# Performance Driven Redundancy Optimization of Data Layouts for Walkthrough Applications


## Abstract

Performance of interactive graphics walkthrough systems depend on the time taken to fetch the required data to render from the secondary storage to the main memory. It has been earlier established that a large fraction of this fetch time is spent on seeking the data on the hard disk. In order to reduce this seek time, redundant data storage has been proposed in the literature, but the redundancy factors of those layouts are prohibitively high. In this paper, we develop a model for the seek time of a layout, and using this cost model, we propose an algorithm that would compute a redundant data layout with the redundancy factor that is within the user specified bounds while maximizing the performance of the system.



**CR Categories:** I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types;

**Keywords:** Data Layout Problem, Out-Of-Core Rendering, Cache Oblivious Mesh Layout, Redundant Data Layout, Walkthrough Application


## 1 Introduction

### 1.1 Walkthrough Rendering

In typical walkthrough systems, data sets consisting of hundreds of millions of triangles and many gigabytes of associated data (e.g. walking through a virtual city) are quite common. Rendering such massive amounts of data requires out-of-core rendering algorithms that bring only the required data for rendering into main memory from secondary storage. In this process, in addition to the rendering speed, the data fetch speed also becomes critical for achieving interactivity, especially when we handle large-scale data. In general, data fetch speed depends on data seek time and data transfer time. Transfer time depends only on the amount of data that is transferred. Seek time is the time taken to locate the beginning of the required data in the storage device and depends on different factors depending on the storage medium.

For a hard disk drive (HDD), its seek time depends on the speed of the disk, and the relative placement of the data units with respect to each other, also called the data layout [Rizvi and Chung 2010]. For a solid state drive (SSD), this seek time is usually a small constant and is independent of the location of the data with respect to each other [Agrawal et al. 2008]. An earlier work utilized this difference between SSD and HDD and designed a data layout tailored for using SSDs with the walkthrough application [Sajadi et al. ]. There have been many other techniques utilizing SSDs for various applications [Saxena and Swift 2009]. SSD, unfortunately, is not the perfect data storage and has its own technical problems, including limited number of data overwrites allowed, high cost, and limited capacity [Rizvi and Chung 2010].

On the other hand, the HDD technology – including disk technologies such as CDs, DVDs, and Blu-ray discs – has become quite reliable and inexpensive thanks to their extensive verifications and testing, and is thus in widespread use. Even for massive data sets hard disk drives (HDD) are still and will be the preferred medium of storage for the foreseeable future [Rizvi and Chung 2010], mainly because of its stability and low cost per unit. As an example, according to [Domingo 2014], as of 2014, an HDD can cost $0.08 per GB, while an SDD can cost $0.60 per GB. As a result, optimizing components of walkthrough systems with HDDs is critical. In particular, addressing the seek time, the main bottleneck of accessing data from HDDs, remains the main challenge for interactive rendering of massive data sets. In this paper, we leverage the inexpensive nature of HDDs to store redundant copies of data in order to reduce the seek time.

### 1.2 Redundancy based data layouts

In this paper, we leverage the inexpensive nature of HDDs to store redundant copies of data in order to reduce the seek time. Adding redundancy for improving the data access time is a classic approach, e.g., RAID [Patterson et al. 1988]. We are also not first to consider the redundancy for the walkthrough applications. Redundancy based data layouts to reduce the seek time were introduced in a recent work [Jiang et al. a], in which the seek time for every access was reduced to at most one unit. However, in order to achieve this property, the redundancy factor, the ratio between the size of the data after using redundancy to the original size of the data, was prohibitively high, around 80.

Another recent work [Jiang et al. b] introduced a transfer volume, which dictates the data transfer time, seek time, and redundancy, and proposed a linear programming approach to optimize the data transfer and seek time in order to satisfy the total data fetch time constraint. In the process, redundancy was a hidden variable that was minimized. Unfortunately, this approach does not directly model redundancy nor seek time, and thus can have unnecessary data blocks and unrealistic seek times.

