Prefetching in Segmented Disk Cache for Multi-Disk Systems *

Valery Soloviev
Computer Science Department
North Dakota State University
soloviev@plains.nodak.edu

Abstract

This paper investigates the performance of a multi-disk storage system equipped with a segmented disk cache processing a workload of multiple relational scans. Prefetching is a popular method of improving the performance of scans. Many modern disks have a multisegment cache which can be used for prefetching. We observe that, exploiting declustering as a data placement method, prefetching in a segmented cache causes a load imbalance among several disks. A single disk becomes a bottleneck, degrading performance of the entire system. A variation in disk queue length is a primary factor of the imbalance. Using a precise simulation model, we investigate several approaches to achieving better balancing. Our metrics are a scan response time for the closed-end system and an ability to sustain a workload without saturating for the open-end system. We arrive at two main conclusions: (1) Prefetching in main memory is inexpensive and effective for balancing and can supplement or substitute prefetching in disk cache. (2) Disk-level prefetching provides about the same performance as main memory prefetching if request queues are managed in the disk controllers rather than in the host. Checking the disk cache before queuing requests provides not only better request response time but also drastically improves balancing. A single cache performs better than a segmented cache for this method.

1 Introduction

Disk I/O has always been slower than CPU processing, and recent improvements in the speed of disk hardware have not kept pace with the increasing raw speed of processors [Patt94]. The problem is compounded in typical parallel architectures that multiply the processing and memory capacity without balancing the I/O capabilities. The most promising solution to the I/O problem is to extend parallelism into I/O subsystems. One such approach is to connect several disks to each of the processing nodes, spreading individual files across all disks. The increase of I/O bandwidth from single disks to multiple disks per processor is essential for database query processing, because it provides a balance for ever-increasing CPU throughput. Multiple disks could provide a significant boost in performance, if there are no substantial bottlenecks in the I/O subsystems.

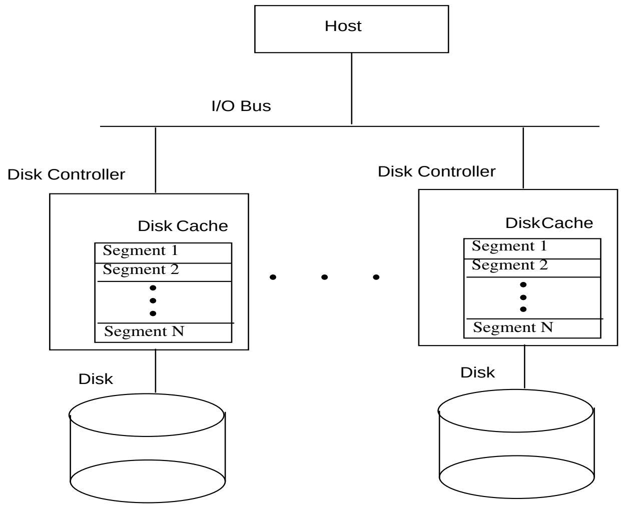

Figure 1: Multisegment disk cache.

Modern disks are equipped with significant amounts of cache memory for performance improvement. A disk cache is a buffer capable of holding portions of the disk address space contents; one of its primary use is for prefetching sequentially accessed data. If such a buffer can capture a significant fraction of the I/O operations then it can improve I/O performance, reducing or eliminating the predicted speed lag of the I/O subsystems [Smith85, Karedla94].

Our study evaluates the impact of disk cache structures on processing multiple file scans. Such a workload is typical for decision support and data warehouse applications, for multimedia systems retrieving large objects, for processing large volumes of raw data in scientific applications, for sorting big files involving reading and sorting multiple runs, etc.

*This research was supported in part by Grants NSF RIA IRI-9409845 and from NSF ASEND-EPSCoR

For this paper, we assume that the a cache of each disk is divided into several equal *cache segments*, as shown in Figure 1. Each cache segment is a unit of prefetching and can store several data pages. The size of a segment determines the amount of data prefetched at a time because one segment's worth of contiguous pages on the disk immediately beyond that page which was requested by a read command is retrieved and stored in a cache segment for transfer from the cache to the host on subsequent reads that request those pages. A disk controller manages the cache segments using a round-robin replacement policy. The operating system manages disk requests using an elevator algorithm (see Section 3).

The idea behind the cache segments is to provide caching benefits to several concurrent file scans. If the cache is not divided into segments, it supports only one file scan efficiently, and only if there are no other concurrent scans. Furthermore, if a workload consists of several parallel scans, each subsequent prefetching would overwrite the total cache, making it useless. That is why a disk cache can efficiently speed up the performance of sequential scans when the number of concurrent scans is no more than the number of cache segments. The number of cache segments is usually considered to be a constant determined in the disk controller at installation time.

It is generally accepted that if the number of sequential scans becomes larger than the number of cache segments, then the benefits of disk-level prefetching are questionable because scans overwrite segments of one another [Graefe94]. This would limit inter-scan parallelism as well as intra-scan parallelism, bounding the number of parallel runs of a sort-merge join or file sorting. In fact, as we observed in our experiments, the disk cache can still provide significant savings even for large numbers of scans, though cache hit ratio is worse. However, the parallel execution of large numbers of scans can degrade system performance because of another reason.

We observed an interesting phenomenon: when the number of scans exceeds the number of cache segments of an individual disk, the load of the entire system becomes unbalanced. One disk becomes a bottleneck, and the performance of all other disks is bounded by the performance of this disk. Most of the time, the bottleneck disk serves a majority of scans in the system whereas the other disks have a very light load. This bottleneck effect takes place for a parallel system as well, when several nodes, each of which is equipped with multiple disks, are connected in a shared-nothing manner.

This paper explore a number of mechanisms to ensure load balancing. We demonstrate that data placement decisions, such as reduced declustering, have a limited impact on load balancing. A load control method achieves better balancing than the placement decisions but discriminates against scans of smaller files. Dynamic dividing of a cache into segments improves balancing however requires changes in the disk controller.

A primary cause of imbalance is a variation of disk queue length across disks, and a segmented cache only exacerbates that difference in the disk's performance. We evaluate two methods to reduce the variation of disk queue length. The first method is prefetching in main memory which supplements or substitutes prefetching in the disk caches. It succeeds in load balancing with limited extra memory needed. Another method is checking disk cache content *before* queuing disk requests. Although such an ability is not always supported by operating systems and/or by disk controllers, we show that the method is the most successful for disk load balancing.

We investigate two classes of simulated systems in our analysis: closed-end systems and open-end systems. Closed-end systems study system performance for a fixed number of processed scans. We employ the average scan response time as a performance metric for closed-end systems. Open-end systems carry a workload of a variable number of scans. We explore the ability of systems to sustain a workload without a saturation as a metric for open-end systems.

The remainder of the paper is organized as follows: We begin by reviewing segmented disk cache organization and previous studies on disk cache use in Section 2. The simulation model used in this study appears in Section 3. We present the phenomenon of disk imbalance in Section 4. Experimental results of comparing a number of balancing techniques are given in Section 5. Our conclusions are summarized in Section 6.

2 Background

This section provides some background information relevant to placing and accessing files in multi-disk systems and reviews disk cache behavior.

Much of the previous work in I/O hardware parallelism has involved disk *declustering*, or stripping [Salem86]. In the declustering technique, a file is interleaved across multiple disks. The performance of data declustering was compared to a scheme that places entire relations on a single disk [Livny87]. The results showed that declustering improves performance except under high utilization. Full declustering as a data placement method is simple and robust. A primary advantage of full declustering is inherited balancing files across disks, even if file access patterns are not uniform [Copeland88]. A potential disadvantage is excessive number of switches between disks causing a larger portion of random disk accesses, and a larger traffic and start-up/termination overhead in multi-node parallel systems compared to non-declustering placement methods [Rahm93]. A recent study [Mehta94], shows that full declustering continues to outperform *partial declustering* (the method where a file is interleaved across a subset of disks rather than across all disks) in disk-bound systems. [Mehta94] demonstrates that full declustering provides better disk utilization than partial declustering whereas the CPU and network costs are not decisive. Data placement has also been studied for file systems in [Wolf89].

Most modern I/O devices are equipped with a significant amount of cache memory. Cache memories are embedded into controllers of individual disks or are placed into multidevice controllers managing several individual disk drives or a RAID. Cache memory sizes are ranged from 256 KB to 1 MB for individual disk controllers, and from 8 MB to 4 GB for multidevice controllers. Cache memory in disk drives tends to be more expensive than main memory because dual-ported static RAM in disk controllers has to be able to work with both the media device and the bus interface.

Disk cache is used for supporting disk reads and disk writes. Although some multidevice controllers contain separate read and write cache, most controllers for individual devices use a single cache for both reads and writes. A read cache can be used for caching frequently used data and for prefetching, i.e. for read-ahead. Popular caching strategies and replacement algorithms such as *random replacement*, *least recently used* and their variations are discussed in [Karedla94]. An analyses of cache polices for disk writing based on *write-through* (where data written to a disk media are not copied to a cache) and *write-back* (where written data must be copied to a cache) methods is given in [Miller93]. The latter method increases a use of cache but can cause some additional disk reads. A good review of modern disk drive features can be found in [Ruemmler94], and a review of I/O cache designs is given in [Karedla94].

