LA-UR- 09-06133

Approved for public release;
distribution is unlimited.

Title:	Managing High-Bandwidth Real-Time Data Storage
Author(s):	David Bigelow, HPC-5 & UC Santa Cruz Scott Brandt, UC Santa Cruz John Bent, HPC-5 HB Chen , HPC-5
Intended for:	The USENIX 2010 FAST Conference FAST - 8th USENIX Conference on File and Storage Technologies (FAST '10) February 23–26, 2010 San Jose, CA

Form 836 (7/06)

Managing High-Bandwidth Real-Time Data Storage

David Bigelow
UC Santa Cruz
Los Alamos National Lab
dbigelow@cs.ucsc.edu

Scott Brandt
UC Santa Cruz
scott@cs.ucsc.edu

John Bent, HB Chen
Los Alamos National Lab
{johnbent, hbchen}@lanl.gov

Abstract

There exist certain systems which generate real-time data at high bandwidth, but do not necessarily require the long-term retention of that data in normal conditions. In some cases, the data may not actually be useful, and in others, there may be too much data to permanently retain in long-term storage whether it is useful or not. However, certain portions of the data may be identified as being vitally important from time to time, and must therefore be retained for further analysis or permanent storage without interrupting the ongoing collection of new data.

We have developed a system, Mahanaxar, intended to address this problem. It provides quality of service guarantees for incoming real-time data streams and simultaneous access to already-recorded data on a best-effort basis utilizing any spare bandwidth. It has built in mechanisms for reliability and indexing, can scale upwards to meet increasing bandwidth requirements, and handles both small and large data elements equally well. We will show that a prototype version of this system provides better performance than a flat file (traditional filesystem) based version, particularly with regard to quality of service guarantees and hard real-time requirements.

Managing High-Bandwidth Real-Time Data Storage

David Bigelow
UC Santa Cruz
Los Alamos National Lab
dbigelow@cs.ucsc.edu

Scott Brandt
UC Santa Cruz
scott@cs.ucsc.edu

John Bent, HB Chen
Los Alamos National Lab
{johnbent, hbchen}@lanl.gov

Abstract

There exist certain systems which generate real-time data at high bandwidth, but do not necessarily require the long-term retention of that data in normal conditions. In some cases, the data may not actually be useful, and in others, there may be too much data to permanently retain in long-term storage whether it is useful or not. However, certain portions of the data may be identified as being vitally important from time to time, and must therefore be retained for further analysis or permanent storage without interrupting the ongoing collection of new data.

We have developed a system, Mahanaxar, intended to address this problem. It provides quality of service guarantees for incoming real-time data streams and simultaneous access to already-recorded data on a best-effort basis utilizing any spare bandwidth. It has built in mechanisms for reliability and indexing, can scale upwards to meet increasing bandwidth requirements, and handles both small and large data elements equally well. We will show that a prototype version of this system provides better performance than a flat file (traditional filesystem) based version, particularly with regard to quality of service guarantees and hard real-time requirements.

Managing High-Bandwidth Real-Time Data Storage*

David Bigelow[†‡], Scott Brandt[†], John Bent[‡], HB Chen[‡]
[†]University of California, Santa Cruz
[‡]Los Alamos National Laboratory, HPC-5
{dbigelow, scott}@cs.ucsc.edu, {johnbent, hbchen}@lanl.gov

Note: This is unpublished work currently under submission. Please do not distribute it to others without the permission of the authors.

Abstract

There exist certain systems which generate real-time data at high bandwidth, but do not necessarily require the long-term retention of that data in normal conditions. In some cases, the data may not actually be useful, and in others, there may be too much data to permanently retain in long-term storage whether it is useful or not. However, certain portions of the data may be identified as being vitally important from time to time, and must therefore be retained for further analysis or permanent storage without interrupting the ongoing collection of new data.

We have developed a system, Mahanaxar, intended to address this problem. It provides quality of service guarantees for incoming real-time data streams and simultaneous access to already-recorded data on a best-effort basis utilizing any spare bandwidth. It has built in mechanisms for reliability and indexing, can scale upward to meet increasing bandwidth requirements, and handles both small and large data elements equally well. We will show that a prototype version of this system provides better performance than a flat file (traditional filesystem) based version, particularly with regard to quality of service guarantees and hard real-time requirements.

1 Introduction

The ability to capture and store data in real time is a common requirement in many aspects of modern life, though perhaps not often thought about by the ordinary person. To record a television show for later viewing, some sort of device is needed to record the signal in real time. To monitor a secure area, security cameras and other sensors need to record their data in real time. Capturing scientific data from an experiment also requires the ability to record data in real time.

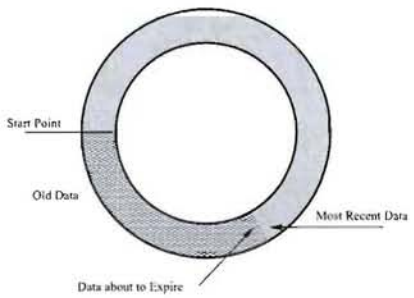

Figure 1: Ring Buffer Diagram

However, there is a world of difference between recording a single television show and recording the output of an experiment or observation in scientific fields. In television, a standard NTSC/ATSC signal provides data at around 20 MB/s[1], a rate easily reached by any consumer level hard drive available today. By contest, the Large Hadron Collider at CERN generates data on the order of 300 MB/s after filtering[6] and has a large machine room (and global network) behind it. What if an even higher date rate is required, and what if there isn't necessarily a high-end storage backend available?