### 1.3 Main Contributions

In this paper, we propose a model for seek time based on the actual distance between the data blocks in the linear data layout. Using this model, and given the spatial proximity of the data set for a walkthrough application, we develop an algorithm to duplicate data blocks strategically to maximize the reduction in the seek time while keeping the redundancy factor within the user defined bound. We will show that our greedy solution generates both the extreme cases of data layout with redundancy, no redundancy (a simple cache oblivious mesh layout with low number of seeks) and maximum redundancy (a layout where seek time is at most one), as well as reasonable solutions for the redundancy factor constraints in between the extremes. When our algorithm was implemented for walkthrough applications, we were able to noticeably reduce the average delay and make the performance much more consistent.

## 2 Related Work

Massive model rendering is a well studied problem in computer graphics. Most of the early works focused on increasing the rendering efficiency. At that time the fundamental problem was not fitting the model into the main memory but the speed of the graphics cards. Hence these works provided solutions to reduce the number of primitives to be rendered while maintaining the visual fidelity. These solutions included level-of-detail for geometric models [**?**], progressive level of detail [**?**; **?**], and image based simplification [**?**; **?**; **?**]. Soon thereafter the size of the main memory became the bottleneck in handling ever increasing sizes of the model. Hence memory-less simplification techniques [**?**] and other out-of-core rendering systems [**?**; **?**] emerged in which just the limited amount of required data that needs to be processed and rendered was brought from the secondary storage to the main memory.

Soon afterward, the researchers realized that the speed at which this data could be brought from the secondary to the main memory in these out-of-core algorithms is limited by the data bus speed, disk seek time, and data transfer time. They realized that these issues could be ameliorated to some extent by better cache utilization that would increase the utilization of data that is brought to the main memory and thus reduce the number of times the disk read is initiated. This meant that subsequent works focused on cache aware [Sajadi et al. ] and cache oblivious data layouts [**?**; **?**] on the disk to reduce the data fetch bottleneck. Our work falls under this class of algorithms that reduces the data fetch time.

Redundancy based data layouts were mentioned in [Patterson et al. 1988; Jiang et al. a; Jiang et al. b] as potential solutions to this problem of reducing seek time. In particular [Jiang et al. b] presented an algorithm that limits the amount of redundancy required but there were major drawbacks. First, it provides a grouping of data units for each seek but it does not provide a data layout. This is because it does not relate one data group with another and it does not consider their relative layout. This could easily result in unnecessary data block duplications. The redundancy minimization is thus not modeled after physical representation of the data layout on the disk. The second major drawback is that the model for seek time is also not based on physical reality. Typically, seek time depends on the relative distance on the disk between the last data unit accessed and the data unit currently being requested. However, in the paper, seek time is simplistically modeled as number of seeks. This means that even if the requested data blocks are adjacent to each other and have no separate seek time to go from one block to another, this model would add them to the cost because it counts individual data blocks as one seek. Our approach seeks to address these issues.

## 3 Greedy Redundancy-based Cache Oblivious Data Layout Algorithm

### 3.1 Definitions

For the purposes of this paper, we are given uniform data units in a linear sequence and each data unit has one or more access requirements assigned to it. An individual data unit consists of geometric primitive data or other information that must be retrieved during walkthrough rendering. The access requirements are determined by the application and represent data units that will be accessed together. An example of this could be vertices that are spatially near each other.

For an access requirement $A$, the total span of $A$ is the total number of data units between the first and last data units that use $A$. If a data unit is not required by $A$ but lies between the first and last unit of $A$ then it is still counted in the span of $A$. Figure 1 shows a set of data units and explicitly labels the blue access requirement in this set. The total number of data units between the first and last blue unit is 14 thus that is the total length of the blue access requirement.

