Reducing Disk Latency through Replication

Gordon B. Bell
Morris Marden

Abstract

Today's disks are inexpensive and have a large amount of capacity. As a result, most disks have a significant amount of excess capacity. At the same time, the performance gap between disks and processors has widened to the point that many workloads have become disk bound. To improve the performance of disks, we propose using the excess capacity of disks to replicate blocks. To do this, the disk controller observes sequences of requests to blocks and replicates blocks on disk so that they are in the same order on disk as in the sequences. By doing this, when the sequence occurs again, no seeks are needed between accesses to blocks in the sequence. Our work shows that these sequences can be reused a large number of times, so they potentially can yield a large benefit. We also have an algorithm, which we implemented in the DiskSim simulator, for detecting these sequences and performing replication.

Abstract

Today's disks are inexpensive and have a large amount of capacity. As a result, most disks have a significant amount of excess capacity. At the same time, the performance gap between disks and processors has widened to the point that many workloads have become disk bound. To improve the performance of disks, we propose using the excess capacity of disks to replicate blocks. To do this, the disk controller observes sequences of requests to blocks and replicates blocks on disk so that they are in the same order on disk as in the sequences. By doing this, when the sequence occurs again, no seeks are needed between accesses to blocks in the sequence. Our work shows that these sequences can be reused a large number of times, so they potentially can yield a large benefit. We also have an algorithm, which we implemented in the DiskSim simulator, for detecting these sequences and performing replication.

1. Introduction

Several technology trends have shaped the way in which disks are used by systems. Capacity has been increasing at a steady rate of about 60% per year, allowing fewer disks to fulfill the same storage requirement that many had previously. Disk seek latency continues to improve (at about 6% per year), but has not done so at the same rapid pace that capacity has. This unbalanced growth can actually lead to poorer performance on systems using modern disks. A system with many disks can stripe data across those disks to overlap seek latency and improve bandwidth. Larger disks can lead to fewer disks, which in turn can lead to fewer concurrent seeks and less overlap. Consider the case of a single, sufficiently large disk, where every disk access results in the system paying the full seek overhead. In order to deal with this "problem" of excess capacity, I/O intensive applications such as online transaction processing (OLTP) often determine the number of disks in a system (which can be in the order of hundreds or thousands) by the number of disk heads, rather than total capacity [11]. This results in a large number of partially empty disks with data placed such that the seek distance of every head in minimized [8]. Other types of systems often have unused disk space as well. Inexpensive disks (around $150) are typically as large as 40GB -- generally more than enough capacity for PCs or workstations.

A second important trend is the rapid increase of CPU performance as it follows Moore's curve. Because a given seek latency will result in a faster processor stalling a larger number of cycles, seek latency is quickly becoming a system bottleneck. This is analogous to the "memory wall" problem, in which waiting for memory access are constituting an increasingly larger percentage of processor activity.

We attempt to answer a simple question in this project: can unused disk capacity be exchanged for reduced seek latencies? We begin with the hypothesis that disks make repeated accesses to data that is physically "spread out" across the disk. Figure 1 illustrates such a case, in which the disk accesses blocks 4, 31, and 12, and then makes the same three accesses later. Each time the blocks are accessed in that order, the disk head incurs a seek between each. We propose a technique that identifies dependencies between these three blocks during their first accesses and duplicates them onto the same track. Subsequent accesses to the three blocks will remap to the area on the disk where they are laid out sequentially and eliminate seeking between the blocks. In this way we remap temporal locality identified in disk accesses to spatial locality in duplicated data.

The structure of this paper is as follows: in Section 2 we outline related work, Sections 3 and 4 explores an upper bound on the effectiveness of such a technique, Sections 5 and 6 discuss the algorithm and implementation of our technique. Section 7 concludes the paper.

