Chapter 6

Cache-Oblivious Mesh Layout and Graph-based Solution

In the previous chapter, we discuss optimization-based solution of the seek time problem and solve it with linear programming. We conclude that this method has two major drawbacks: first, it only clusters data but does not provide information how clustered data should be laied on the disks; Secondly, since the system is blind to the actual seek time of each seek, the total seek time is not well calculated. This fact can lead to unexpected result such as single-seek sequences stored contiguously will still be considered as multiple seeks rather than one.

In this chapter, we propose a graph-based solution inspired by Yoon et al. [45]. Through this solution, we address both of these two problems.

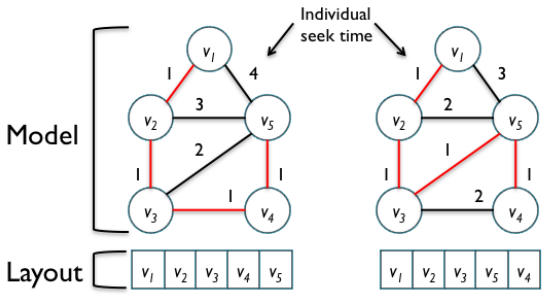

Figure 6.1: Two data layouts of one model with 5 data units are shown. The layout is indicated with red edges. Numbers on edges are seek time. By switching the order of v_4 and v_5, the total seek time is reduced.

6.1 Cache-Oblivious Mesh Layout

In order to integrate the processes of creating data layout and building redundancy to reduce seek time, we need to find a data layout model that can minimize seek time regardless of redundancy. We can then manipulate it by duplicating data units.

Yoon et al. [45] worked on a cache-oblivious mesh layout that is successfully minimizing cache misses without awaring of cache parameters and run time data accessing patterns. The basic idea is by defining the term *edge span* and providing a probability model, they calculated expected cache misses (ECM) and used it or its approximation as metric of evaluating a data layout. After a hierarchical exhaustive comparison, the result is a near optimal data layout that minimizes cache misses.

We follow the approach by first constructing a graph in which each vertex represents a data unit of the model. An edge exists between two vertices of the graph if their representative data units are accessed by the same access requirement or they are spatially adjacent to each other. We then replace edge span with *individual seek time* and ECM with expected seek time (EST). The definition of individual seek time between data units v_i and v_j in a data layout is absolute value of the distance from v_i to v_j. We use S_l to denote the set that consists of all the seeks of seek time l, where $l \in [1, n-1]$. This is essentially equivalent to

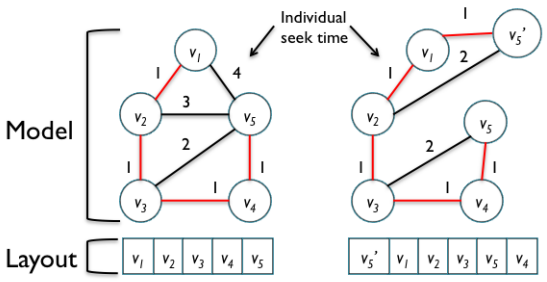

Figure 6.2: Adding redundancy to cache-oblivious mesh layout

edge span. After this modification, the Figure 3 of [45] will appear as Figure 6.1, in which examples of graphs and their corresponding layouts are shown. We define EST as:

$$EST = \sum_{i=1}^{n-1} i \frac{|S_i|}{|S|} \tag{6.1}$$

where $|S_i|$ is cardinalitys of S_i and $|S|$ is the number of edges. EST is actually an approximation of ECM. Since [45] minimizes ECM, it minimizes EST as well. To evaluate if one layout is better than another one respect to seek time, we just check if ΔEST is positive or negative.