[Worthington94] shows that disk scheduling algorithms that effectively utilize disk cache prefetching provide significant performance improvements for workloads with read sequentiality. The *cyclical scan algorithm* (C-LOOK), a combination of C-SCAN [Seaman66] and LOOK, achieves better performance than other seek-reducing algorithms, such as *first come first serve* (FCFS), *shortest seek time first* (SSTF), and LOOK. LOOK is a SCAN-like algorithm changing scanning direction if no pending requests exist in the current direction of movement.

Our paper concentrates on the analysis of prefetching. Prefetching can be implemented in individual disk controllers, multidevice disk controllers, or in the host. A commonly asked question is if caching and prefetching are more efficient at host level or at disk level. [McNutt93] studied this issue for caching, but the question remains open for prefetching. Host caches can use prefetching more intelligently than multidevice controllers or disk caches, since applications can better predict the sequential character of read requests. At the same time, disk controller designers are increasing the amount of cache they have and the complexity of their cache management software. The SCSI-2 interface supports command queuing that permits a disk controller to implement an intelligent optimization, such as seek ordering.

Examples of host-based prefetching include file system caches that prefetch a large portion of the file being read, and query optimizers in database applications. Thus, [Grimsrud93] proposed a caching strategy for a host-based segmented cache. The strategy includes maintaining a record of the order of past disk accesses for predicting future access sequences. Our study has another focus. We concentrate mostly on a disk-based segmented cache and on an analyses of disk load balancing for a multi-disk system. We also assumes predicted access sequences.

Most modern SCSI disks provide some form of prefetching. A read that hits in the cache can be satisfied "immediately", that is, in just the time needed for the controller to detect the hit and send the data across the bus. However, prefetching is one of the least specified areas of disk system behavior. Disks use various prefetching policies, differing in the segment replacement algorithm, the size of prefetch and the ability of prefetch to cross track and cylinder boundaries, etc.

A single read-ahead cache can provide effective support for only a single sequential read stream. If two or more sequential read streams are interleaved in a multiuser or a multitasking environment, benefits of prefetching into a single cache are questionable. Indeed, prefetching might be not sufficiently aggressive for smaller sizes of prefetch, such as the size of request, or can often result in overwriting prefetches of one another for larger prefetches, such as a track. The result is no benefit at all if a length of prefetch equals the cache size. This can be remedied by segmenting the cache so that several unrelated data items can be cached. For example, a 256-KB cache might be split into eight separate 32-KB cache segments by appropriate configuration commands to the disk controller. The popular Seagate Barracuda disk drive can divide its disk cache into up to 16 segments, with a default of 3 segments [Seagate93].

Our goal is to investigate the performance of multiple disks, each equipped with individual multisegment disk cache, for processing multiple concurrent file scans. We concentrate on the case when the total number of executed scans exceeds the number of cache segments of a disk.

3 Simulation Model

A simulator used in this study models a centralized system composed of one CPU and its memory buffer pool, four SCSI disks and a set of external terminals from which scans are submitted. Our simulator is based on the simulator of a parallel database machine that was used in a number of studies, e.g. in [Mehta93a, Shatdal93, Brown93, Mehta93b, Brown94], and is written in the CSIM/C++ process-oriented simulation language ([Schwetman90]).

The terminals model the external workload source for the system. For *closed-end* experiments, each terminal submits a stream of scans, one at a time, waiting for a response to each before sending the next one. The number of terminals regulates the average *MPL* (multiprogramming level) and disk utilization. For *open-end* experiments, the number of terminals is not defined, but new scans arrive at a certain rate, e.g. the rate of arrival can be constant or exponentially distributed with a certain mean value. The scheduler accepts scans from terminals and performs scheduling and memory allocation for all scans. The scheduler can impose a maximum system MPL if such a limit is determined in the experiment. Exceeding scans are queued in the scheduler and wait for execution.

The CPU has a speed of 80 MIPS and is scheduled by a round-robin policy with a 5 msec time slice. The buffer pool models a set of main memory frames, each 8 Kbytes in size, whose replacement is controlled via the 3-level LRU page replacement policy with *love/hate* hints (like those used in the Starburst buffer manager [Haas90]). The hints are given by the query execution operators when a page is unfixed, and they define three levels of value as follows: index pages are considered more valuable than data pages, and randomly-accessed data pages are considered more valuable than sequentially-accessed data pages for caching in the buffer pool for subsequent page requests. A memory reservation mechanism allows query execution operators to reserve memory for their working storage, thus preventing those reserved frames from being stolen while their reservation is in effect. When a scan prefetches into the buffer pool, the prefetched pages are marked as randomly-accessed pages

Parameter	Value
Wire Delay for 8K Message	0.5 msec
Scan Terminals	32 (varied)
CPU Speed	80 MIPS
Number of Disks	4
Page Size	8 Kbytes
Memory	2 Mbytes
Disk Cylinder Size	78 pages
Disk Seek Factor	0.266
Disk Rotation Time	7200 RPM
Disk Settle Time	1.0 msec
Disk Transfer Rate	4.76 MB/sec
Disk Cache Segment Size	4 pages
Number of Cache Segments	8
Tuple Size	200 bytes
Number of Relations	800
Number of Tuples per Relation	40,000 (varied)
Copy 8K Message	10,000 instr.
Start an I/O	1,000 instr.
Send (Receive) 8K Message	1,000 instr.
Read Tuple from Buffer Page	300 instr.
Initiate Scan	20,000 instr.
Terminate Scan	5,000 instr.

Table 1: Simulator and database parameters.

until being processed when the buffer manager downgrades them to the lowest level.

The simulated disks model a slightly simplified[1] Seagate Barracuda Model ST11950 (1.69 GB formatted, 5.25") disk drive [Seagate93]. Each disk is connected with the host by an individual SCSI channel. The disk provides a 256 Kbyte disk cache which is divided in the simulator into eight 32 Kbyte cache segments for prefetching pages of sequential scans. Each I/O request, along with the required page number, specifies whether or not prefetching is desired. If prefetching is requested, one segment's worth of disk blocks (4 pages) is read from the disk into a cache segment immediately after the page originally requested has been transferred from the disk to memory. The requester is not released, i.e. the requested page is not available and a next disk request is not processed, until the entire cache segment is loaded (synchronous cache loading). Subsequent requests to one of the prefetched blocks can then be satisfied without incurring an I/O operation. A simple round-robin replacement policy, used, in particular, in ST11950, is employed to allocate a cache segment if the number of concurrent prefetch requests exceeds the number of available cache segments. The disk queue is managed in OS by an elevator algorithm [Silberschatz94]. In the elevator queuing, there are two request queues at any time: one for the current disk arm direction UP, and another for the current direction DOWN. When the current queue is empty, the current disk arm direction is changed to opposite before reaching a boundary cylinder, that is the elevator algorithm is similar to LOOK rather than to SCAN. A new request is placed in one of the two queues based on the current disk arm direction and the current cylinder, and the requests in each queue are ordered according to the number of the requested cylinder. Multiple requests in a queue for the same cylinder are processed in the order of their arrivals. When the top request from the current queue has been scheduled for execution, the disk checks the cache. If this request is satisfied from the disk cache, the arm does not move. The cache content does not affect the queuing of requests. All deviations of the disk policy described above, such as the variation of the number of segments or the placing of the request queuing after checking the cache content, are described together with related experiments.

[1] The simulator uses a random value instead of an accurate modeling for a latency delay and does not charge for crossing a track bound.

The important parameters of the simulated system are listed in Table 1. The CPU processing costs for various operations are based on actual instruction counts taken from the Gamma prototype on the iPSC/2 [DeWitt90]. The system memory of 2 Mbytes is obviously small, and was chosen solely to keep the simulation time low. The memory size is an unimportant parameter for processing relational scans, because each scan consumes only 2 memory pages (for supporting double-buffering) and the system is never memory-bound, even for in-memory prefetching. "Disk Seek Factor" is multiplied by the square-root of the number of passed cylinders to obtain the seek time. There is at most one outstanding request at each disk at a time.

The database consists of 800 relations, 8 MB each. Each relation contains 40,000 tuples in most experiments. The tuple length is 200 bytes. A relation is stored on disks in four 2 MB partitions, one partition per disk. All parameter deviations are described together with related experiments. All experiments provided for 30 min of simulated time; the number of scan completions is at least 600 in each experiment. This provides a 99% confidence level that the results of evaluating the average scan response time are within 2% accuracy. The workload consists of sequential scans of relations submitted from a number of terminals. Each terminal chooses a relation randomly from the database with a uniform distribution. A file scan processes a partition located on one disk, then a partition on another disk, etc. First partitions of different files are uniformly distributed across the disks, with subsequent partitions stored on the other disks using a round-robin placement policy. Such a workload can be characterized by sequential accesses to a small portion of the database and very low memory requirements and CPU consumption. This permits concentrating attention on the performance of the I/O subsystem. The number of terminals submitting scans is varied with experiments. This number determines *sysMPL*, that we define as the total number of executed scans in the system, unless the sysMPL is specifically limited.

