the clients have enough memory to execute their own queries.

We observe that, in order to have less latency, the broadcast schedule on multiple channels should satisfy the following properties, *ancestor property* and *switch property.* We say that the generated broadcast schedule satisfies the ancestor property if when a node v is broadcast for the first time in a cycle, the parent of v has been broadcast earlier in the same cycle. Let u and v be two nodes, with u being the parent of v. Assume u and v are assigned to channels c_u and c_v, respectively. A *switch* occurs between u and v when c_u is not equal to c_v. We say a broadcast schedule satisfies the switch property if for each switch between u and v we have $c_u < c_v$. The switch property ensures that any root-to-leaf path in a tree experiences at most $c - 1$ switches, if there are c channels. This leads to a fewer number of switches between channels; thus, reduces the power consumption. Our schedule is indeed optimal for some kinds of queries.

We assumes that each node in the tree is broadcast only once in a cycle. In a multiple channel environment, a mobile client can only select one channel to tune into at each time instance; hence, conflicts arise when a mobile client needs to access nodes placed in the same position at some time instance. We refer to such a conflict as *read-conflict.* The read-conflicts will impact the performance of the query processing. For more details about the read-conflict, please refer to [4]. Since it is not easy to control such conflicts, our server broadcast schedule does not consider the read-conflicts too much in this paper. Our schedule will have the ancestor property and switch property as well as minimizes the cycle length. On the client side, we provide a query processing corresponding to the broadcast schedule which minimizes the latency.

4 Broadcasting Protocol

This section introduces our data disseminating protocol which consists of the broadcast scheduling on the server side and the query processing on the client side. We start our discussion with the broadcast scheduling.

4.1 Server scheduling

This section presents our algorithm for scheduling the broadcast for an n-node multi-dimensional index tree on c channels, $c > 1$. We assume that the degree of the index tree must have the following property: whenever the algorithm considers a subtree rooted at r' on c' channels, node r' must have at least c' children. Thus, initially, the root r must have at least c children. Furthermore, the schedule can be generated in $O(n)$ time.

We refer to the algorithm generating the broadcast schedule as Algorithm c-MinCycle shown in Figure 1. The first step (a) arranges the children of every node of input tree T by non-increasing sizes (i.e., the number of nodes in subtree) of their corresponding subtrees, (b) assigns the root r to position 1, and (c) sets the capacity of each channel (i.e., the number of nodes still to be assigned). The second step corresponds to a call to Algorithm AssignChannels and it is the heart of Algorithm c-MinCycle: every node is assigned to a channel, but not to a position in the channel. The channel assignments made result in selected nodes being marked as *filler-nodes* or *root-nodes.* Filler-nodes guarantee that the generated schedule satisfies the ancestor property. Root-nodes terminate the assignment process: When a node u is marked as a root-node, the nodes in the subtree rooted at u are (i) assigned to the same channel with node u and (ii) they do not get marked. The distribution of filler- and root-nodes is such that for a root-node u, all nodes on the path from the root r to u's parent are filler-nodes. The third step of Algorithm c-MinCycle assigns every node to a position in the assigned channel. Filler- and root-nodes are used to determine the positions of nodes within their assigned channel.

We now turn to Algorithm AssignChannels whose description is given in Figure 2. AssignChannels is invoked with four parameters: (1)a filler-node u which is the root of a subtree T', (2)an integer β representing the number of channels used, (3) a parameter *offset*, and (4) a list L containing β target channel capacities.

```
Algorithm c-MinCycle
Input: $n$-node tree $T$ with root $r$
Output: a $c$-channel broadcast schedule with cycle
length $\lceil \frac{n-1}{c} \rceil + 1$
(1)   (a) arrange the children of every node by
          non-increasing subtree sizes
      (b) assign root $r$ to channel 1 and mark $r$ as a
          filler-node
      (c) set the capacity of each channel
(2)   AssignChannels($r, c, 0, L$)
(3)   for channel $i$, $1 \le i \le c$, do
          (a) starting with position 2, contiguously
              place the filler-nodes assigned to
              channel $i$ in order of increasing levels;
          (b) consider the root-nodes assigned to
              channel $i$ by increasing level in $T$;
              root-nodes on the same level are
              considered from right to left;
              starting with the next position in
              channel $i$, place the nodes in the sub-
              trees rooted at each root-nodes in pre-
              order in consecutive channel locations
End c-MinCycle
```

Figure 1: Generating a c-channel broadcast schedule of minimum cycle length satisfying the ancestor and switch properties.