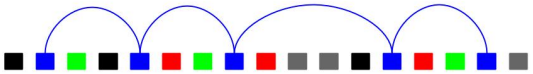

**Figure 1:** *Illustration of data units and a single access requirement. The blocks are data units and the colors represent access requirements. The lines connect data blocks with the same access requirement and represent parts of the span of an access requirement.*

Instead of trying to find the seek time explicitly while running, we use this one dimensional layout to compute a probabilistic estimate for the seek time. A cache-oblivious measure was introduced to measure the expected number of cache misses [Yoon et al. ]. Note that every time we have cache misses, we would have to seek data from the hard disk. We extend this idea by thinking of an individual cache miss as an individual seek. We define the Expected Seek Time (EST) as the total number of individual seeks that could be required. Let $I$ be the set of access requirements and $A_i$ represent the span of access requirement $i$. Because data units in an access requirement will be accessed together, we can assert the following definition.

$$EST = \sum_{i \in I} A_i$$

### 3.2 Algorithm Description

In [Yoon et al. ], the only allowed operation on the data units is the move operation and the optimal solution is computed using only that operation. For our purposes, we are allowed to copy data units, move them, and delete them if they are not used. We want to figure out how to minimize EST while keeping the number of extra copies as low as possible. After constructing a cache-oblivious layout to the mesh to get an initial optimization, we take a data unit and copy it to a place that will mean shortening at least one of the access requirements that use it. If the new data unit reduces the span of all the access requirements attached to it, then we delete the original data unit. We repeat the individual copying and possible deletion procedure until our redundancy limit has been reached.

**Blocks to consider:** Note that the span of an access requirement does not change by moving an interior data unit to another interior location. Cost can be reduced only by moving the blocks that are at the either ends of the access requirement. This will greatly reduce the search space of data units to consider for copying. For the sake of simplicity of the algorithm, we operate on only one data block at a time.

**Location of copies:** Based on the above observation, given an access requirement, we can possibly move the beginning or the end data block of the access requirement to the interior and make it an interior block. This will reduce its span, thus reducing the EST for the layout. In order to maximize our benefit to our

current access requirement, we want to copy the start data unit to somewhere between the one after the first one and the last one. In the same manner, we want to copy the end unit to somewhere between the first one and the one before the last one.

Figure 2 shows the access requirement from the above figure with an arrow showing the locations where the start unit can be moved to as well as what the access requirement looks like after the copy. By moving it to one of the these locations we are guaranteed to reduce the span of the access requirement. The dashed line connects the original data unit with its copy. The solid blue lines represent the new span of the blue access requirement. Because the start unit has been copied and its copy is being used, it no longer is needed and is thus not counted in the new span for the blue access requirement. The span of the blue access requirement has been reduced from 14 units to 12 units.

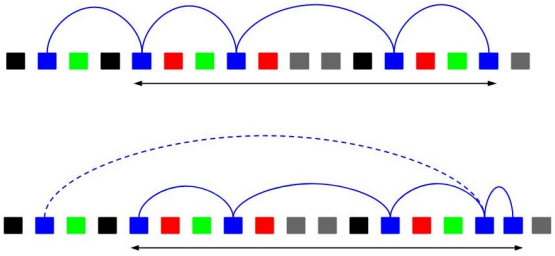

**Figure 2:** *Interval where the start data unit can be copied to (top) and the data units after the copy (bottom)*

For the access requirement under consideration, it does not matter where in the specified interval our data unit is moved. However, if the new location of the data block is in the span of other access requirements, it increases the seek time of those accesses by one unit. We thus want to find a location that in the interior of the access requirement under consideration but is in the span of least number of access requirements. Such a location is identified using a simple linear search through the span of the access requirement. Figure 3 shows the data units in figure 1 before the copying and highlights the gray access requirement that will be affected. As can be observed in figure 3 the gray access requirement has had its span increased by 1. Even though we reduced the total span by 2 with the blue access requirement, we also increased the total span by 1 with the gray access requirement, yielding a net benefit of 1 data unit. If you search through all the spots in the interior of the blue access requirement, then the spot chosen had the least number of overlapping access requirements at 1.

