Virtual Streams
Performance-Robust Parallel I/O

Z. Morley Mao and Noah Treuhaft
{zmao, noah}@cs.berkeley.edu

Computer Science Division
University of California at Berkeley

Abstract

We present Virtual Streams as a solution to performance heterogeneity for I/O-bound applications in a cluster environment. We describe our improvements to the current version of Graduated Declustering, an implementation of Virtual Streams, by exploring a new distributed load-balancing algorithm – response-rate based algorithm that reduces seek overhead, especially when there is no performance heterogeneity. This algorithm not only addresses performance difference among data producers, but also hides the varying rate of progress of data consumers. The system using this new algorithm is shown to be stable and to converge to its final bandwidth value within reasonably amount of time. A new mechanism, server request handoff, is developed to address perturbations occurring when there are many outstanding requests at the server. The system is extended with the ability to handle writes to provide both performance and data availability. Finally, a usable abstraction of our software infrastructure is developed and studied for the convenience of application programmers developing streaming I/O applications in a cluster environment.

1. Introduction

Clusters are an attractive architecture for I/O intensive applications for many reasons. The commodity components that compose a cluster offer excellent cost-to-performance ratios. Furthermore, the ability to start with a modest system and gradually add new components (incremental scalability) affords dual benefits: cluster growth can track application growth closely, and new components can track technology improvements. Most importantly, clusters offer excellent I/O parallelism and scalability. Available network links, I/O buses, and disk drives increase linearly with the number of cluster nodes, ensuring that I/O throughput scales well.

Unfortunately, clusters exhibit performance heterogeneity, particularly within their I/O systems. Incremental scaling leads to a mix of disks and I/O buses with varying performance characteristics. Inside the disks themselves, bad-block remapping and non-sequential file layouts cause unexpected seeks, while multi-zone behavior significantly reduces bandwidth for inner-track files. Finally, at the software level, operating system services and other applications compete for resources, rendering performance of those resources unpredictable.

Performance heterogeneity has a significant implication for I/O-intensive programs. In the simplest case, design of these programs begins with the partitioning of data across cluster disks. A parallel program accessing this data can then be structured with one process per disk. We use the term *physical stream* to describe this one-to-one relationship between processes and disks, emphasizing the access request stream's direct dependence on a single device. Clearly, the behavior of physical streams is subject to the vagaries of performance heterogeneity. When the throughput of a physical stream slows, the I/O-bound process accessing it will slow down, and so in turn will the parallel program to which that process belongs. As a result of this global slowdown, other physical streams will be underutilized.

To achieve consistently high performance, I/O-intensive parallel programs must take advantage of all available cluster I/O throughput at all times. We propose that these programs will be well served by an abstraction over physical streams that yields all available I/O throughput. We call this abstraction *Virtual Streams*. Virtual Streams decouple parallel program performance from that of the underlying physical streams by evenly distributing I/O throughput to all processes. Over the course of the program's lifetime, Virtual Streams deliver to each process an equal share of the aggregate disk throughput, equalizing those shares frequently to ensure that processes see similar throughputs at all times. Running atop such a layer, I/O-bound programs will continually receive the entire available throughput of the cluster I/O system. Because of their adaptivity, Virtual Streams offer precisely what I/O-intensive parallel programs need to achieve near-peak common case performance on clusters.

From this perspective, Graduated Declustering (introduced as part of the River data-flow programming environment and I/O substrate for clusters of computers [1]) can be viewed as the basis for an implementation of Virtual Streams. In this work, we begin to investigate the Graduated Declustering implementation space. We also explore the additional pieces necessary to complete a full implementation of Virtual Streams in a realistic cluster environment.

2. Design Rationale

The design of the system strives to achieve a number of important goals, which we believe are essential in an adaptive parallel I/O system.

2.1 Performance Adaptiveness

The system should quickly adapt to the arrival of any performance perturbations after they reach a certain threshold. This threshold, also known as the low watermark is a benefit vs. cost measure. It is determined empirically and depends on the overhead of invoking the dynamic load balancing mechanism. Load balancing is done only when the perturbation is bad enough to guarantee that the benefit can offset the cost. Another related criterion for adaptiveness is the speed at which the system can adjust to the departure of any existing performance perturbations. Ideally the system should utilize the total available bandwidth and evenly distribute available disk throughput among all parallel processes at any given time.

However, there exists an inherent limitation in the bandwidth equalization. The maximally achievable bandwidth is determined by the sum of the bandwidth of all replicas for every replica. If each of these sums is at least as large as the total bandwidth of the disks divided by the number of consumers, then it is possible for each client to achieve the same bandwidth. This can be shown more clearly by the following equations:

$$C0 = (\sum_{0}^{n} Bi)/n$$

$$\sum Bk \geq C0$$

(The second equation should hold for all files. ΣB_k is the sum of bandwidth for all disks that has the replica of this particular replica k.)