6.2 Minimize Expected Seek Time with Redundant Data Units

Cache-oblivious mesh layout minimizes the EST given the condition that no data units are duplicated, i.e. redundancy free. To further reduce EST, we need to introduce redundancy to the layout. The method is straight-forward: we choose one data unit, copy it to a position such that the ΔEST is negative. Consider the example in Figure 6.1, instead of switching v_4

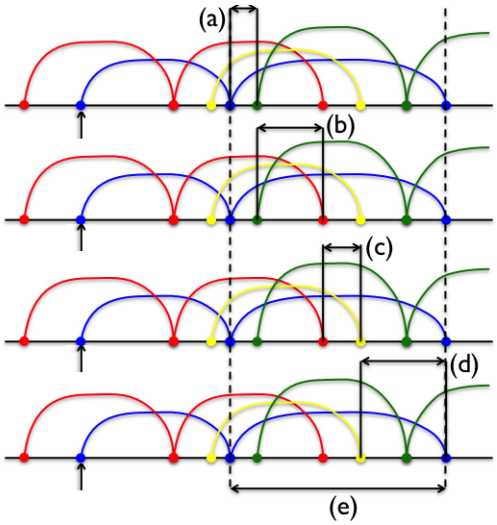

Figure 6.3: Access requirement graph. Each color indicates a different access requirement.

and v_5, we copy v_5 and insert its duplication v_5' next to v_1. Figure 6.2 shows the result and how the seek times are reduced. So far we have introduced the way we add redundancy to cache-obliviously mesh layout. However, to have a concrete algorithm, there are still many questions to be answered. In the rest of this section, we will discuss the decisions to make for each one of them.

Decide the destination of a copy: Assume we have one data unit we plan to duplicate to optimize our layout, one immediate question is where it should be copied to. To answer this question, let us consider the example in Figure 6.3. The black axis is the data layout, and each point on this axis is a data unit. Curves with same color indicate edges from the same access requirement. We want to reduce the seek time of the blue AR by copying the left most data unit of the blue AR, which is marked by arrows, and inserting it to somewhere of the layout. If we only consider the blue AR, inserting this data unit to the range (e) is the best solution obviously. By doing it, we completely eliminate the first part of seek time from the blue AR. Inserting the data unit to range (e) will increase seek time of represented by range (e) by one, which is a modest cost comparing to the benefit we gain. Note that this cost is irrelevant to which exact position the data unit will be inserted to. However, when we consider other ARs as well, depending on which part of (e) we insert this data unit to,

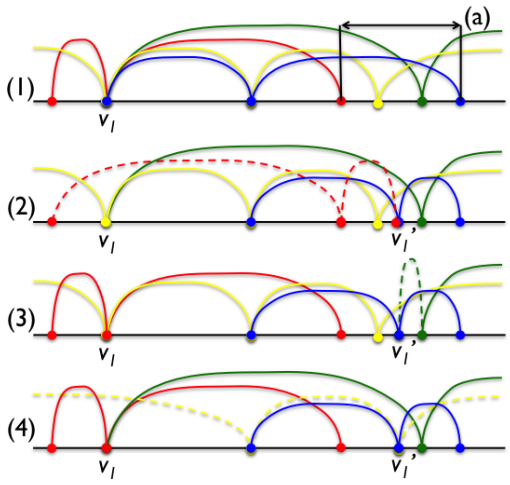

Figure 6.4: Different choices of updating access requirement after insertion. (1) is the initial state. (2) (4) are states after inserting v_1', and corresponding to the red, green, and yellow AR respectively.

the cost can be varied. Within range (a), there are the red and yellow ARs besides the blue one, so the cost will be three. Within range (b), there are the green AR as well, so the cost will be four. Within range (c), the cost will be three. And within range (d), since only the green AR is there besides the blue one, the cost will be two. As we can see, range (a) to (d) provide same benefit, but range (d) has the least cost, so range (d) is our answer for this example.

The process of finding the solution is essentially finding the min-cut of the access requirement graph, where the min-cut means the minimum number of ARs will be affected by the insertion. Thus, to find the optimal range to insert a copy of a data unit, we follow the rules: first, find the range maximizes the benefit regarding the AR we focus on; Then, find the min-cut range within this range which minimizes the cost.

