```
Input: Data units and their access requirements (AR) ;
for start and end unit of each AR i do
    Find optimal location $j$ for copy;
    Calculate $\Delta EST_M(i,j)$ and insert into $E_M$ ;
    Calculate $\Delta EST_C(i,j)$ and insert into $E_C$ ;
end
while true do
    while top element of $E_M$ is positive do
        Pop top element and move the data unit to its destination ;
        Update $E_M$ and $E_C$ ;
    end
    if there is more space for redundancy then
        Pop top element and copy the data unit to its destination ;
        Update $E_M$ and $E_C$ ;
    else
        break
    end
end
```

Algorithm 1: Pseudo-code for our algorithm

$\sum_{t \in T} \Delta A_t$ and k_{old}. But the main advantage of moving instead of copying is that this operation does not increase the redundancy thus it keeps the storage requirement the same. So we perform moving instead of copying as long as $\Delta EST_M(i,j)$ is positive.

Data Unit processing order: We now need to figure out how to use this information to decide in what order the copying and moving should be done. We will make two heaps: E_M and E_C. The E_M heap will organize the move operations and consist of the values of $\Delta EST_M(i,j)$ for the start and end data units for all access requirements i where the units are put in their optimal location j. The E_C heap will be the same thing except it will organize the copy operations and consist of the values of $\Delta EST_C(i,j)$.

We process the E_M heap first as long as the top of the heap is positive and effect the move of the data unit at the top of the heap. After each removal and processing, ΔEST_M and ΔEST_C of the affected access requirements and the corresponding heaps are updated. If there are no more data units where ΔEST_M is positive, then one element from the top of the heap E_C is processed. After processing and copying a data unit from the top of heap E_C, the heaps E_C and E_M are again updated with new values for the affected access requirements. If this introduces an element in the top of E_M heap with positive values, the E_M heap is processed again. This process gets repeated until the user defined bound on redundancy factor is reached. As a summary, the psuedo-code of this algorithm is shown as ~~algorithm~~ 1.

4 Complexity Analysis

We now analyze the running time and storage requirements of our algorithm. Let N be the number of data units and A be the number of access requirements. We will use k as the average span of a single access requirement. Let r be the redundancy factor limit specified by the user so that $O(rN)$ units can be copied. For the sake of analysis each data unit will ~~have~~ $O(A)$ access requirements ~~attached to it~~ and $O(A)$ access requirement ~~spans that overlap it~~ even though the actual number will likely be lower in both cases.

Time Complexity: The construction of the heaps E_M and E_C involves computing the benefit information for all A access requirements and inserting each one into the heap. For a single access requirement, computing the benefit information of moving or copying one of its data units involves scanning each data unit in its span. ~~We justify this approach which takes $O(k)$ operations in Appendix A.~~ Calculating $\sum_{s \in S} \Delta A_s$ and $\sum_{t \in T} \Delta A_t$ will take $O(A)$ operations since there are $O(A)$ access requirements to potentially have to sum over. Inserting this benefit information into the heap takes $O(log(A))$ operations. In total then it takes $O(k + A + log A)$ or $O(k + A)$ operations per access requirement to get the benefit information. The initial construction thus takes $O(A(k + A))$ operations.

After the initial construction, the move and copy loops are executed. In every iteration of move or copy, an element from the top of the heap is removed and processed, the benefit function is recalculated for affected access requirements, and the heap is updated. There are potentially $O(A)$ overlapping access requirements whose benefit information needs to be recalculated. As shown above, for each of these access requirements $O(k + A)$ operations are required to perform the recalculation and update the heap. Each iteration of move or copy thus takes a total of $O(A(k + A))$ operations.

For simplicity we will assume that the move loop runs $O(N)$ times total. This comes from the fact that the cache oblivious layout [Yoon et al. 2005] should be a good approximation so the number of moves that would be useful should be limited. There are $O(rN)$ copies made so there are that many iterations of the copy loop. We thus can assert that there are $O(rN + N)$ iterations of the move or copy loops. We can simplify this to $O(rN)$ operations since $r \geq 1$. In total then the moving and copying loops will take $O(rNA(k + A))$ operations, which is also the running time for the whole algorithm.

Space Complexity: During the run of the algorithm, we have to store the number of overlapping spans at each data unit, which will require $O(N)$ storage. We will also have to store a heap of access requirements, which can be stored using $O(A)$ space. We also have a list of access requirements and that information will take up $O(A)$ space. In total we thus have $O(A + N)$ storage space used during the run of the algorithm.

