THE UNIVERSITY OF NEW SOUTH WALES

SCHOOL OF ELECTRICAL ENGINEERING AND TELECOMMUNICATIONS

I/O Scheduling on RAID

Aaron Carroll

Thesis submitted as a requirement for the degree
Bachelor of Engineering (Electrical Engineering)

Submitted: June 3, 2008
Supervisor: Peter Chubb

Student ID: 3132871

Abstract

The storage world is facing a crisis: while the quantity and cost of storage is improving exponentially, the speed of accessing stored data is not. Two traditional approaches to solving this problem are bigger and smarter hardware (like RAID), and intelligent ways of using existing resources (such as I/O scheduling). The aim of this thesis is to investigate how to marry these two techniques.

In the context of the Linux kernel, we benchmark ten I/O schedulers on a range of RAID configurations and workloads to determine which techniques are applicable to host-based I/O scheduling on hardware RAID devices. We also implement two new schedulers for Linux, and solve several performance issues in the existing ones.

Our results show that I/O scheduling can generally improve performance using the techniques that have been employed for many years. We show that Linux's deadline scheduler generally performs best, and describe some optimisations to improve performance for the other schedulers.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The storage world has, for some time, been facing a crisis; the quantity of storage available has been growing exponentially, while the speed of accessing it has not. Figure 1.1 depicts this growing disparity. While the cost per byte and areal density (that is, bytes per unit disk area) have improved by factors of 100 000 and 10 000 since 1980, respectively, the speed of accessing the data—transfer rate and access time—have improved only by factors of 100 and 10. To make matters worse, processor speeds and memory capacity have been growing by similarly

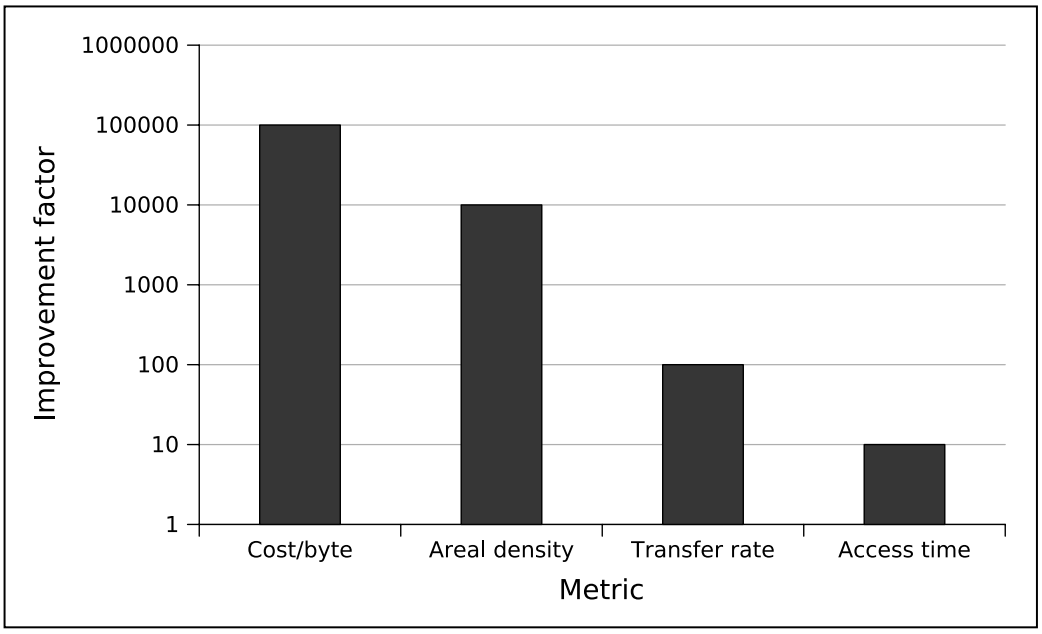

Figure 1.1: Hard disk improvement since 1980 [GH03].

high rates. As a consequence, there are huge quantities of storage available and the system capacity to process that data, but the relative speed of accessing it is slow, and getting slower.

There are two ways in which this problem has traditionally been tackled. The first is to add more hardware to hide performance problems. RAID, for instance, combines multiple physical

disks into a single volume with improved performance and reliability characteristics. This is often done in dedicated hardware which abstracts away such functionality, presenting a disk-like device to the host system.

The second approach is I/O scheduling, which aims to improve performance and quality of service by more efficient use of available disk resources. This is achieved by two main techniques: reordering of disk requests and merging of adjacent requests into single larger requests. While I/O scheduling algorithms have been developing since the 1960s, they all make assumptions about the underlying disk hardware:

1. the logical data layout presented to the host system corresponds to the physical layout;
2. one read/write head, meaning only one location on the disk can be manipulated at any moment;
3. the device has knowledge of a single data request at any time.

These assumptions are sufficiently accurate according to the traditional model of the hard disk.

Both RAID and I/O scheduling can be combined to further improve storage system performance. RAID, however, invalidates all three assumptions of the traditional disk model on which I/O scheduling algorithms are based. To what extent, then, are these algorithms applicable to RAID devices? We hope to answer this question.

We will be taking a measurement-based approach to evaluating I/O scheduler performance using real RAID hardware and I/O schedulers. In particular we will be working on the Linux kernel, which provides four different I/O schedulers. It also provides a fully modular I/O scheduler framework, which allows us to develop new schedulers with relatively little effort.

Using this framework we will develop several new schedulers which implement some of the classic algorithms used in many performance evaluations, and on which real implementations are based. Using these, we can study I/O scheduling on hardware RAID from an implementation-independent perspective, largely free from the requirements of real-world schedulers. In addition, we will study Linux's default schedulers, which are used in millions of production machines. We hope to improve performance of these schedulers and provide guidance to those wanting to use them on RAID devices.

The study will proceed in three parts. Initially, we will study, document, and improve the current state of Linux's four I/O schedulers. We will then benchmark them under a variety of loads and RAID configurations, along with the new schedulers developed for that purpose. Finally, we will analyse the benchmarks and try to identify which of the classic I/O scheduling techniques apply to hardware RAID, while also providing a practical analysis of Linux's schedulers and how they perform on such platforms.

This work is organised as follows. In Chapter 2, we survey the literature on the relevant topics: disk and RAID hardware, I/O scheduling algorithms, and the Linux kernel. Chapter 3 looks at our modifications made to Linux to improve the I/O schedulers and to support the

benchmarks. Chapter 4 describes these benchmarks, and details the hardware and software platform on which they were run. Chapter 5 presents the results of the benchmarks which are analysed in Chapter 6. Finally, Chapter 7 summarises the findings of the study.

Chapter 2

Background

In this section, we review the literature relevant to the remainder of the work. Specifically, Section 2.1 details a model of the hard disk drive, concentrating on properties relevant to I/O scheduling. Section 2.2 describes the purpose and techniques of I/O scheduling and how they have been influenced by the traditional model the hard disk. It then goes on to detail the classic I/O scheduling algorithms, and others relevant to our work. Section 2.3 looks at RAID, including the common configurations and implementations. Finally, in Section 2.4, we describe the relevant components of the Linux kernel, including the I/O schedulers, and look at two previous studies of their performance.