4 Disk Cache and Load Imbalance

In this section we present some basic simulation results on processing multiple scans to illustrate a use of disk cache with a high workload. We estimate the disk cache hit ratio for high workload. We introduce the load imbalance phenomenon that results and explain its nature.

4.1 Disk Cache Hit Ratio for High Workload

A round-robin replacement policy is used to allocate cache segments if the number of concurrent prefetch requests exceeds the number of cache segments. The disk queue is managed by an elevator algorithm (see Section 3). As the disk queue is under the control of the OS while the cache is managed by the disk controller, the cache content does not affect disk request queuing. The CPU and channel time of page processing is small compared to a disk read and varies insignificantly between requests; that is why file scan processes issue requests for new pages at about the same time as they receive previous pages from the disk.

The system workload consists of multiple relation scans and is disk-bound. A scan is implemented using two asynchronous processes per scan: the higher-level process processes page content while the lower-level process checks the shared buffer pool for requested pages and delivers missed pages from a disk. The higher-level process issues a request for a next page after completing processing its current page and passes the request to the lower-level process. The lower-level process returns the requested page from the buffer pool if the page is already there or delivers its from a disk otherwise.

It is presumed that the system can use delivered pages of relations for further memory-intensive query processing, e.g. for building hash-join tables, sorting, etc. Allocating the maximum amount of memory to a hash-join query will allow it to execute with a single scan of each relation. Allocating less memory increases the number of I/Os required in a linear fashion [DeWitt84]. The choice of optimal I/O length along with the optimal memory allocation between different components of database workload such as hash-joins, scans, and transactions is a complex problem and its solution depends on per-component performance goals. Although we do not consider such a complex workload, we assume a lower memory allocation priority for scans and limit the I/O size to one 8 KB page, allocating only two memory pages per scan in the buffer pool. We follow with this setting a number of database performance studies [Brown93, Mehta93b, Brown94].

Relations are of the same length and fully declustered across disks into partitions, one partition per disk. Each partition is stored on a disk in contiguous blocks. A scan processes file partitions sequentially one after another. We denote by *diskMPL* the number of scans executing concurrently in the disk, and by sysMPL the total number of scans executing in the system (see Section 3).

Consider an execution of different numbers of file scans in the presence of disk cache. If diskMPL = 1, a single file scan is executed and monopolizes the entire cache. Each reading from the disk media loads a current cache segment and 4 subsequent reads are made from cache. Therefore, the expected disk cache hit ratio is 80%. The disk processes a single scan workload in the fastest way because a file partition is always allocated in contiguous cylinders and there are no arm movements causing a seek delay.

If diskMPL is slightly larger than the number of segments, performance depends on a request queuing policy. With a FIFO queuing policy, the disk cache would be useless because one prefetching request overwrites the data prefetched by another request. The C-SCAN and C-LOOK

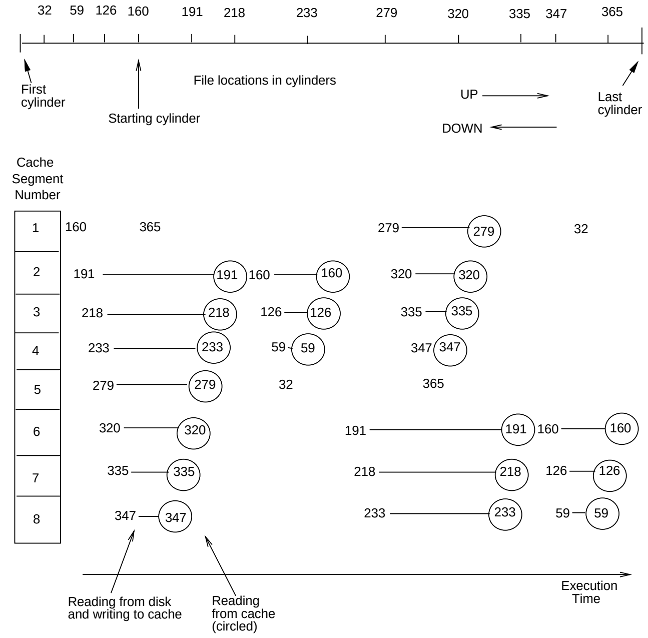

Figure 2: Example of cache use for 8 cache segments and diskMPL = 12.

algorithms, which order all requests by cylinder numbers and uses a single direction of arm movement, would be useless as well. However, a cache remains useful if elevator queuing with two request queues is employed.

Figure 2 illustrates a cache use for diskMPL = 12 and 8 cache segments. Let 12 scanned files be located in 12 cylinders with numbers shown at the top of Figure 2. Let the starting cylinder be 160 and the current disk arm direction be UP. When the arm moves UP, the disk reads from 8 files located on the cylinders from 160 to 347 and prefetches to 8 segments one after another. Then the arm moves UP further, and the disk reads from a file located on the cylinder 365 and prefetches to the first segment (because segments for prefetching are chosen in a round-robin fashion), overwriting a prefetch from the cylinder 160. After reading from the cylinder 160 the arm moves DOWN and the controller processes requests for cylinders in the descending cylinder order starting from 365. The controller reads one page from cache for each of the requests for cylinders from 347 to 160 as they were prefetched when the arm was moving UP, and the arm moves without latency delays from 365 to 160.

As mentioned in Section 3, the chosen elevator algorithm is similar to LOOK rather than to SCAN. LOOK appears to be better than SCAN for the studied workload because it avoids an arm movement to the boundary cylinder after processing the last request in the current queue. This move is rarely helpful as it can pick up only those additional requests from the second queue that belongs to new scans on the disk which happens rarely.

Figure 3 shows the disk cache hit ratio as a function of the diskMPL for a system equipped with only one disk. We include results for round-robin and random disk cache segment replacement algorithms. For this experiment, the size of files is reduced to 10,000 tuples to keep the size of

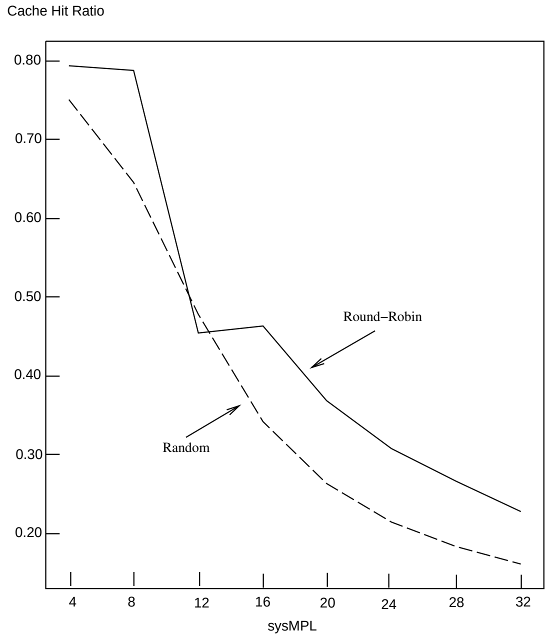

Figure 3: Disk cache hit ratio, single disk.

the partition equal to the size of the partition in the 4-disk system case. Round-robin outperforms random replacement for practically all MPLs. Even when the diskMPL exceeds eight, the number of cache segments, the hit ratio is considerable. For example, the hit ratio for round-robin is 45% for diskMPL = 12.

The hit ratio for round-robin for diskMPL = 16 is slightly better than for diskMPL = 12. This happens because about 8 consequent scans are served from the cache on average at a time for diskMPL = 16 versus only about 4 such scans for diskMPL = 12. This effect of increasing the hit ratio does not appear for larger sysMPLs, e.g. for 20 and 24, due to a larger overall reduction of the caching effect. A scan served from the cache for $diskMPL > 8$ alternates reads from the media and from the cache and has its cache hit ratio about 50%.

Disk response time, our primary metric of interest, is affected by two opposite trends for increasing diskMPL. First, the disk cache hit ratio decreases when diskMPL is larger than the number of disk cache segments. Second, an elevator algorithm is more efficient for a higher diskMPL.

Let us consider a configuration with 4 disks. The disk cache hit ratio remains considerable for experiments with a 4-disk system as shown in Table 2. Each disk has eight cache segments; the total number of segments in four disks is 32. Therefore, if scanned files are chosen randomly, we could expect that all disks are loaded approximately evenly. For example, if sysMPL = 32 and the system has 4 disks, we could expect diskMPL $\approx$ 8.