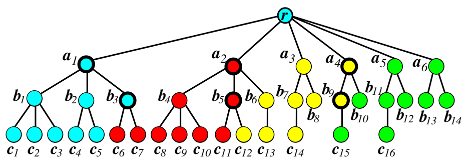
The nodes from tree T' - excluding node u, which is already assigned - are assigned to channels *offset*+1, ... , *offset*+β. Alternatively, we will say node u is associated with channels *offset*+1, ... , *offset*+β. Every channel is assigned the number specified in its corresponding target capacity. Suppose the channels associated with node u are channels $1, \cdots, \beta$,

```
Algorithm AssignChannels($u$, $\beta$, offset, $L$)
(0)   let $v_1, \ldots, v_l$ be the children of $u$ with $\beta \le l$
      set $i = l$; $k = \beta$
(1) Root-Oriented Step:
while ($k \le i$ and $i \neq 0$) do
(1.1) if $s_i < t_k$ then
        assign $v_i$ as a root-node in channel offset+$k$;
        $t_k = t_k - s_i$
      else if $s_i = t_k$ then
        assign $v_i$ as a root-node in channel offset+$k$;
        $t_k = 0$; $k = k - 1$
      else /* invoke a 2-channel problem on $v_i$ */
        assign $v_i$ as a filler-node in channel
        offset+$k - 1$;
        make a list $L' = ((s_i - 1) - t_k, t_k)$; $k = k - 1$;
        AssignChannels($v_i$, 2, offset+$k - 2$, $L'$);
        $t_k = 0$; $t_{k-1} = t_{k-1} - (s_i - t_k)$; $k = k - 1$
(1.2) $i = i - 1$
endwhile
(2) if $i = 0$ then /* all children have been handled*/
      return
(3) Filler-Oriented Step:
(3.1) mark $v_1, \cdots, v_i$ as filler-nodes for channels
        offset+1, $\cdots$, offset+$i$
      and target channel capacities update
(3.2) $r_0 = 1$
(3.3) for $j = 1$ to $i$ do
        • determine $r_j$ such that
          $\sum_{m=r_{j-1}}^{r_j - 1} t_m < (s_j - 1)$ and
          $\sum_{m=r_{j-1}}^{r_j} t_m \ge (s_j - 1)$
        • if $r_j = r_{j-1}$ then
            assign all children of $v_j$ as root-nodes to
            channel offset+$r_j$;
            $t_{r_j} = t_{r_j} - (s_j - 1)$
        • if $r_j > r_{j-1}$ then
            $t_{r_j} = t_{r_j} - (s_j - \sum_{m=r_{j-1}}^{r_j - 1} t_m - 1)$;
            $t'_{r_j} = (s_j - \sum_{m=r_{j-1}}^{r_j - 1} t_m - 1)$;
            make a list $L'$ of target capacities:
            $L' = (t_{r_{j-1}}, \ldots, t_{r_j})$;
            AssignChannels($v_j$, $r_j - r_{j-1} + 1$,
            offset+$r_{j-1} - 1$, $L'$);
            $t_{r_j} = t_{r_j} - t'_{r_j}$;
      endfor
End AssignChannels
```

Figure 2: Assigning the nodes in the subtree rooted at node u to β channels.

$\beta > 1$. Let t_k be the target capacity of channel k, $1 \le k \le \beta$. Let $v_1, \cdots, v_l$ be the children of node u. We have $\beta \le l$. This holds for all calls to AssignChannels since (i) initially every non-leaf node has at least β children and (ii) removing children from future calls results in a corresponding reduction in the number of channels to be filled.