2.1 Disk Model

2.1.1 Physical construction

A modern disk drive consists of a set of rotating coaxial discs, called *platters*, which are coated with a magnetic material onto which data is stored [RW93a]. Typically, both sides of the platter are used, giving two recording surfaces per platter. Data is read and written by a magnetically-sensitive *head*, one per surface, which is mounted at the end of an arm. The arms are coupled to an actuator which moves the heads to the desired angle, a processes known as *seeking*. Only a single head can be active (that is, reading or writing) at any particular time, and since all heads are coupled to the same actuator, they cannot be moved independently. This structure is shown in Figure 2.1.

2.1.2 Data geometry

Data on disk is laid out according to three geometrical attributes. A *sector* is the smallest addressable unit of disk storage, which contains raw data, usually 512 bytes, and information for synchronisation and error-checking [Tan01]. The set of sectors equidistant from the centre of a particular surface form a *track*. Thus, a track is all sectors that can be accessed by a particular head without requiring a seek. The corresponding (i.e. equal radius) tracks from all disk surfaces

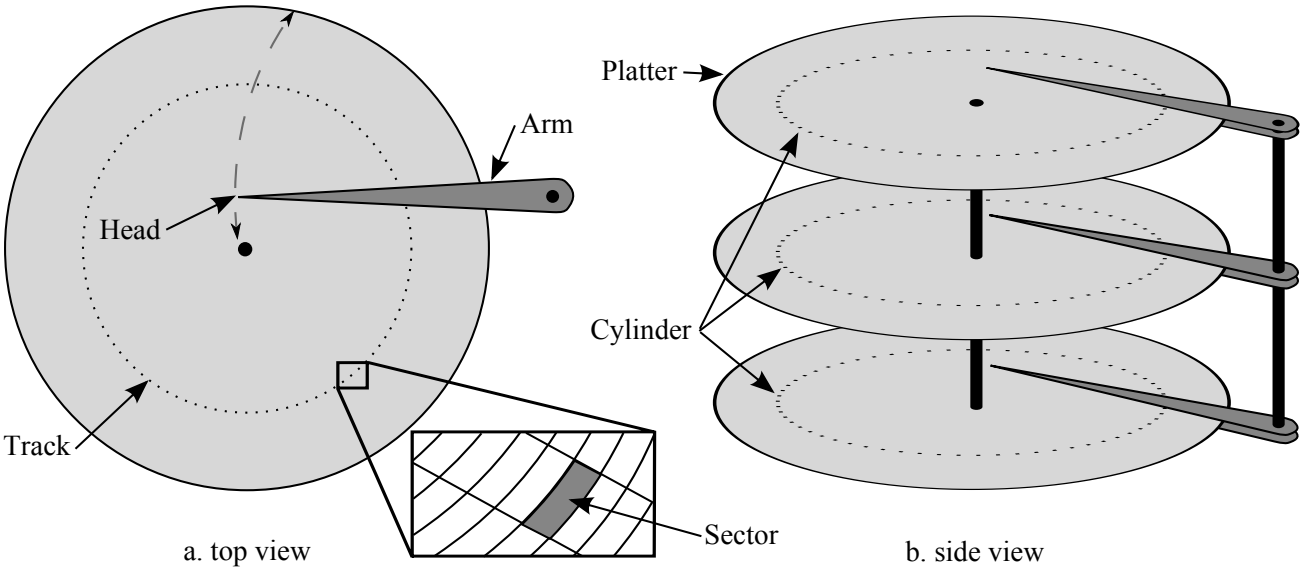

Figure 2.1: Physical structure and data geometry for a three-platter disk.

form a *cylinder*—the set of all sectors that can be accessed without seeking. A sector can be uniquely identified by its cylinder, head (or, equivalently, surface), and sector number; the CHS.

Modern disks, however, typically export an abstract addressing mode to the operating system called LBA, or *logical block addressing*, where the disk is presented as a linear array of blocks. Requests to the disk are made in terms of a *logical block number*, or LBN, which uniquely identifies each block. The LBN to CHS mapping is performed transparently by the disk controller (see Section 2.1.3), but this mapping is complicated by several optimisation and error handling techniques, discussed in the following paragraphs.

Skew When seeking between adjacent tracks, mechanical delay may cause the head to miss the first sector of the next track, requiring an idle period until that sector again rotates under the head. When employing *track skewing*, the position of the first sector of each track is offset from the previous one such that track zero is not missed after a single-cylinder seek. *Head skew* accounts for the latency in switching the active head by offsetting each surface from the previous.

Sector remapping Failed sectors are a common occurrence, during both manufacturing and normal usage. When a sector is detected bad, a new sector is drawn from a pool of spares and remapped over the failed one. Future references to the failed sector are then redirected to the replacement. This results in a LBA to CHS mapping that changes over time, but is completely transparent to the host system.

Zoned bit recording Due to the shape of a disk surface, tracks toward the outer edge are longer than those toward the centre, and hence the number of sectors per track can be increased for tracks toward to the edge of the disk. This is called ZBR, or *zoned bit recording*, and requires the disk controller to keep a mapping between tracks and the number of sectors they contain. To minimise this overhead, the disk is divided into *zones*, groups of contiguous tracks, for which the number of sectors is constant.

As a side-effect of ZBR, the maximum rate of data transfer from the medium is not constant. The rotational velocity of a disk is fixed, and the sector density (approximately) constant. As a result, the linear velocity of sectors passing the head increases the further it is from the centre, and thus the maximum throughput increases for zones toward the edge of the disk.

2.1.3 Controllers

Physical disk management is performed by a microprocessor called the *disk controller*. It provides low-level data storage and retrieval via a host interface such as SCSI, controls the disk head and positioning hardware, and manages the on-board memory which is used for caching, buffering, and request queueing. Only the latter three features will be discussed in more detail, since they are the most important with respect to I/O scheduling.

Caching Read caching is an optimisation technique that exploits locality in disk access patterns by keeping certain sectors in memory. Future access to those sectors can be satisfied out of the cache without requiring access to the physical medium, and can be done with orders of magnitude greater bandwidth, since costly mechanical operations are avoided (see Section 2.1.4).

A *prefetching* (or *readahead*) cache improves sequential read performance by reading more data from the disk than was requested. One such technique is to always read full tracks, regardless of the individual request size (so-called *track buffering*). This is an efficient caching scheme, since no extra seeking is required. Track buffering can also be employed to limit rotational delay by reading the track immediately after the head is positioned, rather than waiting for the starting sector. This is called *on-arrival readahead*.

Another caching policy is to keep recently read sectors in controller memory in the hope that they will be read again in the near future. However, the effectiveness of this cache type is somewhat limited, as modern systems typically employ a *buffer cache* (or *page cache*), where disk data is cached in main memory.