A large subset of data capture, both in scientific and other applications, has the interesting property that most of the data is actually "worthless" in the long run. As a trivial example, consider a security camera positioned to watch over a door. The data generated by such a setup does not need to be kept over the long term: it shows nothing but a door with nothing happening. One needs only look at it long enough to determine that nobody has tried to break

*This work was carried out under the auspices of the National Nuclear Security Administration of the U.S. Department of Energy at Los Alamos National Laboratory under Contract No.DE-AC52-06NA25396. This work received funding from Los Alamos National Laboratory LDRD Project #20080729DR.

in, and then the data is worthless. The same principle applies to data on larger scales as well, particularly in data captured by sensors. If the results can be summarized by "nothing interesting," then there is no need to keep the full set of data. Only when you notice somebody breaking in through the door, or notice something "interesting" on your sensor, do you want to actually preserve the data.

This may be best described as a "write-once, read-maybe," or perhaps a "write-once, read-rarely," storage system. All the data needs to be captured in real time and stored temporarily, but chances are good that it won't actually be needed, and thus the data can safely "expire" after a period of time. This can be easily conceptualized as a ring buffer (figure 1): if the data isn't consumed in a set amount of time, it is automatically overwritten with new data.

In order to address this problem, we created Mahanaxar, which uses a ring buffer architecture to temporarily capture data. Our first priority is to provide quality of service guarantees for incoming data streams, but we have also considered system reliability and scalability. We will present our design for this class of problem, and show that it has superior performance to other methods for managing this type of data.

2 Background

This project was first conceived as a storage system for the Long Wavelength Array (LWA) project[8]. The LWA is a distributed radio telescope currently under construction in southwestern New Mexico. The initial plan is for 53 separate stations scattered over kilometers of desert, with each station generating approximately 72.5 MB/s of data, for an overall data rate of slightly over 3.75 GB/s, generated continuously and without letup over the lifetime of the system.

Radio astronomy generates a lot of data, most of which turns out to be random noise, and "useless." Even if all the data were useful, it would be very difficulty to retain all of it in a system which generates a petabyte of it in just over three days. It may not actually be apparent whether the data is "useful" or not until much later, so we must therefore retain the data for a set period of time and allow an outside observer the opportunity to declare the data interesting if necessary.

As we explored the concept, we realized that there were many other applications that generate lots of "useless" data, but require a portion of it from time to time. Thus we decided to develop a generalized solution that could address all such problems. Broadly speaking, we focused on two "canonical" real-world problems set at opposite ends of the spectrum, with other problems being derivatives and combinations of those two.

1. **Fixed-size, non-indexed data:**

   Fixed-size, non-indexed data is the sort generated by the LWA project, and by many types of sensor systems in general. It arrives at an absolutely fixed rate, never varying, and is only indexed on a single variable: time of generation. Oftentimes such data is generated at too high a rate to be captured on one storage device and must be broken into multiple streams. Such streams need to be correlated with each other in order to regain the full data picture. Any order given to preserve data will be via timestamp only, calculated by an external observer monitoring the data to determine if it is ever interesting.

2. **Variable-size, indexed data:**

   Variable-size, indexed data describes any data source where events are recorded as they happen at variable rates. Such events may be indexed by time, but also by other variables in order for an external process to know exactly why something was recorded. Searching and preserving data may be based on timestamps alone, but will more likely be based on other indices. This problem is more difficult to address due to a non-fixed size and data rate in addition to the difficulties of indexing.

The concept of a ring buffer to gather sensor data is not a new one: Antelope[2] and Data Turbine[14] are both designed along those lines, but can not deal with the data rates that we expect here. Neither service offers actual quality of service guarantees; only best-effort data recording. Other systems like the network traffic capturing "Time Machine"[10] deal with the problem by classifying and prioritizing data streams and dropping what they can't handle. Even then, there are still no real time guarantees in the system; only a promise that it will record data at best-effort capacity and make sure that higher priority streams get first dibs on storage.

The COSS Storage System from Squid[4] utilizes a ring buffer based model, but also functions solely on a best-effort basis in terms of bandwidth, and the mechanism for "preserving" data is simply to rewrite it again at the top of the buffer. Larger storage systems such as Lustre do not make quality of service guarantees from moment to moment[5], which could

be problematic if running a system where data generation rate is very close to the maximum system bandwidth. Larger systems also have no convenient mechanism to automatically expire old data as capacity runs low.

Some Quality of Service work has been focused on providing guarantees of a certain service level from the storage system, as in RT-Mach[11] and Ceph[15], but only to the degree of categorizing traffic for an appropriately fair level of service. Data streams can be guaranteed to get a certain portion of the system resources in both the short and long term, but the guarantee is of the nature "you will get X% of the time every Y time units," rather than "you are guaranteed Z MB/s of bandwidth."

The disk request scheduling system Fahradd[12] is capable of providing QoS guarantees within certain constraints. Fahradd takes the approach of reserving a certain portion of disk head time for requesting processes, and lets each process spend that disk head time however they wish. Unfortunately for the purposes of our problem, that guarantee isn't quite strong enough: a percentage of disk head time does not necessarily translate into bandwidth capabilities, and we need to be able to guarantee a certain bandwidth rather than a certain portion of disk time.