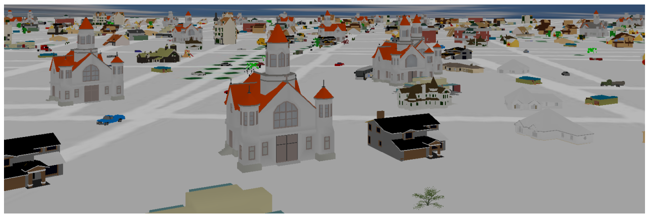

Figure 4: *City model: 110 million triangles, 6 GB*

5 Experimental Results

Experiment context: In order to implement our algorithm, we used a workstation that is a Dell T5400 PC with Intel (R) Core (TM) 2 Quad and $8GB$ main memory. The hard drive is a 1TB Seagate Barracuda with 7200 RPM and the graphics card is an nVIDIA Geforce GTX 260 with 896 MB GPU memory. The data rate of the hard drive is 120 MB/s and the seek time is a minimum of 2 ms per disk seek.

Benchmarks: We use three models to perform our experiments, each model represents a use case or scenario. The City model (Figure 4) is a regular model that can be used in a navigation simulation application or virtual reality walkthough. The Boeing model (Figure 1), on the other hand, represents scientific or engineering visualization applications. The Urban model (Figure

2) has texture attached to it, which is commonly used in games. By comparing performance of cache-oblivious layout without redundancy [Yoon et al. 2005] to our method using redundancy on these three models, our goal is to show that the redundancy based approach can achieve more stable and generally better performance on different real time applications.

To apply our method on these large-scale models, we had to find a proper set of access requirements. In general, that question is deep enough that it can be discussed as a separate research topic. Here, however, we had good performance using only simple schemes for creating access requirements. Each model ended up having a separate scheme. Nonetheless, an access requirement represents a set of data that is highly likely to be accessed together.

For the Boeing model, the predefined objects are used as a conceptual level to create access requirements. Samples of view positions are distributed across the model. For each sample, four fixed directions and four random directions are considered. Objects visible from this position in any one of these eight directions are added to access requirement for this specific sample. The density of these samples depends on the complexity of local occluders to reduce load of each access requirement, i.e. more samples are distributed to places with more complex geometry.

The Urban model is different from the previous two in a way that it involves textures. Building heavy redundancy of textures increases the total size of the dataset significantly, while keeping textures away from redundancy leads to inevitable long seek time, which is completely against the philosophy of this work. To solve this problem, we applied a spatial Lloyds clustering [Lloyd 1982] on objects. By moving centers of clusters, we look for a solution such that each cluster involves almost same amount of texture data. Between clusters, textures can be redundantly stored, but within each cluster, texture data are stored uniquely. In this way, each cluster is used as an access requirement.

For the City model, a 2D grid is used to divide the space into square cells. For each pair of adjacent cells, the difference of the data is considered as an access requirement. By propagating this rule, access requirements are created, and the number of them is determined by the resolution of the grid [Jiang et al. 2013].

The computation time to create redundancy layout is generally linearly correlated to the final redundancy factor for each model. For the examples we used in Figure 5, it took 16 minutes to create the redundancy layout from the cache-oblivious layout without redundancy for the City model. This number is 80 minutes and 38 minutes for the Boeing model and the Urban model, respectively.

Results: Figure 5 shows the results of delays caused by fetching data on the experimental models we used. We compare the results of a cache-oblivious layout without redundancy and one with redundancy. For the layout with redundancy, we set the redundancy factor equal to 4.2. This factor was chosen because as can be seen in Figure 6, it had considerably better performance than lower redundancy factors and did not have significantly worse performance than higher redundancy factors. A factor of 4.2 is also still practical, as the largest model we tested, the Boeing model, becomes 84 GB which is still acceptable given the large capacity of modern secondary storage devices.

It is clear that the performance of the layout with redundancy has generally shorter delays than the cache-oblivious layout without redundancy. As can be observed from the results, 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 more consistent. ~~This is the benefit we get from using our algorithm which adds redundancy.~~ Since the algorithm tends to eliminate seeks with longer seek time first, in practice the larger delays are avoided.