Buffering Under write-back buffering, the disk notifies the host of write request completion immediately after receiving it. To the host, the write completes at cache speed and can continue processing. However, the request is actually queued in the disk controller to be written to the medium at some later point. Under a light write load, this can improve performance due to lower perceived latency. Under heavy load, the buffer fills and performance drops to the level sustainable by the physical medium.

Write-back buffering has several disadvantages. Reliability can be compromised since the host view of the disk can differ from its actual state, and this can result in corruption if the system fails. Additionally, it is possible that write-back buffering results in decreased performance due to the *adversary effect* [RLLR07].

Queueing Command queueing[1] allows a disk to accept and process multiple requests in parallel. The maximum number of outstanding requests is called the *queue depth*. Command queueing can improve performance, but it has important implications for host I/O scheduling.

The first benefit of command queueing is the ability to reorder requests. Since the disk has full knowledge of the platter and head positions, it can modify the order in which requests are serviced to minimise the amount of seeking required, and thus improve throughput.

Another benefit is the ability to overlap disk access with bus I/O [RP03]. For example, while data is being read from the medium for one request, the controller can be receiving the next request from the host.

2.1.4 Sources of latency

The time taken to service a disk request can be attributed to three sources.

Mechanical delay The mechanical delay accounts for the latency due to the intrinsic physical limitations of the moving disk components, and can be broken into two factors: *positioning time* and *transfer time*.

Positioning time (or *access time*) is the time to position the disk head over the desired sector, and has two contributing factors. The first, *seek time*, is the time taken for the arm actuator to move the heads to the desired cylinder. It is a non-linear function of the cylinder distance, arm mass, platter size, actuator power, and other physical disk properties. The second is *rotational delay*, which is the latency incurred while waiting for the target sector to rotate under the head once it is positioned over the desired track.

Transfer time is the time required to read or write data from the physical medium. It is a function of the disk rotational velocity, and the size of the transfer being performed.

Electronic delay Electronic delay encompasses a range of intrinsic delay sources occurring in the controller, including:

1. bus arbitration and transfer;
2. signal processing;
3. error detection and correction;

[1]Command queueing is called TCQ (*tagged command queueing*) in SCSI disks and NCQ (*native command queueing*) in SATA disks.

4. head and platter position calculations;

5. buffer, cache and queue management.

Queueing Using command queueing, controllers can have multiple requests outstanding, only one of which may be operating on the physical medium. As a result, requests can spend time queued on the controller waiting for other requests to complete. This is called the *queueing delay*, and is determined by the disk controller's scheduling policy.

2.2 I/O Scheduling

The *I/O scheduler* is an operating system component whose purpose is to maximise disk performance and to provide quality of service (QoS) between competing disk users. In this section, we outline the techniques, goals, and assumptions of I/O scheduling algorithms, and then describe the classic algorithms which form the basis of many modern I/O scheduling techniques.

2.2.1 Mechanism

An I/O scheduler uses three main mechanisms to achieve its scheduling goals.

Reordering Request reordering is the primary technique employed in practically every I/O scheduler. It involves queueing requests received from applications generating I/O, and modifying the order in which they are subsequently issued to the disk.

Merging Two requests queued at the I/O scheduler can be *merged* if those requests are operating on a contiguous region of the disk. For example, two separate requests for adjacent sectors can be combined into a single, larger request for both sectors at once. This reduces the number of disk commands required.

Delaying An I/O scheduler may delay issuing a request to the disk drive, even if no other requests are pending. Such schedulers are called *non-work-conserving* [ID01], meaning they allow disk resources to go unused.

2.2.2 Performance

The first job of the I/O scheduler is to optimise performance, specifically bandwidth, which it does by exploiting physical characteristics of disk access. As discussed in Section 2.1.4, servicing an I/O request incurs certain delays which limit the performance attained. The most significant of these is the access time, which depends primarily on the geometrical location of consecutive requests on the disk medium. As such, the order in which requests are executed can have a significant impact on the bandwidth extracted from the disk. By reordering requests, one can

reduce the time spent positioning the disk head, and hence increase bandwidth, since more time is spent transferring data.

Request merging can also contribute to bandwidth improvement. Since fewer commands are required, command overhead is reduced [LSG02]. Additionally, if the merged requests are physically contiguous, they can be read or written in a single rotation.

It is believed that the optimal algorithm for the I/O scheduling problem is NP-complete [JW91]. While Andrews et al. proved that the offline[2] algorithm is NP-hard [ABZ02], no such proof is known for the online variant.

2.2.3 QoS

The second function performed by the I/O scheduler is managing quality of service. This is provided by reordering or delaying requests to satisfy some QoS policy. Service quality metrics vary, but may include:

- fair distribution of bandwidth or disk time;
- fixed bandwidth or disk time allocation;
- bounded latency.

These guarantees may be provided to a range of entities, including users, processes, individual requests, or logical blocks.

Different QoS-based scheduling algorithms provide varying levels of quality guarantee. For example, general purpose algorithms might make weak, best-effort attempts to satisfy a scheduling policy, while hard-real-time algorithms might provide strict guarantees. Additionally, the QoS received by each entity may differ according to some priority scheme.

2.2.4 Simplified disk model

The traditional I/O scheduling algorithms operate on a highly simplified model of the hard disk compared to that presented in Section 2.1. In this section, we present a range of approximations and assumptions that are often (implicitly) made, to varying degrees, in practical and theoretical I/O scheduling algorithms.

Strictly speaking, all of these assumptions are false in modern disk drives, and while this disk model may lead to decreased performance, there are several reasons why such a model is frequently used. Firstly, it reduces complexity in the scheduling algorithm. Secondly, it is relatively independent of the actual disks being used, which vary in physical data layout, controller memory size, cache policy, seek times, low-level optimisation, etc. Information required for more complex models is often not exposed outside the disk.

[2] An offline scheduling algorithm assumes all requests are known *a priori*.

Worthington et al. [WGP94] studied the effects of complex data mappings and prefetching caches on I/O scheduler performance. They concluded that knowing the full logical to physical address translation can improve scheduler performance by less than 2%, while effective use of prefetching caches can have a marked improvement.

Data layout Many I/O schedulers work on the assumption of an LBA-like physical data layout and access characteristic. This encapsulates two key concepts: logically-adjacent sectors are physically adjacent, and only a single disk head exists. This implies seek cost is a monotonically-increasing function of LBN.

Controllers I/O scheduling algorithms typically neglect the effect of the disk controller on algorithm behaviour; primarily caching and buffering. While Worthington's study showed this can be detrimental to performance, some algorithms are inherently cache-friendly. Command queueing combined with on-disk rescheduling, however, can subvert an OS-based I/O scheduler. As a result, command queueing must often be disabled for algorithms requiring strict ordering of requests at the disk medium [LSG02].

2.2.5 Classic algorithms

In this section, we present a range of classic I/O scheduling algorithms and summarise the literature comparing their performance. While innumerable other I/O schedulers exist, many are based on the techniques of these algorithms. Figure 2.2 compares how they respond to a particular workload by tracing head movement.

