A Greedy Heuristic for the Data Layout Problem on Cache Oblivious Mesh Layouts using Redundancy

Zachary DeStefano, Shan Jiang, Gopi Meenakshisundaram
University of California, Irvine

June 7, 2014

Abstract

In Computer Graphics, the Data Layout Problem involves figuring out how to lay out the data units for a cache oblivious mesh layout in such a way that minimizes seek time required. Finding a deterministic solution to the problem is NP-hard so various heuristics have been proposed. In this paper, we present a solution to the problem based on introducing redundancy to the layout. It will first use the idea to get an optimal layout without adding extra units. It will then try to reduce seek time while using the least amount of redundancy possible. This will prove to be a better heuristic than existing ones proposed both analytically and experimentally. The use of redundancy will provide better seek time in both cases than the optimal solution without redundancy.

Introduction

The Data Layout Problem can be formulated as follows. The input is a linear sequence of data units. Each of the data units is assigned at least one color and many of them have multiple colors assigned. The length of a color is the distance from its first data unit to its last data unit. The data units can be rearranged as desired. The output we would like is the sequence of data units that will minimize the total length of all the colors.

In Yoon's paper **INSERT CITATION**, the Data Layout Problem is described as well as the metric that is motivating the above definition. We noticed that access patterns described in the paper can be very general and there could easily be data units that are far apart in the sequence but need to be accessed together. This led us to realize that if we copy data units and move them closer to other ones that share the same access pattern then we could save a lot of seek time without adding much storage space. The rest of this paper is about the algorithm developed to optimize seek time while minimizing the redundancy required to accomplish that.

Algorithm Description

The algorithm in Yoon et al. computes a locally optimal solution. Our algorithm is meant to take over after a locally optimal solution has been found or approximated. The basic idea of the algorithm is to take data units and copy them to a place that reduces seek time. If the old data units end up not being used then we delete them. Because there are cases where we would delete old units, the first loop in the algorithm is to take care of those cases first as it would reduce seek time without adding any redundancy. In order to get this algorithm to work, there are a few important issues to consider. We need to know which data units should be copied, where it should be copied to, which data units should be used by each access requirement, and which how many data units should be copied.

Which data unit

Since we only care about the length of each access requirement, we will only be copying the data units that are on the endpoints of an access requirement. This way we can reduce the overall seek time.

Where to locate data unit

We want to copy the data units to somewhere between the one after the first one and the last one. That way we are guaranteed to reduce the seek time for the access requirement we care about. For our own access requirement, it won't matter where in that interval we place our data unit. However, for the other access requirements, we are adding one unit of seek time since that data unit gets inserted. Therefore, we want to find which place will interrupt the least number of access requirements. We now have a problem of given a dynamic set of intervals (access requirement ranges in our case), and an arbitrary range, what is the least number of overlapping intervals in that range?

If we put at each slot the number of overlapping intervals, then this problem is just find the least number in an arbitrary range. For our purposes, we will do a linear search through each entry in the range in order to find the ideal entry. If k is the size of the access requirement, then this gives us $O(k)$ query time. Updates will also be $O(k)$ and construction will be $O(N)$ with N being the number of data units. There are other approaches, such as a range tree or dynamic programming, that may produce better query times, but their construction and update times will be worse as well as their storage.

With dynamic programming, we would have to maintain a matrix where an entry (i, j) would contain the minimum value in that range. This would give us a $O(1)$ query time but the construction and storage would be $O(N^2)$ where N is the number of data units. The update time would be $O(N)$ when we add a data unit. Since the N for this problem domain is in the hundreds of millions, that is an unacceptable storage bound. The construction run time would also be prohibitive given the magnitude of our input.
We could use a range tree. The initial binary search tree would be sorted by index and at each entry would be a pointer to a binary search tree sorted by value. If we put the min value at each of the nodes of the initial tree, we can speed up our queries. We would get a $O(logN)$ query time, but our construction time and storage would be $O(NlogN)$. Updating the data structure would take at a minimum $O(klog(N))$ time if we do careful indexing and only update the nodes that need to be updated. If we have a large access requirement, then this would represent a significant improvement in query time however given our exceptionally large input, the construction, storage, and update bounds are too prohibitive.

Which data unit is used by each access requirement

A given data unit will have a few access requirements attached to it. When you copy the data unit, it will move into a new position and benefit your current access requirement. It may benefit other ones too. For the other access requirements, if they will be shorter if they use this new copy then they should do that. Otherwise, they should stick to the old copy.

Number of data units to be copied

If we want to get the best seek time possible with redundancy, we will copy each data unit as many times as it has access requirements. We will then group all the access requirements into their own blocks as they will now have their own copies of the data that they need. Unfortunately, given memory restrictions, this is not always possible. In practice, the redundancy factor was around 20 when we did this.

For this paper, we tackle the problem of how do we optimize seek time given some limit on our redundancy.