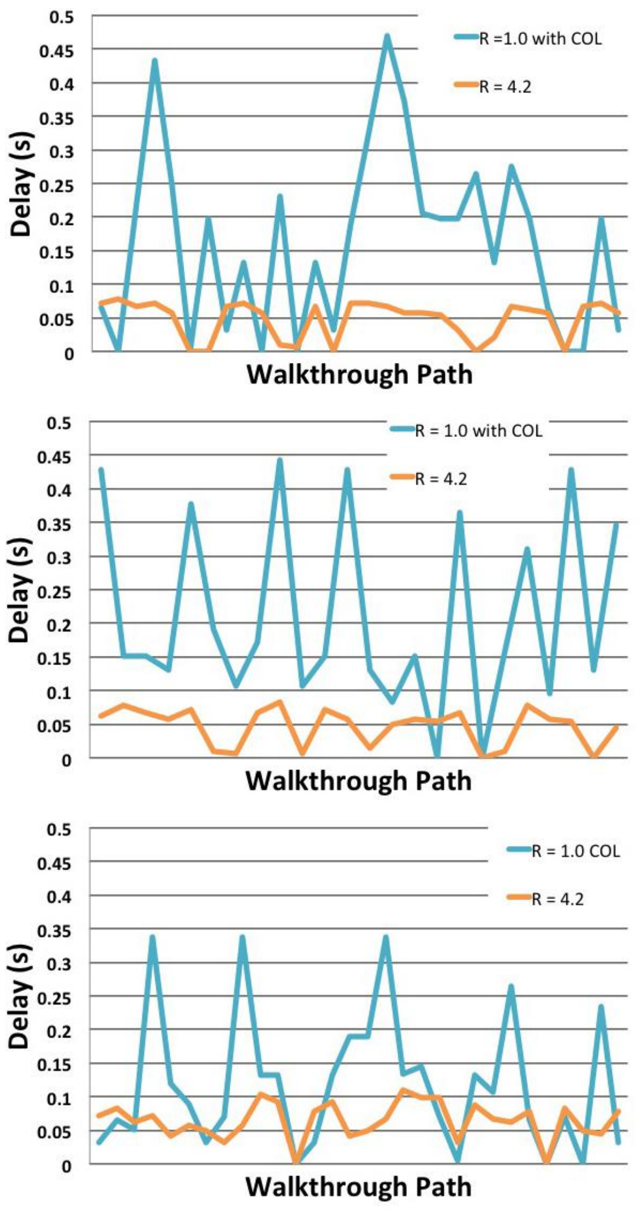

Figure 5: *Statistics of delays caused by the fetching processes for the City model (top), the Boeing model (center), and the Urban model (bottom), with and without redundancy. COL indicates a Cache-Oblivious Layout that does not use any redundancy. R indicates the redundancy factor.*

When we compare our method to the one in [Jiang et al. 2014], there is a performance benefit that can be observed in Figure 7. Additionally in [Jiang et al. 2014], the user does not have any control over the final redundancy factor however each time we duplicate one data unit, we can halt it if the redundancy factor reaches a certain threshold. This helps us create data layouts with arbitrary

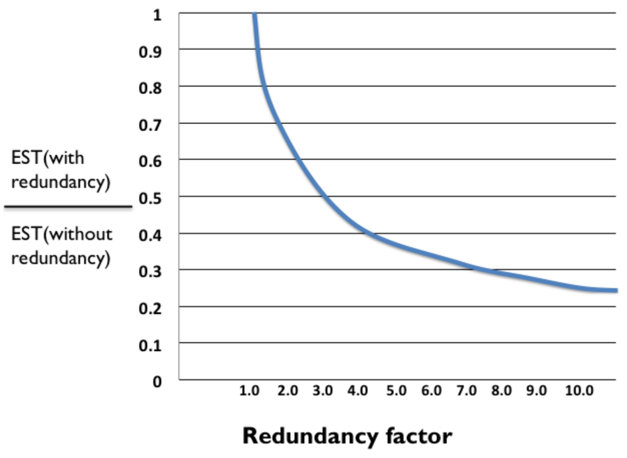

Figure 6: *Plot of the ratio of the EST of layout with redundancy over the EST of cache-oblivious mesh layout without redundancy.*

redundancy factors. We use this fact to test different redundancy factors ~~and see their results~~. In Figure 6, we show the results of using layouts with redundancy factors that range from 1.0 to 10.0. The ~~y-~~axis in this figure is the ratio of the estimated seek time (EST) of the layout with redundancy over the EST of the layout without redundancy. This value starts at 1.0 where redundancy factor is 1.0, meaning no redundancy, 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 are larger than the ones we get later. ~~This implies that most of the performance improvement resides at the earlier phase of raising redundancy factor. This implies that~~ it is worthwhile to limit the redundancy factor used because after a certain point ~~you are using~~ much more secondary storage space without ~~improving seek time by much~~. It also implies that our algorithm dramatically reduces seek time in practice by using only small redundancy factors.