FCFS

FCFS, or *first come first served*, is the simplest I/O scheduler; requests are issued to the device in the order they arrive. This policy does not take advantage of disk characteristics. Given a random workload, it can be shown that, on average, FCFS must perform $\frac{1}{3}$ of a full-stroke seek per request [CKR72].

FCFS generally performs poorly. Jacobson and Wilkes [JW91] showed that, at best, it performs equally with position-aware schedulers, but is usually worse. Seltzer et al. [SCO90] showed that up to 3–4 times performance improvement over FCFS can be achieved with general-purpose algorithms.

FCFS has several advantages. It is trivial to implement, is free of starvation (since no request can "jump the queue") and has the lowest CPU overhead of any I/O scheduling algorithm. It can be used successfully on devices where traditional I/O scheduling techniques do not substantially affect performance, such as RAM-based devices. Finally, FCFS performs well under light workloads, where short queues leave little room for scheduling.

SSTF

Shortest seek time first (or SSTF) schedules the next request such that the seek distance is minimal over all currently queued requests. Seek distance is typically quantified as cylinder difference, but can be approximated as LBN difference [WGP94].

Hofri [Hof80] performed a detailed study of FCFS vs. SSTF performance. He concluded that:

1. SSTF always outperforms FCFS in terms of mean latency;
2. both algorithms have comparable coefficients of variation, with SSTF marginally higher;
3. latency variance in SSTF is lower than FCFS;
4. SSTF has better scalability under increasing load.

The major drawbacks of SSTF are starvation and latency variance. Under sufficiently heavy load, the head has a tendency to "stick" in one area of the disk. Requests outside this area can starve indefinitely, leading to high latency for some requests, and thus high latency variance.

SATF

Shortest access time first (SATF) schedules the next request such that its access time (see Section 2.1.4) is minimal over all currently queued requests. It is a generalisation of SSTF to include rotational delay, and can be considered to be a greedy approximation to the optimal scheduling algorithm.

Worthington et al. [WGP94] compared SATF and SSTF (among others), and concluded that while SATF has very high computational overhead, the improvement in latency is appreciable. They also showed that SATF has a higher coefficient of variance, and thus more susceptible to starvation than SSTF. Jacobson and Wilkes [JW91] showed that, in general, rotational-position-aware schedulers outperform seek-time based algorithms.

Pure SATF is difficult to implement in practice, since disks typically do not export rotational-position information. However, it can be implemented in disk controllers, where accurate positioning information is available [RLLR07].

SCAN

The *SCAN*[3] algorithm, also called *elevator*, moves the arm in one direction only until all requests in that direction have been satisfied. The arm then reverses direction and repeats the process. Thus, the arm "sweeps" in alternating directions across the disk.

[3]SCAN is not an acronym; it is capitalised by convention. In older literature, SCAN is called LOOK, while SCAN refers to a slightly different algorithm.

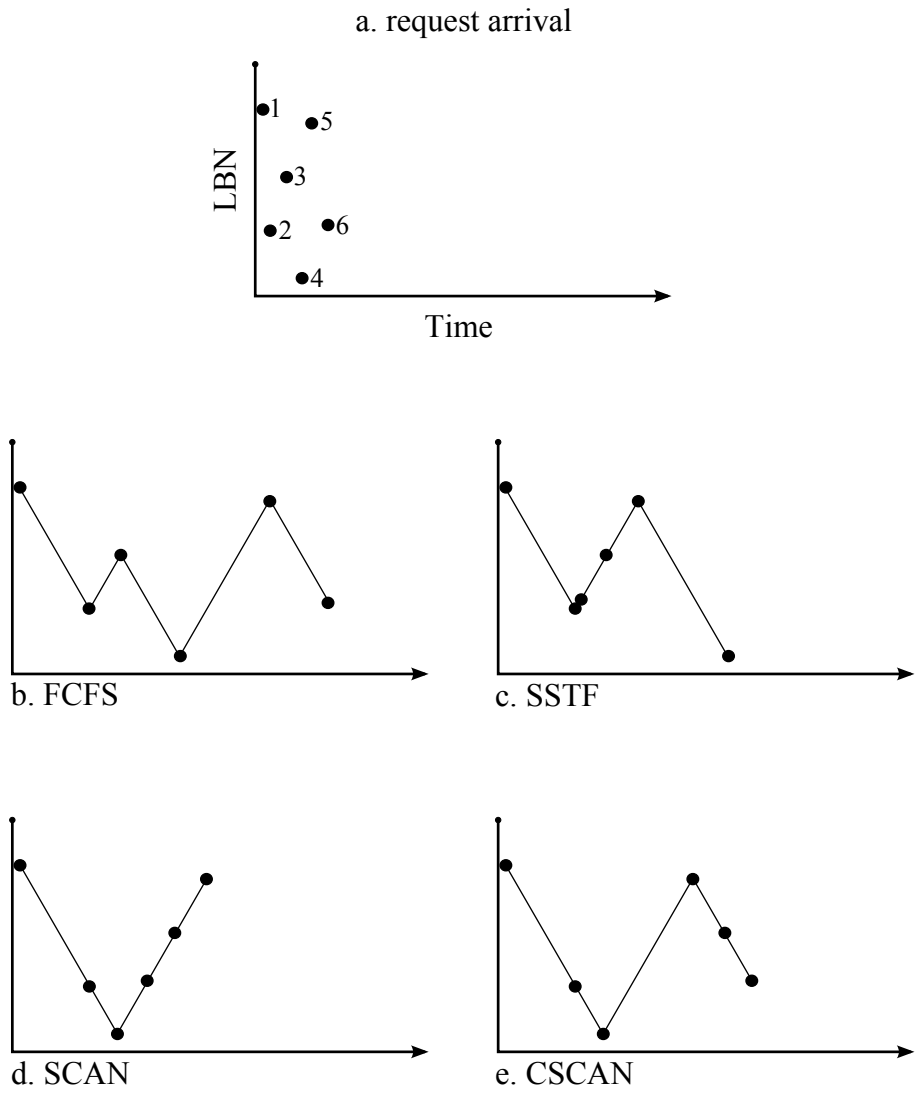

Figure 2.2: Head movement in I/O scheduling algorithms. Each dot represents a request. Figure (a) shows the point at which each request is queued at the I/O scheduler, numbered according to arrival time. Figures (b)–(e) show the resulting seek path for the FCFS, SSTF, SCAN and CSCAN algorithms.

Several studies [WGP94, TP72, SCO90] have shown SCAN and SSTF to have comparable performance characteristics. SCAN is more resistant to starvation—no request can starve indefinitely. However, it is not fair with respect to response time, since the maximum time between service is greater for sectors at the edge of the disk compared with those in the centre. Because prefetching caches are optimised for sequential access, SCAN makes more effective use of them compared with FCFS and SSTF.

CSCAN, or *circular SCAN*, is a variant of SCAN where the arm is constrained to move in the forward (that is, from low to high logical blocks) direction only. When the arm reaches the

last request, it performs a full-stroke seek and starts again from the first request.