However, the actual diskMPL varies drastically between disks. Table 2 shows average values of diskMPL for 4-disk system and sysMPL = 32. We used relations of 40,000 tuples in the experiments. One disk is a bottleneck and the load of disks is unbalanced. This imbalance does not appear in an identical experiment with disks that are not equipped with disk cache and, therefore, do no readahead (Table 3).

This bottleneck effect is typical and was observed for each sysMPL in the range from 20 to 32. Let $avgDQ$ is the average disk queue length for a given disk. Figure 4 shows the standard deviation of avgDQ for four disks in the system for various sysMPLs. The deviation rises sharply starting from sysMPL = 20.

Disk #	Avg diskMPL	HitRatio	DiskUtil
1	0.51	0.793	0.278
2	30.31	0.244	1.000
3	0.50	0.790	0.270
4	0.49	0.793	0.272

Table 2: Disk with 8 cache segments; sysMPL = 32, 4 disks.

Disk #	Avg diskMPL	DiskUtil
1	8.46	0.937
2	8.17	0.931
3	7.70	0.950
4	7.57	0.933

Table 3: Cache-less disk; sysMPL = 32, 4 disks.

The explanation of the bottleneck disk effect is the following: New scans start randomly on disks of the node, and the number of started scans can be different for different disks. It happens that there are more new scans on one disk than on the others. This difference may be arbitrarily large given a sufficiently large number of trials. There are two factors causing a difference in the request response time of the disks processing different numbers of scans. The first factor is a difference in disk queue lengths among disks. Second, when one disk starts to process more scans than it has cache segments, the average response time of this disk increases because the cache hit ratio falls sharply. The processing of scans executed on the disk with a larger number of scans slows down.

Each scan is executed consecutively on each disk of the system. The disks process a scan as a "pipeline": when disk n completes the scanning of its partition of the file, disk $n+1$ starts to scan its partition and so on, until scanning the last partition has been completed. The appearance of a "slower" disk breaks the pipeline. The slower disk fails to complete its portions of scans and to pass these scans to the next disk of the pipeline. This next disk becomes more productive because it completes and passes some of its scans farther into the pipeline but does not receive new scans from its slower predecessor. In contrast, the disk previous to the "slower" disk completes and passes some of its scans to the slower disk, slowing down this disk even more. As a result, the disk which has happened to slacken the pace can eventually accumulate almost all scans and continues to remain a bottleneck. This effect represents a case of a convoy phenomenon [Blasgen79].

4.2 Bottleneck Migration

One interesting phenomenon is the migration of a bottleneck among disks within the system. It happens occasionally, though not often, that the current bottleneck disk completes several scans within a short time interval. This disk passes these scans to the next disk in the pipeline. If the number of passed scans is sufficiently large that the total number of scans of the next disk exceeds the number of cache segments, then this next disk becomes a new bottleneck. As we observed with tracing experiments, the variation of the number of new scans starting on each of the disks is only a secondary reason for the bottleneck migration; the primary reason is migration of scans from a current bottleneck. We observed that such an event does not happen often, maybe once in a thousand file scan completions. We observed the migration in longer (1-2 hours) experiments than reported in this paper and in 30-min experiments with shorter files. The actual frequency of such events depends on the database and workload characteristics.

In most cases, the next bottleneck is the next disk of the pipeline, though we also observed another. It may happen that a disk other than the next disk of the pipeline after the current bottleneck becomes the next bottleneck. It may even be the disk which precedes the current bottleneck disk. The primary reason for this, as we ascertained with tracing, is the migration of a large bunch of scans along the disk pipeline, supplemented by a deviation in the arrival of new scans to different disks.

Although the bottleneck migration itself does not affect the overall performance of disks within the system, a knowledge of this phenomenon is useful in analyses of statistical data obtained from experiments. As we observed repeatedly, after a prolonged period of simulation, values of average diskMPL and disk utilization (i.e. the portion of time when the disk is busy) may look suspiciously balanced between the disks, whereas in reality there was one bottleneck at each time, migrating between disks during the experiment.

4.3 Disk Load Imbalance: Multi-Node Case

The disk bottleneck effect presented above for a centralized system appears also in a multi-node parallel shared-nothing system. Let the system consists of 16 nodes, with 4 disks each, connected in a shared-nothing manner [DeWitt92]. There are still 800 relations stored in the system, of 640,000 tuples each, i.e. each relation is 16 times larger than in the single-node case. Relations are fully declustered across all nodes and across all disks within each node. Each scan is executed *in parallel* on all nodes but still sequentially on the disks within each node. A parallel execution on nodes was chosen in order to limit a pipeline effect to the disks within an individual node rather than to create a pipeline of 64 disks. However, although scans are executed in parallel on nodes, the bottleneck effect still appears.

The effect of a bottleneck disk appearing in one of nodes affects the performance of the other nodes in the following way. For each scan, a node having the bottleneck disk completes its partition of the scan later than the other nodes. The other nodes do not start new scans because the total number of scans processed by the system is constant. All

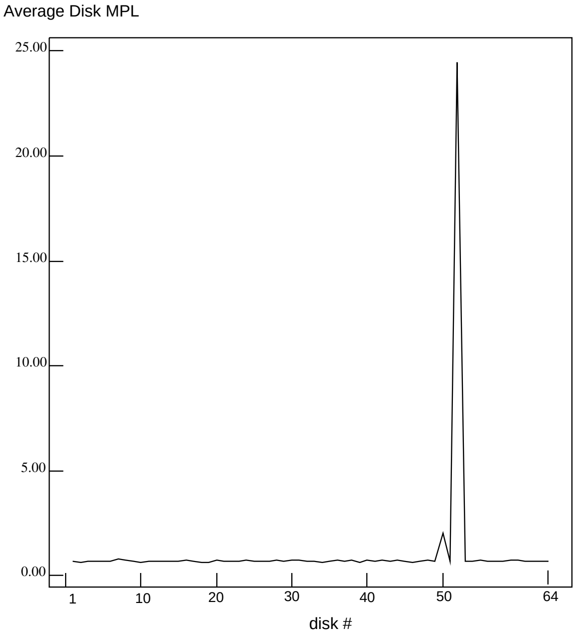

Figure 4: Effect of a bottleneck disk: 16 nodes with 4 disks.

nodes but that with the bottleneck disk process a light load because the frequency of arriving new scans is reduced.

Figure 5 shows the average diskMPL of each disk in the 16-node system for sysMPL = 32. One disk in one of nodes is a bottleneck, and one more disk in the same node has a larger than average utilization because a migration happened some time before the end of the experiment. If the experiment would continue longer, the smaller peak would grow, while the larger peak would diminish. A migration of a bottleneck node also happens but much more rarely than for disks within a node.

The effect of bottleneck disk is located within an individual node but affects a performance of entire system. We limit our consideration of the bottleneck effect to a single node system further in the paper because, as we observed, an achievement of load balancing within a node eliminates an imbalance among nodes as well. This happens due to to a parallel execution of a scan on nodes versus its sequential execution on disks within a node.

5 Load Balancing Methods

This section presents an experimental comparison of several approaches for better balancing disks in a single-node system. They include:

- limitation of declustering degree across disks
- load control
- MPL-adaptive number of cache segments
- partial prefetching in host memory
- checking the cache before request queuing

We conclude the section with analyzing the ability of these algorithms to sustain a variable workload without system saturation. The stored database contains 800 relations of 40,000 tuples each for all experiments in this section.[2]

5.1 Partial Declustering

As we pointed out in Section 4, the reason for a disk bottleneck is a concentration of scans on one disk. The first heuristic method of balancing the load across disks is a reduction in the degree of file declustering. This allows separate workloads of disks and avoids inter-disk convoys. However, the load on disks may sometimes differ significantly because of a random choice of new scans.

Figure 6 compares a full 4-disk declustering (**4D**), 2-disk declustering (**2D**), and no-declustering (**1D**). In full 4-disk declustering, files are interleaved across all disks with 4 partitions; this is our basic placement method. In 2-disk declustering, each file is divided into 2 partitions placed on 2 subsequent disks. In no-declustering, each file is stored on an individual disk. First disks for each file are chosen in round-robin to ensure a balanced placement in each of placement methods.

Each disk stores an equal number of files in partial declustering schemes. No-declustering outperforms full declustering for higher MPLs (sysMPL = 28 and sysMPL = 32) by a modest margin of about 15%, and this does not compensate a loss of robustness provided by full declustering. Full declustering is robust in automatic balancing a disk load for non-uniform file access patterns.

Although each scan is processed on a single disk in the no-declustering case and there is no pipelining, the disk bottleneck effect still takes place here. This shows that a difference in disk queue lengths among disks that happens due to a random choice of scanned files is the primary factor of load imbalance (see Section 4.1). The convoy effect caused by moving scans among disks is the only secondary factor of imbalance.