Using a flat file based approach on a standard file system has the benefit of simplicity and may work in limited circumstances, but fragmentation over time is an inevitable problem as we keep the file system at near full capacity, expiring and writing data constantly. We also explored the possibility of using a database, but while databases are well suited to the problem of indexing and searching, they are less well suited to storing large chunks of data, and preliminary testing quickly demonstrated that a database approach was not realistic.

Because the nature of this problem involves near-constant writes, we assume that any approach will have to remain based on conventional rotational disk drives for the foreseeable future. Solid state storage devices promise to become prominent in the near future, but despite potential bandwidth improvements, it would not be wise to use a device with a limited number of write cycles. Write endurance for one of the latest Intel SSDs is only rated at 1-2 petabytes worth of writes[9], and while that trend will probably improve in the future, the mechanical endurance of rotational disk drives dealing with constant writes will probably remain far superior.

3 Example Use Cases

As briefly mentioned in the last section, there are two primary use cases which stand at opposite extremes of the spectrum. Our first use case is based on the type of data that the LWA generates: continuously streaming fixed size sensor data. It arrives at an unchanging bandwidth, does not need to be indexed, and is uniformly "large." Our second use case is monitoring network traffic: each element is fairly small (perhaps on the order of a few thousand bytes or less), but all of different sizes. Each data element may also have to be indexed on multiple variables other than time.

3.1 Continuously Streaming Sensor Data

Storing continuously streaming sensor data of the LWA sort is the less difficult task of the two. It arrives at the same rate forever, never varying. The layout of each chunk is known in advance, or perhaps it is just treated as a straight stream of bytes which can be broken up in whatever manner the storage system deems convenient.

Interaction with this type of data is extremely limited – we take it and store it with a sequence number (timestamp) and don't worry about it again until it comes time to overwrite it with new data. If an external process decides that the data is worthwhile and should be preserved, the storage system only needs to be told that "timestamp X is interesting" and has no harder task than finding that timestamp and marking it as not to be overwritten until further notice.

It may take some time to determine whether the data is interesting (hence the ring buffer approach). For example, a radio telescope may be collecting interesting data, but nobody knows that until ten minutes later when a stupendous event is suddenly registered, and astronomers need to know what happened leading up to that event. An external process needs only order the system to preserve data up to the limit of its buffering ability and collect it later.

This example is perhaps the most basic use case possible, but it covers a wide variety of sensor-based systems.

3.2 Variable-Rate Indexed Network Traffic

Storing variable-rate indexed network traffic is a far more difficult task than storing fixed-rate sensor data. There is a natural ebb and flow of network traffic that coincides with societal rhythms throughout the

day, and spikes of traffic may arrive at entirely unpredictable times. Both the aggregate bandwidth and the sizes of individual data elements are in a constant state of flux.

Furthermore, data elements must be indexed by more than a simple measure of time. Searching for data packets sent between time X and time Y is useful, but not nearly as useful as the ability to search for data packets sent between time X and Y, between source A and source B, and with protocol type M. A system lacking the ability to index data on-the-fly and search on those indices has severely reduced usefulness. This scenario is more difficult to handle, and it is a use case that no existing system currently handles well.

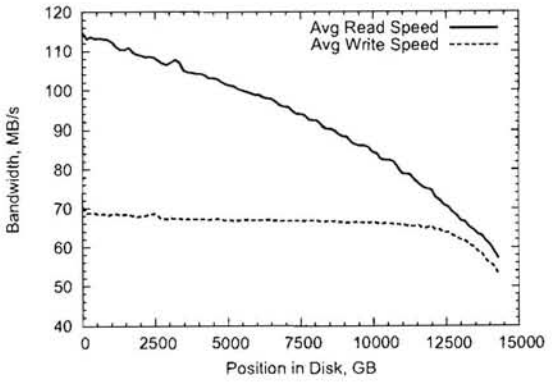

Figure 2: Average read and write speeds on one particular disk

4 Design

We designed our system with three principle factors in mind:

1. **Quality of Service Guarantees:**

   We must be able to guarantee quality of service for the incoming stream, up to a declared bandwidth. If the incoming data stream requires X MB/s of write bandwidth, we need to make sure it gets X MB/s of write bandwidth no matter what. If it exceeds that amount, we'll do the best we can, but no assurances. All other activity on the disk must have lower priority; the key factor is not to lose a single byte of incoming data. Any other processes wanting to use the drive, as well as the task of getting the data back off again, must wait.

2. **Commodity Components:**

   In the case of the LWA project, the physical location for the system may literally be a shack in the desert. We can't assume a high-end network infrastructure or storage backend. On the other hand, we also want to be able to take advantage of a dedicated machine room if we have it available. In all cases we want to take maximum advantage of the hardware we have, and never want to solve anything by "throwing more disks at it" until the problem goes away.

3. **Reliability:**

   The data that we collect can never be regenerated. If there's a hardware failure (and there will be, often, if it's a shack in the desert), we need to be able to retrieve the data if necessary. On the other hand, any reliability mechanism we use can't compromise the first factor, quality of service guarantees.

4.1 Staying Close to the Hardware

One of our first design decisions was that we needed to stay close to the hardware. In order to make real quality of service assurances, we need to know what the underlying hardware is capable of, and what it is actually doing at any particular moment. This is particularly important when it comes to disk drives, as performance can differ by several orders of magnitude depending on use patterns. By carefully mapping out hardware capabilities, we can tell if we need to avoid certain regions of the drive that can't guarantee the rate we need.