Worthington et al. [WGP94] showed that CSCAN has lower variance in latency compared with SCAN, but achieves a lower mean. Teorey and Pinkerton [TP72] claim that SCAN is superior under lightly-loaded conditions, while CSCAN performs better under heavy load.

V(R)

The $V(R)$ scheduler, proposed by Geist [GD87], is a continuum of algorithms parametrised by R, a variable in the range $[0, 1]$. The next request is scheduled according to the SSTF algorithm, with an additive seek penalty of $R\times$(number of cylinders on disk) applied for reversing the head direction compared with the previous seek. Thus, the endpoints of the continuum are $V(0)$, which is pure SSTF, and $V(1)$, which is SCAN.

Via measurement, Geist showed that $V(0.2)$ consistently outperforms both SSTF and SCAN in terms of average throughput, and claims that its implementation is no more complex. He also showed that the selection of R is a trade-off between latency mean and variance; increasing R tends to reduce variance, while decreasing R reduces the mean. Finally, he showed that for $R > 0.4$, performance is indistinguishable from $V(1)$ (SCAN).

Anticipatory

Anticipatory scheduling is a relatively new algorithm, proposed by Iyer and Druschel [ID01] in 2001, which aims to improve performance by recognising and accounting for the *deceptive idleness* problem. It is not a complete I/O scheduling algorithm in itself, but rather augments an existing scheduler.

When applications issue synchronous[4] sequential I/O, there is a small period between requests in which the application is processing—the *think-time*. Normally, sequential I/O achieves good bandwidth, but when multiple streams of I/O are present, a seek-optimising scheduler will service other requests during the think-time. As a result, the performance advantage of sequential I/O is lost. This is called deceptive-idleness.

The anticipatory scheduler solves this problem by detecting sequential synchronous I/O, and inserting short periods of idleness (that is, no request is scheduled) if a sequential request is likely to soon follow the previous one. If this occurs, an expensive seek is avoided. However, any incorrect anticipation results in lost bandwidth.

Despite forced periods of disk inactivity, a well-implemented anticipatory scheduler can provide a significant performance improvement over a non-anticipatory seek-optimising scheduler such as SCAN. Iyer and Druschel showed a consistent performance improvement in a range of micro-benchmarks and real workloads of up to 70% compared to a variant of SATF. However, as a trade-off for increased overall bandwidth, anticipatory scheduling can cause increased latency variance and unfair bandwidth distribution.

[4] A synchronous workload is characterised by a single outstanding request.

2.3 RAID

Redundant array of inexpensive disks[5], or *RAID*, is a technique for solving some of the performance, reliability and capacity problems of traditional single-disk storage systems. A RAID *array* is a set of independent disks, transparently combined into a single logical disk, which achieves the above goals by intelligently distributing data and extra redundancy information over the physical disks.

In their classic paper, Patterson et al. [PGK88] propose five RAID configurations, called *levels*, which vary in their performance, space efficiency, and degree of redundancy. Since then, several additional levels have been adopted as standard, while some of the original levels have seen little practical use.

In this section, we will discuss the basic techniques and terminology of RAID, the RAID levels commonly implemented in modern disk arrays, and the two main implementation methods.

2.3.1 Data striping

In a striped RAID scheme, the physical storage space is broken up into fixed-size chunks called *stripe units*, typically 1–1024 KiB in size (the *stripe size*), which are contiguous within a single physical disk. Stripe units are then grouped into sets called *stripes*, typically formed from the corresponding unit from each disk in the array. A stripe contains a set of data and its associated redundancy information. Stripe units are assigned in a round-robin fashion to the available disks such that the stripe units from a single stripe are logically contiguous. Figure 2.3 shows a graphical representation of data striping.

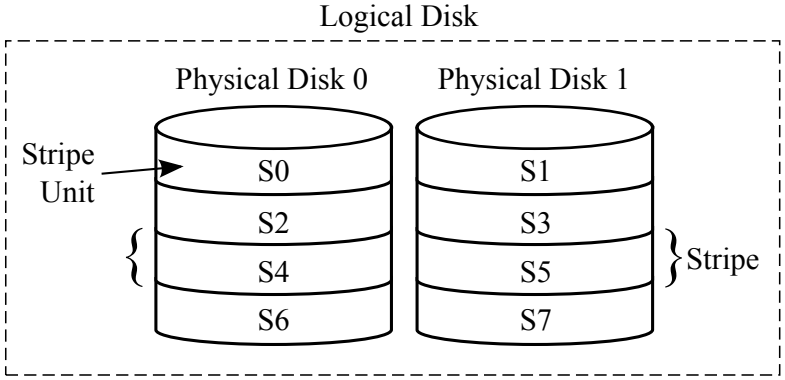

Figure 2.3: Structure of a data striping RAID array consisting of a single logical disk formed by two physical disks with four stripe units per disk. The corresponding stripe units from each disk (S4 and S5, for example) form a single stripe.

The primary aim of data striping is improved performance [CLG+94]. Since a striped RAID array consists of multiple physical disks, it is inherently capable of servicing multiple independent

[5]Often called redundant array of *independent* disks.

requests in parallel. Additionally, due to the round-robin stripe unit placement strategy, striping allows single large requests (larger than one stripe size) to be executed in parallel.

The choice of stripe size affects the performance gains of these two factors. A small stripe size means more requests span multiple stripe units, and these requests can be executed in parallel with improved effective bandwidth. However, this decreases disk utilisation due to increased seeking. On the other hand, a large stripe size decreases seeking at the disk, but limits the effective bandwidth for large requests. Chen and Patterson [CP90] showed that the optimal stripe size depends primarily on the number of outstanding requests, the request size distribution, and the properties of the physical disks.

2.3.2 RAID 0

RAID 0, also known as the *striped set*, is a pure application of data striping with no redundancy. Since there is no redundant information to write, RAID 0 has the highest write performance and space efficiency, but the worst reliability (indeed, worse than a single disk), since failure of any one disk in the array results in data loss. It is primarily used where good capacity and performance (both read and write) are desired.

2.3.3 RAID 1

A RAID 1 (or *mirrored set*) array is the simplest level employing redundancy, where each of two physical drives store a complete copy (*mirror*) of the logical disk. This scheme yields a good degree of redundancy and best read performance, since a request can be serviced by either of the mirrors. However, write performance is average, since both mirrors must be updated for every write. RAID 1 has the lowest space efficiency, so is used when reliability and performance are of greatest importance.

2.3.4 RAID 1+0

RAID 1+0 is a simple combination of mirroring and striping, where n mirrored (RAID 1) sets are striped together to form a single logical disk. This scheme shares the properties of pure RAID 1, however it has the best degree of redundancy, since one disk from each mirror can fail before data loss occurs.

2.3.5 RAID 5

RAID 5, called *distributed parity*, is a striped scheme where one unit from each stripe contains parity computed over the remaining units in the stripe. This configuration can tolerate any single-disk failure, since the data on the failed disk can be reconstructed from the remaining data and parity information.