5.2 Load Control

We define a load control as a limitation of the maximum number of jobs that can be concurrently executed in the system. Load control is a simple method to reduce resource contention. This method is popular in operating systems to prevent thrashing. Limiting the sysMPL to *MaxMPL* by queuing excessive submitted scans before executing, we reduce the diskMPL and improve both balancing and cache hit ratio. We include the performance results of the load control method in Figure 7. We use MaxMPL = 8, denoted as **MaxMPL8**, which achieves the best performance compared with other levels of load control in configurations with 8 cache segments.

The load control is a very simple and convenient method of balancing because it can be used on demand and does not require special data placement. However, the high-level load control has several drawbacks resulting from its unfairness to queries suspended from execution. First, if suspended processes must produce intermediate results consumed by other processes, e.g. pipelining subqueries, or by end users, then

[2]We experimented also with databases containing files of variable lengths and saw results consistent with those reported here.

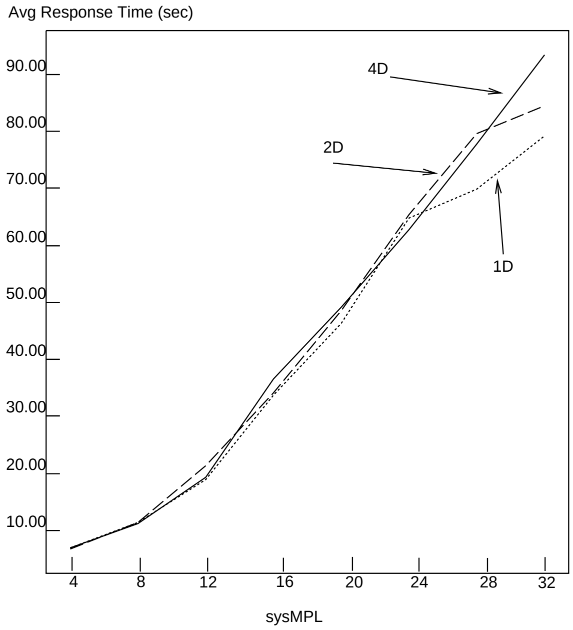

Figure 5: Partial declustering.

the load control leads to degradation of their response time. Second, the load control tends to penalize smaller queries, increasing their response time without regard to query size [Soloviev94]. Third, process schedulers providing the load control are notoriously inaccurate in predicting loads of individual disks when scheduling a mixed workload of queries of different complexity and hence are ineffective in preventing pathological behavior of a disk workload.

5.3 MPL-Adaptive Cache Segmentation

The methods described so far are based on data placement and system tuning actions. They do not affect cache management algorithms in a disk controller. The next method suggests new disk controller algorithms.

We introduce *MPL-adaptive Cache Segmentation*, a disk cache management algorithm where a disk controller estimates dynamically the diskMPL and adjusts the number and sizes of individual segments according to the diskMPL. The main idea is to have as many segments as there are scans in progress and to allocate one segment to each scan, dividing cache memory pages evenly between segments.

A first step in developing MPL-adaptive Cache Segmentation is designing a technique for a disk controller to learn its diskMPL. The general idea is that the controller counts page requests produced by different file scans during each pass over the cylinders. The number of scans counted during an arm pass UP is used as a predictor of diskMPL for a subsequent pass DOWN; the number of scans counted during a pass DOWN is used as a predictor of diskMPL for a subsequent pass UP. This prediction is based on the expectation that the number of scans processed during the previous pass is a good estimation of the current number of scans.

To implement this idea, the controller keeps a list of page addresses requested by prefetching requests during a current

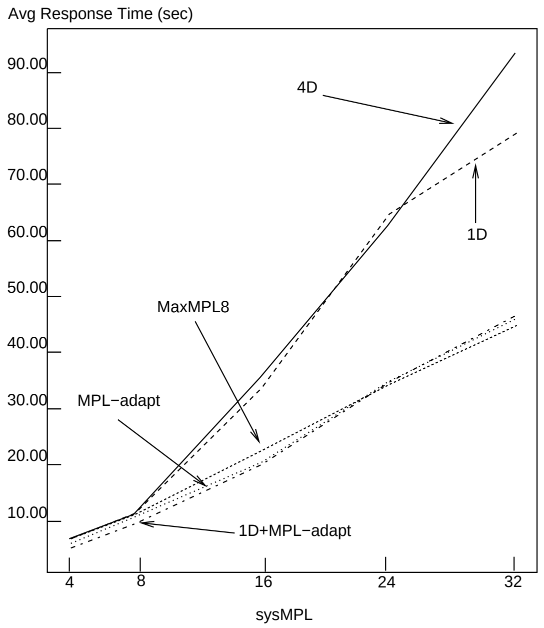

Figure 6: MPL-adaptive Cache Segmentation and Load Control: average response time

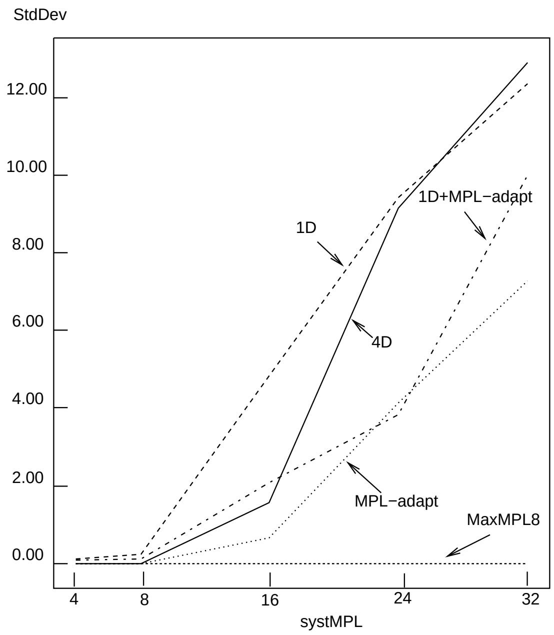

Figure 7: MPL-adaptive Cache Segmentation and Load Control: standard deviation of average disk queue length.

pass. Considering only prefetching requests helps to pick out requests related to file scans, ignoring random I/Os.

MPL-adaptive Cache Segmentation allocates segments as following. Initially, the current segment is a first segment of the cache. For each prefetching request, a new segment is created with a length equal to $\lfloor totalCacheSize/diskMPL \rfloor$. Space for a new segment is allocated starting from the beginning of the current segment. A new segment may occupy more or less space than the current segment. If the newly created segment is larger than the current one then several subsequent segments starting from the current one are destroyed and their space is used for the new segment. If a new segment requires a smaller space than the current segment then the current segment is split into two segments: the new one and the one holding the remaining pages of the current segment. An analogous procedure is applied for splitting the last destroyed segment if a new segment needs its space only partially. The next segment behind the created one becomes the new current segment after the prefetching request has been executed.

The MPL-adaptive Cache Segmentation algorithm has two advantages compared to a round-robin segment replacement. First, it provides better cache hit ratio for low values of diskMPL. For example, if diskMPL = 1 then the algorithm treats an entire cache as a single segment and each prefetching request prefetches several times more pages than with fixed-segment algorithms. Second, the algorithm serves scans fairly, i.e. without penalizing small scans. A drawback of the MPL-adaptive Cache Segmentation is that it requires new disk controller algorithms.

Figures 7 and 8 compare the performance of full declustering (**4D**), no-declustering (**1D**), load control with MaxMPL = 8 (**MaxMPL8**), MPL-adaptive Cache Segmentation (**MPL-adapt**) and a combination of no-declustering with MPL-adaptive Cache Segmentation (**1D+MPL-adapt**) for different MPLs. Figure 7 shows the average response time of the algorithms, and Figure 8 presents the standard deviation of avgDQ. Both variants of MPL-adaptive Cache Segmentation provide a significant performance improvement over full declustering and no-declustering for both the response time and balancing. Their response time is close to the response time of load control. They outperform load control for smaller MPLs. However, disk balancing is significantly better for the load control. MPL-adaptive Cache Segmentation also improves the cache hit ratio. The improvement is from 79% for **4D** to 93% for **MPL-adapt** for sysMPL = 4 and from 66% to 80% for sysMPL = 32.

sysMPL	MPL-adapt	4D
24	4.14	9.17
32	7.27	12.90

Table 4: StdDev of average disk queue length.

5.4 Memory Prefetching

The approaches discussed so far do not provide a satisfactory solution to the balancing problem. The load control penalizes some scans in favor of others and underutilizes resources. The MPL-adaptive cache segmentation requires changes in controller algorithms and, although it improves performance and balancing, it does not avoid disk bottlenecks. Thus, though the standard deviation of avgDQ among four disks is reduced for **MPL-adapt** compared with **4d**, it remains sufficiently high (Figure 8 and Table 4).

A natural alternative to improve balancing is to prefetch in host memory, supplementing or replacing disk cache prefetching. The most important advantage of host prefetching is that a request prefetched in the host is satisfied immediately whereas a request prefetched in disk cache is satisfied after disk queuing. Another plus is flexibility. The system can prefetch in the host for some scans and not prefetch for others based on the amount of available memory and on the total number of executed scans.