As an example of why we need to pay close attention to the hardware, consider one of the hard drives we used in testing: A 1.5 terabyte drive provided a consistent minimum write bandwidth of 68 MB/s or better for the first quarter of the drive. By the last quarter of the drive, we could manage only a consistent minimum write bandwidth of 52 MB/s. The overall performance curve is shown in figure 2, and the other disk drives we tested show similar patterns, with higher capacity drives having a sharper dropoff at the end.

By staying close to the hardware, we are able to determine what data rate we can "advertise," and perhaps more importantly, what performance increases we can gain by ignoring certain regions of the drive. It may be worthwhile to sacrifice extra capacity for extra bandwidth in some circumstances.

However, in order to take advantage of our knowledge of hardware, we need to use the disk without any interface layers. We envision turning our prototype

system into a specialized file system in the future, but in our prototype system (the specific architecture decisions for it are discussed in the next section), we treat the disk as a raw device and eschew traditional file systems.

4.2 Chunk-Based Layout

The first step in taking advantage of our hardware knowledge is to significantly restrict the data layout. Modern file systems are generally good at data placement, but over time, fragmentation is inevitable, particularly in a filesystem constantly at 99%+ capacity. Bandwidth is very difficult to guarantee if a process puts related data on widely-separated portions of the disk, and this problem is only exacerbated by a system that's constantly filling the disk to near capacity, deleting portions to put in new data, and doing it again ad infinitum.

To address the problems of data layout, we took a cue from traditional 512-byte disk blocks, and declare that no data can be written in segments smaller than the chunk size. Chunk size can be customized based on what sort of data the system stores, but for most purposes, the bigger the chunk, the better. The time to write a 1 KB piece of data to the drive is most often dominated by seek time and rotational delay, but those factors are diminished into near-insignificance when writing a single 50 MB chunk.

It is well known that data sequentiality has a very large impact on overall bandwidth[7], and we attempt to exploit this factor as much as possible. There are a few disadvantages to only dealing in very large chunks, but since the workload is intended to be a high-bandwidth stream of quickly-expiring data, the advantages vastly outweigh the disadvantages.

4.3 Chunk Indexing and Consistency

Standard file systems store their indexing information on the disk itself for two main reasons: first, holding the entire indexing structure in memory is at best inconvenient, and at worst impossible (depending on the amount of RAM). It also isn't necessary most of the time because large portions of the file system may not be accessed for large periods of time. Secondly, in the event of a system crash, it is far easier to recover file information from known portions of the disk that it is to traverse the entire disk and try to figure out the structure anew each time.

We have a substantial advantage in this area because our chunk sizes are both large and placed in predetermined locations. We realized that the only information that our system really needs is a piece of data describing the physical layout of the disk, which translates to little more than the chunksize and any skipped regions of the disk. This is somewhat akin to the "superblock" of a file system. With that information, we can assemble an index to hold in memory and never actually commit it to disk, thus saving space and (far more importantly) bandwidth.

Modern disks can usually be filled to capacity within a matter of hours. Because of that fact, any chunk indexing information that we maintain on disk will be entirely out of date within that time, and partially out of date within seconds. Maintaining a consistent index on disk would require very rapid updates, which could be a major problem when we're trying to utilize every megabyte of throughput. If we delay writing an index to disk, much of the point is lost, because it goes out of date so quickly.

Therefore we decided to keep the chunk index entirely in main memory and *never* commit it to disk. We can do this for two reasons: first, with large chunks, reconstruction of the index only takes a few minutes upon startup. Secondly, this is a system which is never supposed to be shut down. If it ever does go offline (perhaps through a power failure), there had better be a backup plan available to ensure that data is not lost, and the time to re-index the old drive upon recovery is much less of a factor. The extra time is a small price to pay for improving our overall performance, particularly since in the event of a data crash, the drive would have to be scanned for consistency anyway.

4.4 Reliability and Recovery

Disks will fail from time to time, whether via a recoverable crash or via outright hardware failure. When that happens, we need to account for two things: making sure that ongoing data collection is not disrupted, and maintaining the ability to recover lost data if the drive is entirely dead.

This problem may be best addressed by redundant drives in a small system, but the more interesting case is a large system: mirroring drives in a large installation is an unnecessary waste of money and energy when there are more elegant solutions available. In this case, the easiest solution is an old familiar one: RAID.

RAID systems offer fault-tolerant behavior and certain performance advantages with the proper workload, but also have disadvantages when they operate in a degraded mode after failure. Traditional storage systems might see drastically increased access times to read data as it gets reconstructed, though writing entirely new data is often unaffected. How-

ever, those disadvantages are mitigated for this type of data load, which we previously characterized as "write once, read rarely."

If a disk fails, it may be that none of its data is interesting, and we never need to deal with the problem of data reconstruction. In this case, degraded RAID mode has no performance impact whatsoever. Even if we do need to recover some of the data, it is highly unlikely that we'll ever need to recover an entire disk's worth, and in any case, we need only do it lazily upon demand rather than immediately upon failure.

However, this technique only works well for related data streams which are deemed interesting (or not) as a single unit. If RAID is done over multiple data streams with no relation to each other, any preservations would have to be mirrored across streams to keep consistency for RAID. Collection of data would never be impaired, but the size of the ring buffer would be dramatically reduced in many circumstances as each stream had to save uninteresting sections to match interesting chunks in other streams.

Reliability is not limited to RAID alone; any erasure-correcting code would work well. Reed-Solomon codes (as an example) are not often used in high-performance storage because they have a high overhead for their encoding and decoding process. While the encoding process remains a factor here, it may be that decoding is never needed, even on drive failure, due to the reasons above.

