UNIVERSITY OF CALIFORNIA

SANTA CRUZ

MANAGEMENT OF HIGH-VOLUME REAL-TIME STREAMING DATA IN TRANSIENT ENVIRONMENTS

A dissertation submitted in partial satisfaction
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

David Bigelow

September 2012

The Dissertation of
David Bigelow
is approved:

Scott Brandt, Chair

Carlos Maltzahn

Charlie McDowell

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Management of High-Volume Real-Time Streaming Data in Transient Environments

by

David Bigelow

In an information-driven world, the ability to capture and store data in real time is of the utmost importance. The scope and intent of such data capture, however, varies widely. Individuals record television programs for later viewing, governments maintain vast sensor networks to warn against calamity, scientists conduct experiments requiring immense data collection, and automated monitoring tools supervise a host of processes which human hands rarely touch. All such tasks have the same basic requirements — guaranteed capture, management, storage, and analysis of streaming real-time data — but with greatly differing parameters. Our ability to process and interpret data has grown faster than our ability to store and manage it, a characteristic which now hinders our ability to exploit it.

The work presented in this dissertation demonstrates a means of integrating data management with the physical storage layer in order to gain superior performance not otherwise achievable. By fusing a close understanding of the disk hardware with the necessary components of a high-performance storage system, a unique method of data handling is constructed. This approach allows for hard performance guarantees and quality of service regulation at near-maximum hardware capabilities in a transient data environment for indefinite periods of time. The core storage system is made to understand the data as more than just a stream of bytes, uniting indexing and query capabilities into basic operations. All such gains are fully compatible with the accoutrements of a large-scale storage system, such as reliability and control mechanisms, which results in a fully viable new storage architecture.

Acknowledgments

I would like to extend my thanks to everybody who supported me in this work. My particular thanks go out to Scott Brandt, my academic advisor for these past several years; and to Carlos Maltzahn and Charlie McDowell, members of my reading committee. This work was funded by the Institute for Scalable Scientific Data Management, a joint collaboration between Los Alamos National Laboratory and the University of California, Santa Cruz, and portions were funded by LDRD Project #20080729DR; my further thanks go out to current and former Los Alamos staff involved in this project, especially John Bent, HB Chen, and Sarah Michalak. Members of the Systems Research Laboratory have frequently contributed useful comments and suggestions which have improved this research effort during my time in Santa Cruz. Finally, above all others, I would like to thank Gillian Smith.

Chapter 1

Introduction

We live in a world that is increasingly fixated on collecting boundless amounts of information, and yet the difficulties in storing and managing this vast accumulation are all too often ignored as we focus instead on understanding and manipulation. Data is processed and interpreted at ever-increasing speeds and in ever-increasing amounts while we simultaneously encounter ever-increasing difficulty in preserving all of it in a timely fashion. This has led us into the curious condition of often being able to recognize the importance of data without having the ability to store it for later use. Thus we find ourselves presented with an unenviable choice, if granted the decision at all: we must either slow down our handling of new data, or we must discard the old and hope that we will not need to consider it at some future occasion. Neither option is palatable.

Decades of evolutionary trends in computing hardware make it unlikely that we will be able to store and manage data at anywhere near the speed at which we can process it, as computational abilities advance at a far greater pace than storage abilities[18]. However, we can use existing storage hardware to greater effectiveness and narrow the performance gap between processing and storage in certain problem domains. To do so, data must not only be stored and retrieved faster, but it must be done *predictably* faster at all times. We are capable of doing this with relatively small amounts of data on expensive systems — for example, in real-time databases which rely entirely upon main memory for storage[42] — but we cannot yet do so with great amounts of data, or on large-scale commodity storage systems. Nor can we search out unknown data in a predictably fast time on such systems. This body of work offers methods by which these desires can be achieved.

Data interaction with a storage system may be generally divided into two broad classifications for the consideration of real-time requirements. Most personal computing tasks do not need to meet strict real-time deadlines for data I/O, and may generally be held to a standard of "complete this before the user starts to get annoyed." Beyond this natural maxim, it is usually not particularly important if a document takes twice as long to commit to disk on one occasion, as opposed to the previous occasion. Many high-performance and scientific tasks also fall within this category, albeit with a much higher "annoyed user" threshold because of the use and expense of such machines. Other tasks are sensitive to real-time requirements, including most multimedia applications. For example, video file playback must maintain a certain data rate to meet user expectations. Some tasks are extremely sensitive to real-time requirements, particularly where real-world sensor data is involved. A storage system recording sensor output from a video camera feed must be able to keep up with data storage on a one-to-one basis, or potentially valuable information will be lost.

All those who capture and store real-time data face an inevitable choice: continuously procure new media to avoid exhausting the available storage space, or discard old data and overwrite it with new. At high enough data rates, this choice may automatically resolve to the latter case for economic reasons alone. Although this latter case may be unpalatable in general, the best way to deal with it is to make sure the data is retained for as long as possible, and ensure that the most "interesting" portion can be quickly located when necessary. A situation where data is constantly cycling in and out of storage is not one that standard file systems and storage architectures are meant to handle, and makes for significant required changes in data management procedures.

A common consumer-grade example of this problem is demonstrated in Digital Video Recorders (DVRs), which are present in over one third of United States households[70], and which record television programs as they are broadcast. DVRs are often configured to constantly record other programs that the owner *may* find interesting, and are thus constantly deleting previously recorded programs to free storage space for new ones. It is possible that such older programs would in fact be interesting to the owner, but there is often no time to investigate them before storage space for the new content is needed. The storage is too limited to retain everything of potential interest. However, although this is a practical example of having too much real-time information, it is also a small-scale one. DVRs only require up to 2.75 MB/s of bandwidth[1] per program, a data rate which poses no difficulty whatsoever to any standard consumer-grade hard drive.

The Large Hadron Collider (LHC) at CERN generates continuous data at a rate of around 300 MB/s[30], two orders of magnitude greater than that of television. This larger data rate is still easily managed by a large backend system and global network, and the data *can* be preserved indefinitely because of the scale of the project. However, this 300 MB/s bandwidth is only reached by filtering out most data at the sensor level. Now, new projects are being developed which generate still more data. The Long Wavelength Array (LWA) Radio Telescope[36] is one such project and has a projected initial data rate of slightly over 3.75 GB/s, another order of magnitude beyond the LHC.

Data at this scale is usually infeasible to store indefinitely. The LWA generates a full petabyte of data in just over three days, and this rate is sustained twenty-four hours a day, three hundred and sixty-five days a year. The greater portion of this data must be *defined* as "unimportant" when judged against the cost of its long-term storage. Though it is continuously collected, data in this magnitude can only be stored for a short time before overwriting it with new, retaining permanently only that which is most interesting. Merely finding the location of temporarily stored interesting data presents a significant challenge if it cannot be immediately identified as such upon its initial capture and storage. Even in current tasks where this class of data exists, such as in certain radio astronomy projects, the data must be captured in "bursts" which are analyzed offline, rather than continuously[52].

No storage system has yet been designed to effectively manage this class of data: continuously written but rarely read. Most storage systems lack even rudimentary quality of service abilities and cannot guarantee an ability to meet hard real-time deadlines. Many storage systems offering performance isolation can only do so for certain workloads, such as multimedia access[26], and even with isolation, cannot guarantee deadlines. Similarly, systems which can provide statistical performance guarantees[74, 16] cannot offer hard guarantees. Other systems which *can* offer hard guarantees[55] do so based on disk head time, rather than absolute bandwidth. It is possible to translate disk head time into bandwidth, but only by cooperating intimately with the storage system itself. Real-time databases[4] are capable of making the performance guarantees that are required, but do so by keeping all information in main memory, which is infeasible for massive amounts of transient data. Systems based entirely on main memory and set to record data at LWA-like speeds would have a lifespan measured in minutes and would be extraordinarily expensive compared to disk, and are therefore unacceptable.

Instead, a new storage system architecture with a specific set of characteristics is required to adequately handle this class of data. The core requirement is an ability to guarantee bandwidth and meet real-time deadlines when using unreliable disk drives, which are nonetheless the only economical online bulk storage devices available. These guarantees must be made in spite of the highly transient data flow, which cycles data out of the system far quicker than in other application areas, creating extra difficulty for management purposes. Furthermore, such data must be quickly indexed and readily searchable, as the system will lose all of its usefulness if data cannot be found in short order. Such a setup must also support a variety of standard storage system accoutrements, which are not necessarily important to the application, but are important in ensuring faithful operation. Examples of this latter category are control schemes for distributed storage networks, and reliability mechanisms to guard against data loss from hardware failure.

Simultaneous fulfillment of these multiple requirements require the adaptation and restriction of the problem area. Existing filesystems, and general-purpose filesystems in particular, cannot adequately handle this specific class of data because they are meant to concurrently handle a broad variety of other data types at the same time. Such generalization hampers the strict handling requirements of this class of data, and thus it must be at least partially unwound and restrictions placed upon formatting, layout, management, and other aspects. By setting generalized filesystem standards aside, many of their associated performance penalties also vanish. This results in a "less capable" storage system for generic data, but a much more efficient one for that which it does support.

The most prominent standard which need be set aside is the virtualization of storage device hardware, which allows users and applications to ignore the physical medium on which their data is maintained. Virtualization of this hardware is a major advantage in general-purpose computing, but it also prevents the maximum utilization of said hardware. A conspicuous example is the rotational hard disk drive. Disks, the standard in bulk online storage, are physical devices subject to mechanical constraints in their operation. Modern trends are to virtualize storage inasmuch as possible, especially in large datacenters[65, 32], thus allowing file systems to work equally well on rotational drives, solid-state drives, main memory "drives" and others. However, if one does not know the details of the underlying hardware, it is impossible to predict what it will do and how it will perform in any given circumstance. By profiling hard drives and understanding their mechanical nature on an individual basis, it is possible to operate very closely to the hardware and obtain better performance than any virtualized filesystem running on the same hardware.

Existing storage systems generally take no special notice of transient data, mainly because there are few mechanisms to explicitly define which of the data is meant to *be* transient. Outside of a few narrowly defined categories, such as data not meant to survive beyond the next system reset, transient data is treated no differently than any other sort. Temporary files, rolling backups, and web caches all have limited (and sometimes predictable) lifetimes, where old data is regularly replaced with new. Normal metadata notations are made, indexing and search systems consider it in the same light as any other data, and there is little to distinguish it to an external observer. These standard methods are ill-suited to a system where *all* data is transient and where it must be replaced in a predictable, regular, and extremely time-sensitive manner. Such a system cannot spare the time to seek out each piece of data due to be replaced and cannot afford to let multiple users and applications proceed with their business in an uncoordinated fashion when deadlines are to be met.

Indexing and search are particularly important problems in this regard. Low-lifetime data requires an equally low-lifetime index, which means that a conventional slower buildup of indexing material[27] is inappropriate. It is often the case that such data is never useful and need not be searched: it may be stored and cycled out again before any search is ever initiated. Pre-emptive indexing and search would thus be wasted in many circumstances. Contrariwise, if one waits on dealing with such matters until a search is initiated, it may be difficult or impossible to find the requested data before it is cycled out of the system. There is no point to storing data in the first place if it cannot later be located, and thus performance guarantees are also needed for queries. This requires that indexing and search be treated as an integral portion of storage system management, inseparable from the data and not treated merely as an add-on component.

Finally, although many standards may be safely abandoned in the design of this new type of storage systems, other standards must be maintained. Reliability schemes are needed to prevent data loss in the face of inevitable hardware failure. At the same time, they cannot be allowed to interfere with the performance guarantees which are vital to the system, and they must be adapted accordingly. Similarly, this system must be able to scale beyond a single storage device, and yet this scalability must not be allowed to interfere with the fundamental performance requirements of normal operation. Certain engineering refinements are also possible to improve general usability and performance.

The primary motivation of this work is to examine this poorly-understood type of high-volume transient data and present a framework by which it can be effectively and efficiently managed. Specifically, the primary research questions addressed by this dissertation are as follows:

1. How can effective bandwidth performance guarantees be made when using unreliable physical hard drives?

2. What performance gains are possible through the combination of data management and organization with data storage?

3. What methods are needed to allow practical real-time search on secondary storage devices?

4. What large-scale data management techniques can be applied to high-volume real-time transient data?

In an effort to answer these questions, this dissertation introduces and demonstrates new integrated methods for storing and managing high-bandwidth real-time streaming data. It shows how physical hard disks may be effectively managed through profiling, thus supporting real-time performance guarantees on a basis of absolute bandwidth rather than disk head time (Chapter 3). Techniques are presented, and the results thereof are shown, for the management of high-volume fleeting data through close hardware integration and strict data layout limitations (Chapter 4). The advantages of careful manipulation and incorporation of indexing information is seen in the ability to guarantee search performance in certain classes of queries (Chapter 5). Lastly, there is discussion of the adaptation of certain storage system peripherals to non-standard modes of operation without compromising the primary quality of service requirements (Chapter 6). Such gains apply not only to this specific problem area, but can confer measurable performance improvements in other storage system architectures.

Chapter 2

Background and Related Work

The problem space of high-bandwidth real-time streaming data has never been directly attacked in storage systems. Nevertheless, there has been prior work in several sub-areas where other storage systems have had need of certain related aspects for different reasons. Many systems have tried to glean advantage from understanding the underlying hardware, and some have attempted to provide statistical, soft, and even hard real-time guarantees for certain types of data storage. Some of this prior work can be extended and applied to the problems confronted in this dissertation, while other aspects of it demonstrate paths which are inappropriate or inadequate to the task at hand.

Section 2.1 opens with an explanation of the physical characteristics of disk drives, and their typical use in modern systems. It goes on to explain previous work in disk profiling, disk-centric scheduling algorithms in file and storage systems, and also addresses the emergence of solid-state disk drives and how they may be applied to this problem space. Section 2.2 discusses existing real-time data capture systems and why they cannot cope with data at much larger scales. Existing scheduling techniques are studied in section 2.3, particularly in how they attempt to provide quality of service and/or performance guarantees. Indexing methods are reviewed in section 2.4. Section 2.5 discusses a number of reliability mechanisms used in modern systems.

2.1 Disks

Hard disk drives (HDDs) were invented and initially developed in the 1950s, popularized in the 1960s, and have been the mainstay of secondary storage hardware for the past five decades[28]. Their use continues to grow exponentially through the present day, particularly in large data centers and in the "cloud" environment[69]. The basic form consists of one or more magnetic platters rapidly rotating around a central spindle, while a series of arms (one per magnetic surface) position read/write "heads" at a desired radius over the platter. Data is magnetically stored on the surface of the platter, and technological advances have increased the density of data storage such that a single consumer-grade 3.5" form factor disk drive may be able to store as much as four decimal terabytes (4×10^{12} bytes) as of 2012[34].

The idealized disk is divided into a number of **tracks**, sometimes also known as **cylinders**, which represent the concentric circular regions of the disk defined by a particular radius. The latter term is often used in multiple platter disks with linked arm/head assemblies, where it can be imagined that multiple single-platter tracks are

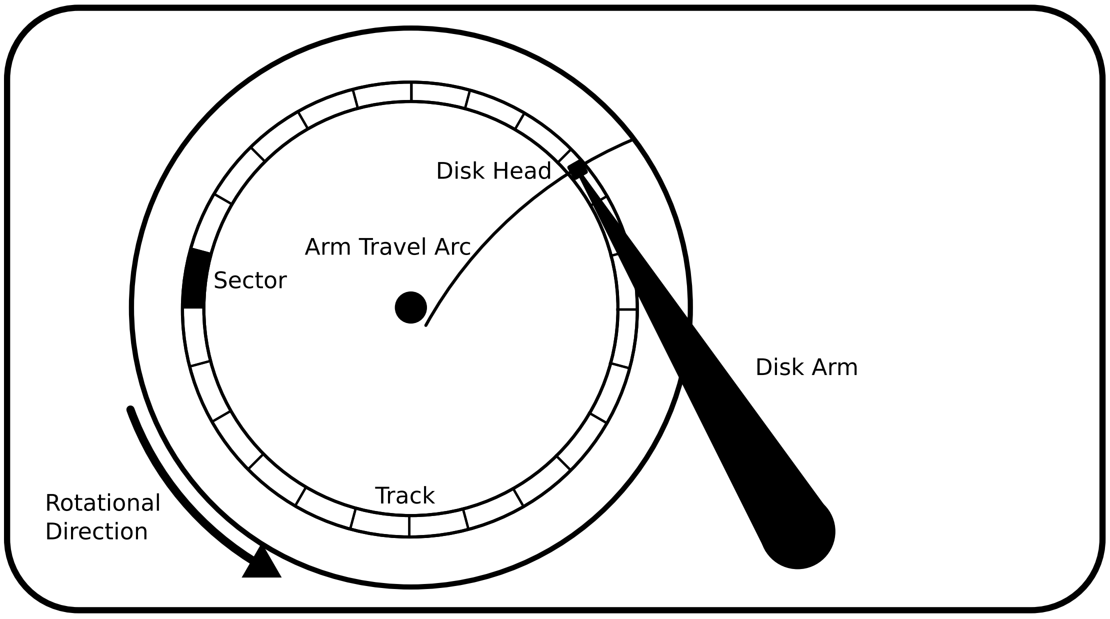


Figure 2.1: Diagram of disk mechanism.

layered on top of each other to form a cylindrical shape. Disk **sectors** are defined as the intersection of tracks and mathematical circular sectors based on the platter, such that the area defined by the sector is capable of storing a certain amount of data (historically variable, but now commonly manufactured to a 4096 byte standard[22]). Decades of advancement in disk technology have led to increased complexity in the internal hardware such that these traditional terms may not always retain a direct correspondence to the actual physical operation. Nonetheless, they remain commonly used when discussing drive characteristics, and are accurate enough for all but the most detailed analysis. Figure 2.1 illustrates standard disk components.

Disk design is thus seen to impose a number of gross mechanical limitations which cannot be easily overcome. This demonstrates why disks are known as "secondary" storage, being far too slow to serve as "primary" storage for normal computer operation. Specific factors in data access time are listed here in ascending order of impact.

1. The **Data Transfer Rate** is the speed at which data is transferred between the read/write head and the magnetic platter. This rate may differ based on the direction of data transfer (reading or writing).

2. The **Rotational Delay** is the lag incurred when the disk head is correctly positioned over the disk platter, but the appropriate sector has yet to rotate to a position under the head. The maximum amount of time lost through this delay is one revolution of the disk platter: for example, if the disk is rotating at 7200 revolutions per minute (a common speed for commodity drives), the delay may be as long as $8.\overline{3}$ milliseconds.

3. The **Seek Time** measures how long it takes the disk head to position itself over the needed track on the platter. At worst, the head will need to move from the innermost region of the platter to the outer edge, or vice versa, a process which generally requires several milliseconds on a modern disk.

4. **Spin-Up Time** is sometimes required if a disk has been spun down (e.g. it has ceased to rotate on the spindle), which is sometimes done to save power. It may take as long as several seconds to bring the drive back up to its required operational speed. When considering the case of high-bandwidth real-time streaming data, disk spin-down may safely be ignored, since the disk will never be clear of pending

I/O operations for a sufficient time to trigger it.

Thus it is possible to imagine the two timing extremes in data movement. In the best case, the disk head is already positioned over the proper track at the time of the I/O request and the platter rotation is such that the appropriate sector is just about to pass underneath the head. In the worst case, assuming no spin-up time, the disk head may need to traverse the entire disk radius (in either direction) and wait for a full rotation of the platter to bring the appropriate sector into range.

It is easy to picture the pathological worst-case data layout: a set of many small data elements alternately located at "opposite ends of the disk," which is to say the innermost (smallest radius) and outermost (largest radius) tracks. A more likely case is that of small elements randomly scattered over the entire disk surface. Conversely, the best possible data layout is a series of consecutive sectors on the same track, which allows the disk head to make one continuous I/O operation. Even when the disk controller reorders I/O operations for the most efficient disk head movement — something which nearly all controllers do — random I/O is orders of magnitude slower than well-structured sequential I/O.

Table 2.1 shows example I/O times and efficiency percentages for a hypothetical disk with 100 MB/s of bandwidth and a 4 millisecond seek time, which are consistent with advertised "average" figures for commodity hardware of 2012. The average case is assumed for these numbers, with the best and worst case times being considerably better and worse. It is evident that disk performance is directly related to the I/O size, with larger I/O sizes yielding significantly greater efficiency up into the tens of

	Transfer Time	Time for one I/O	Efficiency	Total time, 10 GB of I/O
64 KB	0.000625 s	0.004625 s	13.5%	757.76 s
128 KB	0.00125 s	0.00525 s	23.8%	430.08 s
256 KB	0.0025 s	0.0065 s	38.4%	266.24 s
512 KB	0.005 s	0.09 s	55.5%	184.32 s
1 MB	0.01 s	0.014 s	71.4%	143.36 s
5 MB	0.05 s	0.054 s	92.5%	110.59 s
10 MB	0.1 s	0.104 s	96.1%	106.50 s
25 MB	0.25 s	0.254 s	98.4%	104.04 s
50 MB	0.5 s	0.504 s	99.2%	103.22 s
100 MB	1.0 s	1.004 s	99.6%	102.81 s

Table 2.1: Comparison of data transfer times for different I/O sizes at progressive locations. Hypothetical disk bandwidth is 100 MB/s and seek time is 4 ms.

megabytes. These results have been demonstrated many times[35, 23, 79] in practical systems, consistently showing extreme sensitivity to the physical layout of data.

However, the "average" seek time is not necessarily the expected seek time for some modes of operation, or the worst-case disk head movement. Manufacturers usually consider a standard filesystem when determining the average seek time, rather than a situation where the disk head may be moving to random locations on the disk for each new I/O. Figure 2.2 shows a visual depiction of the theoretical I/O times for a disk of 100 MB/s average bandwidth, and seek times of 4, 7, and 10 milliseconds.

2.1.1 Disk-Centric Scheduling

The impact of data locality on data transfer rates is not a recent discovery. A paper by H. Frank in 1969[23] noted that performance could be greatly improved by merely re-ordering the I/O requests to more closely correspond with the physical data layout. Further developments in 1972 by David D. Grossman and Harvey F. Silverman

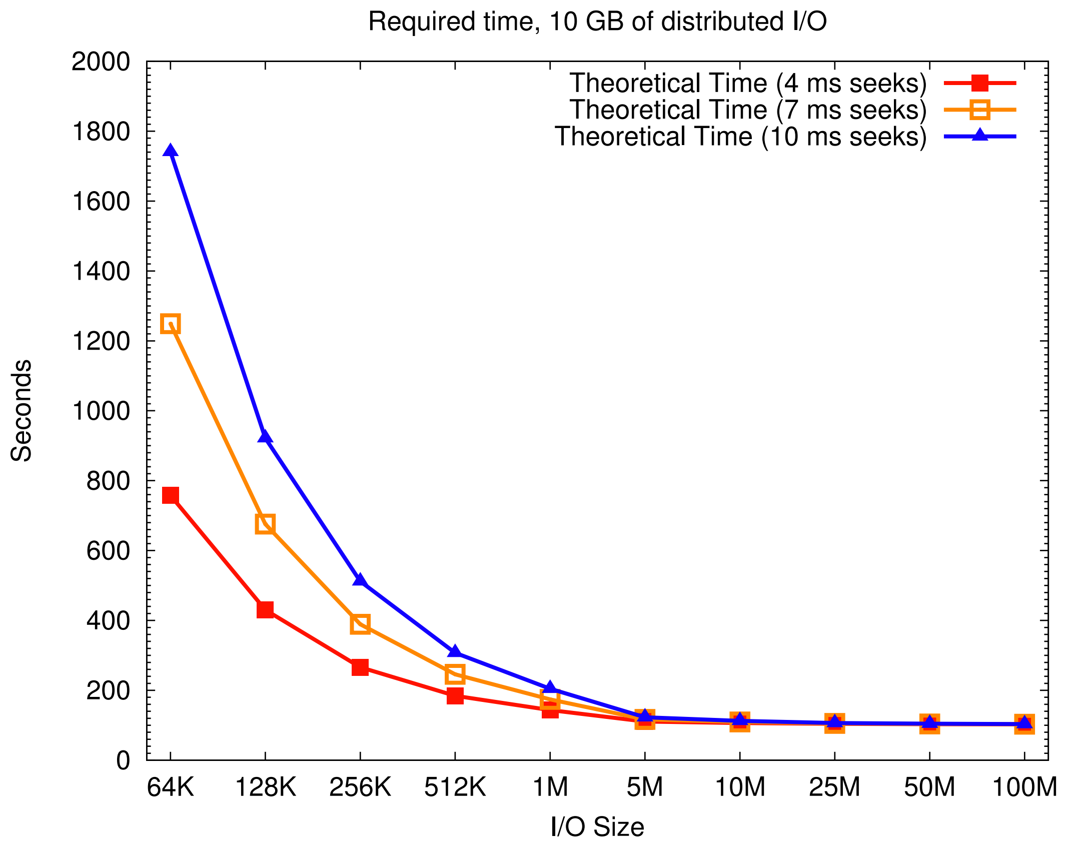


Figure 2.2: Comparison of data transfer times and efficiencies for varying I/O sizes. The disk is assumed to have a theoretical average bandwidth of 100 MB/s.

established a mathematical model for calculating appropriate data locations for access time minimization[31]. Since that time, many disk scheduling algorithms[25, 63] have attempted to minimize seeking and rotational latencies to maximize efficiency.

Despite such efforts, disks are usually regarded as unreliable for real-time purposes with good justification. Their mechanical nature and the opacity of their internal data arrangement causes difficulties when attempting to specify a tight time boundary for I/O operations. Worst-case assumptions must be extremely pessimistic to account for the unreliable data transfer times[42]. These assumptions result in being able to utilize only a fraction of the drive's peak capabilities.

Even those disks which are in constant operation may have significant amounts of unutilized bandwidth available. Christopher Lumb et al. estimated that a "never-idle" disk may actually be able to use 20-50% of the bandwidth to service background applications with no effect on the foreground response times[46]. Certain types of background I/O may take place during the rotational latency delays of normal disk usage, a technique called "Freeblock Scheduling"[45]. Although it is only a passive technique and has limited usefulness, this method can still be used to scan disks for report generation, indexing sweeps, and similar purposes. Certain assumptions must be made about the disk for these techniques to succeed. The authors note that the required coarse-grained assumptions are a specific challenge, and logical block addresses may not map directly to physical configurations.

An interesting aspect of the modern disk drive is best summed up in two words: *disks lie*. The ever-growing complexity of the modern disk drive has led to a necessary expansion of on-board “intelligence” in an effort to increase performance. Part of this expansion has led to a partial departure from the simple historical disk model pictured in figure 2.1. Specifically, disks may internally remap their layout such that a logical block address (LBA) at the software level does not necessarily have a direct correspondence to the physical layout.

Each disk manufacturer does something different, even on a per-model and per-firmware basis, in an effort to increase performance and lessen the burden on a general-purpose filesystem[67]. Although well-intentioned, this has the unfortunate side effect of hindering precise predictions for data transfer times. Bruce Worthington et al. did some online characterization of disk performance[79] in an effort to gain a closer understanding of LBA-to-disk mappings. Although this work allowed for increased performance and predictability, continued disk development has made such characterization increasingly difficult and little follow-on work has yet been done along this line. However, it is precisely in this area that the most significant performance gains may be realized for the problem space of high-volume streaming real-time data.

It is also possible to manage high-bandwidth real-time data by “throwing disks at the problem.” Several existing standard systems are capable of keeping up with real-time demands by doing so, though it is an inelegant solution. Google has developed a reasonably sophisticated mass disk-based approach with Bigtable[17], although it too is not designed for hard real-time guarantees. It is capable of declaring old data obsolete and jettisoning it accordingly, to replace it with the more “interesting” new data, but only on a best-effort basis. Finally, for bursty applications, an intermediate layer can be placed between the memory and the hard drive. This layer (often MEMS based) can smooth out a bandwidth curve for steady performance, but is most useful in an environment where infrequent bursts of data are expected from time to time[59], rather than in one where data is constantly arriving at a high rate.