2.2 Scalability

The system should scale well with both the numbers of disks and parallel processes. The overhead of the system should not be significant enough to prevent it from being usable in a realistic cluster environment with a large number of nodes. In addition, the system should not be limited to any particular configuration of server and client setup. A client should be able to receive data from any number of servers as long as the data reside on these servers' disks. Similarly, a server can serve an arbitrarily reasonable number of clients at any given time. Consequently, when new disks are available, they can be turned into servers and incrementally added to the system without any need to modify the system.

2.3 Minimal Overhead

As a requirement for reaching the goal of scalability, minimal overhead is essential for any system used in a distributed environment. In our implementation, we specifically address the seek-cost, the major overhead of the system.

In the unperturbed case, minimal overhead is incurred using our system. There is inherent seek overhead caused by simultaneously reading more than one file; however, it can be significantly reduced if each process is designated a primary server where most of requests for reading data are satisfied. If perturbation occurs, a primary server still tries to satisfy most of its client's requests if possible. Another strategy for reducing seek cost is to increase the amount of data read for a given file before switching to another file. Nevertheless, this reduces the flexibility in scheduling the bandwidth allocation among the different processes and can only be used to a limited extent.

2.4 Convergence to Stability

Closely related to adaptiveness to perturbation are the ideas of convergence and stability. It is critical that the system quickly converges to a stable state where every client receives roughly an equal share of disk bandwidth. No oscillation should occur, where the amount of bandwidth for any particular client changes constantly in the absence of any new perturbations. The time it takes for the system to reach stability, in our implementation, is linear with the number of nodes present. This is due to the distributed nature of the algorithm. Further, it depends the occurrence of the most recent performance perturbation affecting the system.

2.5 Ease of Use, Good Abstraction

Graduated Declustering should be easy to use by applications. An interface is exposed that is general enough to be effortlessly adopted by any application where a number processes needs to read data from files. The infrastructure resulting from our implementation of Graduated Declustering is self-contained and usable by various applications.

3. Detailed Design and Implementation

In this section, we describe the details about how we reached the above-mentioned design goals.

The evaluation of the implementation is done using NOW cluster consisting of a number Ultra1 workstations connected using Myrinet local area network. Each workstation consists of a 167 MHz Ultra SPARC I processor, two Seagate Hawk 5400 RPM disks, 128 MB of memory. The operating system used is Solaris 2.6. Communication among machines is achieved using IP over AM (Active Message). The programs modeled are assumed to be I/O bound, not network or CPU bound.

3.1 Investigation of load balancing algorithm:

A good load-balancing algorithm should quickly adapt to any performance perturbation and evenly distribute the available bandwidth among all data consumers when possible. It can be either centralized or distributed. Any centralized systems do not scale well and have a single point failure; therefore, we choose to study distributed load balancing algorithms. We studied two alternatives in the distributed algorithm and compared their behavior.

3.1.1 Progress based implementation:

This algorithm has been proposed by the River paper [1]. Adjustments to bandwidth allocation by servers rely on explicit information related to the progress of the clients transmitted to servers. By explicitly telling servers how much bandwidth the client is receiving up to that point helps server to make the bandwidth allocation decision to its clients. In our study, we implemented a slightly different version using the same underlying idea. Rather than piggybacking the bandwidth number to each request sent to the server, we attach a number indicating the number of remaining requests to be served for the particular client. The net effect of scheduling is the same. This is done as a validation of the work presented in the River paper and a good way to evaluate the existing approach.

There are a few shortcomings with this approach. First of all, explicit control information needs to be transmitted between client and server. This incurs both hardware cost due to transmission of the control information and software cost due to its calculation. This becomes a part of the overhead of the system. Furthermore, explicit information transmission also increases the complexity of the system due to the need to process and transmit extra information.

In addition, the information may not be useful when it is received. Despite the fact that the bandwidth information is piggybacked to each request, when the information is arrived at the server, it might already be outdated. This can happen, since a client usually interacts with more than one server. Within the time period that the request has arrived, the bandwidth information could have changed by a relatively significant amount due to the arrival of the replies from other servers. In other words, the bandwidth information is associated with the time when the request is transmitted, but not with the time when the request is received at the server side.

Furthermore, this approach only addresses the performance heterogeneity among the data producers. It assumes the data consumers are running at the same rate. Realistically, it is difficult to achieve due to various factors, such as contention for disks, I/O busses, CPU, and network resources. A particular client running slower than the others will also receive the equal bandwidth as other clients under the current scheme. This will cause this client to finish last. The preferred behavior would be to let the faster client receive more bandwidth since it has more processing capability. Thus, the algorithm only considers load balancing in terms of bandwidth, but does not address the issue of work distribution.