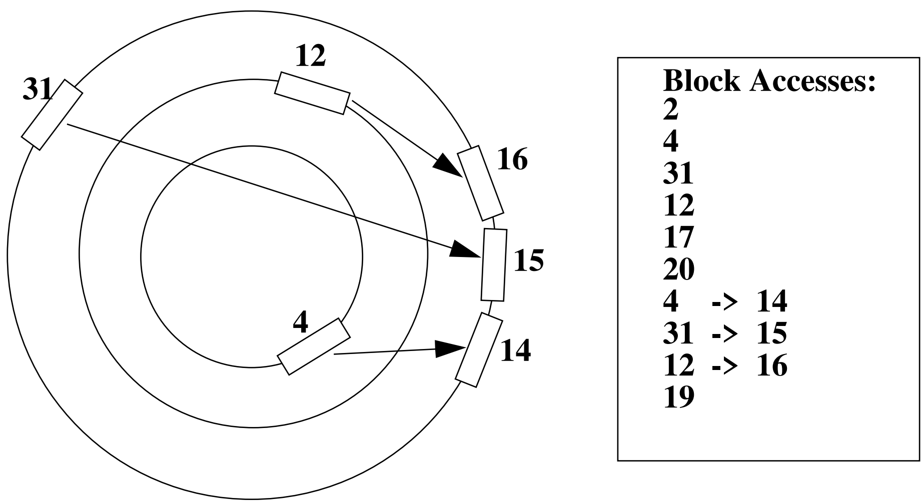


Figure 1: In a traditional disk, the full seek latency is paid when blocks 4, 31, and 12 are accessed. With our technique, the seek latency is paid on the first accesses, but subsequent accesses will map to replicated data

2. Related Work

Many conventional disk driver and controller algorithms attempt to schedule accesses to minimize seek latency. However such algorithms are limited to simply reordering accesses, and do not actually duplicate or relocate data on the disk. Although such reordering can help reduce seek time, not all accesses can be reordered or there may not be enough pending requests to provide sufficient scheduling selection. The example presented in Figure 1 would not benefit from disk scheduling because the disk head would still need to seek between three different tracks, regardless of their order.

One of the first studies that attempted to reduce seek times through rewriting data, rather than just scheduling, relocated blocks based on their access frequency [5]. If a block was accessed a greater number of times than a predefined threshold, it would be relocated to a reserved area in the center of the disk. The metric that was used to determine whether to “shuffle” a block was simply the number of times a block was accessed, and shuffled blocks were placed anywhere in the reserved area. This technique did not attempt to write sequential blocks to disk in the order that they were accessed, unlike our technique. Furtherer, shuffled blocks are relocated, not duplicated as in our scheme. In our mechanism a given block may exist in several “traces” on the disk. The decision of which one to use can be made dynamically based on observed proceeding block accesses.

Akyurek, et. al [1] duplicated frequently accessed blocks, but again did not examine block accesses in the context of preceding or following block accesses. Duplication in order to reduce latency was also examined in [8], but only in terms of replicating data within a single track in order to reduce rotational delay.

3. Traces

A disk trace is a log of the sequence of accesses that a workload performs and some information about those accesses. The information our traces hold is the time of the block request, the device that the request was to (i.e., which disk to request the block from), the number of the block that was requested, the size of the request (i.e., how many blocks to read or write), and the type of access (read or write).

We first attempted to use the disk traces that come with the DiskSim simulator [3]. Unfortunately, when we examined these traces, we found that there was little repetition in the blocks that were requested (i.e., most blocks were only used once and the rest were only used twice). Our optimizations for disk layout can only affect performance when blocks are accessed many times. Therefore, these traces would not work for our study.