4.5 Indexing

Indexing is one of the most difficult problems to deal with. It is reasonably simple to design a system that only needs to consider timestamps for each data element, and work with those alone. It is considerably more difficult to design a system which must index on multiple variables and quickly search through them. Without the ability to quickly search indexed information, data may incorrectly expire.

With large elements and relatively few indices, it would be possible to keep everything in memory. Large chunks indexed only by timestamps could be contained in only a few hundred kilobytes of main memory, and searching would be extremely simple.

However, there are many scenarios where the data elements are tiny and there are multiple indices per element. Consider storing IP packets which are indexed on source and destination addresses (4 bytes each), protocol (1 byte), and data length (4 bytes). Since we have to account for the worst case scenario, all packets may only be 20 bytes in size. Stored on a 2 TB drive, the indexing information would take hundreds of gigabytes of main memory: clearly not feasible. While this is an unlikely scenario, it *is* likely that there would be too much indexing information to store in main memory at some point in the lifetime of the system.

At least some portion of the index must be stored on disk, as there is no room for it elsewhere. Our solution is to attach an "index" segment to each chunk on disk which holds the relevant indices to that chunk. If nobody ever inquires about the data, the index segment expires along with the actual data. If someone does need to search, we can narrow it down as much as possible with whatever index we have in main memory (timestamps, at the least), and then work on searching the rest.

In order to not reinvent the wheel, we have elected to place the secondary search problem in the hands of a mechanism well suited to the task: a database. When we need to search on a chunk of data, we read the index portion off of the disk (a much smaller task than getting the entire chunk off, fortunately) and pass it into a database to perform the search. Afterwards, the data is dropped from the database again, never having to be stored for more than the time it takes to search in main memory. This style of lazy search allows us to optimize data storage according to our own bandwidth needs, but hand off the search problem to other systems already well suited to the task.

4.6 Scaling

Our system may need to scale upwards. A lot. The LWA may only be 53 stations at the beginning, but what if it (or a similar project) requires 500 stations? There is also a large difference between monitoring a network switch in a small complex versus a major router connecting a large site to the outside world.

Since our model is tied so closely to the hardware, we can scale the data capture portion easily. Each disk is bound to only a single data stream, and does not need to make any decisions on its own initiative – some sort of outside process (which may be running on the same hardware) needs to be monitoring the data in real time with the ability to quickly decide what needs to be saved. From the disk point of view, the only connection that its data stream might have with any other is by being part of the same RAID group.

From a control point of view, hundreds or thousands of streams may be tied together, and a controller process might want to preserve the data over a thousand different disks, at least long enough to do

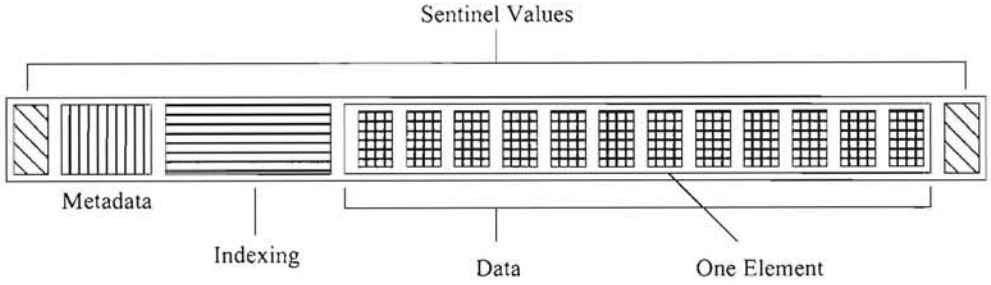

Figure 3: Data chunk layout

a more detailed search on it. As a communications problem more than anything else, we have not yet focused on this aspect, as existing communications methods like MPI are probably suitable. This is an area which we still need to fully explore.

5 Prototype Architecture

Our eventual goal is to create a specialized filesystem and interface layer, but for prototype and testing purposes, we first created a system named Mahanaxar. It is a multithreaded process running in userspace and accessing the disk as a raw device. Multiple processes can be run on the same machine, one per disk, but all access to a given disk *must* be through its associated process in order to manage bandwidth. Each process/disk is governed by a configuration file that specifies, among other things, what the chunk size is, what the element size range is, and how to index each element. State can be restored from scanning and re-indexing the disk upon startup.

Each process is made up of several threads, with the most important two being the data processing thread which is responsible for indexing and arranging the data into chunks, and the I/O thread which puts chunks on disk, and retrieves them again upon request. Our I/O model works upon a simple priority scheme: if there's a chunk of data ready to be put onto the disk, it gets first priority. A chunk is only read off the disk if nothing is ready to be written.

This method produces a somewhat jagged access pattern for reads (nothing for a while and then a chunk delivered suddenly and all at once), it allows us to make the most use of bandwidth by keeping the disk head in place as long as possible. The read process will never be entirely starved, though it may have wait a while.

As each element arrives in the storage system, it is indexed and placed into a chunk. If element sizes are large, one element may be equal to one chunk, and if element sizes are small, hundreds of elements may go into a single chunk. The default indexing is based around chunks: a pair of timestamps (microsecond resolution) which list the time that the first element started to arrive, and the time when the last element was fully received.

Figure 3 shows the chunk layout. Sentinel values are placed at either end of the chunk to manage consistency checking: they're written preceding and following the rest of the chunk, which allows us to determine which segments have been corrupted in the event of sudden failure. Metadata describes how many elements are in the chunk and where they are. This is also in the main memory index, but must be present on disk for recovery purposes. Indexing information follows, and then the actual data segment which may be composed or one or more elements.