Decide how to update access requirements: There are side effects of inserting duplicated data units. One fundamental problem is after copying one data unit, there are two identical units on the layout now. Each access requirement has access to this data unit has choose which one to be included exclusively to avoid rendering the same data twice while

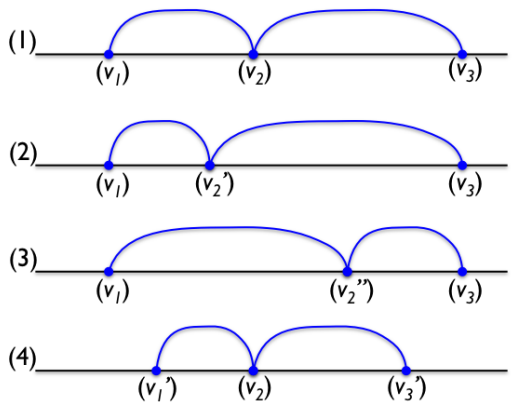

Figure 6.5: Find the effective candidate of data unit to copy.

run time. Figure 6.4 is an example of this choice. Diagram (1) shows the initial state of the layout. We are focusing on the blue AR, and decide to copy data unit v_1 to the range (a) to reduce seek time for the blue AR. The problem rises since the red, green, and yellow AR also have access to v_1. In diagrams (2) (4), we try to use the new inserted v_1' for each one of them. Affected seeks are marked with dash lines. As we can see, by including v_1' instead of v_1, the red AR increases the total seek time, the green AR reduces the total seek time, while there is no different for the yellow AR. This observation provides us the intuition of rules to decide how to update access requirement: if by including the inserted version of data unit, an AR can reduce its seek time, then we choose to do that; otherwise, if no improvement of seek time is found or the seek time increases, then discard this option for this particular AR.

Decide which data unit to copy: So far we have assumed that the data unit to be copied has been given. However, it is not necessary to copy and insert every single data unit blindly. Identifying which data units can affect the EST effectively is equally important to the rest of algorithm. We consider Figure 6.5 as an example to help us find out the sound solution for this issue. Diagram (1) is the initial condition of the data layout. There are three candidates data unit to be considered for reducing seek time of the blue AR. Diagram (2) and (3) show that having a new position for data unit v_2 will not reduce the total seek

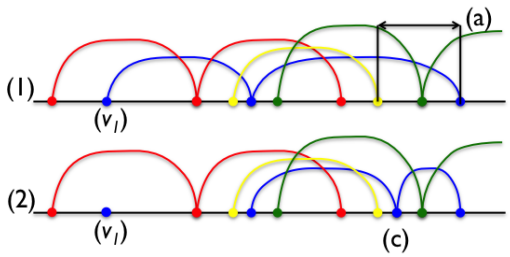

Figure 6.6: Case when discarding a data unit from the layout is possible.

time for this AR, while diagram (4) suggests both v_1 and v_3 can be qualified candidates. The essential difference between v_1, v_3 and v_2 is v_1 and v_3 are end points of the blue AR, while v_2 is a middle data unit. Thus, the rule we should follow is: only consider end points of any AR for copying.

Decide in which order data units are copied: Each AR has two end points. And since we have many ARs, it is clear that we have a good amount of candidates to be copied. Under this condition, in which order to copy these end point units is important to minimize the EST as efficient as possible respect to redundancy factor. We use the following algorithm: 1. find all end points; 2. find the optimal destination for each end point based on the rules suggested by Figure 6.3; 3. calculate the difference between benefits and costs for each end point and its proposed destination; 4. push all end points to a heap based on the difference, the top of the heap has the largest difference, which means maximized benefit over cost, and it will be the one to be executed first; 5. after popping the top of the heap, update benefit-cost difference for each end point in the heap and push new end points if applicable; 6. repeat from step 4 until the EST cannot be further reduced. For the concern of cost, we have already discussed it in Figure 6.3. For the concern of benefit, so far we have seen two ways to gain it: first, the direct reduced seek time in Figure 6.3 for the focused AR; and the direct reduced seek time for the ARs affected (the green AR in Figure 6.4). However, there is indirect benefit we have not yet met. Figure 6.6 shows this case. In diagram (1), we identify v_1 should be copied and inserted to range (a) to reduce seek time for the blue AR. Diagram (2) is the result, where v_1's copy c is inserted and the seeks of the blue AR are updated.