Algorithm AssignChannels starts assigning nodes to channels in the Root-Oriented Step and then in the Filler-Oriented Step. The Root-Oriented Step considers the children of u from right to left. This corresponds to starting with the subtree of smallest size. One iteration assigns either one subtree rooted at a child of node u to one channel or it partitions one subtree among two channels. The Root-Oriented Step operates as long as the number of children of u not assigned to a channel is larger or equal to the number of channels still to be assigned nodes. This is the condition enforcing the ancestor property of the generated schedule. In figure 3, the Root-Oriented Step processes nodes a_6, a_5, a_4, and a_3 and the root-nodes are a_6, a_5, and a_3. Node a_4 is a filler-node.


r
a1 a2 a3 a4 a5 a6
b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 b11 b12 b13 b14
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16

Position	1	2	3	4	5	6	7	8	9	10
Channel 1	r	a_1	b_3	b_2	c_4	c_5	b_1	c_1	c_2	c_3
Channel 2		a_2	b_5	b_4	c_8	c_9	c_{10}	c_{11}	c_7	c_6
Channel 3		a_4	b_9	a_3	b_7	c_{14}	b_8	b_6	c_{13}	c_{12}
Channel 4		a_6	b_{13}	b_{14}	a_5	b_{11}	c_{16}	b_{12}	b_{10}	c_{15}

Figure 3: An index tree of size 37 to be scheduled on four broadcast channels with optimal cycle length 10; filler-nodes (bold lines) are a_1, a_2, a_4, b_3, b_5, and b_9.

When the number of children of u, i, which have not been processed is less than the number of channels, k, which need nodes to fill in, the algorithm switches to the Filler-Oriented Step. Note that this does not happen when the Root-Oriented Step terminates when all of the children have been explored (since channel assignments are complete). The Filler-Oriented Step assigns nodes $v_1, \cdots, v_i$ to channels $1, \cdots, k-1$, respectively, and marks them as filler-nodes (done in Step (3.1)). The remaining children of u are then considered from left to right. Assume we are currently at child v_j, $1 \le j \le i$. Entries r_j (resp. r_{j-1}) represents the largest (resp. smallest) channel index being assigned a node in the subtree tooted at v_j. We refer to Step (3.3) for the exact computation of these indices. Then, when $r_j = r_{j-1}$, all children of node v_j are made root-nodes for channel r_j. Note that node v_j remains labeled a filler-node. It is not a root-node, since it is not assigned to channel r_j, but to a smaller indexed channel. When $r_j > r_{j-1}$, we invoke a channel broadcast problem on v_j using $r_j - r_{j-1} + 1 < k$ channels, as detailed in Step (3.3) of Algorithm AssignChannels. In the figure 3, the Filler-Oriented Step processes nodes a_1 and a_2 and assigns these two nodes as filler-nodes.

To show that Algorithm c-MinCycle satisfies the ancestor property, we first state a number of properties following from the way nodes are assigned to channels and channel positions.

Property 1 *If a filler-node u is assigned to channel k and a descendant of u is assigned to channel k', then $k \le k'$.*

For the following properties assume u is a filler-node associated with channels $\gamma + 1, \cdots, \gamma + \beta$, $\beta > 1$.

Property 2 *The subtree rooted at u contains no filler-node assigned to channel $\gamma + \beta$.*

Property 3 *Let u' and u'' be two filler-node children of u with u' to the left of u''. Assume $\gamma' + 1, \cdots, \gamma' + \beta'$ are the channels associated with u' and $\gamma'' + 1, \cdots, \gamma'' + \beta''$ are the channels associated with u'', where β' and $\beta'' > 1$. Then, $\gamma + 1 \le \gamma' + 1 < \gamma' + \beta' \le \gamma'' + 1 < \gamma'' + \beta'' \le \gamma + \beta$.*

Property 4 *Let u' be a filler-node associated with channels $\gamma' + 1, \cdots, \gamma' + \beta'$. Assume u' is on level i, $i \ge 2$. Then, for any channel k with $\gamma' + 1 \le k < \gamma' + \beta'$, u' or a right sibling of u' is a filler-node assigned to channel k. In addition, level i of tree T contains exactly one filler-node assigned to channel k.*

The following two lemmas show that the broadcast schedule generated by Algorithm c-MinCycle satisfies the ancestor property. We first use the above properties to show that the ancestor property is satisfied for the filler-nodes. Then, we argue that the ancestor property holds for remaining nodes by first showing that all nodes assigned to channel c satisfy the ancestor property. The argument used for channel c is then applied to the remaining channels. Due to the space limit, we omit the detail of the proofs. Following these two lemmas, we conclude this section with a theorem.