**Moving versus Copying:** A data block can be accessed by multiple access requirements. If that data block is an extremal block of access, and if it is moved to its interior, it may affect other access requirements that use the same data block. That is the main reason that we are copying and not moving these data blocks. By copying the data unit the other access requirements can still access it in its old location without any change in their span. Nevertheless, if by using the new copy the span of one or more of the other access requirements reduces, then those access requirements should use the new copy instead of the old copy. If all the access requirements use the new copy and the old data block is not used by any access, then it can be deleted. In this later case, we are really moving the data block.

**Data Block processing order:** We now need to figure out

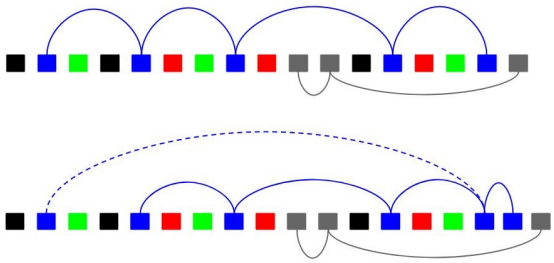

**Figure 3:** *The blue and gray access requirements before (top) and after (bottom) the copying*

how to use this information to decide in what order the copying should be done. For each data unit, its total benefit is the amount that is reduces the total seek time ($EST$). For a given data unit to be copied to a specified location, let $k_i$ be the benefit to access requirement $i$ that is attached to the data unit. We will say that $k_i = 0$ if the access requirement will use the old copy and $k_i > 0$ if the access requirement will use the new copy. Let $I$ be the set of access requirements attached to the data unit. Let $J$ be the set of access requirements not in $I$ whose span overlaps our data units. We can now describe the benefit as follows:

$$Benefit = -\Delta EST = \sum_{i \in I} k_i - |J|$$

Before doing any actual copying, we compute the above described benefit for each start and end data unit for each access requirement. We store all the benefits into a binary search tree sorted in descending order by benefit amount. That way we will easily be able to choose the data unit that provides the most benefit in expected seek time. We will also make a special list $L$ of cases where a data unit is copied then deleted because all the access requirements will benefit.

Because doing the moving for the list $L$ does not increase the storage, we will first go through that list and perform those moves. Once that list is empty we will take the data unit in the binary search tree that provides the most benefit and perform the copy. We will then recompute $L$ and the binary search tree. We will continue to do those steps until we have run out of available space for redundancy. As a summary, the psuedo-code of this algorithm is shown as algorithm 1.

## 4 Run-time and Storage Analysis

We now analyze the running time and storage requirements of our algorithm. We will denote $N$ as the number of data units and $A$ as the number of access requirements. We will use $k$ as the average span of a single access requirement. The variable $Q$ will represent how many executions of the redundancy loop occur. The average number of overlapping access requirements is proportional to the number of data units multiplied by the redundancy factor $r$, which is the amount of redundancy if we have a single-seek layout. Therefore, the number of overlapping access requirements is $O(rN)$.