3.1.2 Response rate based implementation:

We developed an alternative to the explicit, history-based approach – a response-rate-based algorithm. In this implementation, no explicit control information is transmitted between the client and the server. To deduce the available bandwidth of the server, the client compares the number of replies returned by any particular server with other ones. This ratio reveals the relative available bandwidth information of each server. Using this information, client chooses to send more requests to the faster servers. Servers, cooperating with clients, choose to serve the client with more queued requests. In effect, servers implicitly receive information about the progress a particular client has made by the number of requests it has outstanding for that client.

Let's consider the following scenario to show the algorithm in action. Suppose one of the servers is slowed down. The clients requesting data from it will notice a difference in the response rate between this server and other server(s) they have contact with. As a result, the slow server's clients will request more data from other relatively faster servers. These faster servers, in turn, will notice a difference in the request queue length and thus choose to serve more requests out of the longer queue corresponding to the client affected by the slower server.

This approach is "historyless", since servers do not receive any history information, such as how much bandwidth the client has received so far. However, clients arguable do have some history information from the number requests served up by any particular server up to a certain point. Based on this "history" information, clients can deduce the rate server is capable of serving read requests. Accordingly, clients adjust their rate of sending requests to each server.

One important characteristic of the system based on this algorithm is that it is implicitly controlled. No explicit control information is transmitted. It is instead inferred from the behavior of both the clients and the servers. [2] This makes the system naturally adaptive and dynamic.

One important byproduct of this implicit scheme is that it tries to accommodate the performance heterogeneity of both the servers and also the clients. Unless a client is ready to process more replies from the servers, it will not send in more requests for data. Consequently, faster clients will end up receiving more replies and take some of the workload off clients running at a slower rate. As a result, an equalization of disk throughput in light of different client processing capability is achieved. This idea is similar to the Distributed Queue concept proposed by the River project. [1]

Another byproduct of this algorithm is that it is very easy to designate each client with a primary server where most read requests are satisfied to reduce seek cost incurred due to reading from multiple replicas simultaneously. This is achieved by intentionally sending the majority of requests to the primary servers at the start. In response to this, servers will choose to allocate more bandwidth to their primary clients.

Although the implicit information is never incorrect, correct interpretation is needed in order to make good use of it. The experience we had from tuning the implementation of this algorithm taught us some valuable lessons:

1) Careful tuning of various parameters is needed in order to correctly react to the implicit information received. These parameters depend on the particular hardware configuration used, the relative seek cost, the transfer rate of the network. For example, in order to overlap communication with computation, at any given time, each server should never be idle if there are still any corresponding outstanding requests for that server. Therefore, there should be a minimal number of requests outstanding at each server to keep the pipeline full. This number cannot be too big to reduce the flexibility of scheduling. For the particular setup used in our implementation, we empirically determined it to be four.
2) To adjust the sensitivity to perturbation or increase the capacity to adapt to small perturbations, one can vary the number of requests sent to different servers based on any small difference in the number of replies by different servers. In effect, this introduces “artificial” perturbations to detect perturbations in a timelier manner. However, one should establish a threshold to make adjustment to perturbation worthwhile the seek cost incurred.
3) There is an evaluation phase when the clients just start to receive first several replies from its servers. At this stage, it is difficult to determine the relative available bandwidth of the servers before receiving a certain number of replies back. In our implementation, we wait until at least 5 replies have come back from either server, before varying the number of requests sent to the servers. This is necessary, since the initial difference does not truly reflect the response rate of the servers and is mainly due to setup noise.
4) We need to guarantee that no clients will starve by any server. If the server notices a big difference in the number of outstanding requests from its clients, it needs to periodically serve some requests from each client in a round-robin fashion. At the same time, it should devote most of its bandwidth to the client with relatively larger number of outstanding requests, assuming it is not getting sufficient bandwidth from its other servers.
5) If a server is detected to be responding very slowly, the client requesting data from it will reduce to the behavior of sending only one request at a time. Before getting any replies back, it will not send more requests. This occurs only when the ratio of the maximal number and the minimal number of served requests is above a certain threshold. When the ratio goes below the threshold, the number of requests sent is increased to keep the pipeline full.

3.2 Investigation of convergence to stability

We aim at achieving fast convergence to stability. Stability in this case is defined to be the state in which each client is receiving a fixed maximally possible amount bandwidth while maintaining the balance of bandwidth among all clients. When a perturbation occurs, the ideal behavior of the system would be a short adjustment period, then steady state behavior. The two different load balancing we studied have different behavior in the process of adjusting to equal share of bandwidth as shown the graphs below.