Lemma 1 *Let P be any path from the root r to some filler-node u with $P = < p_1 = r, \cdots, p_e = u >$. Then, $pos(p_i) = i$, $1 \le i \le e$.*

Lemma 2 *Let v be a node assigned to position $pos(v)$ in channel c. If u is v's parent, then $pos(u) < pos(v)$.*

Theorem 3 *The schedule generated by Algorithm c-MinCycle satisfies the ancestor and the switch property. The schedule can be generated in $O(n)$ time.*

Proof: The switch property follows from Property 1. From Lemma 2 we know that the ancestor property holds for all nodes assigned to channel c. Remove from the generated schedule and from the tree all nodes assigned to channel c. Then, mark every filler-node assigned to channel $c-1$ as a root-node for channel $c-1$ and apply the argument given in Lemma 2. A repeated application of this process shows that the ancestor property holds for all nodes in all channels. Hence, the generated schedule satisfies the ancestor property. The $O(n)$ time for generating the schedule follows immediately from known algorithms for tree computations and tree traversals. □

4.2 Client processing

This section discusses the query processing on the client side. We assume that the query processing starts at beginning of a broadcast cycle. For one-path search, the query processing is straightforward. Having the broadcast scheduled satisfy the ancestor property and achieve the minimum cycle length, one-path search can always be done in O($\frac{n}{c}$) time in a c-channel environment.

For a multi-path search, after exploring an internal node, one can have each child's position and channel number assigned to it and know which child(ren) should be explored later. We therefore must store the information of these relevant children. We discuss two different methods for multi-path search on the client side.

4.2.1 Method 1: Using a stack (SM)

The first method for multi-path search is straightforward and we use it to compare our method discussed in the following subsection. This simple method uses a *stack* to store the children information the client has received. Due to the FILO property of a stack, we need to push the child assigned to a large channel number first. If there is a tie, we push the child having large position first. After pushing all the relevant children into the stack, the one popped out from the stack is the next node to be explored. However, this methods may still result in a long latency. This will be explained later in Section 5.

4.2.2 Method 2: Two dictionary structures (TDSM)

In a multiple channel environment, nodes to be explored may have the same position but be on different channels. The client uses a *p-element* to hold the nodes to be explored which have the same position. Therefore, a p-element consists of a position and a set of channel indices. It maintains two dictionary structures, T_{act} and T_{next}, which are built on p-elements with p-element's position as the key.

The client uses T_{act} to determine the next node to explore. It first uses the operation *Extract-Min* on T_{act} to get the p-element, e, having the closest (smallest) position. It then decides the next node to explore by selecting the smallest channel index from the channel index set of e. If the channel index set of e is not empty after removing the smallest channel index, the client inserts e into T_{next} for the query processing to extract e in the next cycle.

Consider a client has determined the next node to explore from T_{act}, say node u. If u is a productive node, let $v_1, \cdots, v_l$ be the children of u to be explored. Nodes $v_1, \cdots, v_l$ are then inserted into T_{act}. When inserting a node v_i, $i = 1, \cdots, l$, into T_{act}, the client first checks whether the position of v_i is already in T_{act}. If it finds a p-element having the position of v_i as the key in T_{act}, it inserts the channel index of v_i into the channel index set. Otherwise, the client creates a p-element from node v_i and inserts this p-element into T_{act}.

After the exploration of a productive node u, the next node to be explored can be found in T_{act} as discussed above since T_{act} is not empty. After the exploration of an unproductive node, one of the following situations may occur.

(1) T_{act} is not empty. The next node to be explored can be found in T_{act} as discussed above.

(2) T_{act} is empty but T_{next} is not empty. When this happens, the query processing moves to the next broadcast cycle. T_{next} contains all the information used in the following cycle. The client exchanges T_{act} and T_{next} and continues the query processing.

(3) T_{act} and T_{next} both are empty. The query processing stops since there is no more node to be explored.

One way to implement T_{act} and T_{next} is using a binary search tree. The operations on them include *Extract-Min*, *Search*, *Insert*, and *Delete*. Since the maximum number of p-elements in T_{act} or T_{next} is $\lceil \frac{n-1}{c} \rceil + 1$, each operation on T_{act} or T_{next} can be done in $O(\log \lceil \frac{n-1}{c} \rceil)$ time. Moreover, one can maintain the channel index set of a p-element by a heap. It will takes $O(\log c)$ time to obtain the smallest channel index from the channel index set. Therefore, the client needs $O(\log n)$ time to find the next node to explore in T_{act}.