Name	Description of Workload	# of requests
BigSort	Sorts 200,000 line, 17 MB text file	30,932
J1	Trace driven cache simulator	14,206
J2	Search symbol names in large kernel source	20,799
J3	Search text strings in small kernel source	22,091
J4	Search text strings in large kernel source	34,934
J5	Search for key words in collection of text files	37,938
PQ7	Postgres performing selection	18,261
PJoin	Postgres performing join	25,957
SPECweb99	Web server for static and dynamic pages	2,366

Table 1: Descriptions of the disk traces that we used (note that the first eight are from Pei Cao)

Next, we used eight of the ten disk traces that Pei Cao took [9]. Table 1 has descriptions of these traces. To verify that Pei Cao's traces are representative of disk accesses, we used the emitter interface of SimOS to collect a disk trace of SPECweb99. SPECweb99 is a benchmark for evaluating the performance of a system that is serving static and dynamic web pages [10]. We were only able to simulate SPECweb99 for thirty seconds, since it takes a long time to simulate (it took 14 hours to simulate 30 seconds). The simulated system contained only 128 MB of memory and the size of the file set for SPECweb99 was about 500 MB. As a result very few files in the file set could fit in the OS's or web server's file cache, so most file accesses needed to go out to disk. For the web server, the simulated system used Zeus.

4. Limit Study

Before implementing block replication with temporal locality in the disk simulator, we performed a limit study. This allowed us to see the potential benefit of implementing this optimization without having to worry about the details of how to implement it. This also allows us to better understand the behavior of workloads using this optimization. Lastly, the limit study gives us an idea of how long the sequences of temporally local accesses and whether our implementation can use a static length for the sequences or if it needs to dynamically detect what length to use for the sequences.

The limit study walks through a disk trace one access at a time. For each access in the trace, it constructs a sequence for each possible grouping that was observed so far up to a maximum length that is specified by the user. Note that these sequences are not sequences of blocks, but are sequences of I/O requests to the disk. For the purposes of our study, we chose to limit the length of sequences to ten requests so that the program for the limit study would produce results relatively quickly.

We first studied the relationship between number of times each sequence to the length of the sequence. Figure 2 shows graphs of the number of times the sequences of temporally local accesses (labeled a traces in the graphs) are used for different sequence lengths. The maximum curve corresponds to the number of times that one can use the sequence that has the largest number of hits. For the average curve, the controller would create a sequence for each the request seen (when there are multiple sequences that it can use for starting block, it uses the one with the highest number of hits). We then take the average of the number of times each of these sequences is used. Note that many of these sequences are seen once and never reused, so they lower the average even though the controller probably would not replicate these blocks.

From looking at the graphs, we see two different behaviors in the workloads. In BigSort, J3, J4, J5, and SPECweb99, the number of times one uses the sequences drops off quickly as one increases the length of the sequence. In these cases, long sequences have little benefit over using relatively short sequences. In addition, it is relatively simple to pick a static length to use for the sequences (a sequence of length 7 will hold most of the accesses to these sequences). However, in J1, J2, and PJoin, the number of times one uses the sequences drops off fairly slowly as a function of the length of the sequences. In these cases, large sequences continue to be useful in addition to shorter sequences. Unlike in the first set of workloads, it is difficult to choose a single length to use for these workloads. PQ7 appears to be a hybrid of these two behaviors, since the number of times that it uses sequences drops off more quickly than the second behavior, but much less abruptly than in the first behavior. Therefore, since there is no single length that one can use for the sequences, we believe that one needs to use an adaptive mechanism for choosing how long of traces to use.