The chunk size is configurable, but we have found that larger chunks (at least 50 MB) provide the best performance. In order to avoid wasted space, chunk size should be arranged so that it's a close multiple of the typical element size (including separate indexing information). If elements are 25 megabytes each, a 49 MB chunk size would be a poor choice.

If main memory is of sufficient size, and elements sufficiently large, each element can be individually indexed by timestamp and possibly other "primary ids." The system will not allow extra primary IDs if the worst-case scenario (all elements are minimum sized) means that a full index couldn't fit in main memory. If element sizes are particularly small, timestamp ranges are stored on a chunk basis rather than individual element basis.

Our system does mean that space is wasted if only one small element in an entire chunk is supposed to be saved. As currently implemented, Mahanaxar only allows saving chunks in whole or not at all, meaning that there is potentially a large amount of wasted disk space in worst case scenarios. We do, however,

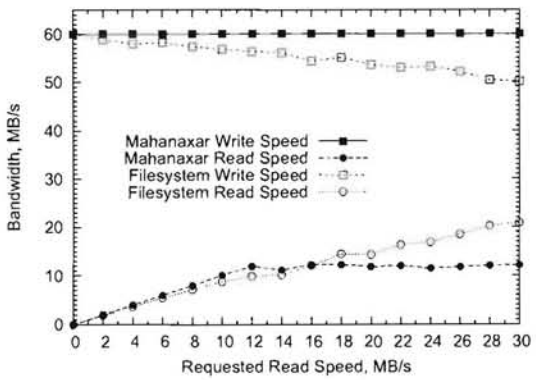

Figure 4: Performance of Mahanaxar against regular filesystem, 60 MB/s intended write speed

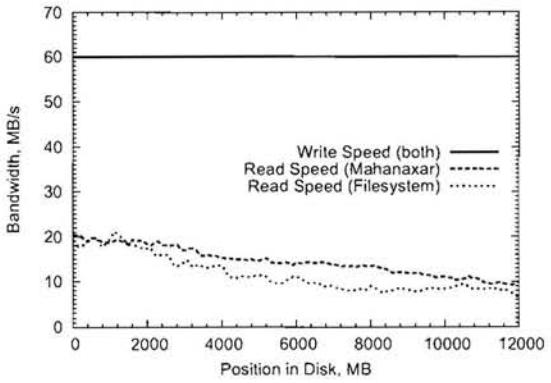

Figure 5: Performance of Mahanaxar against filesystem with strict priorities, 60 MB element size, 60 MB/s write speed

have plans to change this behavior in the future.

6 Testing

Our testing machine used an Intel Core 2 Quad processor clocked at 2.83 GHz (model number Q9550), and had 8 GB of main memory. The operating system was Debian 5.1. Because our system is I/O focused with reasonably light processing requirements, CPU power was never a major factor in our experimentation. We ran tests using several different disks on the same machine and achieved similar results on each disk, adjusted for its individual bandwidth. We carefully profiled one particular disk to use for the experiments presented here so that all comparisons could be made between systems running on the exact same piece of hardware.

The particular disk we used for generating the results we present here was a Western Digital Caviar Green of 1.5 TB advertised capacity. However, we should note that while the drive is measured by powers of ten, we have elected to use the binary convention in measuring KB, MB, etc., from here onward.

The raw write bandwidth of this particular disk averaged from 50 MB/s to 70 MB/s, and its read bandwidth from 57 MB/s to 115 MB/s. Upon profiling the drive, we determined that read bandwidth decreased in a linear fashion from its peak at the beginning of the drive at 115 MB/s to a point about 80% through the drive where it reached about 70 MB/s. Bandwidth dropped off sharply in the last 20% of the drive to a low of around 50 MB/s. Meanwhile, the write bandwidth only dropped from 70 MB/s to 65 MB/s in that same 80%, and followed the same sharp curve down in the last portion of the disk. A performance graph is available in figure 2, in the background section.

Due to this behavior, we elected to not use the "lower" portion of the drive and thus be able to offer a maximum write bandwidth of 60 MB/s, with 5-10 MB/s comfortably left over for reading. This left us with a usable disk size of slightly over one (binary) terabyte.

The disk write cache was disabled for our testing so that we could be (reasonably) sure that the data was actually on disk when we thought it was there. Interestingly, this slightly improved overall write bandwidth on the disks we tested on.

Our primary comparison point was a general purpose file system utilizing flat files. Because journaling file systems can have an adverse effect on raw throughput, we elected to use the ext2 file system in order to get the best performance to compare against (preliminary testing indicated that both ext3 and XFS did not perform as well as ext2 for this type of workload). Data was only sync'ed to disk after around a hundred megabytes were accumulated in each cycle so that the filesystem write cache could reorder to improve its own performance.

Mahanaxar explicitly syncs to disk after *every* chunk write to ensure that data is where we think it is. Doing this with small elements in a flat file system degrades performance massively, so we relaxed that requirement with the standard filesystem, even though it means that it can't guarantee data is saved in as tight a timeframe as Mahanaxar.