### 4.1 Run-time Analysis

The loop that initially constructs the access requirement heap is run on all $A$ access requirements. It involves finding all the benefit information and putting it into a heap so that it is easy to figure out the best access requirement to modify first. With each run of the loop, in addition to a constant number of initial operations, there

```
Start with the data units which each with at least one access
requirement (AR);
Initialize AR heap H and list L;
for each data unit do
    Find number of overlapping access requirements and store the
    number with the data unit;
end
for each AR P's head and tail data units U do
    make old copy list L' empty;
    if U is head data unit then
        Let S = data unit in P after U;
    else
        Let S = data unit in P before U;
    end
    Set BENEFIT=distance(S,U);
    Let U' be the potential copy of U;
    Search data units between S and U for min number of
    overlapping ARs and put U' there;
    for each AR T that also uses U do
        if T will be shorter by using U' then
            Add T's benefit to BENEFIT;
        else
            Add T to list L';
        end
    end
    Add BENEFIT to heap H;
    if L' is empty then
        add U to list L;
    end
end
while BREAK has not been called do
    while L is not empty do
        Take out random element and move the data unit;
        Update heap H and list L;
    end
    if there is more space for redundancy then
        pop best element U from H;
        copy U to its destination;
        update nodes in H and entries in L for affected access
        requirements;
        update H and L;
    else
        call BREAK
    end
end
```

**Algorithm 1:** Pseudo-code for our algorithm

are $log(A)$ operations to insert the data into the heap plus $O(k)$ operations for the search of the AR to get the benefit information. Thus it takes a total of $O(A(k+logA))$ operations to do the initial construction.

With the following loop which actually copies the data units and updates the information, there are a total of $Q$ executions of it. In addition to the constant number of operations, each execution of the loop has to go through each of the overlapping access requirements and update their data. There are $O(rN)$ overlapping access requirements. For each of the affected access requirements, there are $O(k + log(A))$ operations to recalculate the benefit data and reform the heap. Thus the final loop takes $O(QrN(k+logA))$ operations.

This means that in total, our algorithm takes $O((QrN + A)(k + logA))$ operations. In all likelihood, we will have to run the loop at least once, so $Q \geq 1$. Additionally, since $r \geq 1$, we know that $rN \geq A$. Thus we can simplify the expression to say that our algorithm takes $O(QrN(k + logA))$ operations. Because we only have polynomial or logarithmic terms, we have found an algorithm that is efficient given our input size for computing an optimized layout.

### 4.2 Storage Analysis

During the run of the algorithm, we have to store the number of overlapping access requirements 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.

### 4.3 Linear search justification

Here I justify why we do a simple element by element search on the set of possible data units in the part of the algorithm where we decide where to put the new data unit. The original problem is to find the least number in an arbitrary block of a list. This is because each data unit stores the number of overlapping access requirements so we have a list of numbers to search through. There are only a limited number of data units where we could move the copy to hence we only care about an arbitrary part of this list. If $k$ is the size of the access requirement, then the searching 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.

## 5 Experimental Results

**Figure 4:** *City model: 110 million triangle, 6 GBs.*

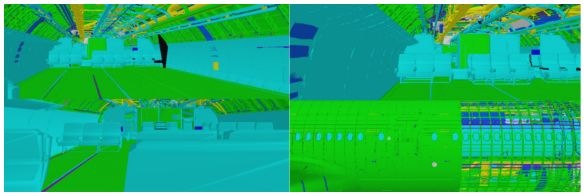

**Figure 5:** *Boeing model: 350 million triangle, 20 GBs.*

**Figure 6:** *Urban model: 100 million triangle, 12 GBs.*

**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 might be used in a navigation simulation application or visual reality walkthough. The Boeing model (Figure 5), on the other hand, represents scientific or engineering visualization applications. The Urban model has texture attached to it, which is commonly used in games. By comparing performance of cache-oblivious layout without redundancy and with redundancy on these three models, our goal is to show the redundancy based approach can achieve more stable and generally better performance on different real time applications.

**Results:** Figure 7 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. 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.