We introduce and model *DiskPrefControl(N)*, an algorithm that limits the total number of scans in the system that prefetch in disk cache. N is a parameter specifying the maximum total number of scans in the system served by prefetching in disk cache at any time. When a new scan starts, the counter of current scans served by disk prefetching in the system is checked. If this counter is smaller than N then the new scan prefetches in disk cache and the counter increments. The counter decrements when a disk prefetching scan finishes. As pointed in Section 3, each read request from the host in our model has a parameter specifying an execution with or without prefetching.

This algorithm makes a decision about a specific scan only once, at the time when the scan starts. Obviously, such a decision is not perfect for full utilization of cache segments of each disk because the workload of disks is varied. N represents the expected total number of disk-prefetching scans that can be handled by all disks of the system without overloading. This number plays the same role as a parameter in the load control method discussed above. But, unlike the load control, this method does not postpone any scan from an execution, it just forbids use of disk cache for some of them. DiskPrefControl(N) = 32 corresponds to the case when all scans are permitted to prefetch in disk cache (albeit this can sometimes cause cache segment overwriting). DiskPrefControl(N) = 0 is the case when no scans are permitted to prefetch there.

We consider two versions of DiskPrefControl(N). *DiskPrefControl(N)-* (**DPC(N)-**) provides no prefetching for scans not granted by disk-level prefetching.

In *DiskPrefControl(N)+* (**DPC(N)+**), each executed scan prefetches either in the disk cache or in the host. Up to N scans are permitted to prefetch in disk at the same time. The remaining scans of a workload are served by prefetching in main memory.

To understand how efficiently DiskPrefControl(N) utilizes a disk cache, we also implemented *LowLevelPrefControl*, the algorithm which makes a decision about disk-level prefetching for a scan more flexible. LowLevelPrefControl decides about disk-level prefetching for each disk accessed by the scan individually rather than only once per scan as DiskPrefControl(N) does. LowLevelPrefControl keeps the number of scans executing disk-level prefetching for each disk less than or equal to the number of disk cache segments. When a scan starts to read from a new disk, LowLevelPrefControl checks the number of scans currently working with this disk. If this number is smaller than the number of disk cache segments, then the scan is permitted to prefetch in the disk cache, otherwise not.

A scan is removed from a list of current scans of the disk when the scan finishes processing the file partition stored on this disk. We assume that a scan tells the system when

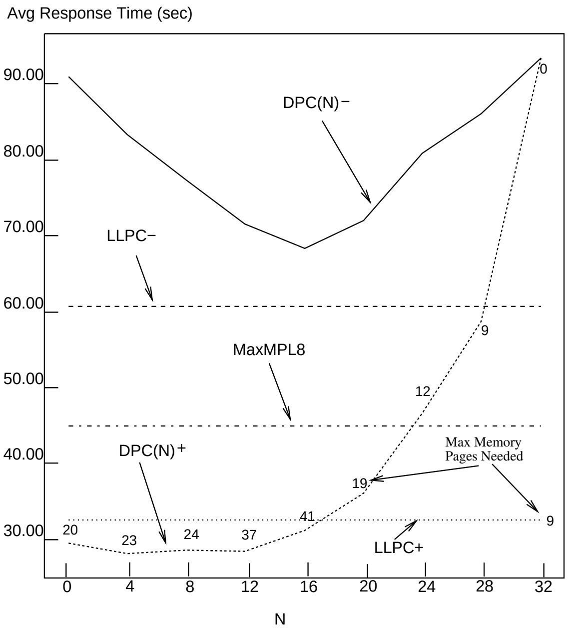

Figure 8: Disk-prefetching control algorithms for sysMPL = 32.

it happens. Note that the prefetching decisions for a given scan can vary from disk to disk.

We tested two versions of the LowLevelPrefControl algorithm. *LowLevelPrefControl-* (**LLPC-**) provides no prefetching for scans not granted by disk-level prefetching. *LowLevelPrefControl+* (**LLPC+**) provided them with memory prefetching instead of disk-level prefetching.

Figure 9 compares **DPC(N)-**, **DPC(N)+** for various N, and also **LLPC-**, **LLPC+**, and **MaxMPL8**. In all cases sysMPL = 32. When **DPC(N)+** and **LLPC+** algorithms prefetch in memory, they use a shared buffer pool to store prefetched pages. Numbers on the graphs for **DPC(N)+** and **LLPC+** show the maximum number of buffer frames occupied by prefetched pages. Response time of **DPC(N)+** sharply reduced for decreasing N, i.e. with an increase of scans prefetching in memory, with simultaneous reducing scans prefetching in disk cache. This shows that memory prefetching makes disk-level prefetching practically useless, with low memory overhead. The number of memory frames for storing prefetched pages reduces with an increase of scans prefetching in memory (i.e. for decreasing N) for $N > 16$. This is because the system becomes more balanced due to increasing CPU utilization (from 25% for N = 32 to 78% for N = 0), and memory-prefetched pages are processed faster.

A significant performance advantage of memory-prefetching over disk-prefetching happens because the prefetch cache memory is searched before queuing in this case, but only after any queuing costs in the disk-level cache case. Placing cache checking before the request queue helps not only to reduce the response time of individual requests but also improves disk balancing. Thus, the standard deviation of avgDQ among four disks is reduced from 12.91 for **DPC(32)+** (prefetching only to disk cache) to 0.80 for **DPC(16)+** (a half of scans prefetch to disk cache and

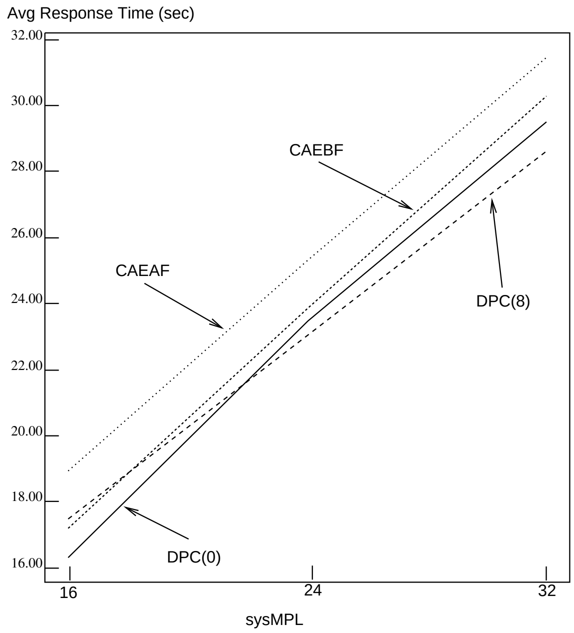

Figure 9: Cache-aware queuing.

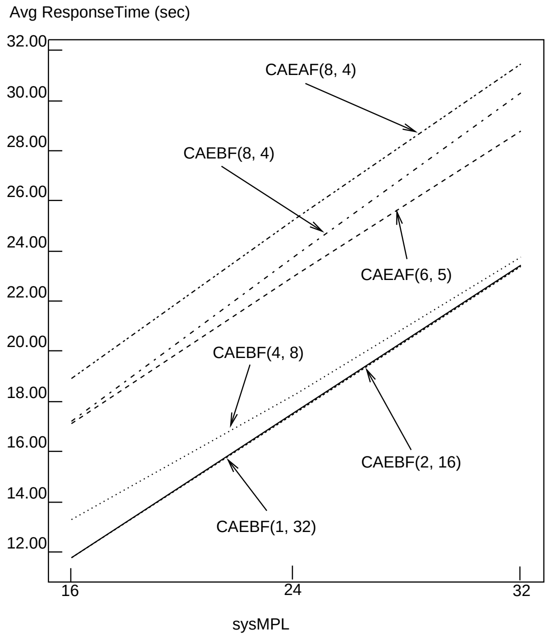

Figure 10: Cache-aware queuing: reduced numbers of segments.

another half of scans prefetch to memory) and to 0.36 for **DPC(0)+** (prefetching only to memory) for the experiment shown in Figure 9.

5.5 Cache-Aware Queuing

As we demonstrate in the previous subsection, memory prefetching outperforms prefetching in multi-segmented disk cache. But being encouraged by a superior performance of *before queuing* prefetching, we experiment with disk management algorithms where a request queue is supported within the disk rather than in the host as we considered so far.

The SCSI-2 supports queuing I/O requests which allows a disk to accept multiple I/O requests for execution at a later time. Although an implementation of queuing I/O requests is optional in SCSI-2, a number of current SCSI disks do command queuing in the drive. For instance, the HP 3.5" products even do 2D (rotation-position) scheduling to get better performance. HP-UX will allow up to 8 outstanding requests to be queued inside each disk drive that has this function.

Researchers of storage systems are also starting to investigate disk scheduling methods based on in-disk queuing and on incorporating the knowledge of the disk cache. [Worthington94] studied the SPCTF and ASPCTF, a cache-aware modification of the SPTF algorithm (*Shortest Positioning Time First* [Seltzer90]). SPCTF and ASPCTF are disk scheduling algorithms which track the contents of the disk cache and execute first those requests that can be satisfied from the cache.