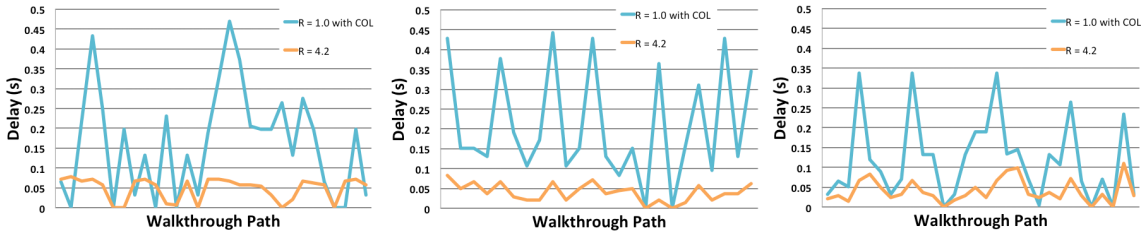

Figure 6.7: Statistics of delays caused by the fetching processes for the Sparse City model (left), the Boeing model (center), and the Dense City model (right), with and without redundancy.

After this operation, however, v_1 is no longer being access by any of AR, which makes it a useless data unit on the layout. In cases like this, we need to delete v_1 from our data layout. It does not only provide extra benefit, since the seek time of the red AR is reduced by one, but also makes this copy-insert operation of v_1 to be free of cost of redundancy.

6.3 Results and Analysis

We implement this algorithm with the same machine we use for the previous chapters and do the experiments on the same models under the same settings. Figure 6.3 shows the results of delays on the three models. This time, we compare the results of cache-oblivious layout without redundancy and the layout we implement with redundancy factor is equal to 4.2. It is clear that the orange curve representing the performance of the layout with redundancy has generally shorter delays than the cyan curve which represents the cache-oblivious layout without redundancy. Note that, although the layout with redundancy does not eliminate delays for most of sample points on the walkthrough path, it reduces delays to a small range and keeps the performance much more consistent. This is the benefit we get from the algorithm we add redundancy. Since the algorithm tend to eliminate seeks with longer seek time first, it practically helps to avoid large delays.

There is another major benefit this graph-based approach brings. Since each time we only

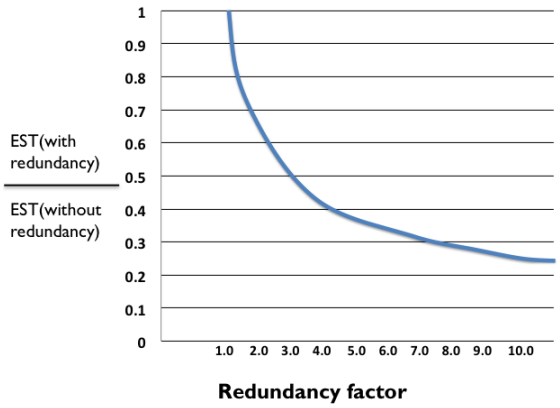

Figure 6.8: Plot of the ratio of the EST of layout with redundancy over the EST of cache-oblivious mesh layout withou redundancy.

duplicate one data unit, we can start the process and halt it any time the redundancy factor reaches a certain threshold. This benefit helps us to create data layout with arbitrary redundancy factor without worrying about adjusting cell size and other parameters as we did in the previous chapter. In Figure 6.8, we show a plot of layouts under redundancy factor range from 1.0 to 10.0. The y-axis in this figure is the ratio of the EST of the layout with redundancy over the EST of the layout without redudancy. This value starts at 1.0 where redundancy factor is 1.0, and decreases as redundancy factor goes larger. We can see that the rate of this decrement is not constant, and the benefits we gain at beginning is more obvious than later. It is clear that the most improvement of the performance resides at the earlier part of raising redundancy factor, which implies that it is not necessary to go all the way down to single-seek layout to have significant boost of consistency of performance. At this point, we finally explore the entire solution spectrum.