A Redundancy-based greedy heuristic for the Data Layout Problem on Cache Oblivious Mesh Layouts

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

May 16, 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 that involves redundancy. We will describe the algorithm in detail as well as its running time and storage requirements. We will then show that in many cases we will get a better seek time than the best one without redundancy while not using much extra space. We will also show that our algorithm obtains a better seek time experimentally than one that does not use 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. 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. The section, min cut approaches, will detail various algorithms for this problem.

Which data unit is used by each access requirement

Number of data units to be copied

Min cut approaches

INCORPORATE THIS PART INTO THE ALGORITHM DESCRIPTION AND THEN ANALYSIS SECTIONS

Range tree approach with data units

Each node will have two bits of info (position,numARs)
This info will be organized into a range tree
Construction will be O(N log N), so $T_2 = NlogN$
Query will be O(log N) if constructed so node contains min in other dimension
thus $T_1 = logN$
Updating the data structure in redundancy loop will be logN operations, thus
$T_3 = logN$
New TOTAL run time: $O(N + n^2 + Q * (logN + nlogn) + (n + N) * logN)$

Segment tree approach with AR info

Each AR is an interval
This info will be organized into a segment tree
Construction will be O(n log n)

Query will be O(log n)
*Work out the details of this idea**
Update to n ARs would be affected by an update, so O(n) update time
In summary, $T_1 = log(n), T_2 = nlogn, T_3 = n$
New TOTAL run time: $O(N + n^2 + Q(nlogn))$

Linear search approach

Each node will still have those two bits of info from range tree approach
A query will linearly search through the nodes
Construction will be O(N) and won't add anything to total, since $T_2 = N$
Query will be O(L), so $T_1 = L$
Updating the info could take up to n*L operations
- this is due to the fact that for each new copy AR, you have to update the number of overlapping ARs for the seek block affected
New TOTAL run time: $O(N + n^2 + Q(nlogn) + nL + Q * n * L)$

Dynamic Programming approach

Store a matrix where each entry (i,j) contains the min from unit i to unit j
It would require $O(N^2)$ storage, which is unusable in our case
Construction would be $O(N^2)$
for every data unit i:
for j from i+1 to N:
matrix(i,j) gets min of matrix(i,j-1) and node j
Query time would be O(1)
Update time would be $O(N^2)$
In summary, $T_1 = 1, T_2 = N^2, T_3 = N^2$
New TOTAL run time: $O(QN^2 + Q * (nlogn) + n^2)$

Algorithm Pseudo-code

```
//Data Structures

accessRequirement:
- head, tail
```

```
dataUnitGroup:
- set of data units that are adjacent
- length will also be a field

dataUnit:
- prev, next
- forEachAR:
-prev, next with same AR
-dist to prev,next with same AR
- ID
- no position parameter as we will just store relative positions

destinationDataUnit:
- subclass of dataUnit
- store dist to prev,next data unit with each AR overlapping

range tree for access requirements:
- binary search tree of indices in first level
- in next level are the endpoint positions

Main loop:
Initialize benefitHeap
for each accessRequirement P:
add info (benefit, destination, chooseOldCopy, chooseNewCopy) from getARbenefitInfo
do the same for the tail node routine
order benefitHeap using benefit
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 need
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, max
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 //seems to measure number of adjacent units
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(acc
//group adjacent units
length,start,end = getLengthOfAdjacentUnitsHead(P)
//this ends up being length between next data unit and the head. that is the potent
potential = end.position  P.head.position - length
//we are finding the best position between the "end" node and the tail of the acces
//putting the nodes before the end might reduce seek time, but we would reduce it b
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 part
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 number
difference2 = destination + length - T.tail.prev.position
/* 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
benefit = benefit + P.length
```

Run-time and Storage Analysis

```
Initial Analysis:
- 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 analysis:
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 + n T_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*(n log n) + Q*T_3) operations
TOTAL Running time:
As stated above, there are O(N + n^2 + Q(n logn) + n*T_1 + T_2 + Q*T_3 ) operations
```

Benefits over Existing Algorithms

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**

Moving a data unit is really just duplicating it to a new position and deleting the old unit. In many cases, we end up doing that in our algorithm.

Experimental Results

INSERT THE CHARTS FROM SHAN'S THESIS

Conclusions and Future Work

Acknowledgments

References