2.1.2 Solid State Drives

Solid-state drives (SSDs) are rapidly gaining prominence in the secondary storage market. Rather than rotating magnetic platers and mechanical arm assemblies, SSDs use integrated circuits with no moving components. The operational opacity of SSDs is considerably greater than that of HDDs, and the physical internal location of data may frequently change without external notice. This opacity is understandable in light of the technological constraints of SSDs, and their greater internal bookkeeping. They are generally faster, smaller, lighter, more durable, and more power efficient than HDDs. However, they also cost considerably more per unit of storage. A 2009 study by Dushyanth Narayanan et al. of Microsoft Research investigated the replacement of HDDs by SDDs in enterprise storage[49]. Its conclusion was that replacement was not cost effective for any studied workload, and that "the capacity per dollar of SSDs needs to increase by a factor of 3-3000 for an SSD-based solution to break even with a disk-based solution."

Not being tied to any mechanical movement, SSD data transfer rate is uniform over all regions of the device, within certain operational constraints. This helps to improve predictability, but those gains are partially offset because overall performance changes based on device usage patterns and physical aging[68]. Tests show that latency and bandwidth are not as variable as mechanical HDDs, but may still impose a significant performance impact.

Solid-state drives are known to have limited write endurance, which can pose a problem in certain storage environments. For example, the endurance of a highly rated Intel SSD is officially put at somewhere between one and two petabytes[38], which may be a rather low limit when working in a data-intensive system. The lifetime of such a drive may be measured in months under maximum workload conditions.

2.2 Real Time Data Capture

The collection of sensor and other real-time data is not a new problem, but existing systems and applications are not designed to cope with a constant cycle of high-bandwidth data. Such data, if too large to save in its entirety, is traditionally dealt with in one of three ways:

- A digest of the data is saved.
- The data is picked through to find only the most "interesting" elements, which are then saved.
- A statistical sampling of the data is saved.

No existing systems attempt to save *all* the data, whether temporarily or permanently, on the scale that this dissertation targets.

2.2.1 Ring Buffers

Ring buffers are a common tool for short-term data handling, frequently found in producer-consumer models. They have also been used for real-time sensor data capture in both commercial and academic settings. The *Antelope* Real-Time System from Boulder Real Time Technologies[10] is an environmental monitoring system incorporating an "Object Ring Buffer" which claims to accommodate any type of data and handles communication between various subsystems. DataTurbine[71] is similar, having started as an environmental monitoring system before evolving into an open source "real time streaming data engine," as it currently bills itself. Rajasekar et al. has presented a virtual object ring buffer framework[58] designed to manage multiple sensor data streams.

However, the focus of these systems is rooted in data processing, analysis, and the facilitation of communications between various subsystems. The underlying storage architecture is always assumed to be adequate the task at hand, and there are no special provisions to manage high-bandwidth data. The target capacity for Antelope is measured in megabytes over a span of minutes, while DataTurbine supports tens of megabytes per second. Neither data rate is remarkable for today's hardware, with single commodity hard drives supporting tens of megabytes of per second as a matter of course.

The COSS storage system[15] from the Squid proxy server also utilizes a ring buffer based model, but has no real time component. It is used as an intermediate web cache and expires data based on a first-in, first-out approach. Interestingly, COSS rewrites requested data at the head of the buffer each time that it is accessed, which simplifies the ring buffer model at the expense of wasting disk time. It functions on a purely best-effort basis.

2.2.2 Network Traffic Capture

Network traffic capture is a domain where there is often more data available than there is capacity to store it. Online systems are often capable of detecting a network intrusion attempt in progress, or of finding the fingerprints of an invader after the fact, but can only track such events in detail from beginning to end when already alerted to their presence. Large government organizations were once able to record all network traffic in and out of their systems[24], but two decades of steady Internet growth has rendered it infeasible to currently do so.

Many tools are designed to address network monitoring tasks, but their storage capabilities are usually limited to data digests, or pre-specified rulesets defining the most "interesting" data that should be captured. Argus[3] is a real-time flow monitor that tracks data network transactions, and can capture packet data in reduced forms. NetFlow[19] is a Cisco-developed protocol intended to collect IP traffic statistics for separate analysis. FlowMon[72] is an autonomous probe that gathers statistical information and passes it off to collectors for storage and analysis. All of these tools assume sufficient storage resources and although they are capable of capturing full packets when specifically instructed to do so, it is only on a basis of "from here on out."

Narus, Inc. builds systems designed for full deep-packet inspection of network traffic, and advertises the capability to fully intercept specified datastreams[50]. However, this capability is also limited to pre-specified rules only, either through explicit instructions or upon encountering packets which meet preset criteria. It is not seemingly capable of maintaining even a short history of the entire datastream. As with ring buffer based systems, it is designed to interact with an external storage system in normal operation rather than provide a storage solution of its own. Data management is kept separate from storage system design, rather than making any attempt to gain advantage through a merged approach.

The network traffic capturing "Time Machine"[40] from Lawrence Livermore National Laboratory *is* designed to consider storage system factors when capturing real-time data, but only in the sense that it is aware of such factors, rather than incorporating them into its design. It does not go so far as to integrate the storage system with the main tool itself, but does recognize the requirements and limitations of a disk-based storage system. It deals with the problem by classifying and prioritizing data streams, giving priority to what it deems the most important and dropping what it cannot handle.

Generalized from network monitoring alone, Frederick Reiss et al. have produced methods for querying data streams[60]. A bitmap indexing technique is applied in order to avoid storing the entire data stream where full archival storage has unacceptable cost. This technique is suitable for trend analysis, but does not actually store the raw data.

2.2.3 Real-Time Databases

Real-time databases are used for a wide variety of purposes, but rarely employ secondary storage[4]. Worst-case assumptions must be set extraordinarily high to ensure that deadlines are assuredly met[39], which leads to unacceptably low system capabilities. There has been some work in high-performance inserts, such as that by Bender et al. with “Cache-Oblivious Streaming B-Trees”[5] and their techniques for fast database inserts. Unfortunately such approaches are still not designed to explicitly meet real-time deadlines, and suffer some performance loss in searching as a tradeoff for the efficiency in insertions.

2.3 Performance Management through Scheduling

Hard real-time guarantees on secondary storage devices are possible but with a mixed record of success. Worst-case performance assumptions may be used to ensure that all deadlines are met, but this necessarily results in severe under-utilization of the storage device. However, quality of service guarantees without strict deadline requirements have proven more successful. Several scheduling techniques have been developed which allow storage systems to fulfill individual user performance requirements without sacrificing overall efficiency. These quality of service schedulers (including those which provide hard real-time guarantees) may be broadly divided into three groupings: reservation-based schedulers, statistical and fair-share schedulers, and schedulers which separate application classes. Some systems may be fairly placed into multiple groupings.

2.3.1 Reservation-Based Schedulers

The YFQ disk scheduling algorithm[12] allows applications to make reservations on both throughput and latency. Unfortunately, as the authors admit, the stateful nature of a hard drive makes bandwidth difficult to guarantee in the face of worst-case usage patterns, and they observe that "such guarantees may come at the cost of excessive disk latency and seek overheads, harming aggregate disk throughput." RT-Mach[48] and Zygaria[78] similarly provide resource reservation capabilities on individual disk drives, and similarly have the same shortcoming of being bound by worst-case assumptions. All provide modes of operation with a lesser guarantee for those applications which do not require strict deadlines. The reservable resources are only a fraction of the total disk capacity (in both throughput and latency), thus leading to a severely under-utilized disk unless best-effort tasks are used to fill in the slack time.

Several disk scheduling algorithms have attempted to leverage knowledge of the hardware, including the aforementioned YFQ. Grossman and Silverman presented theorems for the "Placement of Records on a Secondary Storage Device to Minimize Access Time" in 1973[31] to provide a theoretical basis for arranging data to minimize average retrieval time. Reuther and Pohlack implemented a rotationally aware disk scheduling algorithm in DROPS[61], allowing disk reservations based on throughput, and otherwise scheduling "best effort" requests based on knowledge of the data layout in the hardware. Again, even with detailed hardware knowledge, these systems do not support future predictions for data access patterns, and thus must continue to adhere to all worst-case assumptions.

Fahrrad[55] takes a different approach, guaranteeing disk head *time* rather than attempting to accept reservations based on overall bandwidth or individual I/O request latency. Individual applications are thus able to maximize their bandwidth and latency to the extent that they are able to manage their data layout on the physical hardware. They are not penalized for the extra arm and head movement imposed by other applications, but absolute real-time deadlines are still constrained by worst-case assumptions within their own request stream.

2.3.2 Statistical and Fair-Share Schedulers

Statistical schedulers and fair-share schedulers do not attempt to support real-time deadlines. Instead, they focus on making sure that applications are not “starved,” which is a possibility in systems concerned solely with maximum overall efficiency. In the case of statistical guarantees, starvation remains possible in the short term.

Lottery Scheduling[74] provides a randomized (and thus statistical) scheduling mechanism to support disk I/O. It does not claim to provide hard guarantees on response time, but can be used for “fair” resource allocation. Façade[44] and SLEDS[16] also focus on statistical guarantees, defining “virtual hardware” for external applications. Emphasis is made on isolating workloads from each other, and in throttling offending workloads to prevent them from unduly interfering with others. Both are designed for providing quality of service guarantees on larger storage systems over a variety of workloads, but are not suitable for hard real-time deadlines. The Cello[64] system is similar, providing a two-level architecture to separate application classes from each other in order to provide quality of service, though it is not designed to cope with “misbehaving” applications which make inappropriate resource demands.

Horizon[56] also uses a two-tier approach to manage QoS in distributed storage systems. It has higher-level mechanisms to control deadline assignments and workload shifting, while maintaining individual disk-level tools to handle the details of moving data on and off the hardware itself. It is intended to handle softer “request service guidelines” from a wide variety of applications, rather than hard real-time deadlines over the entire distributed system.

Larger storage systems can often guarantee a certain service level, as in Lustre[21] and Ceph[80], but only to the degree of categorizing traffic to maintain an appropriately “fair” level of service. No guarantees are made from moment to moment; service is allocated on the level of “a certain amount of time on a certain granularity level.” Such constraints are adequate for basic quality of service over a large shared system, but not for meeting hard real-time deadlines.

2.3.3 Application Class Separation Schedulers

Finally, some systems are able to meet real-time I/O deadlines by requiring certain preset conditions and workloads. Clockwise[9] and other multimedia storage servers[26] take advantage of the predictable file layout and workload specification of media applications to meet real-time deadlines in those circumstances. Cello[64] also provides application class separation in the context of statistical guarantees.

Minerva[2] is a tool that can automatically design large storage systems to meet specified performance goals, and can also be used to predict performance. It can support quality of service goals on a specific workload but utilization must be known in advance, and it cannot support applications which deviate from their declared performance settings.

Argon[73] is a storage system that manages resources to insulate applications from each other. Its goal is to provide applications with performance that, in a shared environment, is nearly as good as would be seen in an exclusive environment on the same system. It uses a weighted fair-share system among various request streams, but as with most other systems, performance guarantees are not explicitly made.

2.4 Indexing and Searching

Data storage is most useful when the data can be discovered and retrieved with a minimum of effort. It is possible, of course, to inspect every portion of data until the desired fragment is located, but this process is inefficient at best. A standard filesystem directory structure is adequate when data need only be indexed on one attribute, but different techniques are needed when data must be stored and searched for based on multiple aspects.

Semantic File Systems, as proposed by Gifford et al.[27] date back to 1991. This method of file organization was described as "[providing] flexible associated access to the system's contents by automatically extracting attributes from files." It allows files to be automatically classified and arranged based on the actual data within them, and introduces virtual directories to facilitate access via conventional mechanisms. Virtual file directors are generated by queries.

The work in Semantic File Systems was extended by Gopal and Manber[29] with the integration of content-based access mechanisms. User-defined namespaces are based on queries in combination with content-based access, but there are scope and consistency problems, and the authors admit that its ease of use is questionable. Soules and Ganger went further with Connections[66] which performs file search based on the context of data as interpreted by users. File system calls are traced to "identify temporal relationships between files and use them to expand and reorder traditional content search results."

The Damasc project[11] extends this further with "a configurable layer . . . added on top of the file system to expose the contents of files in a logical data model through which views can be defined and used for queries and updates." This integrates a view of the data into the filesystem itself, something which is vital to any work that makes quality of service guarantees, though this targets long-term data rather than short-term.

Metadata management is often used to assist indexing and queries, apart from the data itself. Spyglass[43] is a file metadata search system designed for very large storage systems, taking advantage of namespace locality and snapshot-based metadata collection to check only modified files. It works best in systems with a long-term outlook, but captures a lot of data which is not necessarily useful. SmartStore[37] is similar to Spyglass and combines a semantic-aware approach with metadata organization and search. It is decentralized and is specifically intended to organize filesystem metadata to prevent full-system searches and disk scans. Ceph[75] offloads metadata from main storage entirely, setting up dedicated servers to handle such matters, a technique which splits certain indexing tasks away from the main storage infrastructure.

Databases are well-suited to the generic problems of indexing and search, but are not necessarily efficient at doing so. Databases which require real-time support do not typically operate on secondary storage[39, 42], however, and are not well suited to larger transient data loads.

2.5 Reliability

There are two common methods used to prevent data loss upon disk failure. The first method is to mirror the data onto another piece of hardware. That is, to keep at least one fully up-to-date backup copy of all data, synchronized with the primary storage device. The second way is through use of a Redundant Array of Independent/Inexpensive Disks (RAID) and similar techniques, which incorporate extra drives (usually called "parity" drives) to provide a safeguard against loss of data through the loss of one or more data drives. The first approach, mirroring, is sometimes also referred to as "RAID 1."

The original RAID design by Patterson et al.[53] defined five different modes of operation. RAID 1 (mirroring) and RAID 4/5 (block-level parity) are still in frequent use today, whereas RAID 2 (bit-level parity) and RAID 3 (byte-level parity) have fallen out of favor. RAID 4 splits a piece of data into n different pieces and performs a bitwise xor operation over them to generate a parity block. These pieces, including the parity block, are then placed on $n + 1$ hard disks in an array; this is usually called a "data stripe." If one of these disks fails, the data block stored on that disk can be reconstructed by another xor of the remaining data blocks, and thus the data is guarded against one drive failure. RAID 5 works similarly, except that it spreads the parity block placement over all the disks for each stripe of data, to avoid placing the entire parity burden on a single disk. A single data stripe example of RAID 4/5 is shown in figure 2.3.

The basic RAID techniques have since been extended into methods capable of guarding against two disk failures. Two-disk protection schemes are frequently labeled as "RAID 6," although there is no specified standard as per RAID 1-5. EVENODD[8] requires a prime number of disks (some of which may be virtual) to tolerate two disk failures, and Row-Diagonal Parity (RDP)[20] is a more recent technique for the same purpose. There are other proprietary commercial implementations that also guard against two disk failures.

Other approaches can support a larger number of drive failures. WEAVER codes[33] can tolerate up to 12 failed drives with only XOR-based erasure codes. General error-correction codes, such as Reed-Solomon codes[54] can guard against any desired number of failures. A beneficial side effect of many of these techniques is the increased availability of bandwidth for I/O operations, when compared to a system using fewer drives.

Declustering approaches for RAID-like behavior, such as distributed sparing[47]

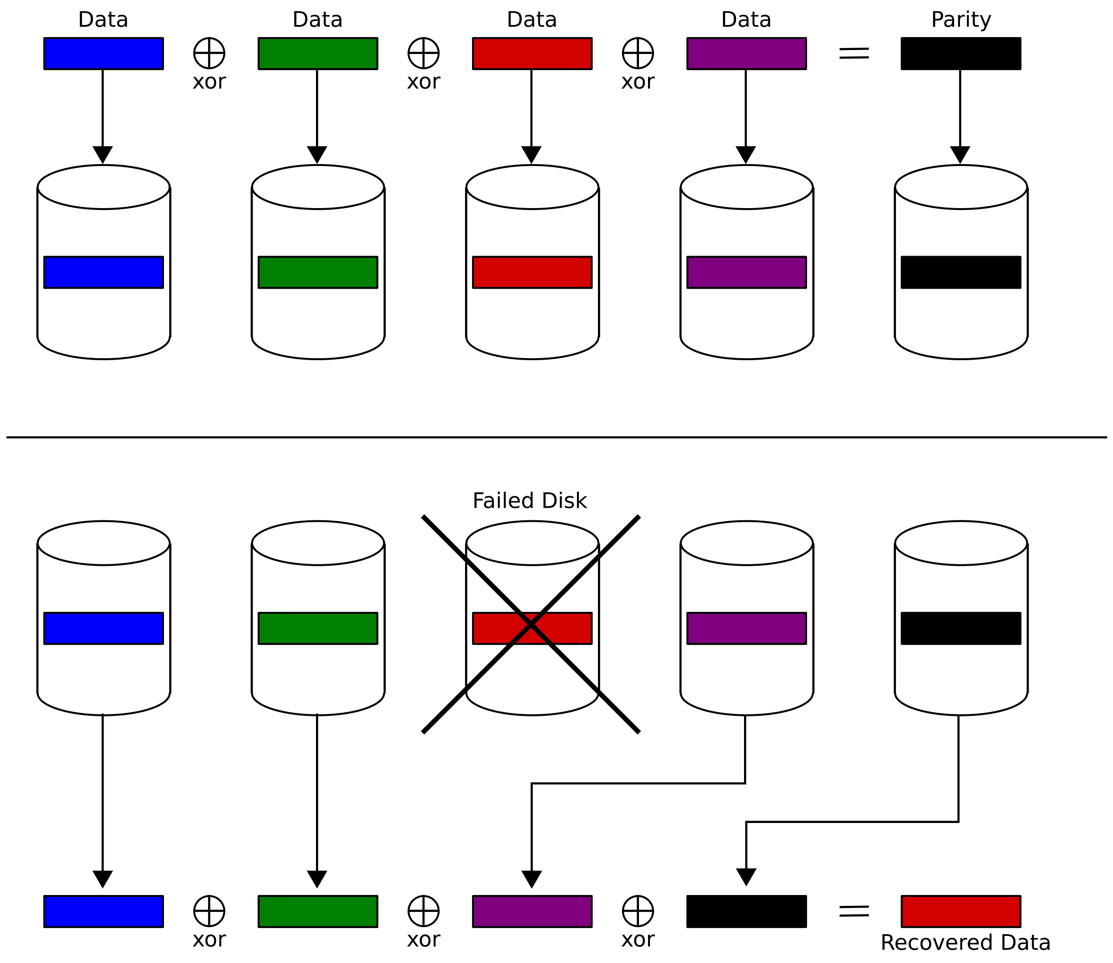


Figure 2.3: Basic RAID operation. The top portion illustrates the initial data distribution/construction, and the bottom portion shows data recovery.

use spare disks to rotate RAID-group placement among many disk combinations to better improve rebuild time and performance in degraded mode, though this requires a reduction in total capacity. Very large scale storage systems often integrate mechanisms of this nature, such as the FAst Recovery Mechanism (FARM)[81], and RAID groups in Panasas[51]. These approaches work well for rebuilding and recovery upon drive failure, reducing the total available bandwidth much less than standard RAID methods, but such approaches are only viable on very large storage systems due to the number of disks involved.

Chapter 3

Real-Time Guarantees with Unreliable Storage Devices

Hard disk drives are unreliable, yet remain the only practical source of bulk online secondary storage. Their operational state is described in section 2.1, along with the challenges that render them difficult to work with in quality of service and real-time deadline environments. Latency may differ by orders of magnitude merely by the order in which one accesses data, and the available bandwidth can change significantly based on whether the data is physically stored on an outer or inner region of the spinning platter. Disk firmware may not even truthfully report this location to an outside process.

Table 2.1 and figure 2.2 in section 2.1 list and picture example performance numbers for a hypothetical disk with a bandwidth of 100 MB/s. Figure 3.1 pictures measured data from an actual physical disk of similar characteristics. As in figure 2.2, the amount of data is ten gigabytes. The specific drive used for this comparison is a

Western Digital Caviar Black, model number WD2001FASS[76]. It has an advertised latency of 4.2 milliseconds, though the figure calculated by disk manufacturers is only "average" when assuming a standard file system and reasonable file layout on the disk. Maximum latency is often higher than 10 milliseconds for many disks, including this one. No specific bandwidth is advertised on the official product datasheet, but testing shows it to be roughly 100 MB/s (see section 3.1 for more information on calculating disk bandwidth).

Figure 3.1 is in three parts. Part (a) shows the bandwidth when the I/O pattern is progressive throughout the course of the disk. That is, all I/O operations take place at regular intervals throughout the disk, calculated such that the first I/O is at the "beginning" of the disk, and the last I/O is at the "end." Part (b) shows the bandwidth when the I/O pattern is truly random, which naturally leads to an increase in I/O times. Part (c) shows the bandwidth with a pathological worst-case data layout, where each subsequent I/O operation takes place at opposite ends of the disk. The disk's on-board write cache allows for re-ordered write requests, improving write performance in all cases, but this is particularly the case in part (c). An additional data line is shown in part (c) detailing write performance with the write cache disabled. This necessitates an expansion of the Y axis in this particular graph in order to accommodate the increased time, as compared to the maximum times in parts (a) and (b).

These graphs show that actual disk performance can very greatly; therefore, the first step toward efficient hard real-time guarantees must lie in a close understanding of the disk.

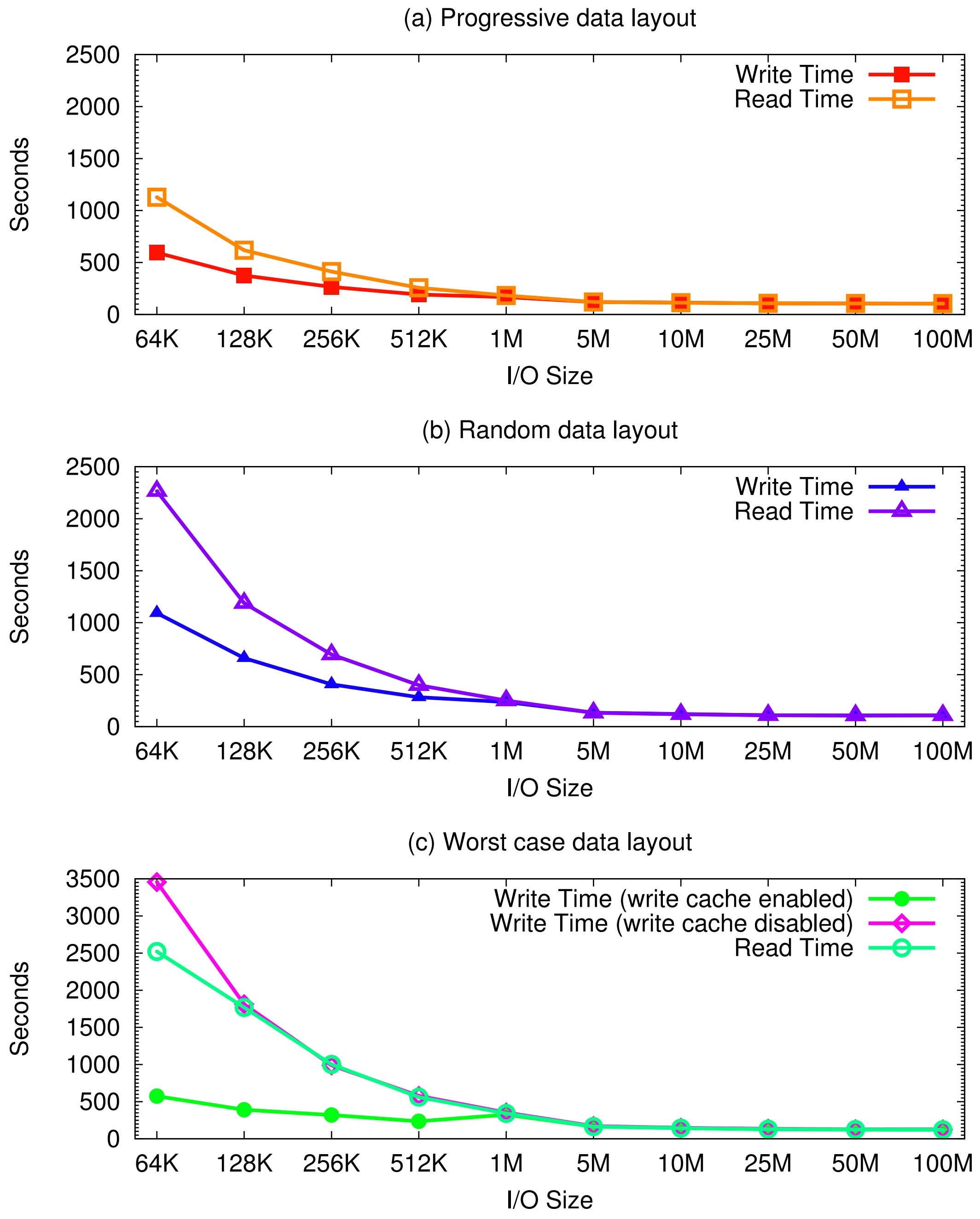


Figure 3.1: Comparison of measured times for varying I/O sizes on an example disk for ten gigabytes of distributed data. Note that the y-axis scale for part (c) is not the same as in (a) and (b) due to a much higher worst-case time.

3.1 Profiling Disk Drives

Disk drives are not all created equal. Though manufacturers provide a single product specification sheet for each model of hard drive, even individual members of the same production run often have measurably different performance[41]. This is partially due to the increased "intelligence" of the hardware which allows disks to provide a logical data view which may not correlate to the physical data layout. The obfuscation allows hard drives to transparently "fix" themselves when the storage medium is damaged by remapping portions of the logical data layout from one physical disk region to another, but this trait complicates all attempts to control the exact physical layout. Fortunately, the logical layout *largely* echoes the physical layout on current disks, and careful measurements allow one to discover the specific regions where that no longer holds true.

Hard drive profiling is necessary in order to understand the data layout well enough to make hard real-time guarantees. This allows for an understanding of, and cooperation with, the physical hardware on which the data is stored. Furthermore, each hard drive in question must be profiled individually because even those of the same model number may be dissimilar. Unfortunately, this information cannot be calculated or predicted without actual testing of the disk in question, since simulators are not well equipped to handle individual disk performance characteristics[13]. Careful testing, using various I/O sizes in multiple passes through the disk, are the only way to find an accurate profile.