We also intended to compare against a pure database model, but quickly discovered that its performance could not come close to matching our own

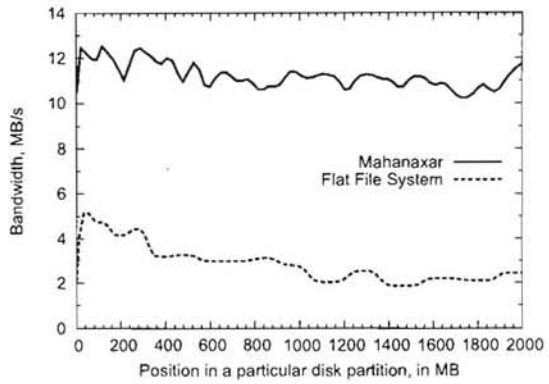

Figure 6: Closeup comparison of Mahanaxar and filesystem read performance, 60 MB/s data rate

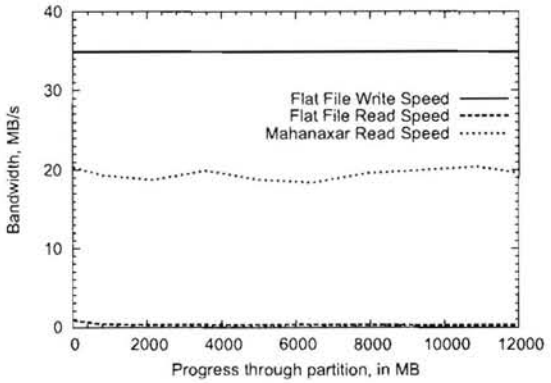

Figure 7: Closeup comparison of Mahanaxar and filesystem read performance, 60 MB/s data rate, 1 MB element size

system or the flat file based system. The initial population of our MySQL database was of comparable speed to our own system and the standard filesystem, but deleting elements took excessive time, slowing performance down to less than a third of the other two. It is excellent for searching data, but not for quick deletion and insertion of new data over the size of an entire disk drive.

Our primary testing procedure was to select various element sizes and measure what read bandwidth was available with each approach when we declared a certain write bandwidth (usually 60 MB/s). We ran tests over the entire space of the first 80% of the drive, and left the slowest portion partitioned into an off limits area. We also created smaller partitions within the space for certain tests in order to gather more finely-grained data.

We modified our testing procedure a few times when it became apparent that the flat file/filesystem approach was starting to fall behind. For example, when dealing with variable element sizes, the filesystem had to delete an appropriate number of files to create room, and create a new file of the appropriate element size. Because the file system was so full (as intended), fragmentation developed quickly and the filesystem was soon unable to keep up with the declared write bandwidth. In order to help it compensate for that problem, we pulled back on the variable element sizes and overwrote files in place to maintain the data locality initially set up by the filesystem.

7 Results

Figure 4 shows the results of Mahanaxar versus a normal flat file system. One process is attempting to write its data in real-time at a rate of 60 MB/s. Another process is attempting to read from the disk at much lower rates (2-16 MB/s). The bandwidth of the drive in that particular area has more than enough capacity to allow 60 MB/s of writing and up to 12 MB/s worth of reading if the two processes are properly sharing access.

The values presented in the graph are the average read and write bandwidths of Mahanaxar and the flat file system over an entire partition which is already populated with existing data at the start of the test. Mahanaxar maintains a constant write bandwidth of 60 MB/s no matter how much bandwidth a writing process asks for. By contrast, the flat file approach starts falling behind the required 60 MB/s bandwidth with a process trying to read only 2 MB/s from the same disk. At 2 MB/s, it has fallen about 2% behind the required write speed. By 10 MB/s (which the disk can easily handle if managed correctly), the flat file approach can only record 94% of the incoming data, while the write process still can't even get its requested 10 MB/s (it gets about 8.5 MB/s).

The write (and read) bandwidth for the flat file approach drop off even more sharply at more bandwidth than what the disk can provide is requested, whereas Mahanaxar simply throttles the read requests in order to maintain its full 60 MB/s write bandwidth. Because of this disparity, we introduced a similar mechanism for the filesystem in our other tests to ensure that writing always had priority.

Figure 5 shows the results of a full-disk test with a write bandwidth of 60 MB/s. With the new mechanism to ensure that the filesystem always had priority for writes, it was able to keep up with Mahanaxar.

However, we purposefully designed this test to give the flat filesystem ideal conditions by specifying a 60 MB element size. By writing elements at no smaller than 60 MB, the flat filesystem gained several advantages of Mahanaxar in that it could do the equivalent of "large chunk" writes. Even so, the maximum read speed available in the regular fileystem was less than Mahanaxar.

Figure 6 shows what happens when the filesystem has been fully populated and is "looping around" again. It shows a detail view (focusing on a small region of the disk) illustrating the difference in read bandwidth as elements start expiring and new ones were added. We should also note that we let the flat filesystem maintain an in-memory list of element ordering rather than looking up the oldest elements via metadata for every deletion – that latter approach leads to quickly-degrading performance.

The write bandwidth for both systems remains at 60 MB/s, but the read bandwidth for the filesystem based approach drops down to 2-4 MB/s, compared to Mahanaxar's 10-12 MB/s. The flat file system seems to stabilize around that 2-4 MB/s range after the collection process has been going on for some time, though only because all elements are of fixed size for this test and can be overwritten in place. When the elements are variable size, the performance of the flat file system slowly degrades over time.

The x-axis may require a bit of explanation: in the case of Mahanaxar, the x-axis is a literal representation of where the data is in the disk. It is measured from the start of the segment by 60 MB chunks. For the flat file system, the units are the same, but they do not necessarily correspond with physical location in the disk because the file system does not place exactly linearly.