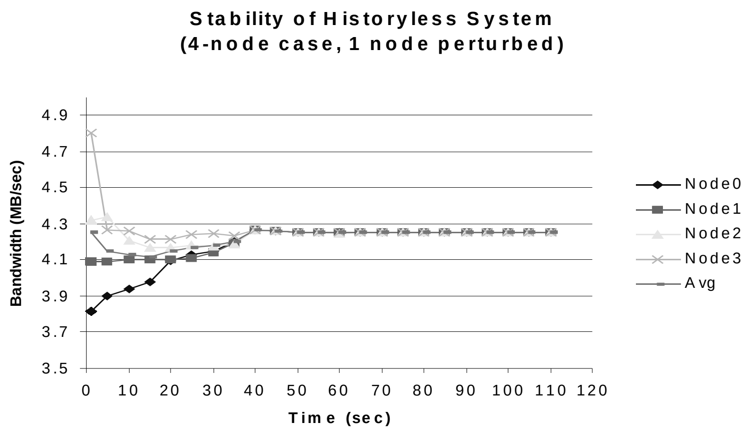

Figure 1: Stability of a History-based System: using progress-based load balancing algorithm, it takes roughly 15 seconds for the system to reach stability in this particular experiment. Perturbation occurs at

time 0, taking away 70% of the bandwidth of one of the disks. The setup is 4 machines acting as data producers with one running at 30% of the full bandwidth interacting with 4 machines as data consumers. The graph shows the bandwidth of the data consumers over time.

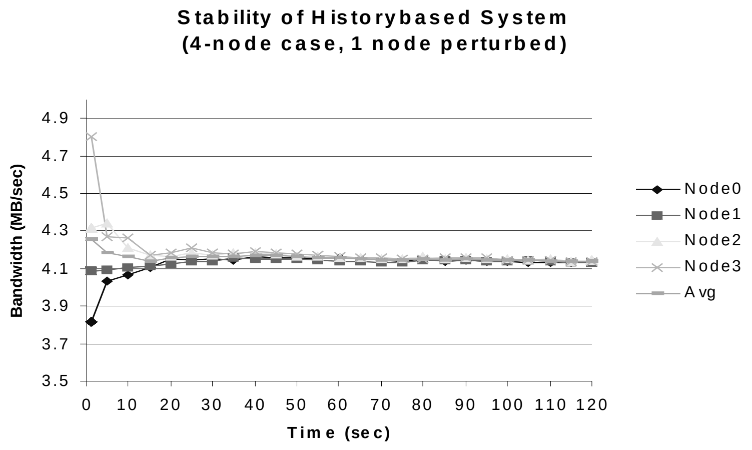

Figure 2: Stability of a History-less System: using response-rate based load balancing algorithm, it takes roughly 30 seconds for the system to reach stability. Perturbation occurs at time 0, taking away 70% of the bandwidth of one of the disks. The setup is 4 machines acting as data producers with one running at 30% of the full bandwidth interacting with 4 machines as data consumers. The graph shows the bandwidth of the data consumers over time. One thing to notice is that the average bandwidth decreases from time 0 before the system converges at around 30 seconds. During this time period, disk throughput is not fully utilized because data consumers are still in the process of guessing data producers' available bandwidth.

3.2.1 Progress based implementation:

In this algorithm, based on the idea presented in the River paper [1], explicit bandwidth information is transmitted from the clients to the servers. This bandwidth information can be an actual number of megabytes per second or the number of remaining request for any client. We choose the second implementation and find out that the convergence to stability is smoother and faster than the response rate based implementation.

3.2.2 Response rate based implementation:

The second algorithm does not rely on any explicit information sent between client and server but rely on implicit information. By observing how fast the replies have come back from any particular server, its bandwidth can be estimated. Since guesswork is involved in determining the available bandwidth of any server, convergence to stability occurs slower. Another reason accounting for this is that guessing the available bandwidth of the servers cannot begin until the servers' bandwidth information is exposed. This occurs only after servers have responded with a number of replies. Only then can the client compare the relative bandwidth of its servers and start to treat them differently by sending various numbers of requests.