A partial traversal of the index tree results in traversing many paths which start at the root and end at internal nodes or leaves. We consider two kinds of paths in a partial traversal of the index tree. One is the path ending at a root-node and the other is the path ending at a filler-node. Because the nodes in the subtree rooted at a root-node are arranged in preorder in the same channel, two (or more) different paths having a common ancestor which is a root-node can be traversed in the same cycle. We hence consider the paths ending at root-nodes. On the other hand, a search path may end at a filler-node u. In such a case, none of u's children is relevant to the query. We refer to a query processing as an m-search if the query processing results in a partial traversal which has m paths ending at a root-node and a filler-node as discussed.

In the rest of this section, we discuss the performance of the query processing in terms of number of cycles. We will show that a multi-dimensional query resulting in an m-search can be executed within at most $\min\{m, c\}$ broadcast cycles. We first claim the following lemma.

Lemma 4 *Any multi-dimensional query can be executed within c broadcast cycles.*

Proof: Due to the space constraint, please refer to [6] for more details.

□

The performance for an m-search is better when $m < c$. Consider a path ending at a root-node v. The nodes on the path from the root to v and in the subtree rooted at v can be traversed in one cycle by the ancestor property. The result also holds for the path ending at a filler-node. Therefore, at lease one path can be traversed in one cycle. Hence,

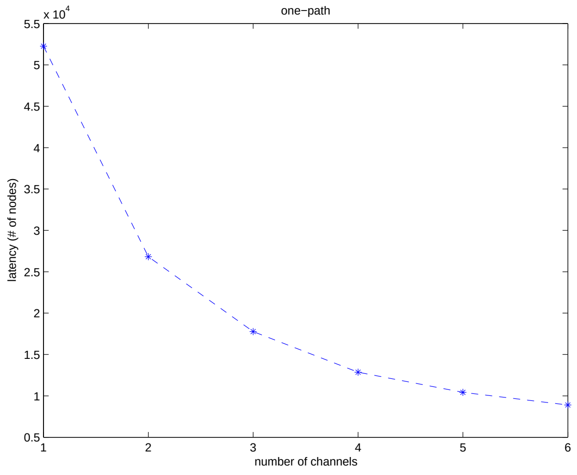


Figure 4: The average latency for a one-path search on an R*-tree of size 105,721 and having 100,000 leaves as well as degree 10.

a multi-dimensional query resulting in an m-search with $m < c$ can be executed within at most m cycles. We therefore have the following theorem:

Theorem 5 *A multi-dimensional query resulting in an m-search can be executed within at most $\min\{m, c\}$ broadcast cycles, where c is the number of channels.*

5 Experimental Results

In this section, we provide our simulation work. The multi-dimensional index tree considered here is an R*-tree. We point out that our protocol can be applied to any kind of multi-dimensional index tree, not only R- or R*-trees. We also assume that the clients always have enough memory to hold all the information during the execution of query. The power efficiency can be achieved using the index. We therefore focus on the latency in single- and multi-channel environments.

We consider synthetic data of rectangles for the time being. All the rectangles are generated using a uniform distribution with the unit square. For random rectangles, the centers of the rectangles are generated uniformly in the unit square and the sides of the rectangles are generated with a uniform distribution between 10^{-5} to 10^{-2}. Having the rectangle data set, we can build an R*-tree where the data are stored in the leaves. In our experiments, we consider the R*-trees which have 10,000 or 100,000 leaves, respectively. For each R*-tree, we generate the broadcast schedule on c channels where c is from 2 to 6. For the 1-channel broadcast protocol, we refer to [4] and compare the performance on latency with our multi-channel broadcast protocol. We use the number of nodes (packets) as our measurement. To measure the latency, we run 100(1000) different queries on the R*-tree with 10,000 (100,000) leaves and then take the average. We first show the result of one-path search and then the result of multi-path search. All the experiments show similar trends.