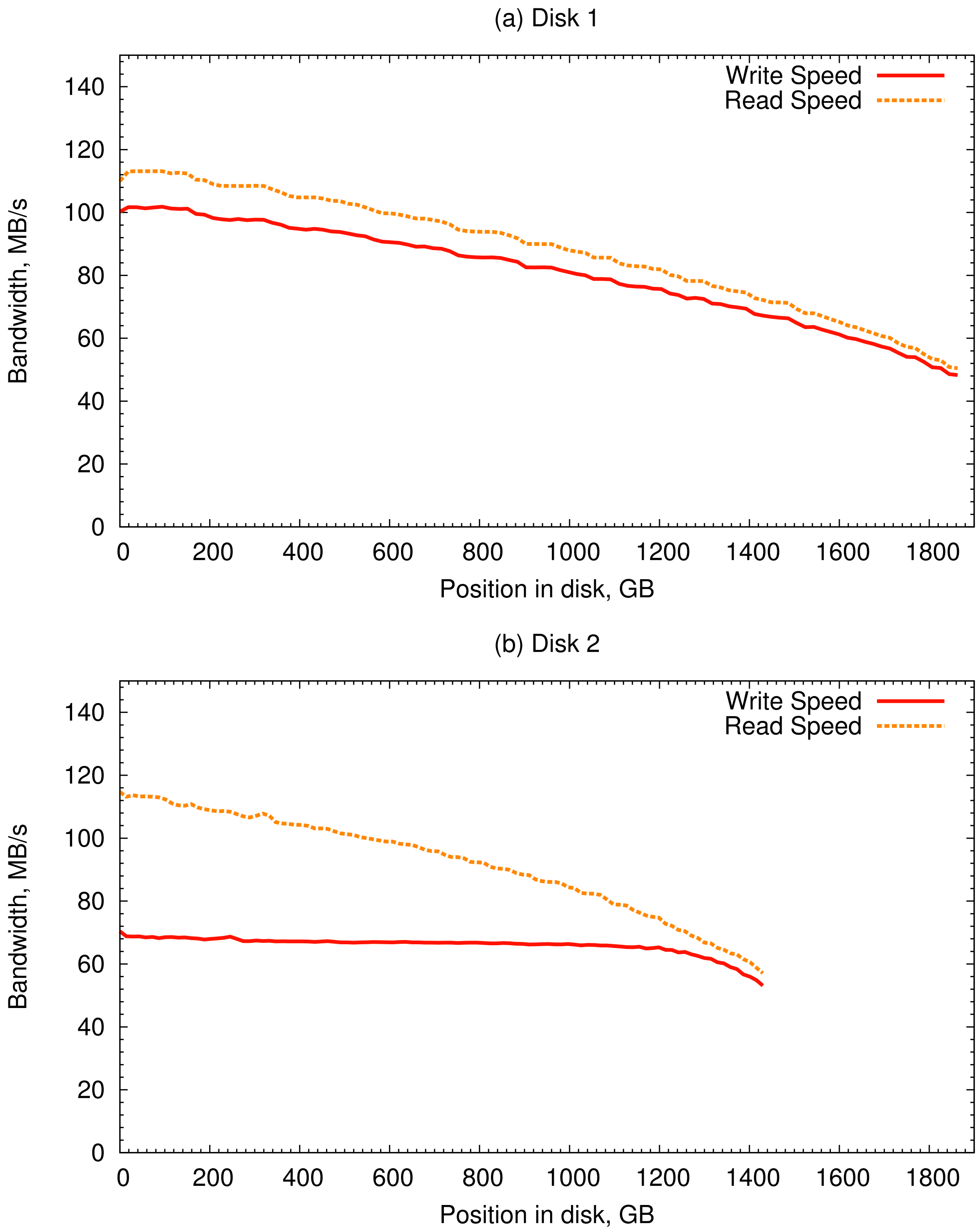


Figure 3.2: Disk performance profiling for a pair of disks. The x-axis measures the logical data position presented by the disk, and the y-axis measures the bandwidth achieved at that logical position.

Figure 3.2 illustrates the read and write bandwidth of two example disks. The logical data layout is measured along the x-axis and the bandwidth is measured on the y-axis. This graph makes it immediately apparent that disk bandwidth is highly correlated with logical data position. Bandwidth cannot be accurately predicted for a given region of the disk unless this correlation is mapped in advance. The overall shape of the bandwidth curve verifies that logical data position roughly corresponds to physical position. The "start" of the disk has a higher bandwidth, corresponding to the outermost regions of the spinning platter where the disk head passes over the data faster than in any other place. Throughout the course of the disk, the bandwidth drops off steadily, corresponding with the shrinking radius of the data track on the platter, until it reaches the "end" of the disk. It should also be noted that write and read bandwidths are not necessarily equal, and may in fact differ by a large amount, as demonstrated by the second disk in this graph. Whenever this is the case, read bandwidth is inevitably greater than write bandwidth. However, in all cases, the available bandwidth at the end of the disk is much less than that available at the start.

Figure 3.3 shows the bandwidth curves for several "identical" disks of the same make and model, each profiled with a tool created as a part of this dissertation. These particular disks are each 250 decimal gigabytes (250×10^9 bytes) in size, with average bandwidths between 40 and 60 MB/s, depending on the logical data position. Each graph, although not individually labeled, measures 250 binary gigabytes (250×2^{30} bytes) of logical data position along the x-axis, and up to 80 MB/s of bandwidth along the y-axis. Only write performance is shown in this figure; the read performance for

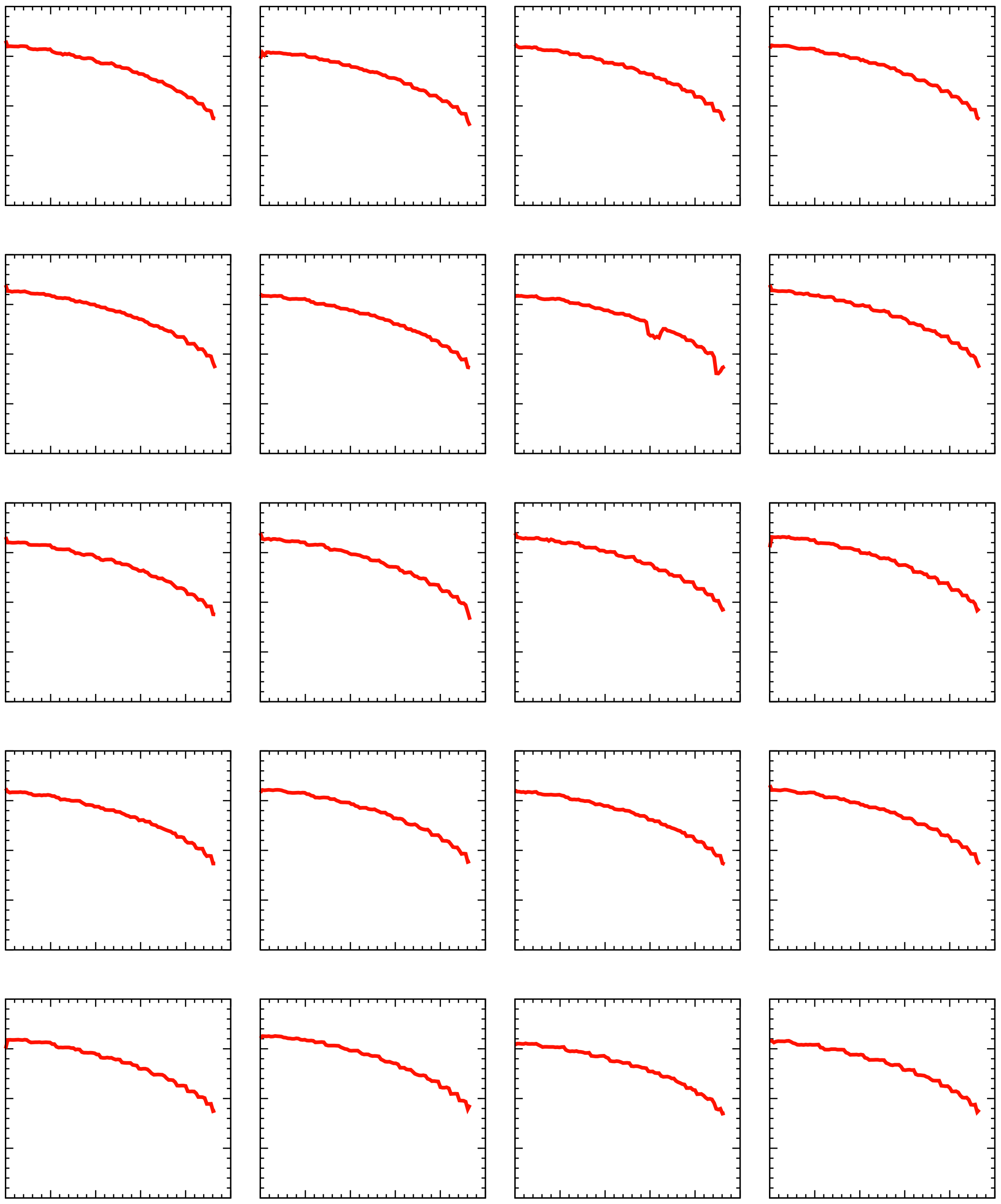


Figure 3.3: Trends in disk performance profiling. The x-axis for each graph measures logical data position from 0 to 250 binary gigabytes, and the y-axis measures bandwidth from 0 to 80 MB/s. Scale is identical on all graphs, and only write performance is shown.

each disk is similar enough to write performance such that it would not be usefully distinguishable at this scale. Note that the performance curve of each disk is of similar shape, but no two curves are exactly identical. Although the overall trend of each curve continues to verify that logical data position roughly corresponds to physical position, variations in performance are visible on several of the disks.

These results have important implications when making real-time guarantees since they show that real time does not necessarily translate to a particular amount of bandwidth. Even setting aside the issue of head positioning and assuming that any given I/O operation can begin immediately, the available bandwidth on one portion of the disk may only be two-thirds that of another portion. The implication here is that even systems capable of guaranteeing disk head time[55] cannot offer a blanket bandwidth guarantee without also having knowledge of the exact data placement on the platter.

This performance curve defines the upper limit of the guaranteed bandwidth: a worst-case assumption may be made that a particular I/O operation must be performed in the area of lowest bandwidth. Unlike many worst-case assumptions, this is actually an entirely reasonable one, since it is guaranteed to occur on any disk approaching full capacity, rather than only happening in an esoteric circumstance. This limit can also be effectively raised by only utilizing the “upper” (outermost) portion of a disk, hence raising the minimum bandwidth available to the device as a whole. No bandwidth guarantee can be made in excess of the minimum bandwidth for the disk as a whole, though certain data regions may have higher minimum bandwidth guarantees when

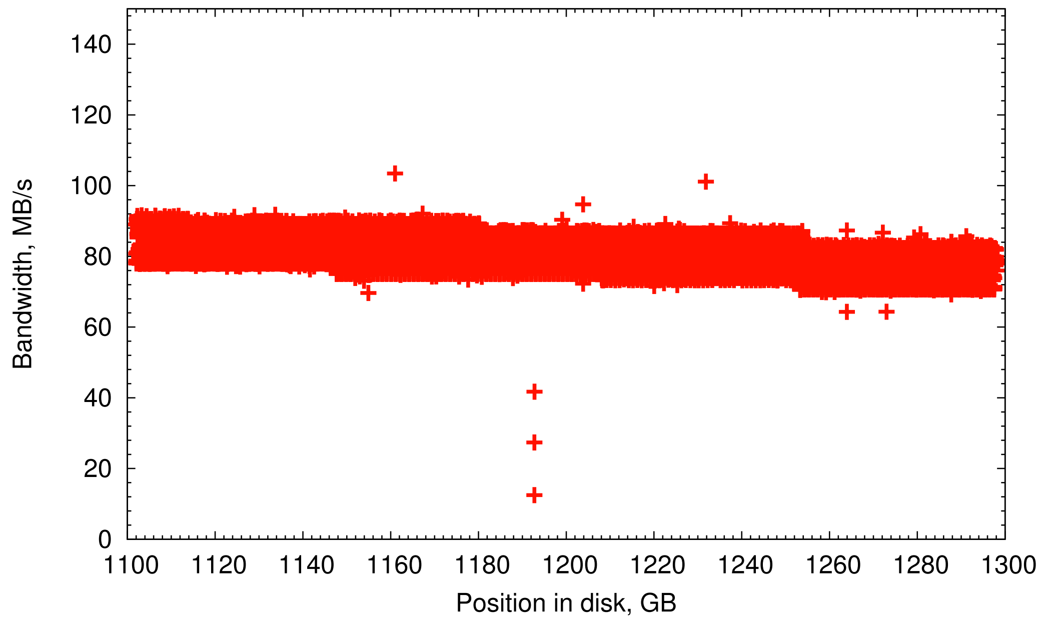


Figure 3.4: Individual performance points in disk profiling.

they are known to be in the outermost portions of the disk.

However, figures 3.2 and 3.3 only show the *smoothed* bandwidth curves. Figure 3.4 shows detail of the read bandwidth on a single disk (the same one from figure 3.2(a)) in the 1000 GB to 1400 GB range, and individual performance points are marked without a smoothed line. Inward head movements are marked by the step function appearance of the points, and nearly all of the measurements are clustered in the same region with few outliers. However, there is a significant bandwidth drop at about 1193 GB, with three significant outlier points.

The drop in bandwidth in this region most likely signifies that the physical medium is damaged at the position to which this logical mapping would ordinarily refer. The disk firmware transparently compensates for this damage by remapping the logical position to a new physical area on the disk, a solution generally beneficial for most storage purposes. This abstraction layer cannot conceal the sudden performance decrease, however, which is noticeable when one specifically looks for it.

This remapping is convenient in general, but potentially problematic when trying to make hard real-time guarantees. Luckily, it is possible to work around it by creating a counter-remapping, and thus applying a set of corrective lenses to one's understanding of the hardware. By maintaining a list of these remapped disk regions, such areas can be avoided when performing a large sequential I/O operation, and thus sudden drops in performance are avoided. It is possible to determine the physical location of the remapped disk portions (relative to logical addressing) and group them with "closer" areas, but this is difficult to do in practice, requiring a significant investment of experimentation time and a great deal of guesswork.

Two benefits are gained by making this remapping information available to the storage system. First, an efficient data layout can be created based on physical locality, if typical I/O operations are of sufficient size to gain advantage from it. Second, it allows one to accurately predict the required time on any particular I/O operation. This information does not yet translate directly into a hard real-time performance guarantee, but it is a necessary requirement to make guarantees at a performance level close to that of the hardware's capabilities.

Fortunately, most disks have few remapped regions. A selection of detailed performance regions for various disks is shown in figure 3.5. Disks are measured by

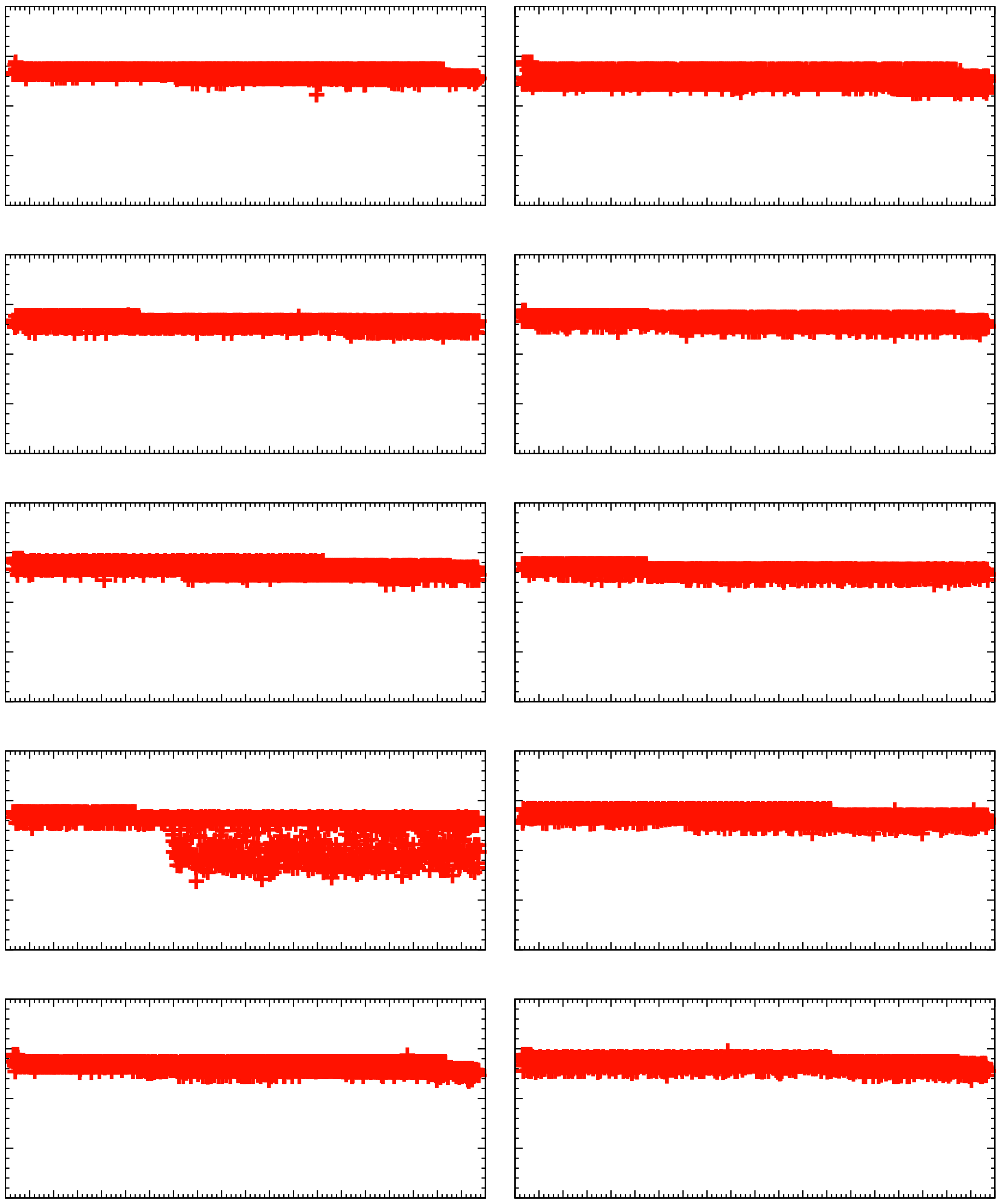


Figure 3.5: Individual performance points for multiple disk profiles. The x-axis for each graph measures the logical data position in the 140 to 160 GB range, and the y-axis measures bandwidth from 0 to 80 MB/s.

logical data position in the 140 to 160 gigabyte rate, with bandwidth in the 50-60 MB/s range (and down to 20 MB/s in one case). One disk in that figure may be seen to have a large aberration in performance, indicating a large remapped region. If such regions are common or particularly large, the disk is likely unsuitable for use in this capacity.

3.2 Data Tuning for Hardware-Aware Layouts

The dominating factor in small I/O operations is the time required to position the head over the appropriate region of the platter. If it were possible to arrange the I/O schedule such that the disk head always ended an operation in the vicinity of the start point for the next operation, there would be no problems in gaining maximum efficiency from the disk. Such systems as can arrange this data pattern see high efficiencies[26], but it is a wholly unrealistic expectation for most use cases. Even in the case of high-bandwidth streaming data, it is impossible to predict which portions of the data may be preserved in-place at some unspecified future point, or to predict what sort of queries or read requests may be made at any point. However, it is possible to do the next best thing: force the data to conform to system-enforced patterns and permit no I/O operation outside the guidance of those patterns.

In other words, the data can be packed into large monolithic chunks of uniform size and known placement. If the data access patterns cannot be made perfectly predictable, it is at least possible to force the data into a form where performance is known and real-time guarantees available. It remains impossible to control which data chunks the user wishes to access, but uniformly large data chunks ensure that seeks are infrequent compared to “normal” modes of operation for standard filesystems This format is obviously not appropriate for general-purpose filesystems, but it can be easily adapted to fit real-time streaming data. Further management aspects of that are discussed in chapter 4.

A consequence of the data chunking model is the loss of a certain amount of flexibility. There can be no such thing as a “small file” except inasmuch as multiple “small files” (or elements) can be packed into a single chunk, and this packing may result in lost storage capacity due to inefficiency. However, performance gains are significant, and more importantly, performance can be made predictable. As shown previously in figure 3.1, larger I/O operations result in lesser time requirements, and those requirements asymptotically decrease as chunk size increases. Not only do larger chunk sizes reduce the worst-case performance, but they also reduce the best-case performance. This allows I/O times to be made more predictable in general, and efficiency is maximized as a minimal amount of overhead time need be set aside to account for worst-case possibilities.

Disks may experience change over their lifespan. Most importantly, the disk may begin to degrade and portions will become unusable. Disk firmware is often capable of detecting and correcting these problems transparently without alerting the user, but such corrections almost inevitably result in remapped regions. The system could be taken offline and the disk re-profiled periodically to detect these changes, but that is fortunately not required. Instead, the same effect can be accomplished merely by keeping track of performance as the disk is in operation. With a baseline profile for comparison, it is possible to detect sudden performance changes and account for them.

Figure 3.6 is a visual depiction of the realignment process which can be undergone when degraded disk performance is detected. First, the entire remapped chunk is taken out of the normal cycle of operations. In almost all cases, most of the chunk will remain suitable for use, save only a small damaged portion that may be located at any point within the chunk. The ongoing performance measurement is capable of determining the particular chunk which has been remapped, though refinement of the exact boundaries (in terms of logical addressing) requires dedicated examination. Happily, this “hard look” only takes a fraction of a second, and does not interfere with normal operations.

Two options are possible at this point. The entire chunk may be taken out of service and no further thought given. Alternatively, since the remapped region of the chunk is likely very small, data chunks may be realigned into a new layout which bypasses the remapped region of the disk. This is a cascading process which may continue for many data cycles before propagating throughout the entire disk, but normal operation is never interrupted, and the worst effect is merely one of temporarily reduced capacity (by the amount of one data chunk). The fourth and fifth “lines” of figure 3.6 show this ongoing process, which may be further delayed by preserved chunks on the disk, but never blocked entirely.

One can never fully account for broken hardware — real-time guarantees are only as “guaranteeable” as the quality of the underlying hardware system — but this

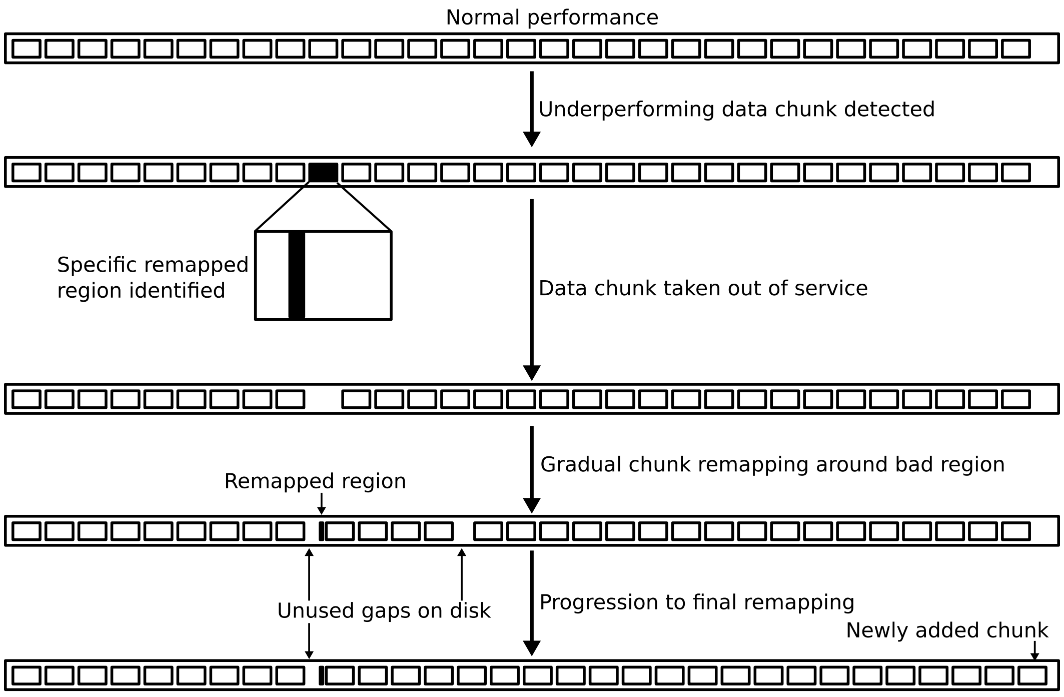

Normal performance
Underperforming data chunk detected
Specific remapped region identified
Data chunk taken out of service
Gradual chunk remapping around bad region
Remapped region
Unused gaps on disk
Progression to final remapping
Newly added chunk

Figure 3.6: Correction of remapped disk blocks in continuing operation. The remapped portion of the disk is taken out of service, and the disk chunks are gradually realigned while normal operation continues.

approach allows in-place correction of what would otherwise be potentially guarantee-breaking problems. If too many errors are detected, particularly in a short span of time, it is almost certainly a sign of imminent disk failure.

A key factor in operations is making sure that data chunks are never split. This requires all I/O requests to be funneled through a single gatekeeper so that they do not interfere with each other. Individual I/O requests may be made outside of the standard chunk size, but guarantees are only issued on the basis of chunks, and chunk alignment is maintained even in the face of smaller amounts of written data.

3.3 Solid-State Storage Devices

Solid-state drives (SSDs) offer generally higher performance than rotational disks, but are not yet developed enough to have stabilized on a general performance trend. For example, SSDs have uniform performance when operating in a 100% write or read mode, but that performance may drop significantly when operating with mixed workloads[68]. Although profiling is still possible, SSDs have a wider operational range and looser worst-case constraints as compared to rotational drives. Figure 3.7 shows the performance curves of a single SSD under 100% write and read loads. The available bandwidth does not depend upon the chunk size. Instead, it depends upon the exact workload the SSD is handling at that particular point in time. This results in SSD performance being more difficult to precisely predict than rotational drives unless the desired workload pattern is known in advance.

It is likely that SSDs and related technologies will find significant use in real-time storage applications due to their superior performance characteristics. However, there remain three significant problems which render them unsuitable for use in large-scale storage systems: capacity, cost, and wear. Commodity drive capacities for SSDs are currently in the 128-512 GB range, while mechanical disks have standard capacities of 2-3 TB. The rate of capacity increase for SSDs is currently no greater than that of mechanical drives, and thus it appears that mechanical drives will retain their advantage in this area for some time. Cost and wear were previously discussed in section 2.1, where the conclusions were that SSDs are not predicted to become cost-competitive for some

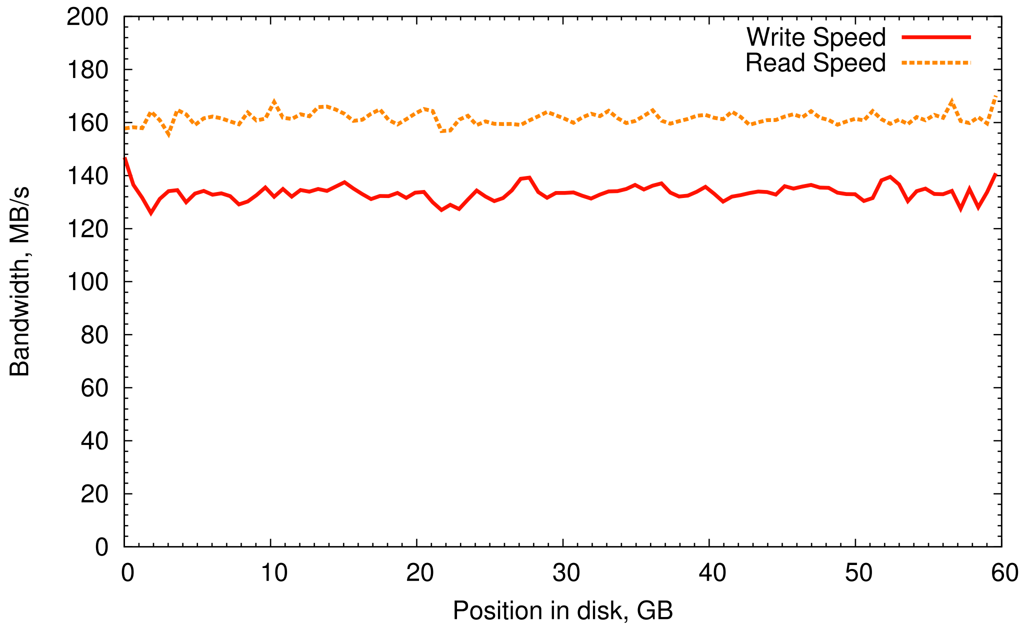


Figure 3.7: SSD performance profiling for a single device. The x-axis measures the logical data view presented by the disk and the y-axis measures the bandwidth achieved at that logical position.

time, and their write endurance makes them currently unsuitable for use in a high-bandwidth real-time streaming data storage system.

3.4 Conclusions

The performance of standard commodity rotational disk drives is unpredictable in the general sense, with three major metrics independently impacting efficiency: I/O operation size, current locale on the disk platter, and the ordering of concurrent operations. Performance may be stabilized by imposing certain restraints on these factors, with large concurrent operations yielding simultaneously the most useful (predictable) and efficient results. Furthermore, even disks which have identical performance characteristics may not behave similarly, either in aggregate, or in specific comparable regions between disks. Thus in order to accurately understand the physical characteristics of any given disk drive, profiling on an individual basis is necessary, which allows use of the disk to its maximum potential.