The maximum number of requests in a disk at a time observed in our experiments is equals the sysMPL. We assume that a disk controller can handle a queue length up to the value of sysMPL for supporting cache-aware algorithms.

We investigate *Cache-Aware Elevator* (**CAE**), the in-disk scheduling algorithm which first checks a disk cache for an outstanding request. If it cannot be satisfied from the cache, the request is scheduled by the elevator method implemented in the disk controller. We consider two versions of **CAE**. *CAE-Before Current* (**CAEBF**) tries to satisfy a new request from a disk cache immediately when the request is received by the disk and does not wait for a completion of the currently processed request from the disk media. This assumes asynchronous accesses to the cache and to the media and can cause a contention of two requests if one of them is read from the cache and another is read from the media. This contention is ignored in our simulations. *CAE-After Current* (**CAEAF**) waits for completion of the currently processed request before checking the cache. This enforces a synchronous execution of requests and eliminates any possible bus contention.

Figure 10 compares the performance of cache-aware elevator (**CAEBF, CAEAF**) and memory prefetching (**DPC(0)+, DPC(8)+**) for various MPLs. Memory prefetching methods have a lower CPU cost than the studied disk-level prefetching algorithms. This is because the memory prefetching methods transfer five pages a time from a disk to main memory (one page that was requested plus four prefetched pages) and the total number of I/O operations at host is five times less than for the disk-level prefetching algorithms. This reduces the total CPU cost for starting I/Os. Memory prefetching outperforms cache-aware elevator due to a lower CPU cost for starting I/Os, but the difference is small (less than 5%). As the CPU utilization in this experiment is 70-80%, this matters. Both memory prefetching and cache-aware elevator achieve good disk load balancing and good balancing of CPU/disk utilization. Thus, the standard deviation of avgDQ, the average disk queue length, among four disks is 0.19 for **CAEAF**, 0.06 for **CAEBF**, and 0.36

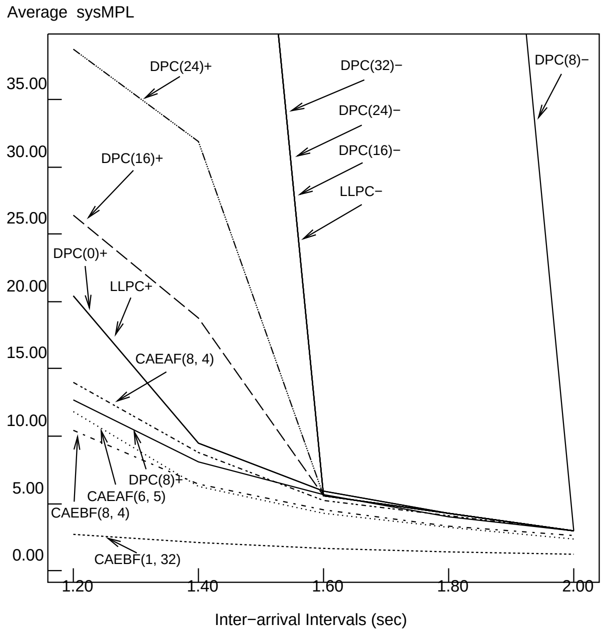

Figure 11: Open system: constant inter-arrival intervals.

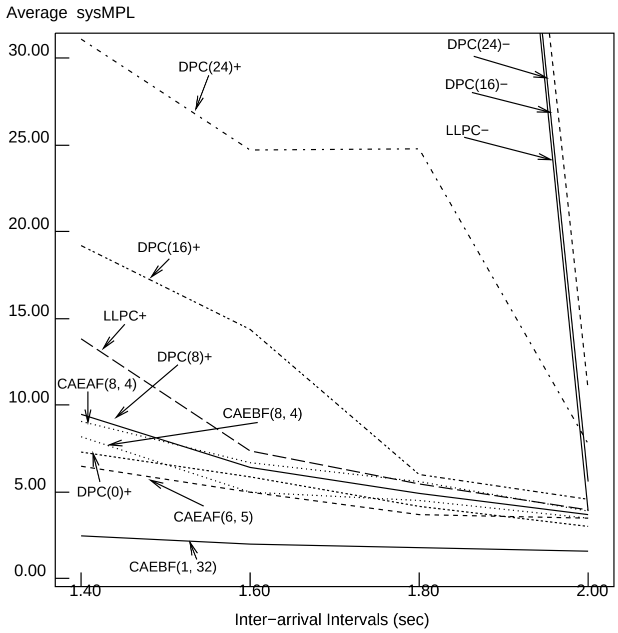

Figure 12: Open system: exponentially distributed inter-arrival intervals.

for **DPC(0)+** when sysMPL = 32. The disk utilization is 90-95% in the experiment of Figure 10.

If a previous scan request was satisfied from the cache the scan sends its next request to the disk much faster in cache-aware elevator algorithms than in algorithms checking the cache after queuing. This encourages dividing the cache into fewer segments increasing their size because it helps to increase CPU utilization and, in turn, to improve balancing between CPU and I/O subsystems. We consider the performance of **CAEBF(S, P)** and **CAEAF(S, P)**, where S is the number of cache segments and P is the number of 8 KB pages per segment.

Figure 11 shows the performance of **CAEBF(S, P)** and **CAEAF(S, P)** for a number of values of S and P such that $S * P \leq 32$, and for various MPLs. **CAEBF(8, 4)** and **CAEAF(8, 4)** correspond to **CAEBF** and **CAEAF** shown in Figure 10. We included **CAEAF(6, 5)** in Figure 11 because this combination of S and P demonstrated the best performance for **CAEAF(S, P)**.

CAEAF(S, P) is most efficient when $S = P + 1$. Indeed, assume that scan A prefetches P pages in one cache segment. Readings a prefetched page by scan A alternate with readings from the disk media by other scans. This is because time of reading from the media is much larger than time between requests from scan A and because a request from scan A waits for a completion of the current media request. Thus, scan A reads the first prefetched page from the cache segment after reading from the media by another scan, say B, which caused one cache segment to be overwritten by B. Scan A reads the second prefetched page from the cache segment after reading from the media by a third scan, say C, which causes another cache segment to be overwritten by C, etc. Scan A reads $S - 1$ prefetched pages from the cache segment before this segment is overwritten by some scan. Therefore, $S - 1 = P$. If $S < P + 1$, then the segment is too large and some scan overwrites the segment before all prefetched pages has been consumed by scan A. If $S > P + 1$, then the segment is too small and scan A reads from the media unnecessary often, i.e. before its segment can be overwritten.

CAEBF(S, P) is most efficient with $S = 1, P = 32$, but pages are delivered to the host too fast, and the system becomes CPU-bound. The cache hit ratio in this case is 97%, the CPU utilization is 100%. The system remains CPU-bound for $S = 2, P = 16$, with the hit ratio 92% and CPU utilization 100%. Thus, response times for **CAEBF(1, 32)** and **CAEBF(2, 16)** are almost identical. The hit ratio drops to 88% for $S = 4, P = 8$ but does not affect the scan response time too much because the system still has CPU utilization 98% for sysMPL = 32. The response time gets 8% worse for sysMPL = 32 because CPU utilization is reduced to 88% here.

5.6 Experiments with Variable Workloads

The experiments reported above were done for stable MPLs with the closed-end system. A reasonable suggestion is that the studied MPLs, such as 32 or 24 concurrent file scans, are too high, and that such intensities happen rarely in real workloads. By contrast, real workloads are highly variable. An important characteristic of a scan processing method is its ability to handle workload bursts without causing saturation of the system. While the results obtained for stable workloads are interesting for comparing relative performance of algorithms, variable workloads are more realistic and their analysis is important.

We changed the experimental model from closed-end to open-end for exploring variable workloads in this section. A performance criteria chosen for variable workloads is an ability of algorithms to sustain a workload without saturation, i.e. without a near-monotonic growth of the number of yet uncompleted scans. The chosen performance metric is the average number of concurrent scans during an experiment. The duration of experiments remains 30 min, the same as before.

An external workload is simulated as a single stream of arriving scans. Inter-arrival intervals are constant or exponentially distributed.

Figure 12 shows the average sysMPL of the memory prefetching and cache-aware elevator algorithms for various constant inter-arrival intervals. Figure 13 shows the average sysMPL of these algorithms for exponentially distributed inter-arrival intervals with mean values shown on the X axis. For exponentially distributed intervals, **DPC(32)-** saturates at interval = 2.0 sec and **DPC(8)-** saturates at interval = 2.4 sec and are not included in Figure 13. Both figures demonstrate that **DPC(N)-** and **LLPC-** saturate quickly, while **DPC(N)+** and **LLPC+** are capable of handling significantly more intensive loads. Cache-aware elevator algorithms **CAEBF** and **CAEAF** perform at the same level as memory prefetching, and their variants with reduced numbers of segments provides the best performance. These experiments point out that the better algorithms improve disk response time, thus improving the load that can be supported.