3.3 Seek overhead reduction

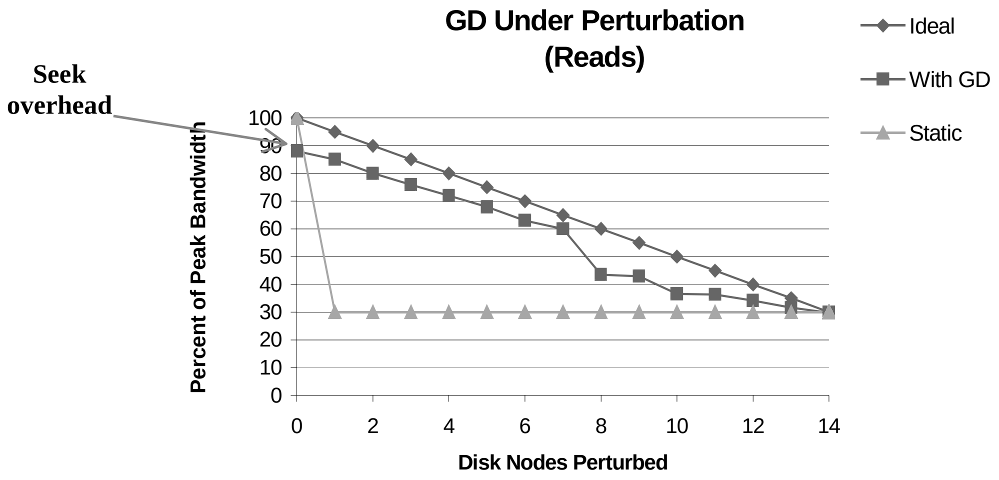

Figure 3: Benefit of Graduate declustering: This graph shows the benefit of graduate declustering implemented using the distributed load balancing algorithm described in the River paper (progress-based). However, when no disk nodes are perturbed, the system loses 10% of peak bandwidth due to seek overhead.

The initial implementation of Graduated Declustering in the River paper [1] does not address any possibility of reducing the seek-overhead incurred by reading more than one replica at a time. Under no perturbation, this cost should be avoided. Although it is rather insignificant in the current NOW configuration (8 to 10 percent of total bandwidth is lost in a 28-node setup shown in the figure above), evidence shows that it is becoming more and more significant as the transfer rate of the disks increases. In fact, we did some measurement to find out the magnitude of the seek-overhead in faster disk drives. These data are shown in the following table.

Disk drive type:	Hawk 2GB	IBM 8.5GB	Cheetah 8.5GB
Transfer Rate:	5400 RPM	10,000 RPM	10,000 RPM
Outer-track Bandwidth:	5.5 MB/sec	13.7MB/sec	17 MB/sec
Bandwidth if another process is reading from inner track:	2.475MB/sec	4.2MB/sec	5MB/sec
Total percentage of peak bandwidth lost due to seek:	10%	38.6%	41.1%

Table 1: Impact of seek-overhead on disks of varying transfer rate

As indicated above, it is crucial to reduce the seek-cost whenever possible, especially in systems with faster disk drives. In addition, as the request size of the data read gets smaller, seek cost also increases. In our experiments we used a fairly large request size (512KB) to amortize the seek cost. Thus, it is critical for such a parallel I/O system to minimize seek overhead. In the response-rate based version of Graduated Declustering that we developed, we used two mechanisms to achieve this goal:

1) Increase the number sequential reads done:
   Each client is assumed to do a sequential access to a particular file. The requests are generated in a sequential order. At the server side, the server decides how much bandwidth to allocate to each client depending on each client's request queue length. By increasing the number sequential reads served at a time without compromising the scheduling flexibility, seek-cost is reduced. The following simple example shows how this can be achieved. Suppose a server decides to allocate 30% of total bandwidth

to client0 and 70% of the bandwidth to client1. And the request queue of client0 has 3 requests; that of client1 consists of 7 requests. Rather than switching between the two files partitions, the server chooses to serve client0's 3 requests first, then does another sequential read for client1's 7 requests.

2) Another way used to reduce seek overhead is to designate a primary server for each client. The idea is that under no perturbation, each client receives most (99%) of its data read from its primary server. Evidence shows that overhead of the system is reduced to less than 1% of the total bandwidth. Nevertheless, this will have a negative effect on the response time to perturbation. If the primary server is slowed down, it will take longer for its client to adjust the request sending rate and switch to send more of the read requests to its other server(s).
   After the perturbation is gone, the ideal behavior would be for the system to revert back to the primary server behavior. However, in practice this is difficult to achieve, because based on local information, client has no knowledge of when the perturbation has finally ceased. It could guess based on the response rate of its servers. If it guessed wrong, it can switch back to original request rate.

The following graph compares the two implementations of GD: response-rate based version is our proposed solution for reducing seek cost; progress based version is the current implementation in the River system. It is quite evident that our proposed solution achieves higher disk throughput under all scenarios shown. Especially in the case when there are no disk nodes being perturbed, the response-rate based approach achieves nearly all of the available peak bandwidth. In other cases, when several disk nodes are perturbed, the overhead incurred by the proposed solution is consistently lower.

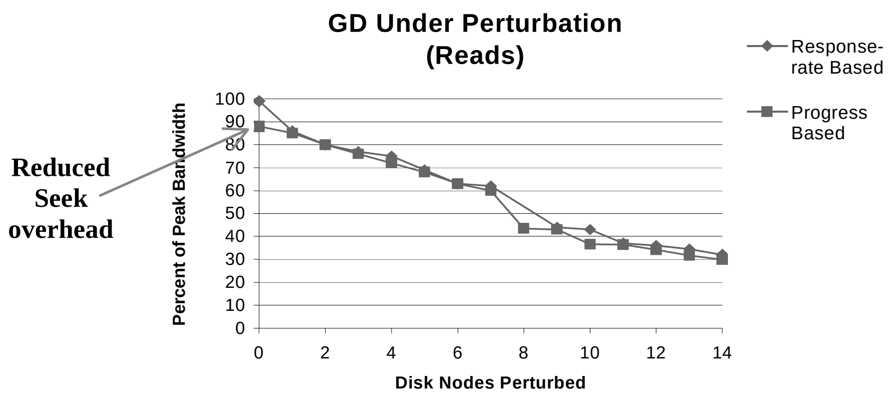

Figure 4: GD under Perturbation. This figure shows that using the idea of primary server, seek cost is significantly reduced when no perturbation occurs. Further by doing more contiguous data accesses, the overall bandwidth is often higher in the response-rate-based system than the progress-based one.