For read requests, RAID 5 behaves similarly to RAID 0, ignoring the parity unit in each stripe. However, when a write occurs, the corresponding parity stripe unit must be updated.

This leads to poor write performance for small requests (so-called *partial-stripe* writes), since computing the parity information requires reading the existing stripe. Large write requests (*full-stripe* writes) do not suffer from this problem, since the parity unit can be calculated from the new data alone.

RAID 5 arrays have the best space efficiency of the redundant schemes, since they require only one additional disk, regardless of the array size. To prevent any "parity disk" becoming a bottleneck, the parity stripe units are distributed equally across all disks in the array (see Figure 2.4).

2.3.6 RAID 6

RAID 6, also called *dual distributed parity* or $P + Q$ *redundancy*, is an extension of RAID 5 to two redundancy units per stripe, allowing two disk failures before data loss occurs. Read and large write performance is similar to RAID 5, but for small writes, two redundancy stripe units must be updated. RAID 6 requires two additional disks over the non-redundant RAID 0 case, and like RAID 5, the redundant stripe units are distributed over all available disks.

2.3.7 Hardware and software implementation

RAID functionality can be implemented in either hardware or software.[6] In hardware RAID, the array is managed by a dedicated RAID controller device, which presents an abstract disk drive to the host system. They often include large memories for buffering, caching, and request queueing. In software RAID, array functionality is performed entirely in software on the host system using existing storage devices.

RAID controllers perform logical-to-physical sector mappings and parity calculations in hardware, relieving the host system from such resource-intensive operations. Hsieh et al. [HSA02] found that, as a result, hardware RAID typically achieves higher throughput for a given host CPU load. However, they discovered that hardware RAID does not always give greater raw throughput.

2.4 Linux

The Linux kernel contains four I/O schedulers: noop, deadline, anticipatory, and CFQ (completely fair queueing). In this section we outline the structure of Linux's block I/O subsystem, then go on to describe the I/O schedulers in detail. The version of Linux described is 2.6.24, which is the latest stable release as of January 25, 2008. Finally, we summarise the existing literature comparing their properties, all of which is based on older versions of Linux.

[6] Hybrid hardware/software implementations also exist, but are less common.

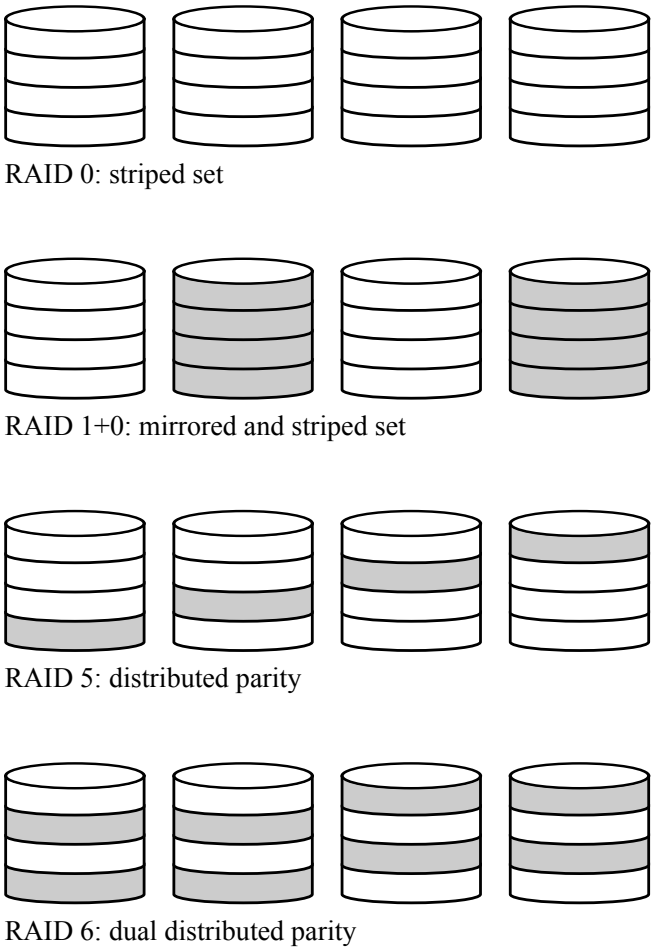

Figure 2.4: RAID 0, 1+0, 5 and 6 arrays with four physical disks. The shaded stripe units represent redundancy information.

2.4.1 Block I/O subsystem

Figure 2.5 shows the structure of the Linux block I/O subsystem. The block layer provides a generic API for accessing block devices such as disks. It deals with *requests*—a representation of individual I/Os performed on the device—which are passed through the I/O scheduler before being placed on the *dispatch queue*. This queue is the interface between the device-independent block subsystem and the device driver.

Two important I/O modes are marked on the figure. Direct I/O operates directly on the block device in terms of LBNs, and bypasses the page cache (also called the buffer cache). File I/O is performed on a filesystem in terms of creating, reading and writing files, and does not bypass the page cache.

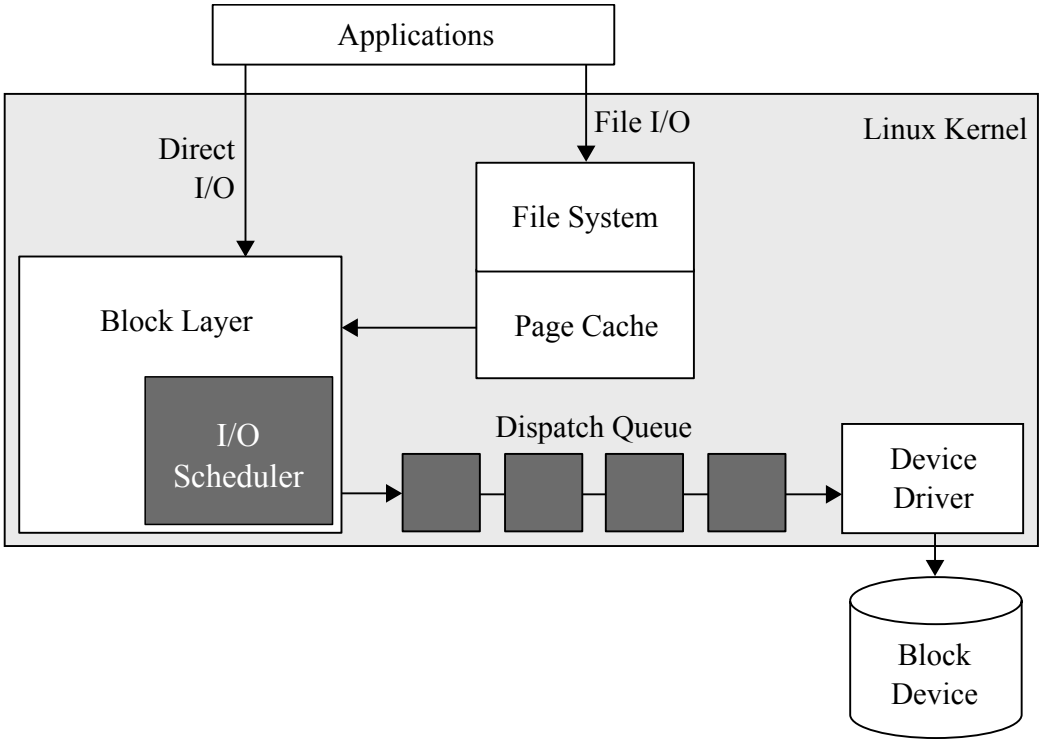

Figure 2.5: Linux I/O subsystem. The arrows show control flow of I/O requests. The important I/O scheduler components are shaded dark grey.