Figure 5 shows performance for 60 MB element sizes, an area where the flat file system performance can almost match Mahanaxar. Figure 7 reduces element size to 1 MB each and shows part of what happens there. Mahanaxar retains a write bandwidth of 60 MB/s (not shown on the graph in order to see detail at the bottom) and maintains a read bandwidth of around 20 MB/s: precisely what it could manage with 60 MB elements. The reason it is able to do so is because it combines elements together into 60 MB chunks and has virtually identical performance for getting the data on and off disk.

However, the flat file based system can no longer keep up once elements reach 1 MB. If not sync'ed, the filesystem can "pretend" to keep up (and even have superior performance) for quite a bit of time as it takes advantage of write caching, but it eventually collapses under its own weight. Synchronizing after every element is unrealistic, however, so we only forced a sync every 100 elements (MB) to keep it honest.

We found that the maximum write performance that the flat file approach could manage was about 35 MB/s. At 38 MB/s and above, it would slowly fall behind over the course of time and eventually start losing data. At 35 Mb/s it loses no data, and can read at about 1 MB/s. This compares very poorly to Mahanaxar's smooth 60 MB/s write and 20 MB/s read.

When we tested variable element size on the flat file system, performance decreased steadily over time without appearing to stabilize at any point, probably because fragmentation gets worse and worse as old elements of variable sizes are deleted to make room for new elements of different sizes. We also tested the performance of a database storage system, but initial performance was only a third that of Mahanaxar, and it collapsed very quickly as old elements expired and new elements arrived.

8 Conclusion and Future Work

The performance of Mahanaxar shows that it has a clear edge over standard filesystems for the "write once, read rarely" workload. By staying very close to the physical hardware and aligning our workload appropriately, we are able to provide real guarantees on quality of service to meet a set of hard real-time deadlines. We are able to reach performance levels very close to the tested maximum of what a hard drive can handle, though this does necessitate generating profiles on a per-drive basis to ensure we can meet deadlines.

Even when standard filesystems are adapted to prioritize one data stream over all other disk activity, they cannot advertise as high a bandwidth as Mahanaxar is capable of, even when element size is large. With smaller element sizes, other file systems have performance at less than half the level of Mahanaxar for this style of workload.

Our future intentions are to turn this project into a full specialized filesystem and develop an API for interaction with it. We also need to address the previously mentioned problem of needing to preserve an entire chunk of data when only one element is worth saving, and develop the appropriate scaling mechanism to allow this system to be controlled over thousands of drives at once. We would also like to explore more complicated reliability schemes, moving beyond the RAID-like model we adapted for the prototype.

References

[1] ADVANCED TELEVISION SYSTEMS COMMITTEE, INC. *A/53: ATSC Digital Television Standard, Parts 1-6, 2007*, 3 January 2007.

[2] BOULDER REAL TIME TECHNOLOGIES, INC. *Antelope: ARTS configuration and operations manual*, 3 November 1998.

[3] BRUNO, J., BRUSTOLONI, J., GABBER, E., OZDEN, B., AND SILBERSCHATZ, A. Disk scheduling with quality of service guarantees. In *IEEE International Conference on Multimedia Computing and Systems* (7-11 June 1999 1999), pp. 400–405 vol 2.

[4] CHADD, A. http://devel.squid-cache.org/coss/coss-notes.txt, 2005.

[5] DATADIRECT NETWORKS. Best practices for architecting a lustre-based storage environment. Tech. rep., DataDirect Networks, 2008.

[6] GRID, L. C. Gridbriefings: Grid computing in five minutes, August 2008.

[7] HSU, W. W., SMITH, A. J., AND YOUNG, H. C. The automatic improvement of locality in storage systems. *ACM Trans. Comput. Syst. 23*, 4 (2005), 424–473.

[8] http://www.phys.unm.edu/~lwa/index.html.

[9] *Intel X25-E SATA Solid State Drive Product Reference Sheet*, 2009.

[10] KORNEXL, S., PAXSON, V., DREGER, H., FELDMANN, A., AND SOMMER, R. Building a time machine for efficient recording and retrieval of high-volume network traffic. In *IMC '05: Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement* (Berkeley, CA, USA, 2005), USENIX Association, pp. 23–23.

[11] MOLANO, A., JUVVA, K., AND RAJKUMAR, R. Real-time filesystems. guaranteeing timing constraints for disk accesses in rt-mach. In *The 18th IEEE Real-Time Systems Symposium* (December 2-5, 1997 1997), pp. 155–165.

[12] POVZNER, A., KALDEWEY, T., BRANDT, S., GOLDING, R., WONG, T. M., AND MALTZAHN, C. Efficient guaranteed disk request scheduling with fahrrad. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (New York, NY, USA, 2008), ACM, pp. 13–25.

[13] RANGASWAMI, R., DIMITRIJEVIĆ, Z., CHANG, E., AND SCHAUSER, K. Building mems-based storage systems for streaming media. *Trans. Storage 3*, 2 (2007), 6.

[14] TILAK, S., HUBBARD, P., MILLER, M., AND FOUNTAIN, T. The ring buffer network bus (rbnb) dataturbine streaming data middleware for environmental observing systems. In *e-Science* (Bangalore, India, 10/12/2007 2007).

[15] WU, J., AND BRANDT, S. Providing quality of service support in object-based file system. In *24th IEEE Conference on Mass Storage Systems and Technologies* (24-27 Sept. 2007 2007), pp. 157–170.