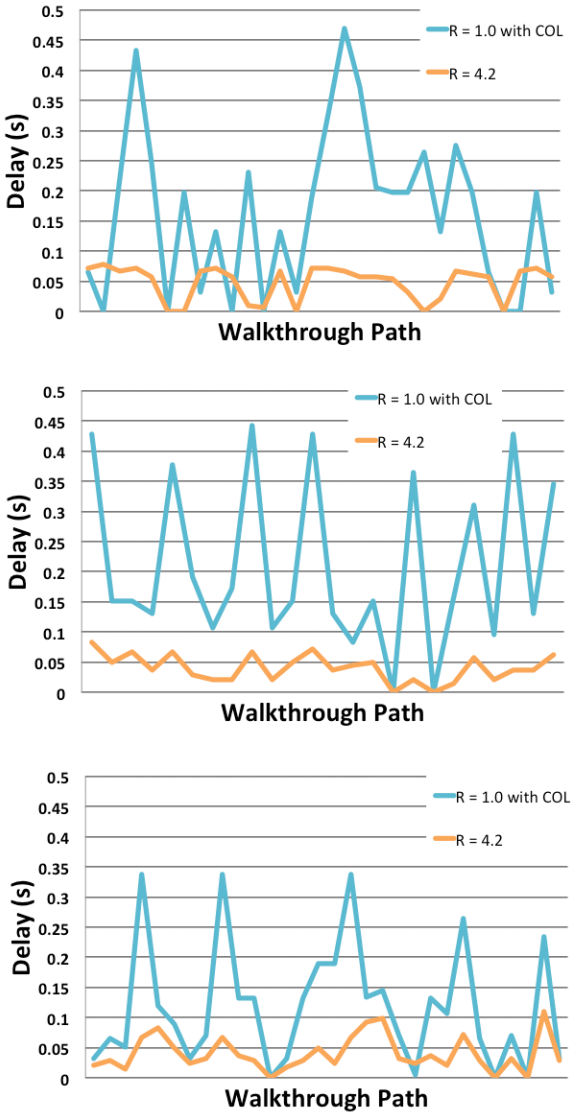


**Figure 7:** *Statistics of delays caused by the fetching processes for the Sparse City model (top), the Boeing model (center), and the Dense City model (bottom), with and without redundancy.*

There is another major benefit to our approach. Since each time we duplicate one data unit, we can halt it when the redundancy factor reaches a certain threshold. This helps us create a data layout with arbitrary redundancy factor without worrying about exceeding the capacity of secondary storage devices. We use this fact to test

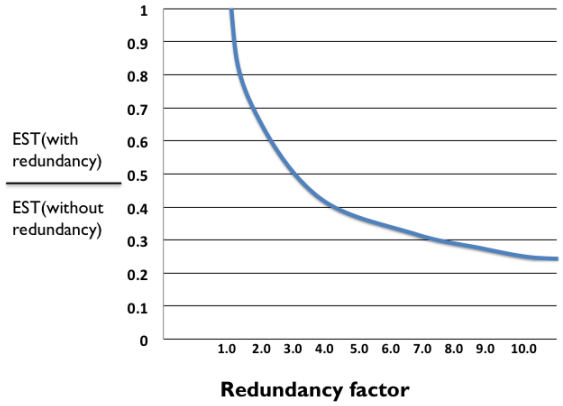


**Figure 8:** *Plot of the ratio of the EST of layout with redundancy over the EST of cache-oblivious mesh layout withou redundancy.*

different redundancy factors and see their results. In Figure 8, 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 worth it 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 Theoretical Improvements over existing algorithms

The initial part of the algorithm where we copy and delete units will produce a better solution than proposed by Yoon without adding extra units. This is because our algorithm will consider cases where data units are close to each other but in different blocks of units that would be arranged in Yoon's algorithm. Consider a case where we have two access requirements of 5 data units each. Figure 9 shows an example of that. Figure 10 shows the result of using Yoon's algorithm. Because its hierarchical it would arrange the units in each block and then arrange the blocks. Thus even though the black access requirement blocks can be grouped together, Yoon's heuristic would not detect that. When our algorithm is run after Yoon's algorithm, it would find that it can shorten the black access requirements without adding redundancy so it would do that first. You would end up with the ideal solution as shown in figure 11

**Figure 9:** *Example of two access requirements of 5 data units each. The red line represents the boundary between blocks that Yoon's algorithm would use*

Existing algorithms for the data layout problem do not consider

**Figure 10:** *The above figure after running Yoon's algorithm*

**Figure 11:** *The above figure after running our algorithm*

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. Figure 12 shows a case where that happens. The total seek time is 11 units. Without redundancy, the optimal solution is shown in Figure **??**. The seek time has been reduced to 9 units. With redundancy, the total seek time is now the minimum required which is 7 units, as shown in Figure **??**. While a reduction from 9 to 7 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 as described in the next section.