Merging The generic block layer performs back-merging, which occurs when a new request is merged behind an existing request; that is, the start sector[7] of the new request is greater than, and adjacent to, the end sector of an existing request. These merge opportunities are discovered by a hash table look-up which is indexed by the end sector of the existing requests.

As a consequence of merging, two queued requests may become adjacent. When this occurs, they can be merged into a single larger request. To clearly distinguish this case, we shall refer to it as *coalescing*.

Tunables The block layer provides access to a variety of *tunables*; per-device variables which affect the behaviour of the block I/O subsystem and I/O schedulers. These variables can be tuned by the user through files in the *sysfs* pseudo-filesystem provided by the Linux kernel. The following table summarises the relevant generic tunables; the scheduler-specific tunables are described in their respective sections.

[7]Linux defines a sector as a 512-byte logical block.

Parameter	Default	Description
`scheduler`	*none*	I/O scheduler selected for device.
`nr_requests`	128	Maximum queued requests allowed.
`queue_depth`	*max*	Maximum concurrent requests on disk.
`max_sectors_kb`	512	Maximum size (in KiB) of a single request.

Table 2.1: Summary of generic tunables.

2.4.2 Noop

Noop (no-operation) is a simple I/O scheduler that provides FCFS-like behaviour; requests are issued in the order they arrive. However, because it back-merges requests, ordering may be perturbed. Strictly speaking, noop performs a sorted insertion of requests onto the dispatch queue. However, testing has shown that this queue very rarely holds more than a single request, since drivers dequeue requests immediately. Hence, this sort has no effect.

Noop has no tunables.

2.4.3 Deadline

The *deadline* I/O scheduler aims to limit the time each request spends queued. This is achieved by assigning an expiry time, or deadline, to each request entering the scheduler. If the deadline is reached before the request is serviced, the scheduler will expedite dispatch of that request. Dispatching of expired requests is rate-limited to prevent excessive performance penalties under heavy load.

Deadline normally issues requests in *batches*, which is a set of contiguous requests in a particular direction (read or write). Up to `writes_starved` read batches are issued before running a write batch. This partitioning ensures that read requests, which often have an application waiting on the response, receive sufficient service. On the other hand, write requests are typically issued asynchronously from the page cache, and hence the latency is not directly visible to the application.

Within a sequence of batches in a single direction, deadline schedules requests according to the SCAN algorithm. However, when switching directions, the request with the earliest expiry time is selected.

Deadline has several tunables which are summarised in Table 2.2.

2.4.4 Anticipatory

The Linux *anticipatory* scheduler (abbreviated AS) is an implementation of the algorithm detailed in Section 2.2.5. It exploits spatial and temporal locality in I/O request patterns to improve performance, but also implements deadlines to prevent starvation. AS's tunable parameters are summarised in Table 2.3.

Parameter	Default	Description
`read_expire`	500ms	Expiry time for reads.
`write_expire`	5s	Expiry time for writes.
`writes_starved`	2	Number of read batches between write batches.
`front_merge`	true	Enables front-merging.
`fifo_batch`	16	Maximum requests per batch.

Table 2.2: Summary of deadline tunables.

Batching AS issues requests in batches, which are defined as a period of time over which requests of the same direction are issued. However, AS does not deal with read and write requests like deadline, rather synchronous and asynchronous. A synchronous request is defined to be a read, or a write to a file opened with the `O_DIRECT` or `O_SYNC` flags, which prevent the request being buffered in the page cache. All other requests are treated as asynchronous.

Requests in a batch are issued in SCAN order, and need not be contiguous. However, small backward seeks of up to 1 MiB are allowed, but are biased at twice the cost of a forward seek. On top of this behaviour is per-request deadlines, which operate similarly to the deadline scheduler. AS always merges requests where possible.

Anticipation AS's anticipation heuristic detects when a process is likely to issue sequential synchronous I/O, at which point the scheduler enters the anticipation state. In this state, an idle period of up to `antic_expire` ms is inserted waiting for a suitable request. The heuristic is based on a table of statistics collected for each running process, which includes a history of seek distances and think-times. Only synchronous requests are subject to anticipation.

Parameter	Default	Description
`read_expire`	125ms	Expiry time for synchronous requests.
`write_expire`	250ms	Expiry time for asynchronous requests.
`read_batch_expire`	500ms	How long to issue synchronous requests before switching to asynchronous.
`write_batch_expire`	125ms	How long to issue asynchronous requests before switching to synchronous.
`antic_expire`	6.7ms	Maximum anticipation time.

Table 2.3: Summary of AS tunables.

2.4.5 CFQ

The *completely fair queueing* (CFQ) I/O scheduler aims to fairly distribute disk time to each competing process, while attempting to minimise starvation with deadlines and improve per-

formance via anticipation. CFQ also supports I/O priorities, which can be used to define the priority and type of service received by each process. The tunables exported by CFQ are shown in Table 2.4.

Priorities CFQ is the only scheduler to support I/O priorities. A priority consists of two components: *level* and *class*. Priority level, a number ranging from 0 (highest) to 7 (lowest), determines priority of requests within a class. The priority classes, one of *real-time*, *best-effort*, or *idle*, have the following meaning:

Real-time Requests in this class are given maximum priority; other classes are not serviced if RT requests are pending. Within the class, requests are serviced in a round-robin-like fashion, with priority level determining the frequency and length of each round;

Best-effort The BE class is serviced in the same manner as RT, but only when no real-time requests are pending;

Idle These requests are only serviced if no requests from other priority classes are pending. There are no priority levels associated with the idle class.

I/O priority can be set with the `ionice` tool. For process with no explicit priority given, they are put into the best-effort class, with the level derived from their *nice*[8] value.

Dispatch Each process issuing I/O is assigned a time-slice during which it has exclusive access to the device and may issued synchronous requests. The number of requests in the slice is determined by the `quantum` tunable, and the base length of the time-slice is specified by the `slice_sync` tunable and adjusted according to priority level.

For asynchronous requests, all processes share a set of seventeen queues; one per priority class/level pair. The scheduling algorithm treats these queues the same as processes; they receive a time-slice (determined by priority level and `slice_async`) and are scheduled round-robin.