The results presented in this chapter demonstrate that disks may be understood in sufficient detail as to dictate an effective data layout. This allows mechanical disk drives to make hard real-time performance guarantees in terms of absolute bandwidth, at near-maximum potential, rather than the lesser guarantees of disk head time, or by requiring high worst-case assumptions. Constraints must be placed on the data itself in order to take advantage of this technique, which reduces overall flexibility somewhat. Profiling may continue during active disk operations to detect online failure, and data may be relocated appropriately to maintain performance guarantees without visible interruption to the user.

Solid-state storage devices are less volatile than rotational disk drives, but come with significant downsides in price and durability, which precludes their use as a substitute in the field of high-bandwidth real-time data management.

Chapter 4

Management of High-Volume Fleeting Data

"Write-once, read-maybe" data is not typical in storage systems. Checkpointing and backup systems make regular copies of the active data in order to ensure it is not lost through primary system failure. In those cases, data copies are maintained as a precaution against loss in the primary system, and thus may be appropriately classified as "write-once, read-maybe." However, the intent of such a system is to maintain the data for failure recovery purposes, rather than to interact with it on a regular basis. It does not have constant real-time requirements — except in the broader sense of making backup copies frequently enough to be useful — and is only overwritten with more up-to-date copies of itself. In contrast, the problem space of managing high-bandwidth real-time streaming data requires the erasure of old data in order to store new, and most of the data will leave the system before it is ever referenced. It is possible to indefinitely preserve a smaller subset of the data from time to time, but it is infeasible to preserve all of it. The operational state of this type of storage system may be described thusly:

1. Current data is preserved for a limited time only. The extent of that time depends on total system capacity and system policy configuration.

2. Current data may be specifically marked for preservation, with the understanding that such preservation will reduce the total available capacity.

3. Old data is automatically overwritten in a first-in, first-out manner as new data enters the system.

4. Incoming data must be preserved immediately; it is generated in real-time and is lost if it cannot be immediately collected.

This manner of data collection describes a system commonly known as a "ring buffer," though the standard model does not usually incorporate in-place preservation of existing data. A visual representation of the ring buffer model is pictured in figure 4.1. Two blocks of data are preserved in place, and the replacement cycle of incoming data moves clockwise. The oldest region of data currently in the buffer is noted, and newer data is marked in progressively lighter shades.

Ring buffers are simple data structures, and have previously been used to manage low-rate data flows[10, 71, 58, 57, 15]. With suitable adaptations, the ring buffer model may also be used to manage a high-bandwidth stream of real-time data.

Gathering information is easier than understanding it, or using it, and the difficulties in storing information long enough for it to be useful are not new. Scientific

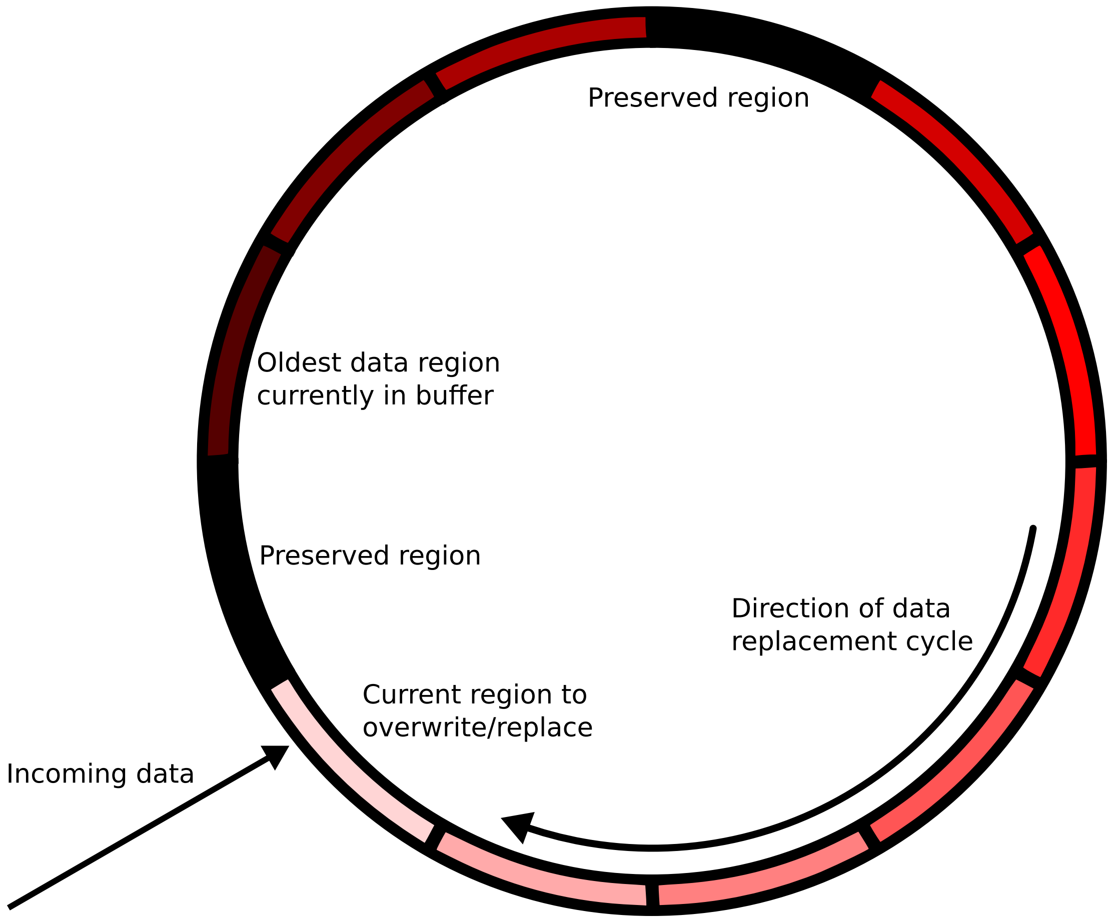


Figure 4.1: Ring buffer model.

data capture in particular has often had this problem: it is easier to construct the data recording instruments than it is to store all the data that such instruments can collect. The Large Hadron Collider, perhaps the current epitome of “Big Science” was previously mentioned as having a continuous data collection rate of about 300 MB/s[30]. That data rate is stepped down three orders of magnitude from what the instruments actually collect: 300 GB/s, which must be filtered down to only the most “interesting” events. It is not difficult to imagine events worth capturing which must be filtered out before reaching the storage system and are thus “lost.” New generations of experiments attempt to capture more of this data in order to lessen this possibility.

Understanding the data is the first step in managing it, and thus a storage system must “understand” the data as more than just a stream of bytes or a collection of files. Such abstraction is useful and proper for general-purpose storage systems, but this work targets a more specific area that can benefit from the tight integration of data with storage. Thus, the specific mechanics of the ring buffer storage model are highly dependent on the type of data.

4.1 Data Types

Three primary characteristics distinguish incoming real-time high-bandwidth data for the purposes of storage and management: data rate, element size, and indexing complexity. The data rate is measured by bandwidth (bytes per second), and may be variable. Data elements are treated as indivisible units for the purpose of management, and may also vary in size. Example data elements would be a single photograph, a sensor reading, or an IP packet. The indexing complexity refers to the number of aspects on which each data element is indexed. At the least, each data element is indexed by timestamp. At the most, the indexing metadata might exceed the size of the data itself. Two real-world data patterns may be regarded as “canonical” examples:

1. **Fixed-size, non-indexed data:** Fixed-size, non-indexed data is often generated by sensor systems. It arrives at a fixed rate, does not vary in timing or content layout, and does not need to be indexed beyond time, and possibly source. Searching for and preserving such data can be done merely by specifying its space-time coordinates.

2. **Variable-size, highly-indexed data:** Variable-size, highly-indexed data is the type generated (for example) through network traffic monitoring. Its arrival rate and exact data element sizes cannot be precisely specified in advance, and individual elements are subject to multiple indexing requirements based on the data contents. Any searching and preservation is likely to be requested on the basis of a database-style query over complex indexing data.

Large fixed-size data elements are easiest to manage, whereas small variable-sized elements indexed on multiple vectors pose additional difficulties. Other combinations are possible, but these two examples represent the easiest and hardest ends of the scale. If a storage system has a native view of these data patterns and requirements, it becomes possible to reach higher performance levels. Instead of a faceless amalgamation of random bytes with no more than a directory/file structure to group them by, the data is understood in its native form, and the storage system can adjust its access and I/O patterns accordingly. The differences may be better understood by a closer study of two example data patterns: continuously streaming sensor data, and variable-rate indexed network traffic.

4.1.1 Continuously Streaming Sensor Data

Continuously streaming sensor data is predictable and unvarying. That is, the data itself is constantly changing, but there are no changes in its size, layout, or rate. Such parameters are set in advance, and the indexing complexity of this data is extremely limited, requiring perhaps only a source location and a timestamp. The nature of this data makes it very easy to align with any internal formatting requirements imposed by the physical storage medium. The equivalent of a "file boundary" is deterministic, as is the equivalent of metadata.

When an external process decides that some portion of the data is interesting and that it should be preserved, it need only tell the storage system something such as "preserve data from source A, where the timestamp is between X and Y." The data is marked as preserved; the ring buffer ordering is (logically) rearranged to bypass the newly-preserved data, and operation continues normally with a known loss in total buffering time and a changed determination of when old data is now due for expiration.

4.1.1.1 Variable-Rate Indexed Network Traffic

Variable-rate indexed network traffic is unpredictable in several respects. The previously-described continuously streaming sensor data has a fixed size, a fixed rate, and a relatively simple indexing requirement. None of this applies to network traffic, where "variable" can be used to describe all the major parameters. Consider, for example, a standard IPv4 packet. It can vary in size from 20 to 65535 bytes (though 1500 is the more common upper bound), it has no set rate, and there are several possible ways to index each packet, even aside from any indexing based on the data itself.

Continuously streaming sensor data can be indexed based on the timestamp, and perhaps the source of the data packet if that is not already part of the data layout. An IP packet, however, has several unique aspects which may need to be indexed and tracked in order to make use of the data in a timely fashion. A partial list of such aspects might include the source IP address, the destination IP address, the protocol, the packet size, and the time index. There might additionally be a need for indexing based on aspects of the data contents, or other metadata-like information provided by an external process, which should be stored with the raw data for later searching purposes.

These extra indexing requirements add a new layer of complexity when it comes to properly managing the data, particularly for preservation or retrieval requests to the storage system. Instead of a request to preserve data based on a timestamp range, a request may take the form of "preserve IP packets which are traveling between source address A and destination address B, have a length of at least N bytes, and were sent at time X or later." The storage system must be able to act on such a request in a short amount of time, or the data that it has been instructed to preserve may be overwritten before it can be found. This aspect of indexing is further discussed in Chapter 5.

4.2 Data Chunking and Layout

Individual data elements may be large or small, but in either case their effective management depends on tight integration with the physical hardware. Chapter 3 discussed the sensitivity of mechanical disk drives to the physical layout of data, and the methods by which one may gain control over such placement. This knowledge is now combined with the formatting of data in order to make effective quality of service assurances and hard performance guarantees. Modern filesystems, such as ext4, are generally good at data placement with regard to physical locality on a disk[14]. However, they are prone to fragmentation over time, particularly as the capacity approaches full. Since a utilization rate of near-maximum on each individual drive may be reasonably anticipated, ordinary filesystem fragmentation would increasingly accumulate if one followed the standard practice. Thus data layout must be precisely controlled to avoid the problems of ordinary filesystems. Two terms are now defined:

- A **Data Element** is a single piece of data which should be treated as an indivisible unit. For example, a data element might be a single snapshot from an optical telescope, or a single IP packet intercepted from network monitoring.

- A **Data Chunk** is a unit that the storage system interacts with. It may contain a single large data element or many smaller ones. It may also contain other metadata-like information related to the datastream.

As per the results in table 2.1 and figure 3.1, the minimum data chunk size must be on the order of megabytes, and no ordinary I/O operation may be smaller than this chunk size. The data chunk size is customizable based on the exact nature of the data, but the general rule of thumb is that “bigger is better” from the perspective of the hardware. Large data chunks allow performance guarantees closer to the maximum performance of the hardware. Small data chunk I/O times are dominated by the positioning of the disk head, which is the greatest factor in the worst-case timing assumptions necessary for real-time guarantees. Large data chunk I/O times are dominated by the actual data transfer time, which, although not entirely constant, is far less variable than disk head positioning time and allows for tighter worst-case assumptions.

When data elements are small, multiple elements can be packed into a single data chunk. For example, nearly thirty-five thousand 1500-byte IP packets (data elements) can fit into a single 50-MB data chunk. Though useful for bandwidth purposes, this has two unfortunate effects. First, flexibility is lost: elements can only be stored, preserved, and expired in large chunks. Second, depending on the exact chosen layout, some storage capacity can be lost through inefficient packing, a concept sometimes referred to as “internal fragmentation” in standard filesystems. This latter effect can often be minimized through careful selection of the data chunk size. If the data element size is n bytes, for example, it would be particularly unwise to pick a data chunk size of $2n - 1$ bytes.

In practice, a certain amount of each data chunk must be reserved in order to describe the internal chunk layout of the data elements. If the data elements are of constant size, or if their internal structure is well-defined, this extra bookkeeping information is minimal. When data elements are small and arbitrarily defined, it may become necessary to reserve a higher percentage of the capacity for this layout data.

In the same way that elements are packed and tracked into data chunks, data chunks themselves must somehow be packed and tracked on the storage hardware. Standard filesystems store their indexing information on the disk for two main reasons. First, it is inconvenient at best to hold an entire filesystem index structure in active memory at all times. It may even be impossible to do so, depending on the size of the disk, the number of files, and the available memory. The second reason is by far the more important: in the event of a system shutdown, it is both easier and faster to read back indexing information from the disk itself, than it is to traverse it in its entirety to reconstruct the structure file by file.

Naturally, maintaining indexing information on the disk itself carries some penalty, since a portion of the bandwidth must be used to write updated indexing information to the disk for each relevant I/O. This reduces the amount of bandwidth available for actual data, and requires frequent repositioning of the disk head, since the relevant file structure information is not necessarily adjacent to the data itself. Most modern filesystems take data locality into account and the performance penalty is *usually* not overly burdensome. However, every extra megabyte counts in an attempt to make real-time guarantees close to maximum hardware capability, and relatively small changes in data placement can lead to large losses of bandwidth as the disk is forced into making multiple extra seeks per I/O operation.

It is possible to gain substantial advantage in this area by use of a uniform disk chunk size, a side effect of which is that each chunk is deterministically placed, both with regard to the file system equivalent data layout and the physical media itself. The only information required to understand the entire physical disk layout is: the chunk size, the total number of chunks, disk remapping information (see section 3.1), and a starting point on the disk itself. This information may be thought of as akin to the superblock in a standard filesystem. It is the only disk structure information which must be stored on the device itself, and only for the purpose of reinitialization after a shutdown. This indexing information need only be stored once at the commencement of operations, rather than continuously being updated as the system works.

The implication of this approach is that the entirety of the disk structure may be held in main memory and need never be committed to disk. The deterministic placement of chunks ensure that the position of any given chunk can be calculated upon demand, rather than looked up, whether stored in memory or on the disk itself. Therefore this approach gains measurable advantage by writing only data to disk, rather than constantly re-updating an on-disk index and metadata structure, and there is no disadvantage to not maintaining the otherwise standard structure.

Regular startup takes longer than as in a normal filesystem, but this model is intended to work in an environment where shutdowns are not expected. Any startup after initial configuration is likely going to be crash recovery, which requires extra at-

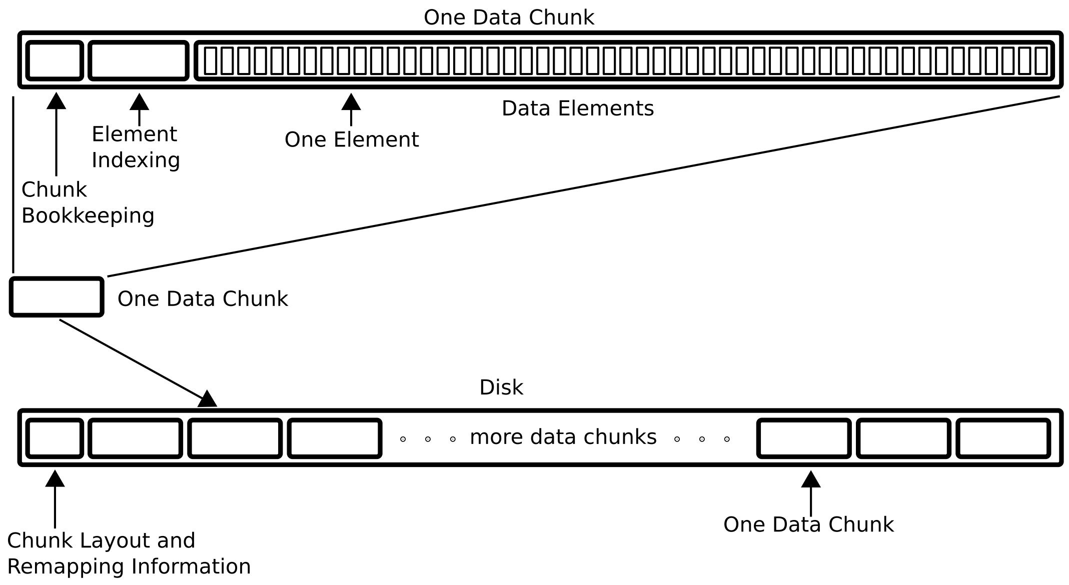


Figure 4.2: Data chunking model. Data elements are packed into data chunks along with their internal bookkeeping and indexing information. Data chunks are arranged on the storage device in a similar manner. The relative size of the aspects in this diagram are not to scale.

tention to the data layout in any filesystem. The exact time for startup depends on the chunk size, since the disk must touch each chunk on disk to ascertain its exact status. Figure 4.2 shows the data chunking model described in this section.

4.3 Mahanaxar Data Management System

The initial testing and evaluation for these data management techniques involved the creation of a prototype model named "Mahanaxar," which was developed and implemented in Linux for testing and evaluation. Mahanaxar is a multithreaded process which runs in userspace and accesses disk drives as raw devices. Multiple processes may run on the same machine, one process per disk, and all access to a given disk must take place through the associated process in order to manage bandwidth. Each process/disk is governed by a configuration file that specifies, among other information, the chunk size, the element size (or size range), and basic indexing aspects. It is possible to restore state after a controlled or uncontrolled shutdown by scanning the disk and re-initializing the in-memory index.

The system architecture for a single Mahanaxar process is shown in figure 4.3. All components of Mahanaxar are enclosed by the dashed line, and an external data source and outside process are also depicted. Raw data is generated from the source and passed into the processing thread, which handles all aspects of "data arrangement." It is responsible for assembling all the individual data elements into a single data chunk (see figure 4.2) and collecting appropriate indexing information. Indexing aspects are sent to the indexing module, which assigns and keeps track of element/chunk assignment. The external communication aspect refers to coordination of reliability and search aspects; see chapter 6 for further details.

Full data chunks are passed off to the I/O manager, which controls all access to the disk. The Mahanaxar model works on a simple priority scheme: if there is a chunk of data which needs to be written to disk, it has first priority and is written as soon as possible. Reads are only performed when no chunks need be written. As long as the chunk size is sufficient (see section 4.4) and the data rate does not exceed the capacity of the physical disk, this system guarantees that real-time deadlines are met.

Outside processes may request chunk reads or preservation, either in the form of an explicit chunk reference, or in the form of a query. In the case of preservation, the

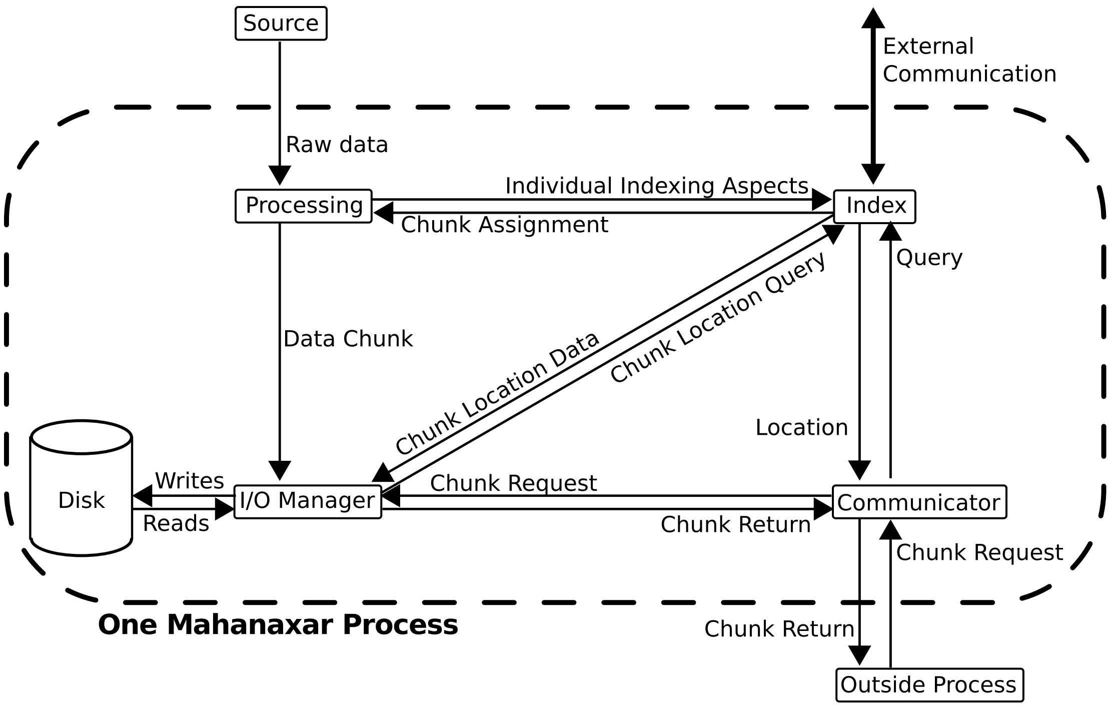


Figure 4.3: Mahanaxar storage model for one process.

ring buffer is automatically shortened, and the data preservation range adjusted. When an outside process requests the data itself, the request is accommodated insofar as it is possible to do so while still meeting real-time deadlines for incoming data. Depending on the physical location of the data chunk, and how close the drive is operating to actual capacity, it may take some time to complete a read operation. Naturally, by policy configuration, the system may set aside a dedicated portion of the bandwidth for reading only.

This method produces a jagged access pattern, particularly for reads, as a chunk read request may be delayed for some time until the system has time to schedule it, and then it is delivered all at once. This is an unfortunate but necessary effect of making quality of service guarantees and maximizing bandwidth, since the system must not fritter away disk head time by seeking back and forth over the disk in pursuit of a smoother curve on the bandwidth graph. Long-term trends average the read bandwidth into smooth curves, but priority must be given to the incoming data which cannot be regenerated if lost. It is possible to provide real-time guarantees to both writing and reading aspects by lowering requirements for absolute bandwidth capability, if such behavior is desirable.

As data elements arrive into Mahanaxar, they are indexed and placed into chunks. If element sizes are large, one element may be equal to one chunk. If element sizes are small, hundreds or thousands of data elements may be placed into a single chunk. The default indexing model is based entirely around chunks and timestamps, and secondary indexing components may be stored inside a data chunk when necessary.

This simple model is sufficient for initial capability testing of the system, but a more detailed model is required to handle complex data, which is discussed in Chapter 5.

4.4 Performance Evaluation

Tests of the Mahanaxar system were primarily designed to demonstrate absolute performance, maximum exploitation of hardware, quality of service, the ability to meet real-time deadlines, and a comparison of chunk size on overall performance. All tests were performed on multiple disks and multiple machines in order to fully demonstrate the validity of these management techniques, but comparative results presented in this section are based on single disks. Because no existing purpose-built systems are intended to handle continuously streaming high-bandwidth real-time data, comparisons are shown against a system built on the standard filesystem model. The comparison system, like Mahanaxar, was implemented and tested in Linux.

4.4.1 Comparison Systems and Testing Procedure

The principle alternative approach for handling non-indexed data elements is a general-purpose filesystem which uses flat files to store the data. Multiple data elements may be bundled into single files to improve performance; the results of this approach may be seen with larger element sizes. Multiple general-purpose single-disk filesystems were tested in order to determine which among them had the best relative performance for this mode of operation, with the ext2 filesystem coming out best. Modern filesystems often use data journaling techniques, trading bandwidth for data consistency, which is a useful and valid technique in a general-purpose filesystem, but undesirable for the type of data modeled here. Even when journaling is disabled for modern filesystems, performance is not any better than ext2.

Standard filesystems often use write caching to insulate applications from the physical hardware, and improve performance. This technique allows many gigabytes of data to be "written" to disk almost instantaneously, when in reality it is only cached in the RAM and is trickled out to disk at the slower disk bandwidth rate. Data may not reach the disk for many seconds in this case. Obviously this is not an acceptable method in an environment where incoming data arrives at a constant rate, and any write caching in excess of actual disk bandwidth cannot be maintained over the long term.

However, write caching serves the secondary purpose of allowing the filesystem to internally rearrange its writes for maximum performance. Thus, it should not be disabled entirely in the standard filesystem model. In the experiments presented here, the filesystem was only explicitly synchronized to disk every several seconds. In contrast, Mahanaxar explicitly synchronizes every data chunk to disk and guarantees that data is successfully stored when the operation is reported complete.

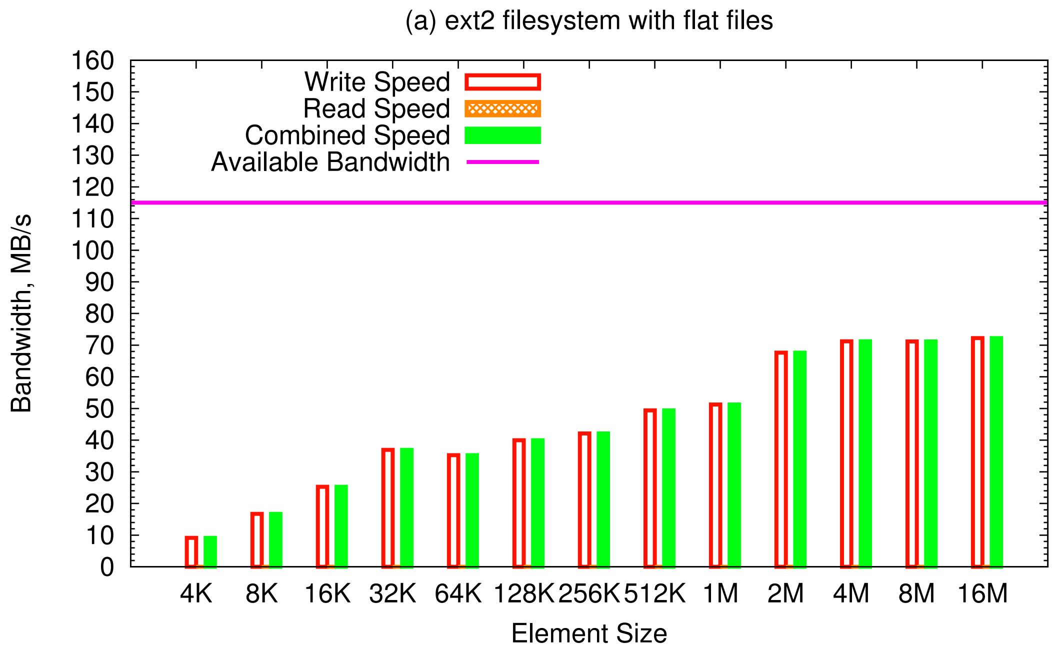
4.4.2 Performance

As previously described in section 2.1, many filesystems use only 50-80% of the disk's supportable bandwidth[46], and may use significantly less with non-optimal storage layouts. This loss is especially serious when storage performance is the limiting factor of the system. The Mahanaxar model allows disks to be used at near-maximum capability, whereas the ext2 filesystem model has performance roughly between 8% and 65% of peak drive capabilities.