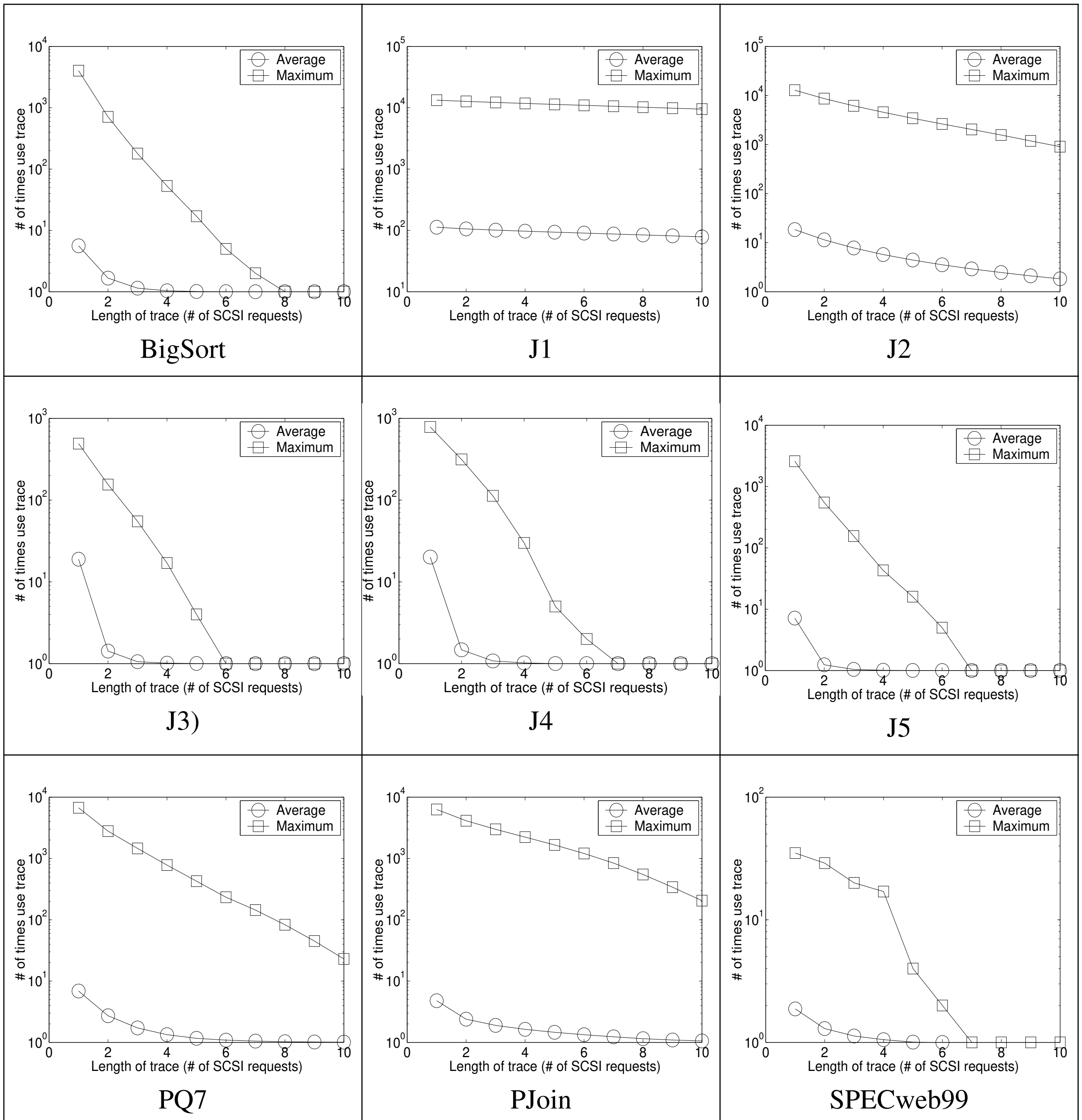
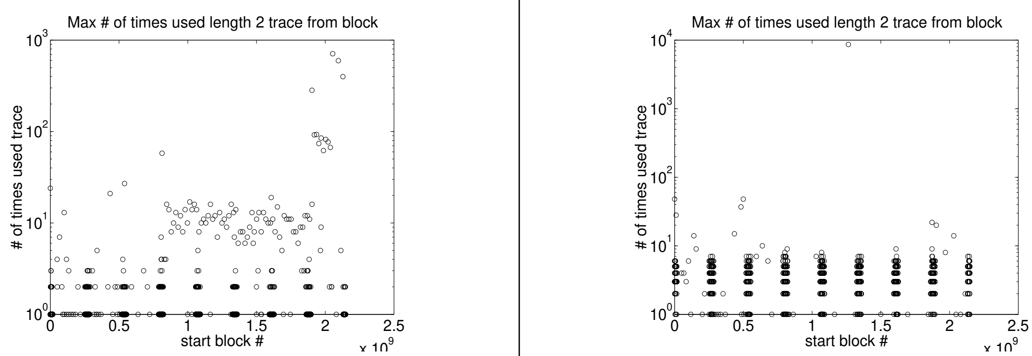