Algorithm Pseudo-code

```
Main loop:
     Initialize benefitHeap
     for each accessRequirement P:
          add info (benefit, destination, chooseOldCopy, chooseNewCopy) from
      getARbenefitInfo to benefitHeap node
if chooseOldCopy is empty, add P to oldCopyList
     do the same for the tail node routine
     order benefitHeap using benefit
     while oldCopyList is not empty:
           take out random element and move the node to its destination
           update benefit info for affected ARs as well as oldCopyList
     while there exists more space for redundancy:
           pop best element from benefitHeap, getting node,length,destination
           destination object will tell us affected access requirements
                whose information needs to be updated
           copy the elements node to node+length to destination
           for every access requirement P from destination to destination+length:
                update benefit info in heap
          for every access requirement T in chooseNewCopy:
               update head,tail info on access requirement
           update head node to not contain the old access requirement
           if chooseOldCopy is empty
              delete node
                for every accessRequirement P from node to node+length:
                      update benefit info in heap
           reform Heap
```

```
//subroutine for finding best spot
position findMinOverlappingARs(rangeTree for Access Requirement data,
     minPoint, maxPoint):
     search range tree to get nodes between minPoint and maxPoint
     min = Infinity
     For each node:
          update min if its min is better than the current one
     return position corresponding to min

number getLengthOfAdjacentUnitsHead(AccessRequirement T)
     length = 0
     start = T.head
     end = T.head.next
     while end.position  start.position = 1
          length = length + 1
          start = start.next
          end = end.next
     return length,start,end

//subroutine for getting benefit of using AR
(benefit,destination,list chooseOldCopy, list chooseNewCopy) =
     getARbenefitInfo(accessRequirement P):
     //group adjacent units
     length,start,end = getLengthOfAdjacentUnitsHead(P)

     //this ends up being length between next data unit and the head.
     //   that is the potential that can be saved.
     potential = end.position  P.head.position - length

     //we are finding the best position between the "end" node and
     //     the tail of the access requirement.
     //putting the nodes before the end might reduce seek time,
     //     but we would reduce it better by putting it after the "end" node
     destination = findMinOverlappingARs(rangeTree, end.position, P.tail.position)
           -this should probably check both sides of the head/tail
           -make sure that destination stores the overlapping
                 Access Requirements, as
                 well as links and distinations to the next nodes for each AR
           -**TODO: Work out the details of this idea**
      benefit = potential
      //calculate cost to other ARs
```

```
 for each other AR uses the data units P.head to (P.head + length), T
      if P.head is T.head //in this case, they will both benefit
           T.length,T.start,T.end = getLengthOfAdjacentUnitsHead(T);
           //this gets us the benefit that we can guarantee, hence the min par
           benefit = benefit + min(potential, T.end.position  T.start.position
     add T to chooseNewCopy list
  //in this case, we will definitely want the old copy
  else if any of the nodes in P.head to (P.head+length) is T.tail
           add T to chooseOldCopy list
      else
           //decide which copy to use in T
           if P.head is T.head.next
           {
                 //difference1 is future benefit from using the new copy for T
                //use relative positions to get this number
                 difference1 = P.end.position  P.start.position
                 if destination > T.tail.prev.position
                       //difference2 is future penalty
                       //     from using the new copy for T
                       //don't use straight arithmetic,
                       //      but use the relative positions to get this numb
                       difference2 = destination + length - T.tail.prev.positi
                       //TODO: Figure out whether to
                       //     consider P.length or T.length
                 else
                      difference2 = 0

                 if difference1 > difference2
                       add T to chooseNewCopy list
                 else
                       add T to chooseOldCopy list
       }
             else
                  add T chooseOldCopy list

if chooseOldCopy list is empty
     discard P.head
     for each AR, A
     if A.head.position < P.head.position AND A.tail.position > start.postion
          //by discarding, for each AR covering it,
          //     this is the benefit of deleting that data unit
```

```
benefit = benefit + P.length
```

**TODO: In the algorithm description, for the part where we decide whether to keep the old or new copy, just say that we are choosing the copy that will keep the AR shorter. Similarly, use that description in the other parts of the algorithm.

Run-time and Storage Analysis

Parameters

- n: number of ARs
- m: average number of overlapping ARs
- N: current total units
- L: length of AR
- Q: number of runs of redundancy loop

Initial Construction

Being run on n access requirements
Takes O(N) operations to get all the initial AR data
For each AR, there are log(n) operations to insert data into heap
Let T_1 be running time to get the min number of affected access requirements
For each AR, there are $O(n + T_1)$ operations to get benefit info
There are T_2 operations to construct data structure for mincut operation
TOTAL: In worst case, initial construction takes $O(N + n^2 + nT_1 + T_2)$

Redundancy loop

Popping and copying takes O(1) operations
There are O(n) access requirements affected by the operation
Updating relative positions will take O(n) operations
Reforming the heap is O(n log n) operations
There are T_3 operations to update the data structure for mincut operation
TOTAL: In total, we have $O(Q * (nlogn) + Q * T_3)$ operations

Total Running Time

As stated above, there are $O(N + n^2 + Q(nlogn) + n * T_1 + T_2 + Q * T_3)$ operations. This proves that we have a quadratic time algorithm for finding a good layout.

Benefits over Existing Algorithms

The first part of the algorithm will produce a better solution than proposed by Yoon without adding extra units, thus it is an improved solution to the Data Layout Problem. Here is an example layout produced by Yoon's algorithm:
*INSERT PICTURE OF LAYOUT PRODUCED BY YOON**
With our algorithm we will produce the following layout:
*INSERT PICTURE OF OUR NEW LAYOUT**

Existing algorithms do not consider redundancy. Even if we find a polynomial solution to the data layout problem, we can actually achieve a seek time better than the optimal one without redundancy. Here is a case where that happens:
*INSERT PICTURE OF DATA LAYOUT**
Without redundancy, this is the optimal solution:
*INSERT PICTURE OF LAYOUT WITHOUT REDUNDANCY**
With redundancy, this is the result
*INSERT PICTURE OF LAYOUT WITH REDUNDANCY**

Experimental Results

INSERT THE INFO FROM SHAN'S THESIS

Conclusions and Future Work

We have shown that we have a quadratic time algorithm with quadratic storage space for the Data Layout Problem. It achieves significant results analytically and experimentally.

THINK MORE ABOUT POTENTIAL FUTURE WORK

Acknowledgments

References