Figure 4.4 shows a performance comparison between ext2 and Mahanaxar for varying element sizes between four kilobytes and sixteen megabytes. The results presented in this graph are from a small region of the disk with an average read speed of about 115 MB/s, which is shown as a solid line. As shown in section 3.1, the average bandwidth of each disk varies depending on region, and thus this particular test was isolated to a very small region of the disk (about 10 gigabytes) in order to minimize bandwidth variation, and the entire region was used for each element size so that the same amount of I/O was scheduled in each case. Furthermore, this particular disk was chosen to have similar write and read bandwidths in order to best measure and compare total combined performance. Requested write bandwidth was maintained at 100 MB/s, while read bandwidth was set to "best available." Mahanaxar chunk size was set to 80 MB/s.

Performance of ext2 is shown in figure 4.4(a). At no element size did performance reach the requested 100 MB/s write speed, and thus the read speed during the test was zero in all cases. Bandwidth was slightly less than 10 MB/s when element sizes were set to 4KB. Performance generally increased with increasing element sizes, until it starts to level at 4MB and above. Element sizes greater than 4MB show minimal additional benefit. Thus, performance is limited to 70-80 MB/s total bandwidth.

Mahanaxar's performance is shown in figure 4.4(b). The requested 100 MB/s



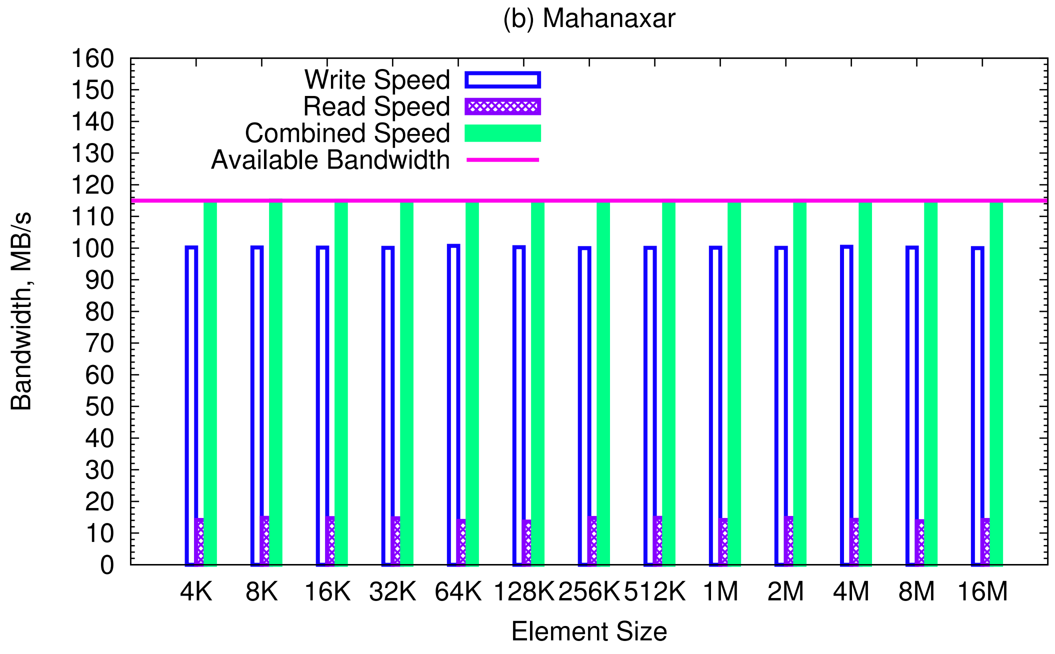


Figure 4.4: Elementary performance comparison between ext2 and Mahanaxar, with varying element sizes.

write speed is easily met for each element size, allowing up to 15 MB/s of spare bandwidth for reading. In each case, the measured read bandwidth is around 14 MB/s (the average over all element sizes is 14.35 MB/s), thus leading to total bandwidth utilization of over 99% of what the disk supports. Mahanaxar is thus shown to have far superior performance compared to ext2.

If element sizes are variable, rather than static, ext2 performance decreases significantly. Elements cannot overwrite each other "in place" on the disk, and thus become excessively fragmented. When the ext2 system is run with variable element sizes, performance never stabilizes to the point where average bandwidth is reliable for predicting future performance.

4.4.3 Quality of Service Ability

Performance results in figure 4.4 show that an ext2 flat file-based system can maintain 70 MB/s of write bandwidth when element size is 4MB or greater. At that rate, for ext2, a few MB/s of bandwidth is theoretically left over for read performance. Accordingly, for the purpose of demonstrating quality of service ability in the next set of tests, the requested write bandwidth is set at 70 MB/s which thus allows ext2 the capability to fully meet the desired write speed.

Figure 4.5 shows the performance of ext2 and Mahanaxar in an environment of increasing read requests. The x-axis in these graphs measures the requested read bandwidth from an external user. Again, these experiments were run on a small region of the disk in order to minimize the impact of variable maximum bandwidths on the average numbers. Mahanaxar chunk size is 80 MB, and element size for both systems is set at 5 MB.

The performance of ext2 is seen in figure 4.5(a). At 0 MB/s requested read speed, write performance is the requested 70 MB/s. At 2 MB/s requested read speed, write performance drops to about 66 MB/s, and read performance is 2 MB/s. Total performance thus drops to around 68 MB/s, which is below the requested write speed alone, and even further below the requested combined bandwidth. Disk performance is theoretically capable of supporting 115 MB/s of total bandwidth, which translates to 70 MB/s write and 45 MB/s read. However, the total performance of ext2 never again reaches 70 MB/s, let alone surpasses it.

In contrast, as shown in figure 4.5(b), Mahanaxar maintains a steady 70 MB/s write speed as requested. Read bandwidth also increases as requested, up to around 44 MB/s, where it levels off at nearly the maximum achievable rate of the disk itself. Total combined performance thus tracks the desired/available bandwidth, showing that Mahanaxar can provide the requested read speed up to the physical limitations of the drive itself, without failing to meet the real-time deadlines of the incoming data.

The reason for this disparity, aside from the performance aspects previously discussed, is that ext2 (and most other file systems) allocates performance “fairly” rather than for maximum efficiency. Although ext2 is fully capable of maintaining a 70 MB/s write speed and meet low levels of read bandwidth, the “fair” allocation paradoxically results in a lesser combined bandwidth and a violation of needed performance guarantees for real-time incoming data.

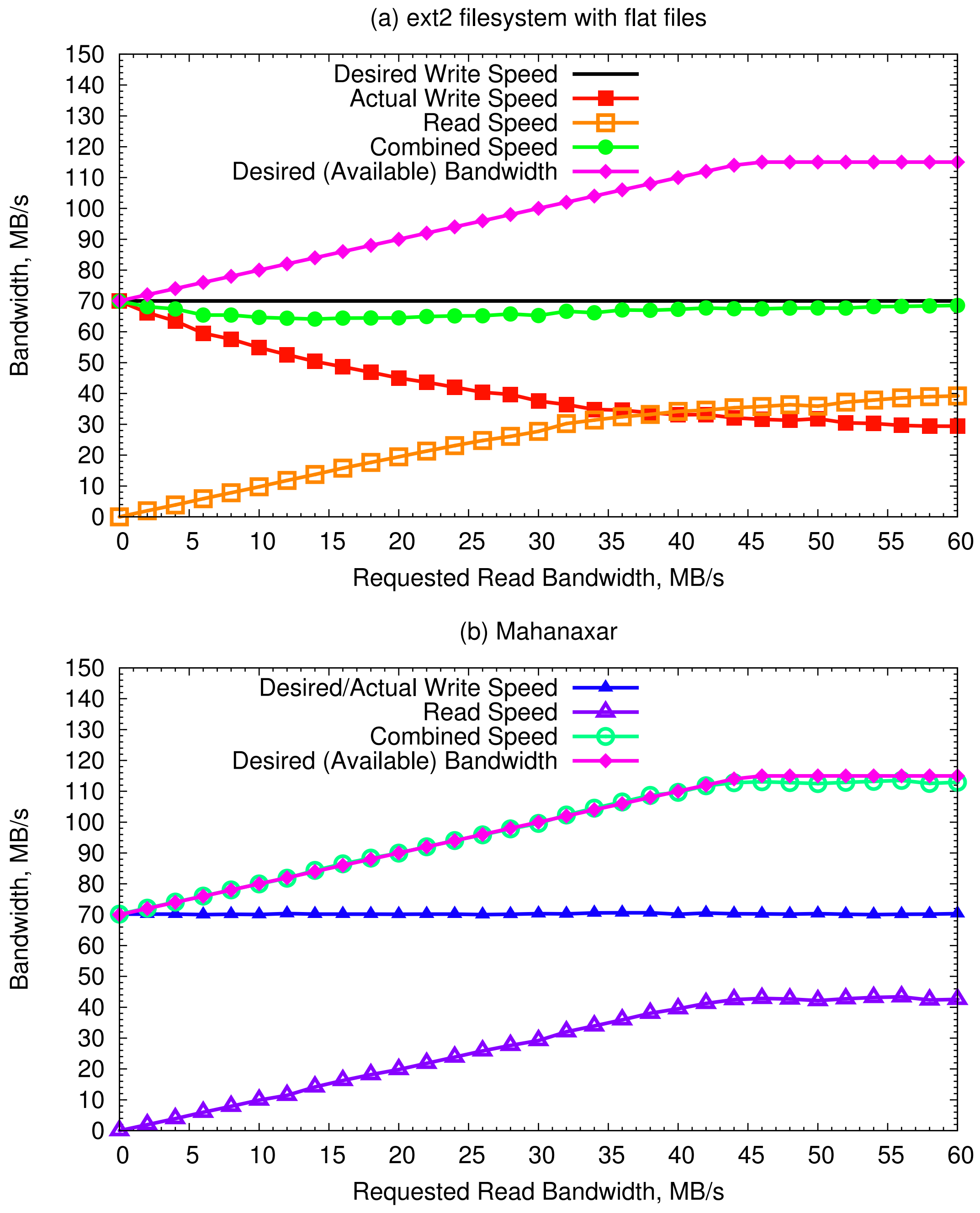


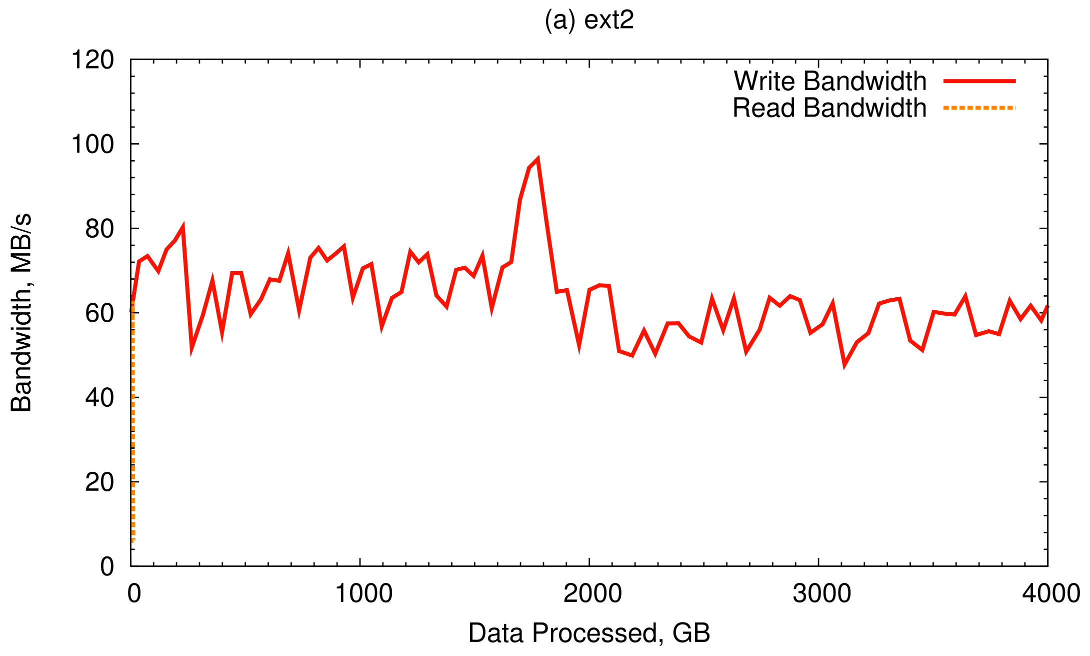
Figure 4.5: Quality of service comparison between an ext2 system and Mahanaxar. The requested read speed is 70 MB/s, the disk's maximum bandwidth is about 115 MB/s, and element size is 5 MB.

4.4.4 Operational Comparison

Figures 4.4 and 4.5 show the average figures for overall performance and quality of service abilities. These measurements were collected over only a small disk region in order to minimize the distorting effect of the whole-disk performance curve. Use of the entire disk would have rendered these average figures less reliable. However, actual system operation must take place over an entire disk, and the performance differences between regions play an important part. Therefore, any operational comparison cannot be restricted to only small regions of the disk.

Figure 4.6 demonstrates the whole-disk performance of ext2 and Mahanaxar. Data cycles were constrained to about 1.5 TB, in comparison to the 1.8 TB size of the disk. That is, the system expectation was to hold the most recent 1.5 TB worth of data. This reduced capacity was chosen in order to avoid use of the innermost portion of the disk platter, which has the lowest bandwidth (see section 3.1), and thus offer a higher overall capacity. The entire drive was made available for use in ext2 after discovery that restricted partitioning of the disk actually led to worse overall performance.

Write bandwidth was set at 80 MB/s over the entire course of the disk, which was theoretically supportable by the ext2 system in this particular case and disk. This leaves significant extra capacity available for read bandwidth (whether such is desired or not). The exact amount of read bandwidth depends on the position of the disk in the data cycle, and the exact location of the data on the platter, and varies from an average of around 20 MB/s to almost 40 MB/s Element size is set to a static 5 MB,



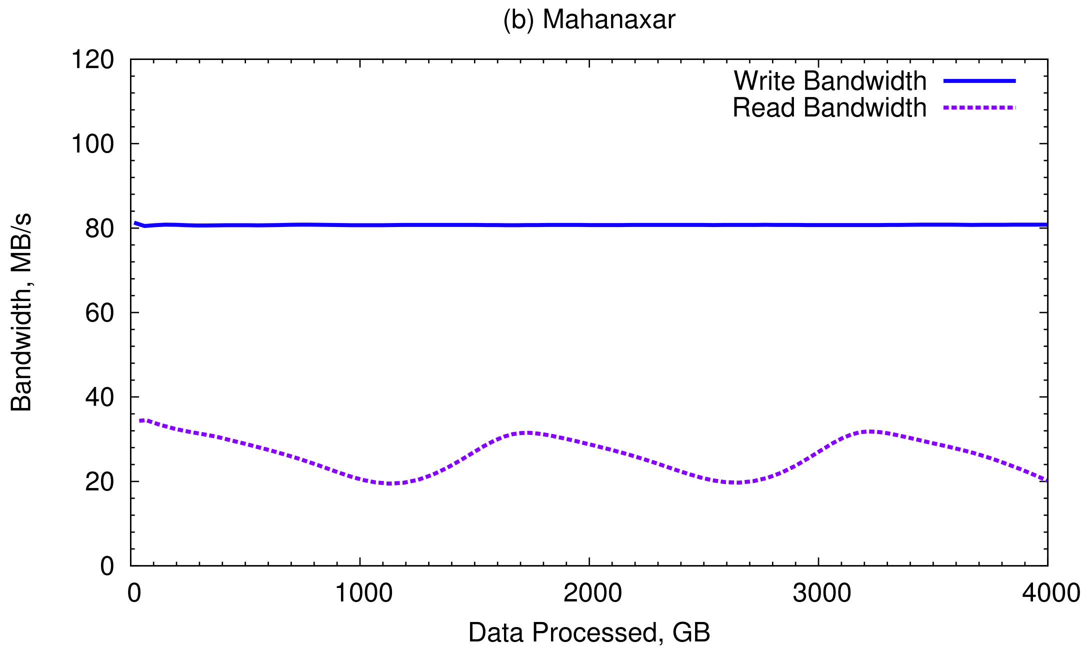


Figure 4.6: Basic operational comparison between ext2 and Mahanaxar. The disk used for these results is about 1.5 TB in size and thus the processed data spans the entire disk platter over the course of one data cycle. Element size is 5 MB.

and Mahanaxar uses an 80 MB chunk size, conveniently corresponding to exactly one second's worth of data.

Figure 4.6(a) shows the unsuitability of ext2 for this data type. Although the raw disk ability can easily support 80 MB/s of write bandwidth, the filesystem is unable to keep up. Instead, it has an extremely jagged performance curve, usually spiking between 60 and 70 MB/s bandwidth, and once spiking above 80 MB/s in a particularly favorable region, but never stabilizing at the rate it needs. Furthermore, as the second cycle of data enters the system (thus requiring old data to be deleted/overwritten), performance drops below the first data cycle. During the course of the test, the general-purpose filesystem lost about 26.1% of the total data. Since writes were prioritized, reads were extremely rare, and read bandwidth remained zero for nearly the entire test (only 650 MB were read in total at various points during the course of the test).

In contrast, figure 4.6(b) show stable performance with Mahanaxar. Write bandwidth stays steady at 80 MB/s, and the read bandwidth follows a sinusoidal/sawtooth pattern in accordance with the disk head's movement from the outermost to the innermost regions of the disk platter. The read requests made here are each made from random portions of the disk; performance would be slightly higher if read requests happened to take place "close" to the disk head's write location, but that would be an ideal (and unreasonable to expect) mode of operation. The system is capable of continuing this mode of operation indefinitely, precisely as required.

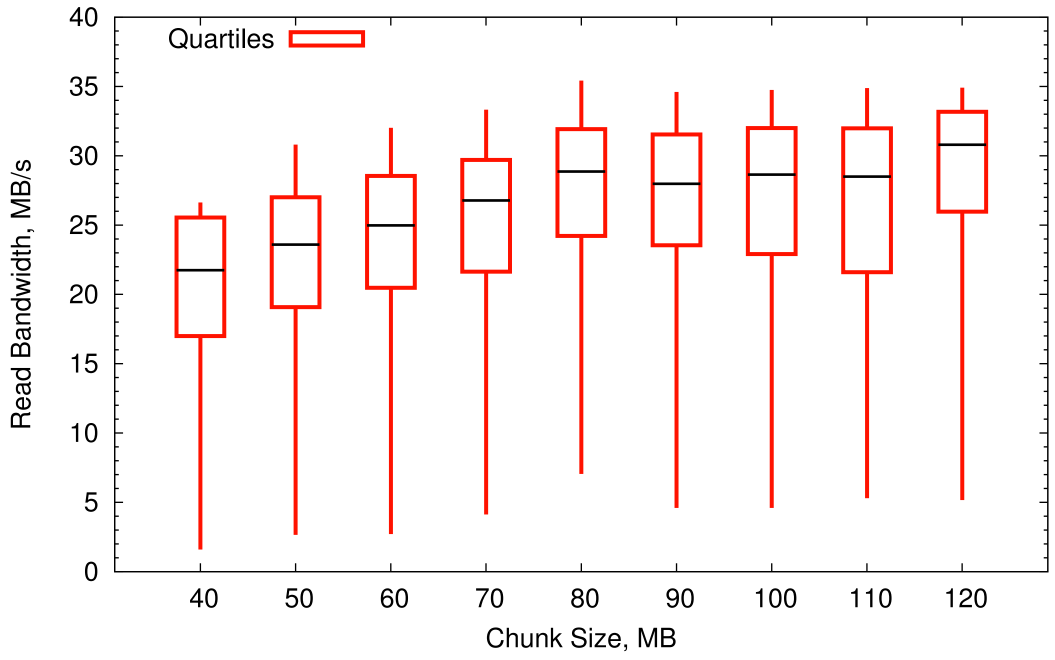


Figure 4.7: Chunk size relationship to read bandwidth comparison. Measurements are in quartiles, by the available read bandwidth when attempting data reads over the course of a data cycle.

4.4.5 Effects of Chunk Size

Mahanaxar depends on a large data chunk size for efficient operation. The minimum size of each chunk must be determined on a disk-by-disk basis if such is desired, but performance is better with larger data chunks as discussed in section 4.2. The relationship between chunk size and available read bandwidth is shown in figure 4.7.

Write bandwidth is a constant 80 MB/s for all chunk sizes, and is not pictured on this graph. Chunk sizes of 30 MB or less occasionally result in the write bandwidth dropping below 80 MB/s in significantly aged systems, and thus data may be potentially lost. Chunk sizes of 40 MB or more never lose data, and the available read bandwidth gradually increases. At about 80 MB chunk sizes, average read bandwidth stabilizes and ceases to increase. As a general rule of thumb, a chunk size equivalent to $\frac{1}{2}$ of a second's worth of bandwidth is a safe and reasonable size, but larger chunk sizes up to about 1 second's worth of bandwidth results in additional read performance.

4.5 Conclusions

It is possible to make performance guarantees and meet quality of service requirements for incoming real-time data by closely integrating data management techniques with the underlying physical hardware. Moreover, this is possible at data rates which are near to a disk's maximum potential bandwidth. Write speed may be guaranteed and average read bandwidth is predictable based on the drive profiling work from the previous chapter, although individual read operations are often irregular due to the mechanical operation of the disk. Where previous systems were only able to make worst-case bandwidth guarantees, or timing guarantees rather than bandwidth guarantees, the techniques presented in this chapter are robust and reliable.

The Mahanaxar model offers constant performance for indefinite periods of time at near-maximum disk potential. This stands in contrast with standard filesystems, which are not intended to handle this type of data. Previous approaches offer substandard performance compared to theoretical capabilities, no quality of service guarantees, and continuously degrading performance when maintained in constant operation. Results presented here demonstrate reliable performance in the face of variable element sizes, quality of service guarantees in spite of increased demands for additional bandwidth, and smooth and predictable performance curves in accordance with the maximum capabilities discovered in disk profiling.

Chapter 5

Data Indexing and Search in Transient Environments

Merely being able to store and manage rapidly cycling low lifetime data is not enough if that data cannot also be found again in short order. The need for rapid location is driven by the reality that this data is not long-lived: an inability to locate specific data quickly may convert into an inability to *ever* locate it, because it will soon no longer exist. If this happens, it may as well be that it was never captured in the first place. Not only must a management system meet real-time deadlines and maintain quality of service guarantees for the data storage itself, but the categorization and indexing of the data may be simultaneous in real time to support potential searching.

This search process shares some characteristics of a standard file system and some characteristics of a database system, but requires different techniques and stricter deadlines than either. The rapid cycle of data precludes a gradual buildup of an appropriate index, and the most likely outcome is that no search will ever be required. Data, along with its indexing information, will quickly expire. When a search is performed, it will almost certainly be a one-time occurrence to preserve or retrieve a region of data, which also means that the search will likely not be repeated. This characteristic renders the system subject to an unusual usability paradox: categorization and indexing must be performed when data first enters the system if it is to be of use, and yet the majority of pre-assembled indexing serves no purpose except to waste work and storage.

Therefore, indexing and search capability must be integrated into the storage system itself. General-purpose storage systems generally limit their indexing capabilities to file metadata, but that is not sufficient for these purposes. A constant inflow of new data, combined with a highly limited storage capacity means that data search must also meet real-time deadlines. The indexing information must also be integrated with the physical layout on disk, meaning that indexing information must be treated similarly to the data itself.

Indexing may take place on several different scales according to several factors. When data elements are large and differ mainly by timestamp, it can be stored in memory and need never be committed to disk. At the other end of the scale, indexing information may approach the size of the original data, and may actually exceed the original data in size in extreme situations. Indexing and search at that scale is problematic from any perspective: a full search requires the equivalent of reading the entire data set at the highest end. Nonetheless, data must be indexed at its arrival rate, and in a manner which allows for an efficient search when required.

In a system where data lifetime is measured in hours, it is safe to assume that a search has only one purpose: to find data that is “interesting,” and by implication, data that should be marked for preservation. Therefore, a real-time search may be defined as follows:

1. A real-time search is conducted over all data present in the system at the time the query is initiated.

2. A real-time search is considered successful if it locates all specified data in the system, and that data is still present in the system by the time the results are returned.

3. A real-time search is considered at least partially unsuccessful (or, as having missed its deadline) when it is unable to consider data because of expiration, or if it locates data which is no longer present in the system by the time the results are returned.

There is no absolute time requirement on such a search in the primary definition, although faster is always preferable.

Data collection cannot be interrupted while a search is conducted. It is thus possible to pose a query that cannot avoid being at least partially unsuccessful. Queries with no time-based component, or with a time-based component that includes the oldest portion of data, must simultaneously search data which is being overwritten in favor of new, and there may be no time to search the expiring data before it is overwritten. It is possible to engineer solutions around this quandary by maintaining reserve storage capacity, but this is equivalent to shortening the ring buffer.

5.1 Redefining Search into Data Movement

When data elements are large and indices few, such as in some types of time-based sensor data, there are few problems. For example, only a few hundred kilobytes are required to index two terabytes worth of data (a current-generation hard disk) stored as 50 megabyte elements and indexed by 8 byte timestamps. When the data element size is reduced by three orders of magnitude, the indexing structure only grows to a few hundred megabytes in size, an amount easily held many times over in main memory. Even if data elements are tiny, they may be indexed based on a range model, or with chunk-based resolution instead of element-based, so long as there is only a single axis of indexing required.

Search is a non-issue in this environment: any indexing structure already held in main memory can be exhaustively searched in a negligible amount of time when compared to the time required to move one chunk of data on or off of a disk drive. This is the best approach whenever possible, since it requires that no indexing information be stored on the disk itself, and thus maximizes total capacity while minimizing search times. Unfortunately, this approach is not always possible when the data elements are small and multiple indices required.

Consider the problem of storing IP packets indexed by source and destination addresses (4 bytes each with IPv4), as well as by an 8 byte timestamp. If these packets are between 20 and 1500 bytes in size (the typical range of an ethernet packet), the total amount of indexing for a two terabyte drive would be somewhere between (roughly) 15 gigabytes and 1.5 terabytes. This is far too large to fit in main memory for a commodity system. Therefore the indexing information must be kept at least partially on disk.

The greatest factor in search time is in the amount of time required to move data from the disk into memory[42]. Once the data has been moved from disk to memory, even a grossly inefficient search algorithm will still have performance that is orders of magnitude faster than the time needed for the disk to retrieve the next portion of data. Therefore, the best search technique is one that minimizes the data movement time from the disk. This implies in turn that the best indexing techniques are those which minimize data movement itself. Information should therefore be arranged in a manner to facilitate rapid retrieval from the disk with a minimal number of I/O operations.

Indexing information stored on disk is thus properly treated in the same manner as the data itself: by gathering indexing elements together into chunks, which can then be treated as single I/O units. These indexing chunks can then be stored throughout the drive alongside data chunks. There are three methods for determining placement of these indexing chunks: interspersed alongside the data in all regions of the disk, isolated to a separate region of the disk, or a combination of the two.

Interspersing data and indexing chunks over all regions of the disk is the most generalized solution and works well in all cases. If the number of indexing chunks can be computed in advance, they can all be placed in a specific region of the disk. If this specific region is the highest-bandwidth area (the outermost portion of the platter), all queries are sped up for uniformly better search performance. However, when data element sizes are variable and the total number of indexing chunks unknown, a specific region of exact size cannot be set aside. A combination of the approaches allows most indexing chunks to be stored in the highest bandwidth regions of the disk, with “overflow” indexing chunks distributed normally elsewhere. Working with a default assumption that writes are most common and reads of any sort are rare, dispersed indexing chunks are most efficient overall.