Figure 2: Number of times uses temporal sequences (called traces here) of different lengths

To get more insight into the two different behaviors of temporal sequences, we also looked the distribution of how often different sequences are used. In particular, we looked at the number of times each length two sequence is used. In BigSort, the accesses between different sequences are fairly distributed, in that there are a large number of sequences that are used a significant number of times. The accesses to the sequences does not follow much of a pattern. At the same time, there are a few sequences that are used much more than the rest of the sequences, so it is important to make sure that one includes these sequences in the replication.

In contrast, the behavior of J2 is very different than that of BigSort. First, there are not that many sequences that are used a large number of times. Also, the accesses to the sequences appears to have a pattern. Lastly, there is a single sequence that is used a large number of times and is used more often than in the case of the sequences with high use in the case of Bigsort. In particular, this sequence in J2 is used 42% of all requests to the disk. In comparison, the sequence with the highest usage in BigSort consists only of 2% of all requests.



BigSort J2

Figure 3: Maximum number of times that use a length 2 temporal sequence for each starting block

5. Algorithm

The basic data structure in our algorithm is a table indexed by block number. When a disk request is made, the starting block of that request is looked up in the table. The corresponding table entry indicates all of the duplicated “traces” that block exists in. The algorithm will identify the most likely candidate of these “traces” and remap the incoming request to the duplicated block number. It consists of three phases: identifying “traces” to duplicate, the duplication itself, and updating the table to reflect the disk’s updated state.

Each entry in the table corresponds to a unique block on the disk, and consists of an arbitrary number of “sections” that correspond to blocks observed to immediately precede this block. Each section consists of a previously identified block number, a count of how many times that block was observed to precede the block number for this entry, a mapping to a block the request should be substituted for, and a valid bit to indicate if the mapping is valid. This table is illustrated in Table 1. For every block request, the disk checks to see if an entry exists for that block number. If it does, it checks to see if a section in that entry exists for the block that was observed immediately before the current request. If such a mapping exists and is valid, the disk will fetch the block identified in the mapping field rather than the one actually requested and increment the counter. If either of entries do not exist, they will be added to the table.

If the count field exceeds a predefined threshold and no mapping exists yet, the algorithm will insert a request to duplicate the block in the disk queue. When the disk processes this request and writes the data (to an unused block indicated by a free list), it updates the valid bit and mapping field in the corresponding entry to indicate that the mapping is valid. Reads and writes are handled almost identically; the major difference is that writes need to be propagated to all duplicated blocks. We deal with this by invalidating all the entries that a block to be written appears in and lazily updating them on disk. The table entries are updated at the point that the updates occur to the disk.

At this point we have said nothing about where this algorithm and table are implemented. They could be added to the device driver or inside the disk at the controller. Although adding it at the driver-level may be more feasible than changing the controller hardware, more information is available to the controller, such as disk head position (although [8] implements a disk head position predictor in). In either case the table need not be retained in non-volatile memory, as blocks always exist in their original location -- losing the information in the table only means that the disk will not benefit from our optimizations.

Block Req	Prev Block	Cnt	Remap	V	Prev Block	Cnt	Remap	V
2	??	??	??	?				
4	2	1	----	0	20	1	----	
31	4	2	15	1				
12	31	2	16	1				
17	12	1	-----	0				
20	17	1	------	0				
19	12	1	------	0				

:Table 2: Table state for the trace in Figure 1.

6. Implementation

We used the DiskSim Simulation Environment v2.0 [3] to model our disk replication algorithm. DiskSim models all aspects of a storage subsystem, including the OS driver, controller, cache, and scheduling algorithms. It has been validated against production disks and is reputed to be accurate and reliable.