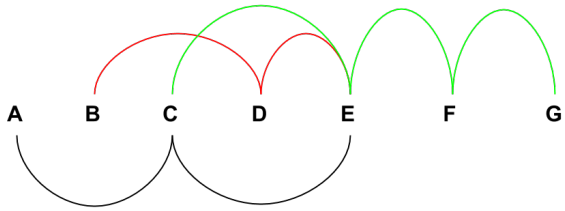


**Figure 12:** *Data Units with varying access requirements*

## 7 Conclusion and Future Work

We have shown that we have an algorithm with an efficient running time and storage space for the Data Layout Problem. It achieves significant results analytically and experimentally. When walking through an extremely detailed 3D model, this algorithm can be used to ensure that the performance will not suffer. If we give the algorithm the proper access requirements with this 3D model, then the performance will be even better.

This leads to a logical extension of this work. Since we have a good algorithm that takes over once we know the access requirements, we should figure out how to ensure there are good access requirements to begin with. One idea on how to ensure this is to check the usage history of an application and group data units together if they are accessed together with high probability. This could even be done dynamically in the sense that after a certain amount of usage and repeating on a regular basis, you recompute the optimal access requirements and then use that to recompute the optimal layout.

## References

AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. 2008. Design tradeoffs for ssd performance. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, USENIX Association, Berkeley, CA, USA, 57–70.

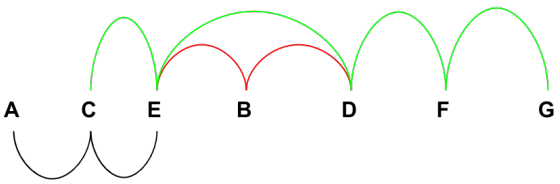


**Figure 13:** *Optimal layout without redundancy*

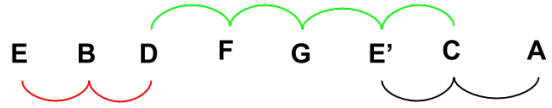


**Figure 14:** *Optimal layout with redundancy. The data unit E' is the redundant copy of E.*

DOMINGO, J. S. 2014. Ssd vs. hdd: What's the difference. *PC Magazine* (February).

JIANG, S., SAJADI, B., , AND MEENAKSHISUNDARAM, G. Single-seek data layout for walkthrough applications.

JIANG, S., SAJADI, B., IHLER, A., AND MEENAKSHISUNDARAM, G. Optimizing redundant-data clustering for interactive walkthrough applications. *CGI 2014*.

PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. 1988. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, ACM, SIGMOD '88, 109–116.

RIZVI, S., AND CHUNG, T.-S. 2010. Flash ssd vs hdd: High performance oriented modern embedded and multimedia storage systems. In *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, vol. 7, V7–297–V7–299.

SAJADI, B., JIANG, S., HEO, J., YOON, S., AND MEENAKSHISUNDARAM, G. Data management for ssds for large-scale interactive graphics applications. *ACM SIGGGRAPH 2011*.

SAXENA, M., AND SWIFT, M. M. 2009. Flashvm: Revisiting the virtual memory hierarchy. In *Proc. of USENIX HotOS-XII*.

YOON, S., LINDSTROM, P., PASCUCCI, V., AND MANOCHA, D. Cache oblivious mesh layouts. *ACM SIGGGRAPH 2005*.