6 Conclusions

Modern disk drives are equipped with a significant amount of on-board cache. Many of them have the cache divided into multiple segments. Multisegment disk cache can help to support prefetching for several concurrent read streams. Using detailed simulation, we investigated an execution of multiple concurrent file scans in a multi-disk system. We concentrated on the performance of multisegment disk caches for prefetching.

A disk cache has the potential to reduce I/Os even when the number of processed scans exceeds the number of cache segments. However, a small variation in the load between disks caused a convoy effect if the number of scans on one disk was larger than the number of cache segments. This led to a disk load imbalance and the appearance of a disk bottleneck degrading performance. The bottleneck effect could migrate between disks but continued to exist at one of the disks over the workload execution.

We investigated several approaches to achieve better disk load balancing. Changing from full declustering of files across disks to partial declustering had a limited impact. A load control limiting the MPL of the system achieved better results but tended to penalize smaller scans. An MPL-adaptive cache segmentation policy in the disk controller increased the cache hit ratio for both low and high MPLs and remained fair to all scans.

Main memory prefetching, replacing disk-level prefetching or combined with them, outperformed both load control and the MPL-adaptive cache segmentation and required only a modest amount of host buffer memory. Memory prefetching worked better than disk-level prefetching because the requests prefetched in memory were satisfied before disk queuing while requests prefetched in the disk cache were satisfied after queuing. The resulting decrease of the disk queue length not only reduced the request response time but allowed balancing the load of disks and avoided a bottleneck. Memory prefetching made disk-level prefetching obsolete and questioned the benefits of using the disk cache for prefetching.

However, prefetching could be supported on the disk side if a request queue was managed in the controller rather than in the host. In this case the prefetched requests could also be satisfied before queuing, as well as with memory prefetching. Cache-aware disk scheduling supported disk-level prefetching with a performance comparative to memory prefetching. Cache-aware scheduling solved the load balancing problem for studied system and workload parameters and appeared to be the only disk-level prefetching method that provided about the same performance as host memory prefetching. With cache-aware scheduling, a cache divided into a smaller number of larger segments outperformed a cache with the larger number of smaller segments, up to the limit where the system became CPU-bound.

In practice, workloads are highly bursty. Experiments with variable workloads showed the importance of disk load balancing for handling workload bursts without saturating the system. Relative merits of discussed algorithms remained the same as for stable workloads.

For future work, we would like to explore more complex workloads that include hash-join queries and short transactions and to evaluate the impact of larger sizes of reads, such as a track a time, on the performance of different job classes.

Acknowledgments

We thank David DeWitt for a suggestion to look at the problem of disk cache management and Kurt Brown, David DeWitt, David Lomet, and John Wilkes for their insights on the study and on various I/O issues. We are grateful to John Grosen for help in editing the manuscript and to John Wilkes and Liddy Shriver for a significant improvement of the presentation of this paper.

References

[Blasgen79] M. Blasgen, J. Gray, M. Mitoma, and T. Price. The Convoy Phenomenon. *Operating Systems Rev.*, **13**(2) (1979), pp. 20-25.

[Brown93] K. Brown, M. Carey, M. Livny. Managing Memory to Meet Multiclass Workload Response Time Goals. In *Proc. 19th Int'l VLDB Conf.*, Dublin, Ireland (1993), pp. 328-341.

[Brown94] K. Brown, M. Mehta, M. Carey, M. Livny. Towards Automated Performance Tuning For Complex Workloads. In *Proc. 20th Int'l VLDB Conf.*, Santiago, Chile (1994).

[Copeland88] G. Copeland, W. Alexander, E. Boughter, T. Keller. Data Placement In Bubba. In *Proc. ACM SIGMOD Conf.*, Chicago, IL (1988), pp. 99-108.

[DeWitt84] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, D. Wood. Implementation Techniques For Main Memory Database Systems. In *Proc. 1984 ACM-SIGMOD Conf. Management of Data*, Boston, MA (1984), pp. 1-8.

[DeWitt90] D. DeWitt *et al.* The Gamma Database Machine Project. *IEEE Trans. on Knowledge and Data Engineering*, **2**(1) (1990), pp. 44-63.

[DeWitt92] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Processing. *Comm. ACM*, **35**(6) (1992), pp. 85-98.

[Graefe94] G. Graefe. Sort-Merge-Join: An Idea Whose Time Has(h) Passed?. In *Proc. 10th IEEE Int. Data Engineering Conf.*, Houston, TX (1994), pp. 406-417.

[Grimsrud93] K. Grimsrud, J. Archibald, and B. Nelson. Multiple Prefetch Adaptive Disk Caching. *IEEE Trans. on Knowledge and Data Engineering*, **5**(1) (1993), pp. 88-103.

[Haas90] L. Haas, *et al.* Starburst Mid-Flight: As the Dust Clears. *IEEE Trans. on Knowledge and Data Engineering*, **2**(1) (1990), pp. 143-160.

[Karedla94] R. Karedla, J.S. Love, and B. Wherry. Caching Strategies to Improve Disk System Performance. *Computer*, **27**(3) (1994), pp. 38-46.

[Livny87] M. Livny, S. Khoshafian, and H. Boral. Multi-Disk Management Algorithms. In *Proc. ACM SIGMETRICS Conf.*, Alberta, Canada (1987), pp. 69-77.

[McNutt93] B. McNutt. I/O Subsystem Configuration for ESA: New Roles for Processor Storage. *IBM System J.*, **32**(2) (1993), pp. 252-264.

[Mehta93a] M. Mehta, V. Soloviev, and D. DeWitt. Batch Scheduling in Parallel Database Systems. In *Proc. 9th IEEE Int'l Data Engineering Conf.*, Vienna, Austria (1993), pp. 400-410.

[Mehta93b] M. Mehta and D. DeWitt. Dynamic Memory Allocation for Multiple-Query Workloads. In *Proc. 19th Int'l VLDB Conf.*, Dublin, Ireland (1993), pp. 354-367.

[Mehta94] M. Mehta and D. DeWitt. Data Placement in Shared-Nothing Parallel Database Systems. *Submitted for publication* (1994).

[Miller93] D. Miller and D. Harper. Performance Analysis of Disk Cache Write Policies. *To appear in* Microprocessors and Microsystems, (3) (1995).

[Patt94] Y. Patt. The I/O Subsystem - A Candidate for Improvement. *Computer*, 27(3) (1994), pp. 15-16.

[Rahm93] E. Rahm and R. Marek. Analysis of Dynamic Load Balancing Strategies for Parallel Shared Nothing Database Systems. In *Proc. 19th Int'l VLDB Conf.*, Dublin, Ireland (1993), pp. 182-193.

[Ruemmler94] C. Ruemmler and J. Wilkes. An Introduction to Disk Drive Modeling. *Computer*, **27**(3) (1994), pp. 17-28.

[Salem86] K. Salem, and H. Garcia-Molina. Disk Striping. In *Proc. 2nd IEEE Int. Data Engineering Conf.*, 1986.

[Schwetman90] H. Schwetman. *CSIM Users' Guide.* MCC Technical Report No ACT-126-90, MCC, Austin, TX (1990).

[Seagate93] *Barracuda 1 and 2 Disc Drive. Product Manual (Volume 1).* Publication number: 83327780, Rev. B, Seagate Technology (1993).

[Seaman66] P. Seaman, R. Lind, and T. Wilson. An Analyses of Auxiliary-Storage Activity. *IBM System J.*, **5**(3) (1966), pp. 158-170.

[Seltzer90] M. Seltzer. J. Chen, and J. Ousterhout. Disk Scheduling Revisited. In *Proc. Winter 1990 USENIX Conf.*, Sunset Beach, CA (1990), pp. 313-323.

[Shatdal93] A. Shatdal and J. Naughton. Using Shared Virtual Memory for Parallel Join Processing. In *Proc. 1993 ACM-SIGMOD Conf. Management of Data*, Washington, DC (1993), pp. 119-128.

[Silberschatz94] A. Silberschatz and P. Galvin. *Operating Systems Concepts*, 4th edition, Addison-Wesley (1994).

[Smith85] A. Smith. Disk Cache-Miss Ratio Analysis and Design Considerations. *ACM Transactions on Computer Systems*, **3**(3) (1985).

[Soloviev94] V. Soloviev. *MPL-Adaptive Algorithms for Multisegment Disk Caches.* Comp. Sc. Tech. Rep. TR 9407, North Dakota State University, Fargo, ND (1994).

[Worthington94] B. Worthington, G. Ganger, and Y. Patt. Scheduling Algorithms for Modern Disk Drives. In *Proc. ACM SIGMETRICS Conf.*, Santa-Clara, CA (1994), pp. 241-251.

[Wolf89] J. Wolf. The Placement Allocation Program: A Practical Solution to the Disk File Assignment Problem. In *Proc. ACM SIGMETRICS Conf.*, Berkeley, CA (1989).