There are several methods in which the remapping table could be implemented in a real system. Our model uses a hash table indexed by starting block numbers. In this way it can grow dynamically based on how many different blocks have every been requested. Although typically an entry exists for every physical block on disk, the table's size varies depending on how many mappings are retained for each block (in previous terminology, the number of sections per entry). Because it is likely that many blocks on a disk have never been accessed or are accessed infrequently, it is not crucial that we allow a mapping to occur for every block on disk. By limiting the table entries to a subset of blocks, we can duplicate these blocks a greater number of times. The location of this table may also be a factor in its size or structure. If added at the disk controller, it could be in RAM inside the disk and would probably be a small, fixed size. If added at the driver, more elaborate algorithms could exist that harness the processing power of the main CPU and utilize surplus system memory.

Other issues exist in manipulating table entries, such as when and how to replace mappings for a given block. Each entry can hold a certain number of mappings (in our implementation this is fixed, but one can imagine a design that allows a variable number); when this limit is reached, do we evict entries to make room for others? What is this replacement policy based on (LRU, access frequency, etc.)? When an entry is evicted is it just dropped or would it be beneficial to store it out to disk for possible retrieval later? We leave many of these issues for future research.

7. Conclusion

At this point we have implemented the table and the algorithms that manipulate the table into DiskSim. The modified simulator is able to identify repeating sequences of blocks, update the table if given threshold is exceeded, and substitute the original request for the remapped block number. Unfortunately, due to simulator complexity (and poor documentation) and lack of time, we were not able to completely integrate our code into DiskSim and collect average seek latencies.

However, based on several assumptions, we anticipate our algorithm would indeed reduce seek times: enough excess disk space exists for a reasonable amount of duplication; disks are idle enough that they can afford to spend time duplicating data; disk controller processors are sufficiently powerful enough that they can keep track of and manipulate the required data structures; RAM is cheap (if the table is implemented in the controller); and as processors get faster and disks get larger, workloads will be bound by disk seek latency. We plan to continue this work and finish integrating our table into Disksim to collect latency results.

We also believe that there is a plenitude of other interesting data to collect and analyze. For example, how does cache behavior affect the effectiveness of our technique? It seems that a sufficiently large cache may be able to capture many accesses that would otherwise cause seeks. Are there ways for the controller to exploit its knowledge of where the disk head is? Perhaps this could be used to complete lazy updates of writes (in the case that the head is passing over an updated region anyway). We leave these issues for future work.

8. References

[1] S. Akyurek and K. Salem. Adaptive Block Rearrangement.

[2] R. English and A. Stepanov. Loge: a self-organizing disk controller. In *Proceedings of USENIX Winter 1992 Technical Conference*, January 1992.

[3] Greg Ganger, Bruce Worthington and Yale Patt. *The DiskSim Simulation Environment.* Version 2.0, December 1999. Available at *http://www.ece.cmu.edu/~ganger/disksim/*.

[4] J. Matthews, et al. Improving the Performance of Log-Structured File Systems with Adaptive Methods. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, October 1997.

[5] C. Ruemmler And J. Wilkes. Disk Shuffling. *HP Laboratories Technical Report HPL-91-156,* October 1991.

[6] C. Ruemmler and J. Wilkes. An Introduction to Disk Drive Modelling. *IEEE Computer*, March 1994.

[7] R. Wang, T. Anderson, and D. Patterson. Virtual Log Based File Systems for a Programmable Disk. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, February 1999.

[8] X. Yu et al. Trading Capacity for Performance in a Disk Array. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, October 2000.

[9] Cao. Disk Traces. http://www.cs.wisc.edu/~cao/traces/.

[10] Systems Programming Evaluation Cooperative. SPEC Benchmarks. http://www.spec.org.

[11] Transaction Processing Council. TPC Benchmarks. http://www.tpc.org.