Indexing chunks “expire” when they no longer reference current data, and thus maintain the same lifecycle pattern as data chunks. If certain data chunks are preserved, partially-expired indexing chunks must also be preserved to properly reference the data, which can introduce a storage overhead. An engineering refinement is to use spare bandwidth to periodically consolidate partially-expired indexing chunks so as to keep wasted storage space at a minimum.

The main problem with chunk-based indexing is one common to many indexing systems: the arrangement of the data is focused and sorted around a single primary key. For example, the most probable primary key in this storage system is that of time, since the whole structure of the system is based around limited-lifetime data, and the method of expiring data is based on time. Consequently, a search based on timestamps will be reasonably efficient through a bird’s-eye view of all the indexing chunks, which allows one to target a range of times. Conversely, if the required search is based on a source IP address (to continue the example from above), there is no bird’s-eye view of that aspect of the index, and each indexing chunk must be searched in turn.

However, a search of every single indexing chunk must be avoided whenever possible. The speed at which indexing chunks may be read back from the disk is limited by the quality of service guarantees made for incoming data, which has the first call on bandwidth. A non-time-based query that needs to search every index chunk will almost certainly miss data which should be preserved, as incoming data overwrites it before it can be found. The solution to this problem is to store multiple copies of the indexing information sorted by additional indices.

5.2 Lazy Indexing and Query Prediction

A *successful* search may safely be assumed to be relatively rare in this high-turnover data model. Specifically, queries may be frequently executed, but the percentage of data matching any given query will be low compared to the total volume managed by the system. If this were not the case, this style of data management would be inappropriate for that particular problem space.

Therefore, preemptive indexing and search will yield wasted effort and resources in the vast majority of cases. Aside from computational resources, which may or may not be a concern, disk resources are always precious in a storage system. Any time spent writing extra indexing information, or reading back data for a preemptive search, is disk time left unavailable for the execution of real queries. Contrariwise, if no extra steps are taken to preemptively classify the data, data will likely be lost before it can discovered, as discussed at the end of section 5.1. When indexing is left until the last moment, the natural cycle of data will inevitably put some of it out of reach before it can be distinguished. A balance is needed between full indexing and no indexing.

The best middle ground is a lazy indexing model which predicts the most likely queries and indexes accordingly.

As previously discussed, it is possible to trade total storage capacity and a portion of the available bandwidth for additional indexing capabilities. Each data element is stored only once, but may be referred to by multiple indexing vectors, each pointing to the same element. Each indexing component is then stored in its own indexing chunk, which is written to disk alongside data chunks. When a query is made, the most "useful" indexing chunk is read back from disk, and the appropriate data chunk located by that information.

Figure 5.1 depicts a visual example of indexing chunks using IP packets as an example data source. Each IP packet is indexed according to the source address, destination address, and by a generic data marker of some sort (the specifics are unimportant for illustrative purposes). A timestamp is also available as metadata outside the raw packet elements. As each element enters the system, the entire contents of that packet are stored in a data chunk. Additionally, the "useful" components for indexing are extracted and placed into their own indexing chunks. In this case, the source address, destination address, data marker, and timestamp are all indexed and placed into indexing chunks. Each entry is merely a pointer to the data chunk containing the relevant packet, and thus each indexing chunk holds element pointers to many data chunks worth of IP packets. As indexing elements approach their maximum size, they are stored on disk alongside the data elements and new indexing chunks are prepared.

This technique allows queries to be executed on predefined indexing vectors

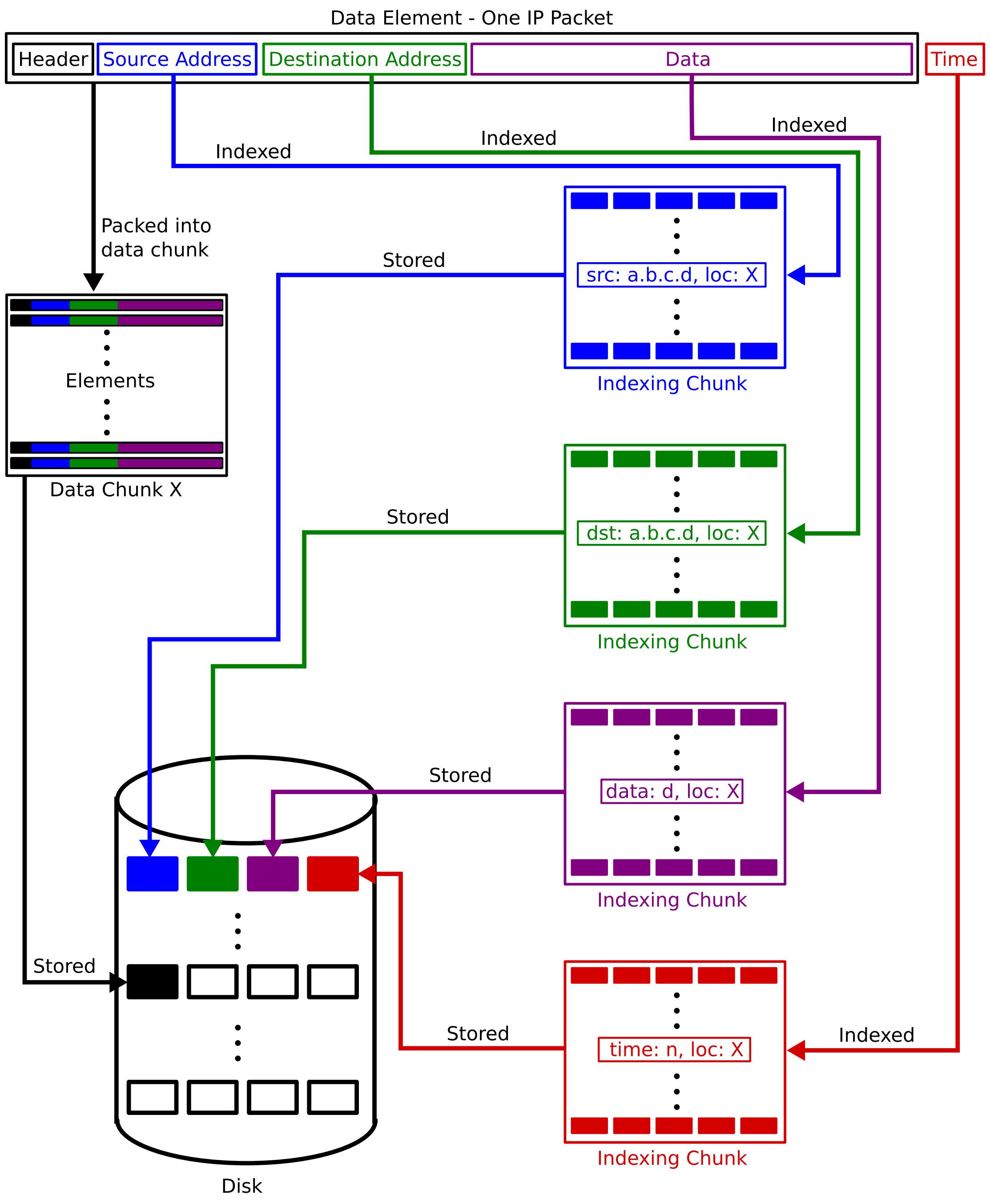


Figure 5.1: Indexing chunk construction using IP packets as the data source example.

at the same speeds as timestamp-only queries. Search performance is vastly improved, albeit at the cost of additional required bandwidth and a lessening of overall storage capacity. Both overheads are calculable based on the data element size, indexing vector size, and rate of data element arrival. Quality of service guarantees are thus not compromised. The chief difficulty of this method is that of determining which additional indexing vectors are to be managed. It is not unreasonable to assume that such vectors may be known in advance. In a system designed to manage transient data, the user is expected to know what information is most likely to be “interesting.” This may be preconfigured in the initial system setup and the indexing information made available at the commencement of operations.

Unfortunately, not all queries are necessarily predictable in advance. Nothing may be done to speed the processing of one-off queries, but a more sophisticated approach may allow for the speeding of subsequent queries related to previous ones. The key to this approach is to allocate a certain amount of bandwidth and storage capacity to flexible system-defined indexing processes which are able to change at need, and allow prior queries to direct the indexing expectations of future ones.

The simplest form of query prediction is merely to adjust indexing information to match the recent queries. For example, if data elements are being indexed on indexing vector X and a query is made for those data elements matching criteria based on indexing vector Y, the indexing of new elements can shift to vector Y instead. This is of no help for searching older data, but will speed up future queries for all new data.

A somewhat more refined method of query prediction focuses on the arrangement of indexing information within preexisting indexing vectors, rather than in the entire reallocation of indexing vectors. When multiple queries are executed which look for a limited set of specific criteria on a specific indexing vector, it may be useful to further refine indexing operations to isolate those specific criteria. For example, if searches are frequently performed which specify a value of $[a, b]$ for indexing vector X, the indexing chunks may be further subdivided into groupings which specifically hold values of $[a, b]$. This allows for future searches to more quickly turn up the desired data elements by focusing on indexing chunks dedicated to the known criteria, which may confer additional advantage beyond that of merely referring to indexing chunks on the general indexing vector. Because overall indexing information is not increased, and the sorting and storage order is modified, no additional bandwidth or storage capacity is needed in order to implement this technique.

Both types of query prediction are dependent upon the storage system understanding the data elements as more than just a binary stream of bytes. A combination of pre-known indexing vectors and adaptive indexing vectors may be used for maximum flexibility. It is also possible to allow for the possibility of "temporary" indexing information which is only stored when storage resources are not otherwise occupied. This indexing information is only recorded when disk bandwidth is available, and does not interfere with desired data reads, either for search or data retrieval purposes. It may impose a cost of extra storage overhead, but will never interfere with read bandwidth, albeit at the cost of not being a comprehensive index.

5.3 Performance Evaluation

The Valmar system is an extension of the Mahanaxar data management model, which further incorporates the indexing and search aspects discussed throughout this chapter. As discussed in section 5.2, the indexing aspects are both pre-configurable along initial parameters, and capable of adjusting future indexing distribution to better maximize performance according to past queries. Like Mahanaxar, the testing and evaluation version of Valmar is implemented in Linux as a multithreaded process running in userspace and accessing disks as raw devices.

5.3.1 Comparison Systems and Testing Procedure

Results presented in section 4.4 made comparisons between the Mahanaxar system and a standard ext2 filesystem using flat files for data storage. This comparison type is appropriate for simple management models, but is not well-suited to data elements with complex indexing requirements. No provisions for complex indexing techniques are available beyond standard system metadata, thus making this approach less than ideal for data which needs to be indexed on several vectors simultaneously. In order to make an appropriate comparison, the ext2 based system used "indexing" chunks similar to Valmar in order to support search capabilities.

The more appropriate comparison system in this case is a database model. Database systems are designed to handle the indexing requirements necessary for complex queries, and are also capable of handling larger data elements with simple indexing requirements. Therefore, comparisons are made against a database system, in addition to the standard filesystem. In this case, the database was mysql-based on top of a standard ext2 filesystem, which once again proved more efficient than more complex filesystems, and implemented in Linux. The exact database layouts were constructed in advance according to the type of data needed.

A hybrid comparison system was also attempted, where all indexing information was stored in the database, and data elements stored as flat files. Unfortunately, this hybrid approach proved to have far worse performance than either "pure" approach, and was thus dropped from testing.

Both comparison systems were constructed such that they prioritized data capture over data reading, rather than rely on the unwanted "fair" allocation as shown in section 4.4. Furthermore, an external time-based indexing structure was maintained in main memory for both comparison systems. Without this external indexing structure, all expiration had to take place based on system metadata (ext2) or queries (mysql), which was hugely detrimental to their respective performances. It takes a great deal of time to even *find* the data due to be expired before actually overwriting it.

Both comparison systems also employed a policy of in-place element replacement in an effort to prevent excessive fragmentation. Again, performance was miserable if individual elements were deleted and new files or database entries created for the incoming elements. Therefore, all elements were assumed to be of maximum possible size and overwritten in-place, at the expense of lowering system total capacity.

5.3.2 Large Element Handling

The Valmar model was first compared against other systems with large fixed-size 5 MB elements. These elements were indexed only according to time, with no complex data categorization scheme. The system was allowed to run for multiple data cycles, randomly preserving data chunks, reading them back to return to an outside process, and un-preserving them again. As in previous tests, the data cycle was constrained to about 1.5 TB (out of a disk size of 1.8 TB) in order to maintain a constant write bandwidth of 80 MB/s, with margin left over for a healthy read bandwidth. Figure 5.2 shows the results from this test.

Part (a) shows performance for ext2. Again, even though the raw disk ability can easily support 80 MB/s of write bandwidth, the filesystem is unable to keep up. The performance curve is again jagged, and demonstrates that no constant write bandwidth above about 45 MB/s may be expected. Read bandwidth is zero for all intents and purposes, with only a few reads possible at the points where bandwidth temporarily spikes above 80%.

Part (b) shows performance achieved by the database system. Although databases are not usually used to store large amounts of relatively uncomplicated binary data, it is included here for comparison purposes. Its performance is roughly comparable to that of the flat file system for just over 2 TB of processed data, and performance is far more stable. However, during the second data cycle, performance drops rapidly from about 60 MB/s to 40 MB/s, and continues to slowly decrease from that point. Although

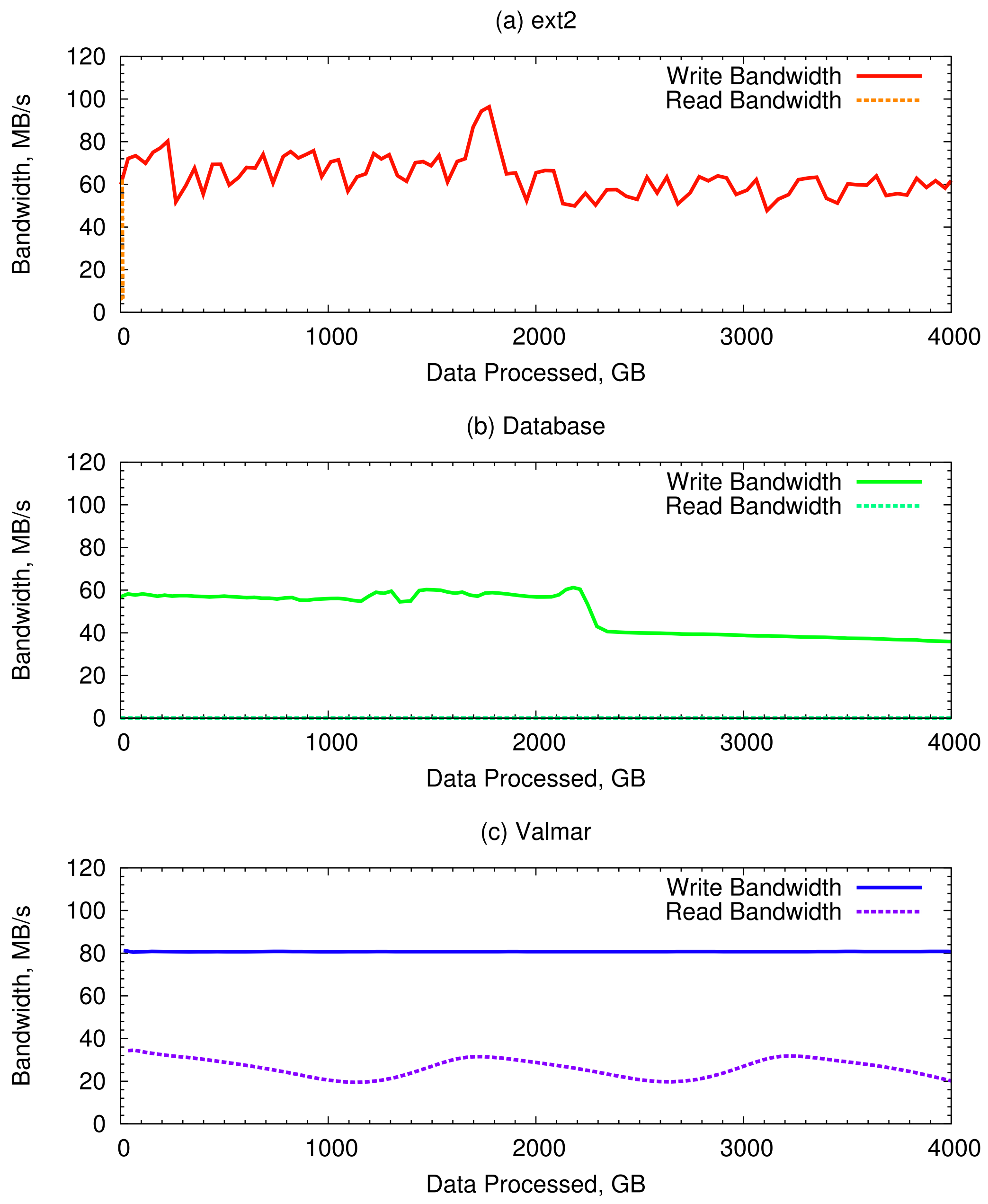


Figure 5.2: Performance comparison of ext2, mysql, and Valmar with large (5 MB) data elements.

the database has performance which is far more stable than ext2, it never achieves the desired write demand, even temporarily, and thus has no read bandwidth during the course of the test.

Valmar's performance is shown in part (c). Performance is again stable, and shows no surprises, being basically identical to the Mahanaxar model previously shown in section 4.4. In fact, the performance is practically indistinguishable from the results shown in figure 4.6(b). Read performance varies between 20 and 40 MB/s as expected, while write performance is steady at 80 MB/s.

5.3.3 Small Element Handling

Baseline performance having been established with large elements, small element handling with variable indexing requirements came next. IP packets are a suitable real-world example due to their diverse nature and highly structured format. It would have been impractical to use an actual network trace for a data source due to the size and rate of the data (a single test run requires multiple terabytes of data streaming into a single point source over the course of the experiment), so simulated IP traffic was generated instead. The parameters were set as 10,000 source IP addresses communicating with 1,000,000 destination IP addresses, which could perhaps be seen as a large facility communicating with the external world. A deployed system would obviously need sufficient hard drives available to handle the total bandwidth; it may be reasonably assumed that this test was of a particular process/disk combination tasked to handle "about" 80 MB/s of data. The exact bandwidth was not fully precise due to the varying nature of packet sizes.

IP packets in this test were indexed according to timestamp, source address, destination address, and a metadata-like classification number based on the contents of the packet. Each packet varied between 20 and 1500 bytes in size, randomly and uniformly distributed in that range. Although real (content-bearing) Internet traffic is weighted towards larger-sized packets, smaller packets place a greater strain on the indexing system, which is part of the desired testing. As previously discussed, the comparison systems were limited to in-place element replacement, which necessitated maximum element size assumptions. Therefore the average amount of history maintained by the comparison systems was approximately half that of Valmar, which is capable of handling all element sizes.

Figure 5.3 shows the performance of small-element handling with indexing. As before, the write bandwidth was set to 80 MB/s and the data cycles were constrained to 1.5 TB for all systems.

Part (a) shows performance of the ext2 based filesystem. This test was performed with a 128 KB element size, since general-purpose file systems generally do not react well when attempting to store hundreds of millions of 20-1500 byte files in a constantly rotating data cycle. Even with an element size over a hundred times larger than an IP packet, performance was extremely erratic. Write bandwidth steadily decreases during the first cycle of data, roughly corresponding with head position on the disk platter. It continues to spike and dip irregularly after that point, starting to stabilize around 40 MB/s of write bandwidth. Read performance was zero due to the inability

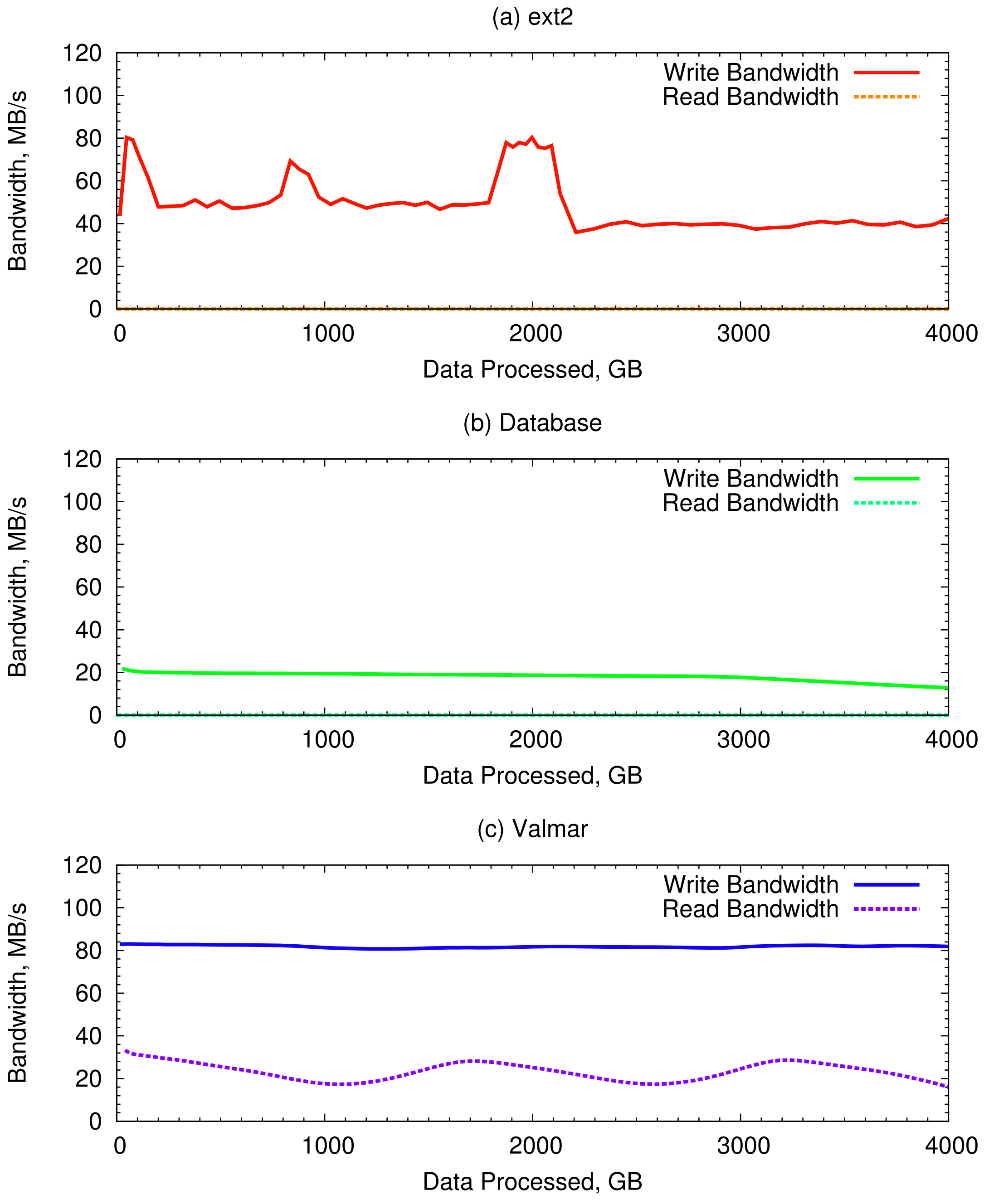


Figure 5.3: Performance comparison of ext2, mysql, and Valmar with small data elements.

to keep up with the writing data cycle. During the course of the test, 51.3% of the data was lost.

Part (b) shows database performance. Although a database is well-suited to indexing and querying over a large number of small elements (or "rows" in database terminology), these results show that it is not well suited to record that data in real time. Performance is limited to no more than a 20 MB/s write speed, and it drops further as the system continues into further data cycles. There is no opportunity to read data due to write priorities, leading to a constant read bandwidth of zero. This means that there is no opportunity to make a query while the system is in operation, without losing even more data. When the system is no longer recording new data, queries still take an extremely long time, since the database may need to search over the entire drive's worth of data in the worst case. During the course of this test, 79.4% of the data was lost.

Part (c) shows the performance of Valmar. The write bandwidth is not exactly pegged to 80 MB/s as it was in large-element testing, but instead is slightly above. The rate is for two reasons. First, the use of indexing chunks as described in section 5.1 imposes a small but constant overhead. Second, as discussed in section 5.2, Valmar redistributes IP packets into different data chunks based on past queries and trends. This has the side effect of adding a slight jitter to the write bandwidth as data chunks are irregularly ready for writing, compared to the constant readiness of large elements.

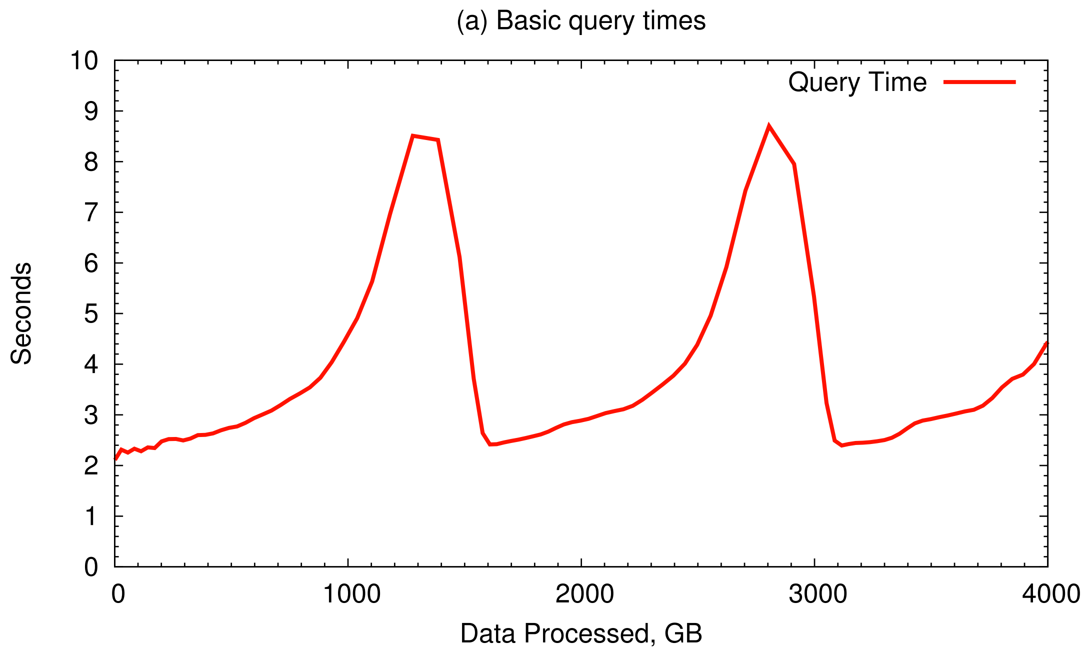
Both of these aspects are expected for this type of data and do not result in any lost information. The exact amount of overhead bandwidth required is calculable in advance, but it depends heavily upon the size of the data elements, indexing aspects of each data element, and core arrangement of the data. In the particular case presented in figure 5.3(c), the overhead is about 2 MB/s.

Read bandwidth again follows a sinusoidal/sawtooth pattern in progression with the disk head position on the disk platter. Overall, even with data elements 0.02% the size of the previously-tested large elements, performance is steady. Again, the system can continue in this mode of operation indefinitely, losing no packets, precisely as required.

5.3.4 Query Times

If Valmar is able to contain an entire index structure in memory, the ability to preserve data based on a query is near-instantaneous. Data may then be retrieved for full inspection at an average rate of 20-30 MB/s (in this particular setup on this particular disk), depending on the current position in the data cycle. When Valmar is unable to contain the entire index structure in main memory, query speed depends on indexing chunk density and the time period covered by the query.