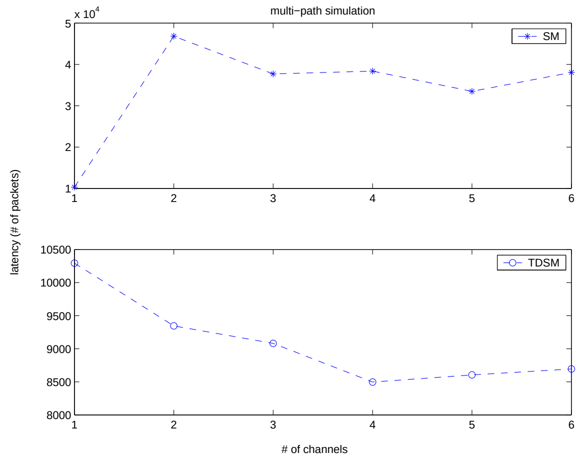


Figure 5: The average latency for a multi-path search using SM and TDSM respectively on an R*-tree of size 11,025 and having 10,000 leaves as well as degree 6.

Figure 4 shows the average latency for a one-path search on an R*-tree of size 105,721 and having 100,000 leaves. The degree of the R*-tree is at least 10. In the plot, the average latency decreases as the number of channels increases. Furthermore, the average latency is reduced with a factor of $\frac{1}{c}$.

We now discuss the average latency for a multi-path search using SM and TDSM respectively on an R*-tree. Although our multi-channel broadcast schedule satisfies the ancestor and switch properties as well as achieves the minimum cycle length, the resulting average latency for SM is not as good as the one in a 1-channel environment as shown in fig. 5. Using a stack will skip some nodes closer to the current explored node. This leads to finish the execution of query in more than c cycles. On the other hand, TDSM makes the result better. In all our experiments, TDSM has a better latency compared to a 1-channel environment.

6 Conclusions

We consider to generate the schedule by directly mapping the index tree into channels without considering the read-conflicts. Our schedule consists of two steps, channel assignment and node placement. For the channel assignment, the procedure focuses on which channel a node should belong. The node placement will place the node into the right position in the assigned channel to keep ancestor property. In order to make the algorithm easy to keep the ancestor and switch properties, the index tree should be rearranged by the subtree sizes. We show that our schedule satisfies the ancestor and switch properties by counting.

Besides, we provide the corresponding algorithms for executing query on the client side to integrate the broadcasting schedule and query process into a proper protocol. Having the protocol, we implement it and compare the performance with the protocols in a one channel environment in terms of latency. Our protocol not only achieves the optimum latency for a one-path search but also leads to a shorter latency for a multi-path search.

REFERENCES

[1] S. Acharya, M. Franklin, and S. Zdonik. Balancing push and pull for data broadcasts. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 183–194, May 1997.

[2] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 322–331, May 23-25 1990.

[3] M.-S. Chen, P.S. Yu, and K.-L. Wu. Indexed sequential data broadcasting in wireless mobile computing. In *Proceedings of the 17th International Conference on Distributed Computing Systems*. IEEE Computer Society Press, May 1997.

[4] S. Hambrusch, C.-M. Liu, W. Aref, and S. Prahakar. Query processing in broadcasted spatial index trees. In *Advances in Spatial and Temporal Databases - 7th International Symposium (SSTD 2001), Lecture Notes in Computer Science 2121*, pages 502–521. Springer-Verlag, July 2001.

[5] T. Imieliński, S. Viswanathan, and B. R. Badrinath. Data on air: Organization and access. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):353–372, May/June 1997.

[6] C.-M. Liu. *Broadcasting and blocking large data sets with an index tree.* PhD thesis, Purdue University, 2002.

[7] S.-C. Lo and A.L.P. Chen. Optimal index and data allocation in multiple broadcast channels. In *Proceedings of 2000 IEEE International Conference on Data Engineering*, pages 293–304, February 2000.

[8] N. Shivakumar and S. Venkatasubramanian. Efficient indexing for broadcast based wireless systems. *Mobile Networks and Applications*, 1(4):433–446, May/June 1996.

[9] K.-L. Tan and J. X. Yu. Generating broadcast programs that support range queries. *IEEE Transactions on Knowledge and Data Engineering*, 10(4):668–672, July/August 1998.

[10] K.L. Tan, J. X. Yu, and P.K. Eng. Supporting range queries in a wireless environment with nonuniform broadcast. *Data Knowledge Engineering*, 29(2):201–221, 1999.

[11] W.G. Yee, S. B. Navathe, E. Omiecinski, and C. Jermaine. Efficient data allocation over multiple channels as broadcast servers. *IEEE Transactions on Computers*, 51(10):1231–1236, 2002.