3.4 Server request handoffs:

Perturbation can occur at any time in a real cluster environment. After clients have finished sending any of their read requests to their servers, servers might not respond in a timely manner. When this occurs, they can either choose to time out and re-send the requests to the faster server, or they can be passive and just wait for the response. The downside of the first approach is that it is difficult to choose a good time-out interval. Extra requests can go wasted, if replies will come back very soon. In addition, the server has the overhead of keeping track of which request is repeated and making sure to disregard the duplicate ones. The client, similarly, needs to keep track of which request has been served and re-send the request when necessary. Therefore, this approach increases the complexity of both the client and the server.

In our implementation of the response-rate based system, we choose the second approach for the client. We introduce a way to address this type of perturbation by introducing server to server interaction. On the serve side, to expedite the request handling, finished servers "work-steal" from slow servers. Server to server request transfer occurs, whenever any server has finished serving its clients. In order to achieve this, each server needs to know on what other servers the replicas of its files exist. When any server has finished all the requests, it will actively contact these servers to reduce their workload. These servers will send some of the read requests containing information about the client to the work-stealing server, so that it will reply directly to the clients when requests are satisfied. The goal is to keep all servers busy whenever possible in order to achieve maximal bandwidth utilization for each client.

Some of the design decisions involved in implementing server request handoffs are the following. There should be a minimum number of requests transferred at a time to offset the cost of transfer. This number is determined empirically and depends largely on the underlying hardware configuration. Borrowing from the idea of diffusive load-balancing [13], the exact number of requests transferred is determined by the bandwidth information of both servers involved. Thus, this requires the servers to periodically note down its available bandwidth value. Using these two bandwidth values, the formula used to calculate the number request transfers is show below:

$$\mathbf{N = N_0 * B_{fast} / B_{slow}}$$

N: the number of requests transferred from the busy server to idle server.
N_0: the total number of requests outstanding at the busy server.
B_{fast}: bandwidth of the idle server; persumably it has a bigger bandwidth.
B_{slow}: bandwidth of the busy server, persumably it is slower.

Since replies of data read can come from servers other than the one the request was originally sent to, any client need to keep connections to all servers open before all of its replies have been received. Servers on the other hand needs to detect when they have finished serving their share of work so that they can contact other servers to offer help. A separate thread of execution at the slow server does the work of accepting help and transferring requests; therefore, request handoff does not stall the process of serving clients, but only expedites it.

3.5 Implementation of writes:

In its original incarnation, Graduated Declustering supported only read accesses. A complete implementation of Virtual Streams must handle writes as well. In this section we discuss the extension of Graduated Declustering to support writes.

In adding support for writes, our primary concern is to maintain performance robustness. However, before attacking the complete problem, it is useful consider a simpler scenario. Suppose that we want to write a file to disk as quickly as possible, but that we do not need the ability to read that data back. In other words, the data's destination is not important, but writing it to *some* disk is. In this case, our write requests may be distributed across a set disks in any order, and at whatever rate those disk will accept them. Faster disks will accept more requests and receive more data, lending performance robustness to the write operation. Notice that after sending all of our data, we do not know which disk actually has any particular byte. We do, however, know which disks might have it.

The more complex scenario, in which we do wish to read our data after we write it, can be addressed by straightforward extension of the above algorithm. If each of write-request uniquely identifies its block data (for example, by its byte offset from the beginning of the file being written), then the disks can track this metadata. Later, after all data are written, the disks involved in the write can swap blocks, allowing each to construct a complete replica. The only additional information necessary in this step is knowledge of which disks participated in the write stream, and the client can supply that information. At the disk, replica construction can be performed easily if blocks are initially written to their ultimate offsets within the incomplete replica. In essence, performing writes in this fashion leaves holes to be filled with blocks from other servers at a later time. The metadata required to track these holes is a single presence bit (or byte in our case, for the ease of implementation) per block.

This strategy for writes fits nicely into the Graduated Declustering framework. Our load-balancing algorithms require no changes at all. We experienced little difficulty in adding write support to our implementation, which previously supported only reads.