For this particular set of results, using IP packets as described in the previous set of results, the indexing structure is set up such that only one to two indexing chunks need be read to cover the vast majority of queries. Indexing chunks are divided to cover specific ranges for each indexing element, such that a single indexing chunk can cover many gigabytes of incoming data. For example, all IP source addresses of 10.100.p.q can be indexed in one particular chunk for a large period of time, while



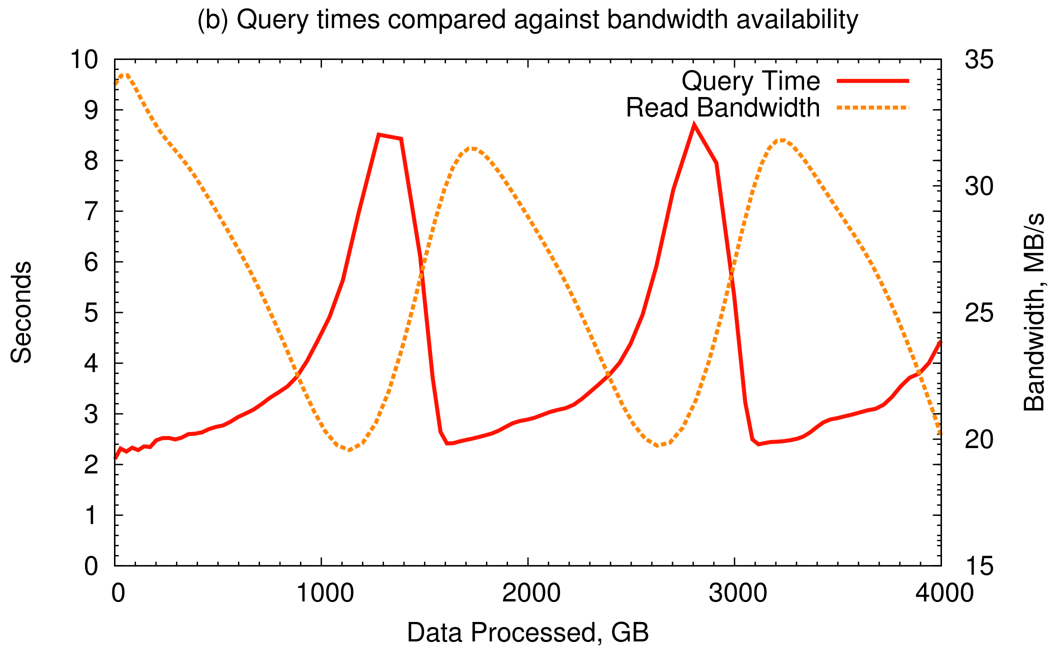


Figure 5.4: Queries with Valmar.

all destination addresses of 100.10.x.y are indexed in another particular chunk. Thus any query seeking data from a specific source address to a destination address need read two indexing chunks at most, unless the desired time span happens to cross chunk boundaries. Figure 5.4 shows these data query results.

Part (a) shows the average time requirements to answer any given query. This value does not include the time required to return the data chunk, but only the time required to locate (and preserve) it. The time pattern is directly related to the read bandwidth curve of figure 5.3(c), since indexing chunks are being read from the disk drive itself.

Part (b) superimposes a rescaled version of the read bandwidth curve with the average query time curve. As expected, the query time curve is an inverse of the read bandwidth curve. The sawtooth pattern shows that the query time is low during the point of the data cycle with the highest read bandwidth, and query time is high when the read bandwidth is lowest.

5.4 Conclusions

The ability to retrieve desired data is an integral part of any storage system, but the difficulty in doing so is much greater in an environment where data may only exist for a matter of hours, and where any attempt to find it cannot be allowed to interrupt the ongoing collection of new incoming data, which consumes a majority of the available storage resources. The Valmar model solves this problem by close integration of indexing information alongside the raw data itself, and appropriate arrangement of that information to minimize search time based on both predetermined indexing information and the history of previous queries. It is not possible to fully account for all data search possibilities, and thus it is not possible to absolutely guarantee that data may be found in a timely fashion for worst-case situations, but it *is* possible to conduct queries over a large amount of data in a relatively short amount of time.

Valmar continues to offer performance at close to a disk's full potential, maintains all quality of service guarantees, and adds a comprehensive indexing model such that complex queries may be executed on the data, and results returned in a reasonable time frame. Neither standard filesystems nor database systems are capable of handling this type of data at this scale, and provides obviously inferior performance. The results presented in this chapter demonstrate Valmar's continued reliable performance for variable element sizes down to twenty bytes, and the ability to answer queries at a rate directly proportional to the disk's available read bandwidth.

Chapter 6

Large-Scale Data Management

Systems which employ only a single hard drive are suitable for data rates of up to tens of megabytes per second, but larger volumes of data require correspondingly larger systems to handle it. To a certain extent, multiple copies of any single-drive storage system, can be run in parallel on multiple hard drives, and even over multiple nodes. However, a large-scale storage system is expected to operate as more than multiple concurrent copies of a smaller setup. Features are needed for ensuring reliability, scalability, and control, and engineering refinements are available which are not practical in small-scale work.

Reliability mechanisms are very common in larger storage systems, and for good reason. Hard drives have a high aggregate rate of failure[62], such that hardware failure is expected on a regular basis whenever large numbers of drives are involved. There being no method to regenerate lost data in this problem area, not only must continuing collection be uninterrupted whenever there is a case of hardware failure, but any data previously stored on the now-inaccessible drive must be recoverable in a reasonable amount of time.

I/O balancing is frequently an issue in large storage systems, and requires careful monitoring. Write load is obviously not important in a system where all hard drives are expected to constantly record new data, and the read load on its own is entirely unpredictable. However, data distribution may be modified in real-time to better support queries, and control mechanisms for searching large-scale storage systems remain important in this model.

Furthermore, there are various engineering refinements possible in larger storage systems, which are not possible in smaller ones. For example, extra drives may be utilized to improve indexing and search performance, and stand by as hot spares to swap in when other drives suffer failure, or perhaps become "overloaded" by an excessive number of search requests.

The Valmar model is designed to integrate with this style of large-scale data management and basic operation remains unchanged. Performance is additive such that many drives acting in concert behave as they would independently, and administrative overhead is minimal. Overall storage capacity and bandwidth is only reduced by the number of resources assigned to reliability and other performance-enhancing purposes, and few computational resources are needed.

6.1 Reliability Mechanisms

All mechanical devices fail from time to time, and disks are no exception. Such a failure can lead to an inability to record new incoming data, temporary loss of access to current data, or permanent loss of current data. These failures may be only temporary (a momentary loss of power) or permanent (physical destruction, as in a “head crash”), but are particularly important to guard against in a system designed to capture real-time data. As previously discussed in section 2.5, RAID techniques are frequently used for general purpose reliability.

RAID-like methods may also be adapted for handling large volumes of high-bandwidth transient data. The constant data rate even allows for certain performance advantages and repair techniques which are not available in a standard storage system. Recalling that the “data chunk” is defined as the minimum unit of storage, two choices are available for RAID operation. An $n + 1$ reliability scheme based on RAID 4 is assumed here for the purpose of example, but the techniques work equally well with complex RAID and error-correcting code schemes.

The first choice is that RAID groups may be defined on an ad-hoc basis over n separate and unrelated data chunks, and each group of chunks is distributed over a different set of disks. Such groups are connected only by the fact that they are all being written to individual disks at approximately the same time. This approach is the most generalized and allows groups to be defined over multiple nodes and data sources, but also creates an implied connection between data chunks in a group, which imposes future restrictions on their order of expiration.

The second choice is that data chunks may be redefined as n times their original size, and then broken down into sub-chunks for storage on individual disks. This ensures that related data is kept in close proximity and imposes fewer connections between otherwise unrelated data chunks. However, it is also less flexible in terms of what data can be arranged into RAID groups, and hardware failure imposes a higher burden on the remaining disks in a group whenever that data must be recovered.

RAID-like operation imposes no performance penalty or loss of ability to maintain real-time deadlines. However, it does require that all disks used in a RAID group have similar performance characteristics, since total bandwidth is naturally limited to that of the worst performing drive in any given group. When reliability is not considered, a large-scale implementation of this storage methodology has no reliance on drive homogeneity for normal operation, so long as each drive is expected to handle no more data than their individual profiling results indicate. However, since all drives in a RAID group are expected to maintain an identical data rate, reliability requires that drives be divided into classes. Overall heterogeneity is acceptable, but disks in individual RAID groups must operate at about the same data rate.

Alternatively, if data chunks are of identical size but bandwidth is variable, it is possible to rotate RAID groups to achieve the same effect. A simple example of this concept is shown in figure 6.1 using four disks of identical bandwidth, and two other disks which have a combined bandwidth equal to that of one of the first disks. Each of the identical disks records data in normal fashion, while the two lesser disks trade off

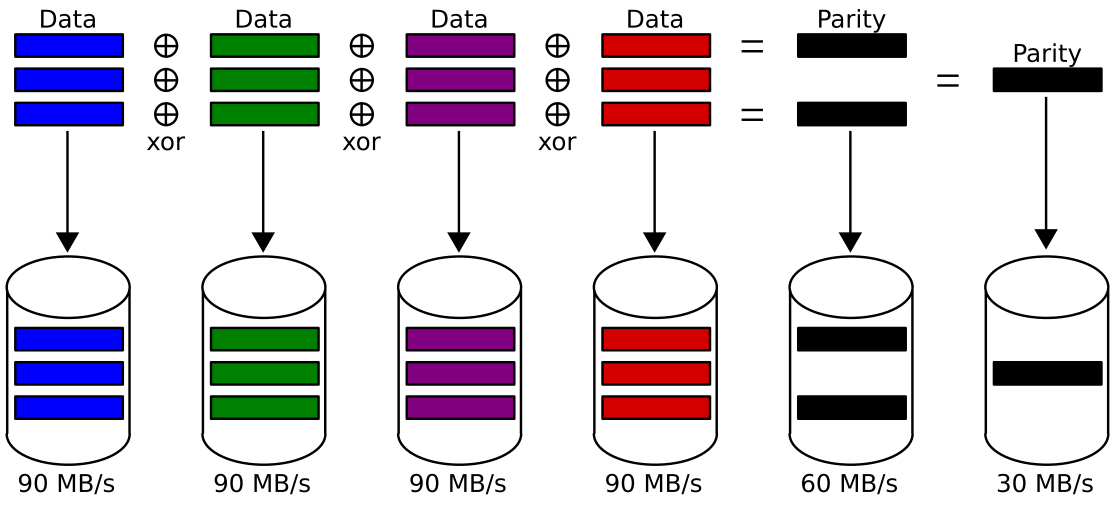


Figure 6.1: Heterogeneous disks in RAID operation. All disks are assumed to be writing chunks as their maximum speed, and thus the parity disks (with lower capabilities) must alternate.

storage responsibilities for parity chunks, according to their respective bandwidths.

Conventional RAID systems provide fault tolerance and performance advantages in normal operation, but can have difficulties operating in degraded mode. I/O times increase dramatically as the system is required to reassemble small pieces of data from elements across all other disks. Fully reassembling the data can take a great deal of time, as disk capacities have grown much faster than disk bandwidth. It may take hours to reassemble the data lost from the failed disk if all other performance is suspended, or it may take a day or longer to reassemble the data if continuing normal operation is expected from the array during the period of reconstruction. In particularly large systems, there is a significant chance that a second drive will fail (through an unrecoverable read error or worse) during the reconstruction process, which leads to data being lost entirely.

Perhaps counter-intuitively, this type of disk failure has almost no impact on quality of service requirements and the ability to meet real-time deadlines. In this data model, only the vulnerability to data loss is increased. This is because of the mode of operation: all disks are in use at all times, whether there are n or $n + 1$ of them. Any I/O operation involves every disk at near-maximum performance at all times, and hence the failure of one disk does not negatively impact overall performance.

Recalling that this system can be characterized as "write once, read maybe," it becomes apparent that one of the main disadvantages of a RAID system might never actually come into play: the data may never need to be rebuilt. In a system where data lifetime is measured in hours, there is no reason to attempt a full rebuild of the data, although there is still value in recovery of the indexing chunks in order support future search. If a failed disk contains no preserved data, one *never* need reconstruct any of it. There may be a need to regenerate a portion of a failed disk's data if some of it is preserved, but there can never be a situation where an entire disk is made up of preserved data (or rather, if there is, the ring buffer has ceased all collection of real-time data and no quality of service considerations are in effect anyway). This does have the same limitations of any other RAID-like system for the portion of data which *does* need to be reconstructed, unfortunately, though the magnitude of the problem is far less.

The constant cycle of operation over a group of disks also creates the opportunity to smooth out performance curves. By rotating data placement such that data chunks are placed on different portions of the drive platters for each chunk in a RAID group, the I/O time for the entire chunk may be averaged over the combined disk bandwidth. Note that this only works with constant groups of homogeneous disks. By placing some data chunks on the high-bandwidth portions of the disk platter, and other chunks on the low-bandwidth portions, I/O time for the entire group can be averaged.

This technique cannot be used to increase average bandwidths, but can be used to *stabilize* read bandwidth when reading entire RAID groups at once. Data chunks on individual disks are located at specific points on the platter, and thus may be subject to high or low bandwidths when read. When a group of data chunks is distributed over a broader region of the platter, read bandwidth is stabilized by averaging all the per-disk I/O times. This method is only useful when the cycle of data permits disks to operate in lockstep with regard to their physical layout, and is thus best suited to the method of breaking single data chunks into sub-chunks, rather than combining unrelated chunks.

By trading total bandwidth for flexibility in rebuilding, it is also possible to implement rotating RAID groups among a large set of disks, storing each incoming chunk on a different combination of disks to produce an “m choose n” number of layouts. Upon the failure of any one disk, other disks can be taken out of the normal rotation to completely reassemble the lost data in short order. Perhaps more importantly, taking a disk out of normal rotation can also allow for faster search and faster data reads, as real-time write deadlines are suspended for a single disk, and thus this technique has possibilities beyond simple reliability purposes.

6.2 Scalability

As storage systems grow in size, there is often difficulty when trying to ensure that the workload is adequately distributed over all of the involved hardware. Certain files and directories may be more "popular" than others and form hot spots in the storage hardware. This leads to a disproportionate number of I/O requests being focused on only a few disk drives, and leaving the others idle. Not only is this an inefficient use of the storage hardware, but it leads to bottlenecks and a low level of overall performance.

Fortunately, this is not a major issue when dealing with high-volume real-time streaming data. Hard drives involved in this mode of operation have a constant duty cycle according to their relative performance. Each is continuously writing new data, and no hot spots are possible for writes. Minimum aggregate write bandwidth is thus the sum of the individual write bandwidths on all drives.

However, hot spots when *reading* are possible. Each element of data is written to only one disk at a time — reliability methods excepted — and each element quickly expires. The worst case scenario is to have many simultaneous pending reads for data elements which, by happenstance, are all on the same piece of hardware. Read bandwidth is limited to whatever is left over from guaranteed write performance, and data chunks may only be read one at a time. Meanwhile, all other read bandwidth on the other disks is left idle.

Since it is not possible to predict all future read requests with perfect accuracy, this problem is not solvable in the general case. However, it is possible to attempt to anticipate future read requests and adjust future data distribution to render it a less likely outcome. Section 5.2 discussed the concept of query prediction and rearranging data and indexing chunks as to maximize predicted future efficiency. The same concepts can also be applied to data distribution in large-scale storage systems, and data can be redistributed based on past reading patterns.

For example, if it is determined that a large proportion of the outstanding read requests are directed at data with a particular set of characteristics, it is possible to reroute that data onto different hard drives as new chunks are assembled. This has no effect on already-written data and there is no guarantee that future requests will be made for the rerouted data chunks. It may even worsen matters if, by happenstance, the rerouted data patterns lead to a new logjam of needed data on the same piece of hardware. However, it does not compromise existing performance guarantees.

There are two potential downsides to this mode of operation: increased network traffic bandwidth if data is routed to a different storage node, and increased difficulty in locating the appropriate data once it has been rerouted to a different piece of hardware. Both problems may be avoided if data redistribution only takes place within a single node containing multiple disk drives, and thus all requests for reads continue to go to the same location.

Basic data distribution is best handled statically from data source to data destination. This method allows for the greatest correlation of data elements, since users need only look in one location (or set of locations) for desired data. For example, all data from a sensor may be routed to one particular data node, or all traffic from a single network router distributed over a known set of storage nodes. All search requests may thus be directed at the node(s) known to hold the appropriate data. Since this data is best distributed statically, it is better not to confuse the issue by redistributing data outside of the same node, and thus forcing the corresponding reroute of data retrieval requests to unfamiliar locations.

6.3 Performance Evaluation

The previous chapter having established the validity of the Valmar system (see section 5.3), this set of results adds additional large-scale storage system accoutrements as described in this chapter. No comparison systems are modeled in this section, since the intent is to demonstrate that previous techniques remain acceptable. The tests in this section all involve multiple hard drives, and multiple instances working in unison.

Results continue from the "small element handling" portion of section 5.3, with the data elements being generated IP packets, simulated as 10,000 source IP addresses communicating with 1,000,000 destination IP addresses. Individual element size remains uniformly distributed between 20 and 1500 bytes, and are indexed according to timestamp, source address, destination address, and a classification aspect based upon the data contents of the packet.

6.3.1 RAID Performance

Valmar's RAID performance was tested in a 4+1 configuration on one storage node. That is, four data disks (with largely identical performance and of similar size)

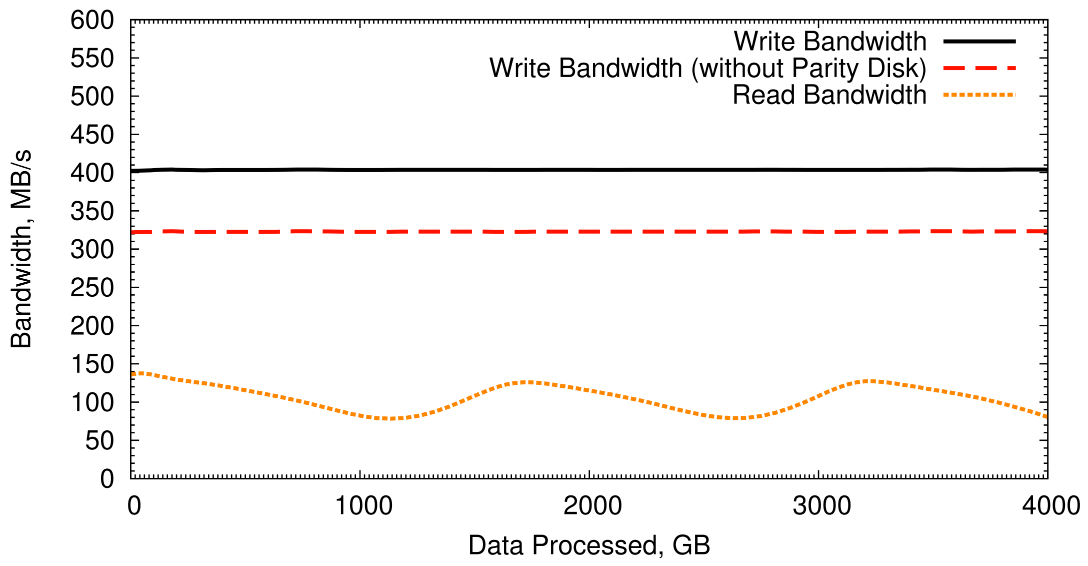


Figure 6.2: Valmar performance with RAID in a 4+1 configuration. Data chunks are in the same order on all disks.

recorded incoming data, while a fifth disk recorded the parity information calculated from the four data streams. Each disk was tasked to record 80 MB/s of data (as in section 5.3), with each of the chunks unrelated to each other except by participation in the same RAID group. Multiple methods of data chunk configuration were tested, and four of them are shown here in figures 6.2, 6.3, 6.4, and 6.5. In all cases, read requests are made simultaneously to all data disks, as fast as bandwidth is available.

Figure 6.2 shows the results from four data disks working in unison with regard to physical layout. That is, each disk starts its data cycle from logical "byte zero" on the disk, and proceeds in unison from the outermost regions of their individual platters to the innermost regions. Write bandwidth is just above the desired 400 MB/s, of which 320 MB/s is data. As in the previous chapter, the slight overhead in

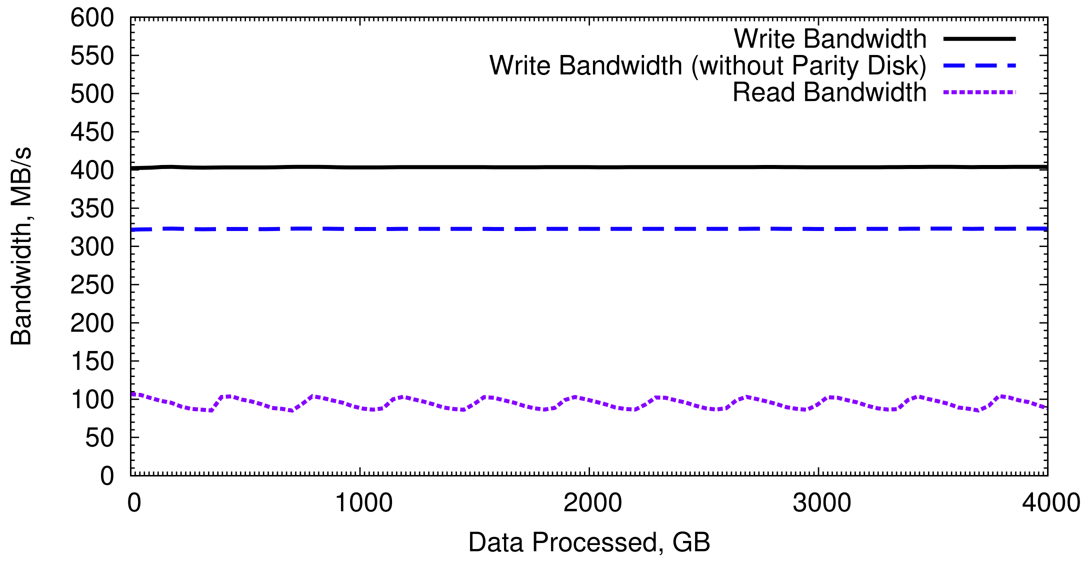


Figure 6.3: Valmar performance with RAID in a 4+1 configuration. Data chunks are logically shifted so that "chunk 1" is offset by roughly a quarter in each subsequent data disk.

recording is caused by the indexing chunks on top of the normal data chunks. Read bandwidth is extremely similar to figure 5.3(c), except that it is four times greater in magnitude (as expected). Since each disk is working in unison, areas of high bandwidth and low bandwidth are encountered simultaneously over all disks, leading to the same sinusoidal/sawtooth pattern previously seen on similar graphs.

Figure 6.3 shows the results from four data disks, each with a regular offset in regard to physical layout. The first disk starts its data cycle at the beginning of the drive, the second disk starts one quarter of the way through the drive, the third disk starts one half of the way through the drive, and the fourth disk starts three quarters of the way through the drive. The parity disk is not normally involved in data reads and operates in standard fashion. Again, write bandwidth is exactly as expected and

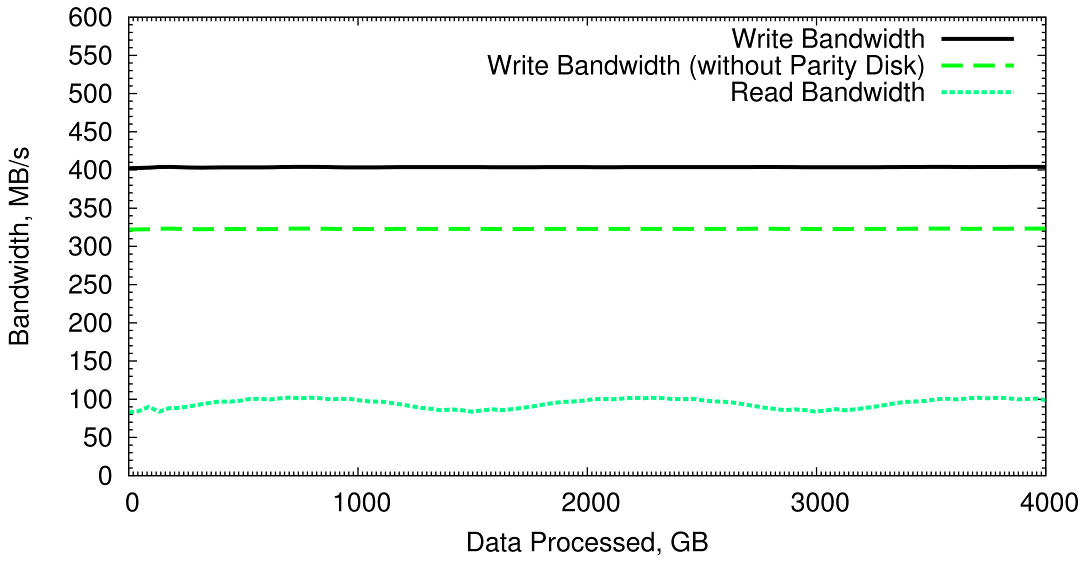


Figure 6.4: Valmar performance with RAID in a 4+1 configuration. Data chunks are reversed on every other disk, so that the "odd" disks are working from the innermost portion of the platter, out.

desired. Read bandwidth maintains a sinusoidal/sawtooth pattern, but with a lower variance and a period four times that of figure 6.2. Thus, the regular offset of the four data disks allows for a greater stability in read bandwidth, as discussed in section 6.1, though it obviously still has peaks and valleys in the performance.

Figure 6.4 shows the results from four data disks, with pairs operating in different "directions." Two disks are proceeding in standard logical order, while the other two disks are proceeding in reverse logical order. Thus two disks start their data cycles in the highest bandwidth regions, and two start in the lowest bandwidth regions. Again, the parity disk is not involved in data reads and operates in standard fashion, with no bearing on the data disks. Write bandwidth remains exactly as desired with no changes. The read bandwidth curve combines the benefits of figures 6.2 and 6.3, with

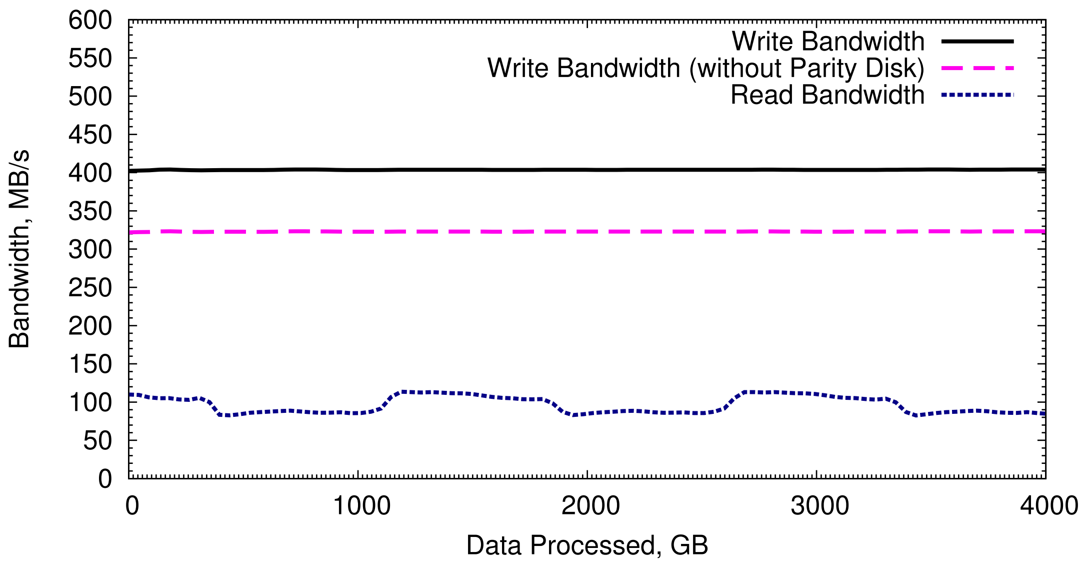


Figure 6.5: Valmar performance with RAID in a 4+1 configuration. Data chunks are logically shifted by roughly a quarter in each subsequent data disk, and every second disk is working from the innermost region of the platter, out.

both a lower overall variance and fewer peaks and valleys. This method of data chunk organization presented the most stable read bandwidth of all tests.