6 Analysis and Comparisons over Prior methods

In the algorithm, we make a heap of data units that will reduce seek time by just moving instead of copying them. We perform these moves first before working with data units that need copying. This initial step will produce a better solution than proposed by [Yoon et al. 2005] without adding redundant units. This result is possible mainly because our optimization algorithm searches wider sets of potential locations for moving cases in an efficient manner. To show this, consider a case where we have two access requirements of 5 data units each. Figure 8 shows an example of that kind of layout. In the middle of that figure is the result of using the cache oblivious layout. Because it hierarchically constructs blocks and arranges the units in each block, it does not detect that the units with the black access requirement can be grouped together. On the other hand, the algorithm we propose would shorten the black access requirements without adding redundancy, as shown in the bottom of that figure.

The algorithm in [Yoon et al. 2005] did not necessarily produce the best cache oblivious mesh layout. However, even if we had the best layout without redundancy, we would actually achieve a better seek time than it using redundancy. We have such an example with Figure 9. As can be seen in the figure, the total seek time is 7 units which turns out to be the minimum possible seek time without redundancy, as found through a brute-force search. With redundancy, the total seek time is the minimum required which is 6

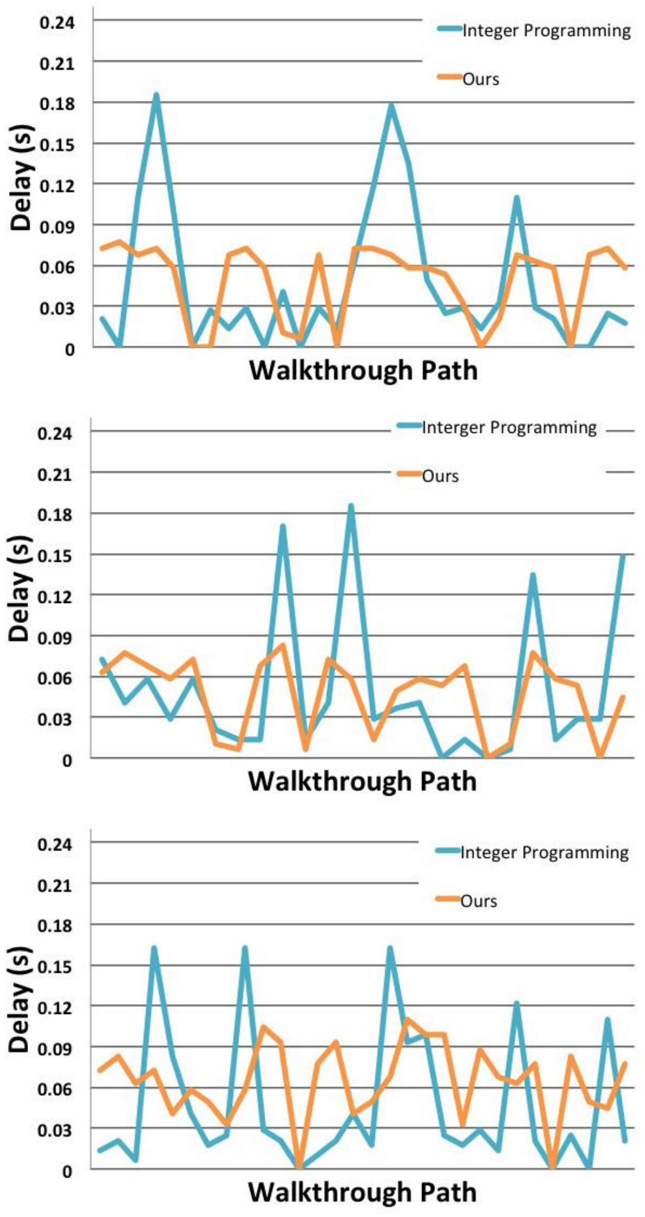

Figure 7: *Statistics of delays caused by the fetching processes for the City model (top), the Boeing model (center), and the Urban model (bottom), using integer programming and our method.*

units. While a reduction from 7 to 6 units may not seem dramatic, when this result is scaled up to the hundreds of millions, this makes a big difference in seek time, which we saw in practice.

7 Conclusion and Future Work

Given the data units, access requirements, and the desired upper bound on redundancy factor, we have proposed an algorithm that would create a cache oblivious layout with the primary goal of reducing the seek time through duplicating the data units. We proposed a cost model for estimating the seek time, and in our algorithm we can move or copy data units in appropriate locations such that it reduces the estimated seek time. We have shown