The performance of writes under perturbation is shown in Figure 5. It exhibits behavior that is similar to that of reads. Given the software commonality, this comes as little surprise. The percentage of peak bandwidth achieved while writing is somewhat lower than that achieved while reading. This is due to the extra seeks incurred when logging metadata. Although not shown in the figure, the absolute bandwidth for writes is also lower than that of reads, again due to metadata writes.

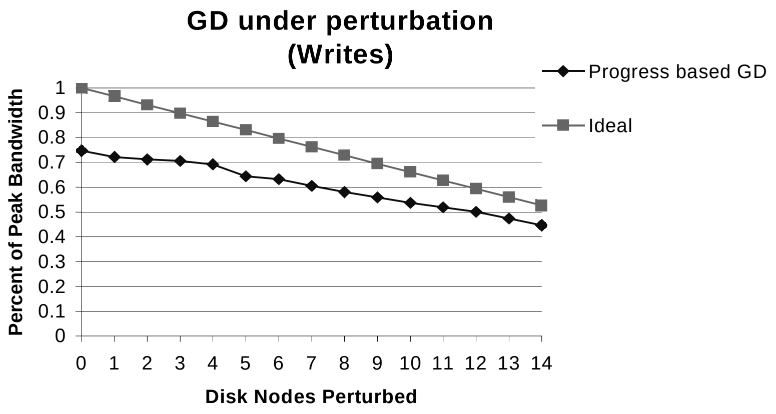

Figure 5: Performance of writes in GD under perturbation: This graph illustrates writes performance under perturbation. The setup is similar to that in Figures 3 and 4, where 14 machines in the NOW cluster produce data and 14 of them consume data. As the graph shows, tracks ideal behavior closely.

3.6 Client library design:

An important part of any infrastructure software is the interface that it exposes to the applications. In this section, we discuss the important issues surrounding the design of libraries that provide client access to Graduated Declustering.

Our primary goal in this work is to provide high I/O throughput to applications. Under any workload (be the access pattern random or sequential), keeping disks busy with useful work is the key to maximizing throughput. Simply put, this requires that at least one request be queued at the disk at all times. In Graduated Declustering, requests pass through a pipeline (network, scheduler, disk, and network again). This pipeline must be kept full in order to keep requests queued at the disk. An effective interface to Graduated Declustering must either fill the pipeline on the application's behalf or permit the applications to fill the pipeline itself.

Low resource overhead is also a goal, as it is for any interface. We want to avoid consuming CPU and memory resources in the library that would otherwise be used by the application. Issuing, tracking, and retiring requests demand relatively little work. Data buffering, on the other hand, has the potential to consume a significant amount of memory. Furthermore, copying large amounts of data to and from buffers can burden the processor. This effect is particularly noticable in I/O-intensive applications, which by their nature touch many bytes. Therefore, an effective interface to Graduated Declustering should avoid the cost of unnecessary buffering.

For unordered scans over files containing fixed-sized records (e.g., unsorted database relations), we have implemented a simple-to-use library that automatically fills the request pipeline and performs no data buffering. This interface consists of three entry points: open(path_name, request_size, server_list), next(stream_id), and close(stream_id). In addition to establishing connections to servers, open() specifies the size for all requests and launches an empirically-determined number of requests into the pipeline. As replies come back from the server, they are placed directly into buffers allocated by the library. One of these buffers is returned by each invocation of next(), to be used and freed by the application. Next() also issues another request. Close() performs any necessary cleanup.

With the addition of buffering, this unordered-scan library can support true (i.e., ordered) sequential scans over files containing variably-sized records. For true sequential access, buffering is needed to maintain ordering between requests sent to different servers. (For example, suppose requests n and n+1 are sent to different servers. If the reply to request n+1 arrives first, it must be buffered until the reply to request n arrives.) Once buffering is in place for ordering, only a small amount of bookkeeping and an extra copy operation are needed to allow each invocation of next() to specify the amount of data that it will return. This allows next() to handle records of any size. As mentioned, buffering comes at a cost whose significance must be evaluated by application writers.

Although the simple interfaces described above may be convenient for use in some applications, access to full Virtual Streams functionality *with high performance* requires an asynchronous interface in the style of POSIX asynchronous I/O. For random access patterns, a library cannot possibly fill the request pipeline automatically. To do so, the library would have to predict the application's accesses. A synchronous interface in the style of POSIX I/O (the familiar open(), read(), write(), and close()) cannot offer peak throughput. Still, it may help the casual application, whether parallel or sequential, avoid severe perturbations.

4. Related Work

This work is based on the Graduated Declustering introduced by the River project [1]. Like River, it has relation to a number of areas of research: parallel file systems [3,5,6,9,10,14], programming environments, and databases. The major difference between our work and other I/O work is that we develop a system that adjusts bandwidth allocation dynamically and aims to provide a usable coherence abstraction to hide the underlying hardware heterogeneity.