Figure 6.5 combines the techniques shown in figures 6.3 and 6.4. The data disks each start their data cycles at quarter-disk offsets, except that two of the disks work “forward” in those offsets, while the other two work in a reversed direction. Once more, the parity disk works normally, not being involved otherwise. The write bandwidth is still unchanged, providing the needed write guarantee as usual. The read bandwidth bandwidth curve shares characteristics from the previous two methods, with a period half that of figure 6.3, but with a lesser performance stability.

Since all data chunks were unrelated to each other, individual chunk read times on a per-disk basis were still dependent on their disk’s present position in their data cycle. Thus, individual queries are still governed by the results seen in figure 5.4. Other performance curves for read bandwidth are possible with other configurations of data cycle order on the disks making up a RAID group. The technique of reversing the order of every other disk in a RAID group, as shown in figure 6.4 does not always have the most stable performance of those configurations possible, but it is a good general-purpose combination of simplicity and stability. The number of disks used in each RAID group changes the curve, and a larger number of disks presents a greater opportunity for greater overall stability in the read bandwidth for that group.

6.3.2 Large-Scale Operation

Large-scale operation of Valmar was tested on a twenty four node computing cluster, each node of which was equipped with three same-model hard drives of theoretically similar performance characteristics. Three of these seventy-two hard drives had very poor performance and were thus unsuitable for use in this test. Of the other sixty-nine disks, twenty of them were previously profiled in figure 3.3, and the remaining forty-nine have similar performance characteristics. Figure 6.6 shows the individual metrics for twenty drives operating in unison. Each drive is tasked to maintain a write bandwidth of 50 MB/s, and whatever read bandwidth is available. The write performance of each drive is steady at 50 MB/s as expected, but the read performance is variable based on the individual disk characteristics.

All storage nodes were set up to handle IP-style indexed traffic, as per the experimentation previously shown in section 5.3, and RAID operation earlier in this

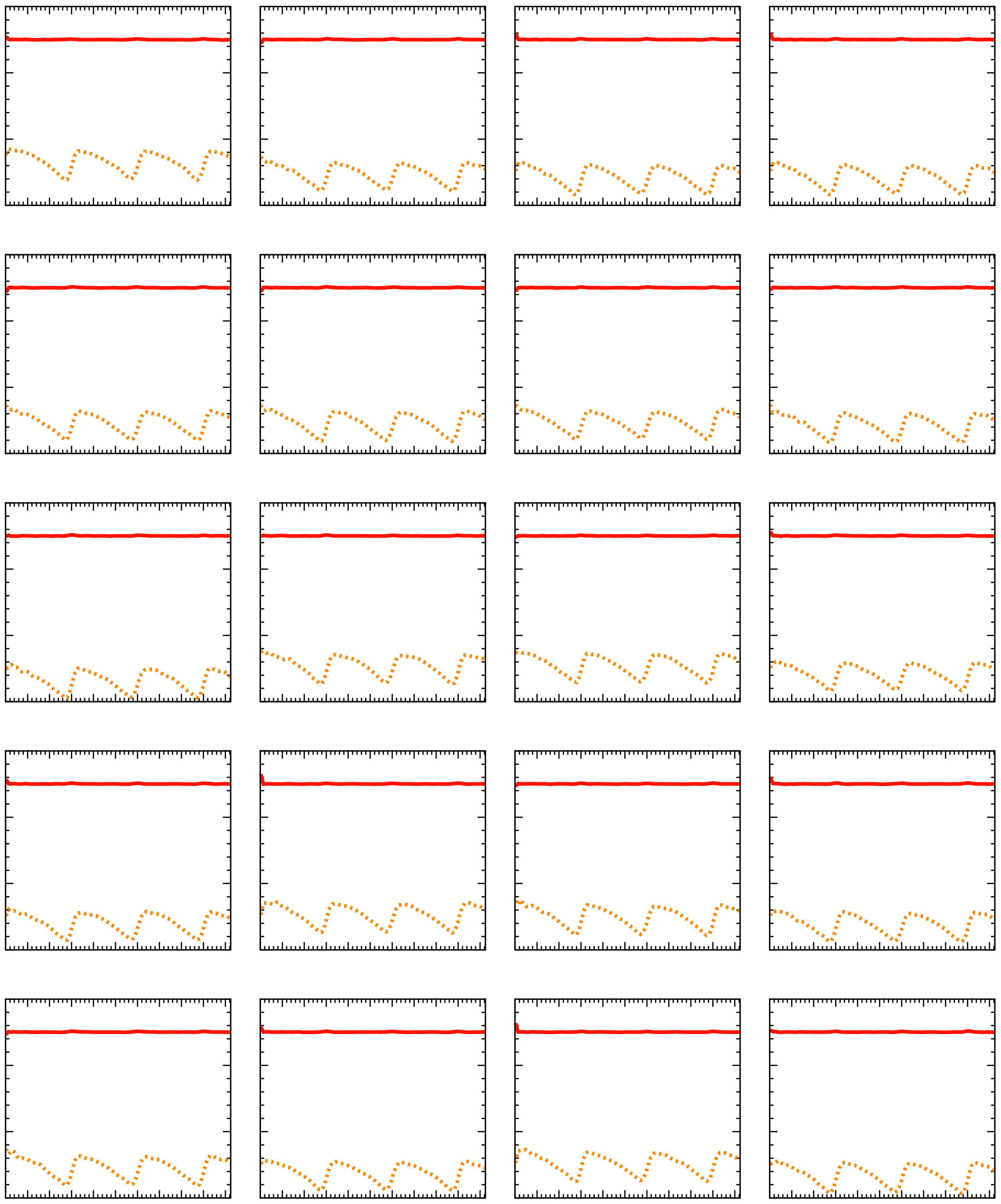


Figure 6.6: Performance of twenty drives operating in unison. The scale of each graph is the same, with the x-axis showing data processed (from 0 to 512 GB), and the y-axis showing bandwidth (from 0 to 60 MB/s).

chapter. Operating in unison, the sixty-nine drives have a combined data handling rate of just under 3.37 GB/s, and a total capacity of 10.1 terabytes (each disk having a useful capacity of about 150 GB). Maximum read performance, which depends on an equal distribution of read requests over all disks, is variable between about 100-800 MB/s depending on the exact operational scheme chosen. Figure 6.7 shows overall performance for large-scale operation for two different data-ordering methods.

Figure 6.7(a) shows the performance of large-scale Valmar when individual disks are coordinated such that their logical data ordering is identical in each case. Because all disks have similar size and performance characteristics, the net effect is that of a single disk performance curve, writ large. The write bandwidth is steady at the desired level. The minor variations previously seen in figure 5.3(c) are still present, but the scale of this graph does not reveal them. Read bandwidth looks exactly the same as a single disk, due to logical data position coordination between all disks.

In contrast, figure 6.7(b) shows the performance of large-scale Valmar when half of the disks are set to work in reverse logical data order. Again, write bandwidth is steady at the desired level, and all performance guarantees are made. Similar to figure 6.4, the read bandwidth has a much lower variance in this mode of operation, and performance is much more stable. Individual per-disk performance points are still highly variable, but overall performance is more predictable than with synchronized disks.

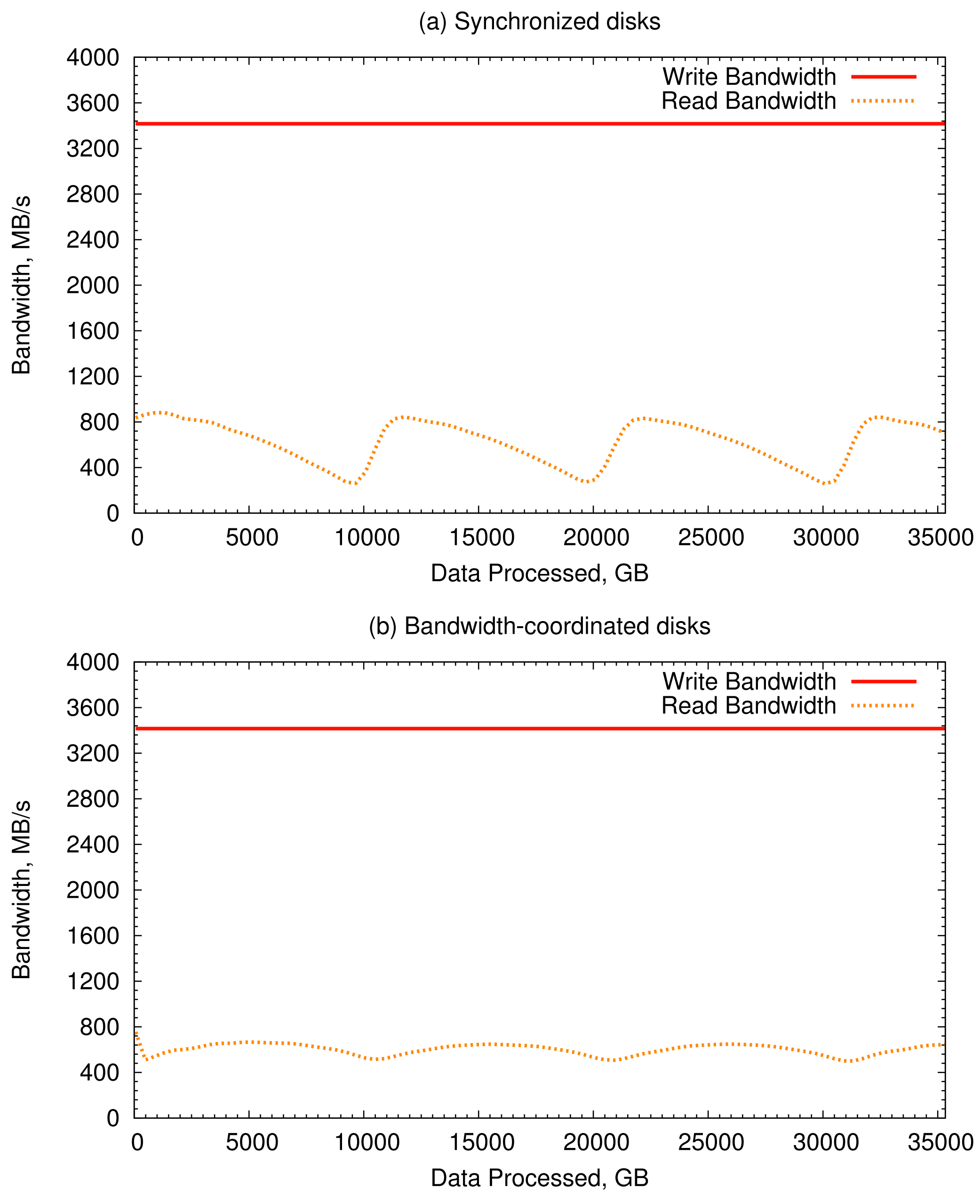


Figure 6.7: Valmar large-scale operation. This graph shows the results of coordinated Valmar processes running on 69 disks over 24 storage nodes with two different methods of disk coordination.

6.4 Conclusions

Large-scale storage systems have certain additional requirements beyond that of smaller single-drive and single-node setups, with reliability mechanisms and appropriate scalability being the most important. These additional requirements are needed to guarantee efficient performance, but cannot be allowed to interfere with the performance guarantees required in this mode of operation. Reliability mechanisms must not hinder normal operational requirements – even when operating in a degraded mode due to hardware failure – and any large system must scale appropriately such that the performance burden is distributed equally among all available devices and does not impose undue burden upon one, or a few, units only.

These additional requirements can be easily adapted to the management and indexing of high-bandwidth real-time streaming data, as detailed in previous chapters. Results demonstrate that not only may reliability mechanisms be adapted without compromising performance guarantees, but that they may also be used to "smooth" the aggregate bandwidth curves for reads and stabilize that performance within a node. The Valmar model is also shown to scale upward to a large number of nodes, including the ability to redistribute data allocation while continuing operation, and maintain the same steady performance that single-node operation offers.

Chapter 7

Conclusion

This dissertation has addressed several problems surrounding the management of high-bandwidth real-time streaming data. Specifically, it has answered the questions of how to make real-time guarantees with unreliable hardware, how to manage the indefinite operations of a transient data storage and management system, how to successfully index and search this data in a high-bandwidth environment, and how to combine all these techniques into a large-scale storage system. Integrating each of these concepts into a single body of work has yielded a comprehensive approach to handling this hard-to-manage data type.

As demonstrated in chapter 3, it is possible to make real-time guarantees on otherwise unreliable storage devices. Rotational disk drives, despite their unpredictable behavior, can be forced into conformance by careful profiling and data tuning to match access patterns to hardware constraints. These techniques allow them to be used at nearly their maximum rate of performance where other storage systems are often unable to reach even half of the disk's potential capabilities.

It has furthermore been demonstrated that the management of high-volume fleeting data is possible through a careful schedule of data chunking and layout control. The Mahanaxar data management model allows for the indefinite management of incoming data in an environment where it cannot be long preserved. Performance evaluation shows that this model can be used to achieve far better results than a conventional filesystem utilizing flat file based storage. This model is capable of making hard performance guarantees, allowing concurrent write and read access to the disk, and continuously operating at near-maximum potential.

This approach has been further extended to complex data types which require multiple indexing vectors and rapid search procedures. The Valmar extension of the Mahanaxar model allows very complex data elements to be arbitrarily indexed and a configurable approach allows for various performance levels in query response time. A real-time response model further allows the ongoing adaptation of indexing to better serve future queries. None of these techniques compromise quality of service guarantees, and offer a combination of performance and capability which exists in no other system, including databases. These methods have been integrated with conventional storage mechanisms to create a large-scale storage system capable of operating over many individual disks.

Thus, it is possible to construct a comprehensive system capable of supporting the indefinite management of high-bandwidth real-time streaming data where previous approaches have had no success.

Bibliography

[1] Advanced Television Systems Committee, Inc. *A/53: ATSC Digital Television Standard, Parts 1-6, 2007*, 3 January 2007.

[2] Guillermo A. Alvarez, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Trans. Comput. Syst.*, 19:483–518, November 2001.

[3] ARGUS FAQ. `http://www.qosient.com/argus/faq.shtml`, 2011.

[4] J. Baulier, P. Bohannon, S. Gogate, C. Gupta, and S. Haldar. Datablitz storage manager: main-memory database performance for critical applications. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, SIGMOD '99, pages 519–520, New York, NY, USA, 1999. ACM.

[5] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming B-trees. In

Proceedings of the nineteenth annual ACM symposium on parallel algorithms and architectures, SPAA '07, pages 81–92, New York, NY, USA, 2007. ACM.

[6] D. Bigelow, S. Brandt, J. Bent, and H.B. Chen. Mahanaxar: Quality of service guarantees in high-bandwidth, real-time streaming data storage. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1 –11, May 2010.

[7] D. Bigelow, S. Brandt, J. Bent, and H.B. Chen. Valmar: High-bandwidth real-time streaming data management. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1 –6, April 2012.

[8] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: an optimal scheme for tolerating double disk failures in raid architectures. In *Proceedings of the 21st annual international symposium on Computer architecture*, ISCA '94, pages 245–254, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[9] P. Bosch, S.J. Mullender, and P.G. Jansen. Clockwise: a mixed-media file system. In *Multimedia Computing and Systems, 1999. IEEE International Conference on*, volume 2, pages 277 –281 vol.2, July 1999.

[10] Boulder Real Time Technologies, Inc., Boulder, CO. *Antelope: ARTS configuration and operations manual*, November 1998.

[11] Scott Brandt, Carlos Maltzahn, Neoklis Polyzotis, and Wang-Chiew Tan. Fusing data management services with file systems. In *Proceedings of the 4th Annual*

Workshop on Petascale Data Storage, PDSW '09, pages 42–46, New York, NY, USA, 2009. ACM.

[12] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *IEEE International Conference on Multimedia Computing and Systems*, pages 400–405 vol 2., June 1999.

[13] John S. Bucy, Jiri Schindler, Steven W. Schlosser, and Gregory R. Ganger. The disksim simulation environment version 4.0 reference manual. Technical Report CMU-PDL-08-101, Parallel Data Laboratory, Carnegie Mellon University, May 2008.

[14] Mingming Cao, Suparna Bhattacharya, and Ted Ts'o. Ext4: The next generation of ext2/3 filesystem. In *Proceedings of the 2007 Linux Storage and Filesystem Workshop*. USENIX Association, February 2007.

[15] Adrian Chadd. http://devel.squid-cache.org/coss/coss-notes.txt, 2005.

[16] D.D. Chambliss, G.A. Alvarez, P. Pandey, D. Jadav, Jian Xu, R. Menon, and T.P. Lee. Performance virtualization for large-scale storage systems. In *Proceedings, 22nd International Symposium on Reliable Distributed Systems*, pages 109 – 118, October 2003.

[17] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable:

A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26:1–4, June 2008.

[18] P.M. Chen and D.A. Patterson. Storage performance-metrics and benchmarks. *Proceedings of the IEEE*, 81(8):1151 –1165, aug 1993.

[19] Cisco Systems. Introduction to Cisco IOS NetFlow - A Technical Overview. Technical Report C17-408326-01, Cisco Systems, October 2007.

[20] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST 04)*, pages 1–14, 2004.

[21] DataDirect Networks. Best Practices for Architecting a Lustre-Based Storage Environment. Technical report, DataDirect Networks, 2008.

[22] Western Digital. Advanced format technology. Technical Report 2579-771430-A02, Western Digital Technologies, Inc., April 2010.

[23] H. Frank. Analysis and optimization of disk storage devices for time-sharing systems. *J. ACM*, 16(4):602–620, October 1969.

[24] Simson Garfinkel. Network Forensics: Tapping the Internet. `http://www.oreillynet.com/pub/a/network/2002/04/26/nettap.html?page=1`, 2002.

[25] Robert Geist and Stephen Daniel. A continuum of disk scheduling algorithms. *ACM Trans. Comput. Syst.*, 5(1):77–92, January 1987.

[26] D.J. Gemmell, H.M. Vin, D.D. Kandlur, P. Venkat Rangan, and L.A. Rowe. Multimedia Storage Servers: A Tutorial and Survey. *IEEE Computer*, 28(5):40 – 49, May 1995.

[27] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole, Jr. Semantic file systems. In *Proceedings of the thirteenth ACM symposium on operating systems principles*, SOSP '91, pages 16–25, New York, NY, USA, 1991. ACM.

[28] K. Goda and M. Kitsuregawa. The history of storage systems. *Proceedings of the IEEE*, 100(Special Centennial Issue):1433 –1440, 13 2012.

[29] Burra Gopal and Udi Manber. Integrating content-based access mechanisms with hierarchical file systems. In *Proceedings of the third symposium on operating systems design and implementation*, OSDI '99, pages 265–278, Berkeley, CA, USA, 1999. USENIX Association.

[30] LHC Computing Grid. Gridbriefings: Grid computing in five minutes, August 2008.

[31] David D. Grossman and Harvey F. Silverman. Placement of records on a secondary storage device to minimize access time. *J. ACM*, 20(3):429–438, July 1973.

[32] Ajay Gulati, Ganesha Shanmuganathan, Irfan Ahmad, Carl Waldspurger, and Mustafa Uysal. Pesto: online storage performance management in virtualized datacenters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 19:1–19:14, New York, NY, USA, 2011. ACM.

[33] James Lee Hafner. WEAVER Codes: Highly Fault Tolerant Erasure Codes for Storage Systems. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, pages 16–16, Berkeley, CA, USA, 2005. USENIX Association.

[34] Hitachi Global Storage Technologies, San Jose, California. *Deskstar 7K4000*, 2012.

[35] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young. The automatic improvement of locality in storage systems. *ACM Trans. Comput. Syst.*, 23(4):424–473, 2005.

[36] http://www.phys.unm.edu/˜lwa/index.html.

[37] Yu Hua, Hong Jiang, Yifeng Zhu, Dan Feng, and Lei Tian. Smartstore: a new metadata organization paradigm with semantic-awareness for next-generation file systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 10:1–10:12, New York, NY, USA, 2009. ACM.

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

[39] B. Kao and H. Garcia-Molina. An overview of real-time database systems. Technical Report 1993-6, Stanford University, 1993.

[40] Stefan Kornexl, Vern Paxson, Holger Dreger, Anja Feldmann, and Robin Sommer. 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*, pages 23–23, Berkeley, CA, USA, 2005. USENIX Association.

[41] Elie Krevat, Joseph Tucek, and Gregory R. Ganger. Disks are like snowflakes: no two are alike. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, HotOS'13, pages 14–14, Berkeley, CA, USA, 2011. USENIX Association.

[42] Kam-Yiu Lam and Tei-Wei Kuo, editors. *Real-time Database Systems: Architecture and Techniques*. Springer-Verlag New York, 1 2001.

[43] Andrew W. Leung, Minglong Shao, Timothy Bisson, Shankar Pasupathy, and Ethan L. Miller. Spyglass: fast, scalable metadata search for large-scale storage systems. In *Proccedings of the 7th conference on file and storage technologies*, pages 153–166, Berkeley, CA, USA, 2009. USENIX Association.

[44] Christopher R. Lumb, Arif Merchant, and Guillermo A. Alvarez. Façade: Virtual storage devices with performance guarantees. In *Proceedings of the 2nd Usenix Conference on File and Storage Technologies*, March 2003.

[45] Christopher R. Lumb, Jiri Schindler, and Gregory R. Ganger. Freeblock scheduling outside of disk firmware. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST '02)*, Monterey, California, January 2002.

[46] Christopher R. Lumb, Jiri Schindler, Gregory R. Ganger, Erik Riedel, and David F. Nagle. Towards higher disk head utilization: Extracting free bandwidth from busy

disk drives. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, California, October 2000.

[47] J. Menon and D. Mattson. Distributed sparing in disk arrays. In *Compcon Spring '92. Thirty-Seventh IEEE Computer Society International Conference, Digest of Papers.*, pages 410 –421, February 1992.

[48] A. Molano, K. Juvva, and R. Rajkumar. Real-time filesystems. Guaranteeing timing constraints for disk accesses in RT-Mach. In *The 18th IEEE Real-Time Systems Symposium*, pages 155–165, December 1997.

[49] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to SSDs: analysis of tradeoffs. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 145–158, New York, NY, USA, 2009. ACM.

[50] Inc. Narus. Narus intercept solution. `http://www.narus.com/index.php/solutions/intercept`.

[51] Panasas. `http://www.panasas.com/products/software.php`.

[52] A. Parsons, D. Werthimer, D. Backer, T. Bastian, G. Bower, W. Brisken, H. Chen, A. Deller, T. Filiba, D. Gary, L. Greenhill, D. Hawkins, G. Jones, G. Langston, J. Lasio, J. Van Leeuwen, D. Mitchell, J. Manley, A. Siemion, H. K.-H. So, A. Whitney, D. Woody, M. Wright, and K. Zarb-Adami. Digital Instrumentation for the

Radio Astronomy Community. In *astro2010: The Astronomy and Astrophysics Decadal Survey*, volume 2010 of *ArXiv Astrophysics e-prints*, page 21, 2009.

[53] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, SIGMOD '88, pages 109–116, New York, NY, USA, 1988. ACM.

[54] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience*, 27(9):995–1012, September 1997.

[55] Anna Povzner, Tim Kaldewey, Scott Brandt, Richard Golding, Theodore M. Wong, and Carlos Maltzahn. Efficient guaranteed disk request scheduling with fahrrad. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 13–25, New York, NY, USA, 2008. ACM.

[56] Anna Povzner, Darren Sawyer, and Scott Brandt. Horizon: efficient deadline-driven disk i/o management for distributed storage systems. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 1–12, New York, NY, USA, 2010. ACM.

[57] A. Rajasekar, M. Wan, R. Moore, G. Kremenek, and T. Guptil. Data grids, collections, and grid bricks. In *Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*, pages 2 – 9, April 2003.

[58] Arcot Rajasekar, Sifang Lu, Reagan Moore, Frank Vernon, John Orcutt, and Kent Lindquist. Accessing sensor data using meta data: a virtual object ring buffer framework. In *DMSN '05: Proceedings of the 2nd international workshop on data management for sensor networks*, pages 35–42, New York, NY, USA, 2005. ACM.

[59] Raju Rangaswami, Zoran Dimitrijević, Edward Chang, and Klaus Schauser. Building mems-based storage systems for streaming media. *Trans. Storage*, 3(2):6, 2007.

[60] F. Reiss, K. Stockinger, Kesheng Wu, A. Shoshani, and J.M. Hellerstein. Enabling real-time querying of live and historical stream data. In *Scientific and Statistical Database Management, 2007. SSBDM '07. 19th International Conference on*, page 28, July 2007.

[61] L. Reuther and M. Pohlack. Rotational-position-aware real-time disk scheduling using a dynamic active subset (DAS). In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 374 – 385, dec. 2003.

[62] Bianca Schroeder and Garth A. Gibson. Understanding disk failure rates: What does an mttf of 1,000,000 hours mean to you? *Trans. Storage*, 3(3), October 2007.

[63] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. In *Proceedings of the 1990 Winter Usenix.*, pages 313–324, 1990.

[64] Prashant J. Shenoy and Harrick M. Vin. Cello: a disk scheduling framework for next generation operating systems. In *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*,

SIGMETRICS '98/PERFORMANCE '98, pages 44–55, New York, NY, USA, 1998. ACM.

[65] A. Singh, M. Korupolu, and D. Mohapatra. Server-storage virtualization: Integration and load balancing in data centers. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1 –12, nov. 2008.

[66] Craig A. N. Soules and Gregory R. Ganger. Connections: using context to enhance file search. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 119–132, New York, NY, USA, 2005. ACM.

[67] M.J. Stanovich, T.P. Baker, and A.-I.A. Wang. Throttling on-disk schedulers to meet soft-real-time requirements. In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, pages 331 –341, April 2008.

[68] STEC. Benchmarking enterprise ssds. Technical Report 61000-006002-005, STEC, Inc., March 2012.

[69] Murray Stokely, Amaan Mehrabian, Christoph Albrecht, Francois Labelle, and Arif Merchant. Projecting disk usage based on historical trends in a cloud environment. In *Proceedings of the 3rd workshop on Scientific Cloud Computing Date*, ScienceCloud '12, pages 63–70, New York, NY, USA, 2012. ACM.

[70] The Nielsen Company. Snapshot of Television Use in the U.S., September 2010.

[71] Sameer Tilak, Paul Hubbard, Matt Miller, and Tony Fountain. The Ring Buffer

Network Bus (RBNB) DataTurbine Streaming Data Middleware for Environmental Observing Systems. In *e-Science*, Bangalore, India, 10/12/2007.

[72] Martin Žádnik, Petr Špringl, and Pavel Čeleda. Flexible FlowMon. Technical Report 36/2007, CESNET, 2007.

[73] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance insulation for shared storage servers. In *Proceedings of the Fifth Conference on File and Storage Technologies (FAST '07)*, San Jose, California, February 2007.

[74] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, November 1994.

[75] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[76] Western Digital Technologies, Inc., Irvine, California. *WD Caviar Black Desktop Hard Drives*, June 2012.

[77] John Wilkes. Traveling to rome: Qos specifications for automated storage system management. In *International Workshop on Quality of Service*, pages 75–91. Springer-Verlag, 2001.

[78] T.M. Wong, R.A. Golding, Caixue Lin, and R.A. Becker-Szendy. Zygaria: Storage performance as a managed resource. In *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pages 125 – 134, April 2006.

[79] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, and John Wilkes. On-line extraction of scsi disk drive parameters. In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '95/PERFORMANCE '95, pages 146–156, New York, NY, USA, 1995. ACM.

[80] J.C. Wu and S.A. Brandt. Providing quality of service support in object-based file system. In *24th IEEE Conference on Mass Storage Systems and Technologies*, pages 157–170, September 2007.

[81] Qin Xin, E.L. Miller, and T.J.E. Schwarz. Evaluation of distributed recovery in large-scale storage systems. In *Proceedings, 13th IEEE International Symposium on High Performance Distributed Computing*, pages 172 – 181, June 2004.