5. Future Work

Currently, the order of arrival is guaranteed only on a per-server basis. Requests sent to any particular data producer are satisfied in order. However, there is no global order guarantee across data producers. Proper buffering is needed if processing of the data needs to proceed in order. This can be easily implemented in the response rate based scheme. Once the relative response rate has been determined, requests can be sent in an intelligent order so that replies will arrive in order. For example, if server0 is replying at three times the rate of server1, by sending request0, request1, request2 to server0 and request3 to server1 will guarantee that not much buffering is needed to receive orderly replies.

Another area in which we would like to improve our system is seek cost reduction after the cease of perturbations. It is preferable for the system to intelligently and quickly revert back to primary server designation to minimize seek overhead incurred. Maybe each client's primary server should be dynamically determined to increase the flexibility of allocating bandwidth.

In addition, we would like to add the mechanism of "hot file" replication to our system. The idea here is to replicate files on the fly if it is detected to be frequently accessed, acting as the bottleneck of the system. Since statistics of accesses to files are currently maintained in our implementation, decisions about replicating files can be easily made if extra disk space is available.

Finally, we would like to write a few applications to demonstrate the performance and usability of our system. The applications we are currently considering are hash-join and sorting.

6. Conclusion

We have developed a new distributed workload-balancing algorithm for Graduated Declustering to achieve equal share of disk bandwidth among a number data producers in a cluster environment. This algorithm not only addresses the problem of performance heterogeneity of data producers, but also that of data consumers. We determined the impact of seek overhead on the performance of the original Graduated Declustering implementation and developed an effective way to address the problem in the response-rate based algorithm proposed. Through the evaluation of this algorithm, we demonstrated its stability and convergence to the final optimal bandwidth value. During the evaluation process, we identified the possibility of perturbation occurring after load-balancing decision has been made. To address this kind of perturbation, we have designed and implemented a new technique called server request handoff to fully utilize all available server disk bandwidth. Furthermore, we have extended the system with the capacity to handle writes with performance robustness. Finally, we have explored the development of usable interfaces exposed to the application programmers using our software infrastructure.

Acknowledgements

We thank Remzi Arpaci-Dusseau for many insightful discussions and professors Culler, Hellerstein, Patterson, and Yelick for giving us valuable suggestions. One of the authors thanks Eric Anderson for his input.

References

[1] R. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. Culler, J. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the Fast Case Common. *In Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, Atlanta, Georgia, 1999.

[2] Andrea C. Arpaci-Dusseau. Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed Systems. *Ph.D. Dissertation*, University of California, Berkeley, December 1998.

[3] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. singh, R. Thakur. PASSION: Parallel And Scalable Software for Input-Output. ECE Dept., NPAC and CASE Center, Syracuse University, Syracuse, NY. *NPAC Technical Report SCCS-636*, September, 1994

[4] Peter F. Corbett, Dror G.Feitelson. The Vesta Parallel File System. *Scalable High-Performance Computer Conference*, May 1994.

[5] Roger L. Haskin. Tiger Shark – a Scalable File System for Multimedia. IBM Almaden Research Center

[6] James V. Huber, Jr., Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, David S. Blumenthal. PPFS: a High Performance Portable Parallel file System. *ICS 1995*, Barcelona, Spain.

[7] David Kotz. Disk-directed I/O for MIMD Multiprocessors. Department of Computer Science, Dartmouth College, Hanover, NH, *First Symposium on Operating Systems Design and Implementation (OSDI)* November, 1994.

[8] Edward K. Lee. Highly-Available, Scalable Network Storage. Systems Research Center, Digital Equipment Corporation

[9] S. J. LoVerso et. Al. Sfs: A Parallel File System for the CM-5. *Summer Usenix*. 1993, June 21-25, Cincinnati, OH

[10] Nils Nieuwejaar, David Kotz. The Galley Parallel File System. PCS-TR96-286. Department of Computer Science, Dartmouth College, Hanover, NH.

[11] Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. *Report RND-92-020* December 1992. NAS systems Development Branch, NAS Systems Division, NASA Ames Research Center

[12] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, J. Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile Application-Aware Adaptation for Mobility. *Proceedings of the 16th ACM Symposium on Operating Systems Principle*s, Saint-Malo, France, 5-8 October 1997.

[13] Raghu Subramanian, Isaac D. Scherson. An Analysis of Diffusive Load-Balancing. *SPAA* 94 –6/94 Cape May, New Jersey.

[14] C. A. Thekkath, T. Mann, E. K. Lee. Frangipani: A Scalable Distributed File System. Systems Research Center, Digital Equipment Corporation

[15] S. Kuo, M. Winslett, Y. Chen, Y. Cho, M. Subramaniam, and K.E. Seamons. Parallel input/output with heterogeneous disks. In *Proceedings of the 9th International Working Conference on Scientific and Statistical Database Management*, pages 79-90, Olympia, Washington, August 1997.