Within a time-slice, requests are fully-merged and issued according to the SCAN algorithm, with backward seeks of up to `back_seek_max` KiB allowed but biased at `back_seek_penalty` times the cost of a similar forward seek. Additionally, the scheduler will wait up to `slice_idle` ms within a synchronous time-slice for the process to issue more I/O. This is for both anticipation, and to ensure processes issuing bursty I/O get a fair share of disk time.

2.4.6 Previous studies

There have been two recent studies looking at the Linux I/O schedulers, one by Pratt and Heger [PH04], and another by Riska et al. [RLLR07]. In this section we review the experiments and results obtained in these two studies.

[8]The nice value refers to task-scheduling priority.

Parameter	Default	Description
back_seek_max	16MiB	Maximum backward seek size.
back_seek_penalty	2	Reverse seek penalty factor.
fifo_expire_async	250ms	Expiry time for asynchronous requests.
fifo_expire_sync	125ms	Expiry time for synchronous requests.
quantum	4	Maximum requests per-queue to service each round.
slice_async	40ms	Base time slice for asynchronous requests.
slice_sync	100ms	Base time slice for synchronous requests.
slice_async_rq	2	Base number of asynchronous requests per round.
slice_idle	8ms	Maximum anticipation time.

Table 2.4: Summary of CFQ tunables.

Pratt 2004

Pratt and Heger [PH04] studied the Linux 2.6.4 I/O schedulers: noop, deadline, anticipatory and CFQ (which was back-ported from Linux 2.6.5). While those schedulers share their names with the current generation, their implementations and behaviour are somewhat different. Notably, the 2.6.5 version of CFQ uses a completely different algorithm based on *stochastic fairness queueing* [McK90] and does not support I/O priorities. Additionally, there are many other minor changes in all four schedulers which together could result in significantly different performance characteristics. Care must be taken when comparing results between versions.

Experimental setup Three different hardware configurations were used: a single CPU, single disk system, an 8-way NUMA machine with a 5-disk RAID 5 array, and a 16-way SMP system with a 28-disk RAID 0. It is unclear from the paper whether hardware or software RAID was used. The benchmarks consisted of four workload profiles simulating a file server, web server, mail server, and a metadata-intensive load. In addition, they benchmarked sequential read and write performance. These workloads were run on both the EXT3 and XFS filesystems.

Results The key results from this study are as follows:

- the performance difference between the schedulers is more pronounced with the XFS filesystem compared with EXT3;
- AS typically performs poorly, except for sequential read workloads, for which it performed well;
- deadline performs well in all situations;
- anticipation causes loss of performance on TCQ-enabled disks or hardware RAID arrays.

The conclusion from the study is that I/O scheduler choice is heavily dependent on workload I/O pattern, disk hardware, and filesystem used.

Riska 2007

Riska et al. [RLLR07] studied the I/O schedulers of the Linux 2.6.16 kernel. This version of the I/O schedulers are similar to 2.6.5, but variations still exist and thus behaviour may have changed.

Experimental setup Each I/O scheduler was studied on top of the ReiserFS filesystem using a single 18 GiB SCSI disk with a TCQ depth of 4. Two workloads were used: the Postmark benchmark with different file sizes ranging from 0.1–3 MiB, and concurrent compilations of the Linux kernel source.

Results The study showed the following for Postmark:

- deadline achieves the highest throughput overall;
- noop always performs significantly worse than the other schedulers—as low as 25% throughput;
- AS performs marginally better than the other schedulers with large files.

The kernel compile benchmark, which they classified as a medium write-intensive load, showed that deadline, anticipatory and CFQ yielded approximately equivalent throughput, with noop marginally lower.

The conclusions of the study were:

- for heavy loads, merging is critical to performance;
- for light and medium workloads, reordering is the most important factor leading to performance gains; up to 20% improved throughput.

Chapter 3

Implementation

In this chapter, we describe the important code we have written as part of the study. Firstly, this includes two new I/O schedulers for Linux: FIFO, described in Section 3.1, and V(R), in Section 3.2. Secondly, in Section 3.3, we look at a modification made to the driver for the RAID controller used in our benchmarking. In Section 3.4, we look at problems discovered in the existing Linux I/O schedulers, and the way in which they were solved.

3.1 FIFO

Linux does not include a strict first come first served (FCFS) I/O scheduler. While noop does not sort requests, it does merge and coalesce them, which means order is not always preserved. Since FCFS is often used as a baseline for comparing scheduler performance, we implemented a Linux I/O scheduler called *FIFO* (first in first out—so named to distinguish the algorithm from the implementation) that provides pure FCFS behaviour; that is, requests are never merged or reordered. Since the final request ordering under noop and FIFO should be similar, this helps isolate the effect of merging on I/O scheduler performance.

FIFO is based on the noop scheduler, modified to provide the desired behaviour. The major change is in `elevator_allow_merge_fn()`. This function allows an I/O scheduler to control when merging is allowed to take place. The FIFO implementation of this function unconditionally denies all attempts to merge. Since coalescing occurs as a result of a merge, this ensures both merging and coalescing do not occur.

The other change made compared to the default noop implementation is to remove request sorting of requests when placing them on the dispatch queue. However, as noted in Section 2.4.2, this change should not affect performance, since the dispatch queue never holds more than a single request with the drivers used in this study.

A patch which adds the FIFO I/O scheduler to the Linux 2.6.24 kernel can be found in Appendix A.

3.2 V(R)

We implemented a variant of V(R) (see Section 2.2.5) which was used primarily for benchmarking. It provides both SCAN and SSTF behaviour in a single implementation (by tweaking the R parameter, named `rev_penalty`), both of which are commonly used for comparing I/O scheduler performance.

Our V(R) implementation schedules read and write requests together (unlike the other schedulers), and implements back and front merging and coalescing. It also provides deadlines, which can be disabled at run-time. We will first describe the differences in behaviour between our implementation and the theoretical algorithm, then go on to detail the implementation. Table 3.1 summarises the V(R) tunable parameters.

The published V(R) algorithm defines R such that the effective distance for a backward seek is $R\times$(number of cylinders on disk) larger than an equivalent forward seek. In our implementation, we redefined R such that the effective distance of a reverse seek is multiplied by R. Hence, the domain of R becomes $[1, \infty)$, with $V(1)$ behaving as SSTF, and $V(\infty)$ as SCAN (for simplicity we allow $R = 0$ as an alias for SCAN). This was done to simplify implementation, tuning and debugging, because the scheduler behaviour is no longer dependent on disk size.

Internally, V(R) is structured similarly to deadline. When the scheduler queues a new request, it is placed into a red-black tree sorted by start sector. In parallel, pointers are kept to the request before and after (in sector order) the last one issued. In addition, the sector of the last request and the current head direction are stored. In selecting the next request, the sector distances to the next and previous requests are calculated, and the distance corresponding to the current head direction is divided by `rev_penalty`. This is equivalent to multiplying the reverse seek distance, but avoids overflow. The request with the smallest effective distance is selected and issued to the device.

Deadlines If deadlines are enabled, requests are assigned an expiry time when queued. If that deadline expires, V(R) expedites dispatch of the corresponding request. However, it is not scheduled immediately, since such behaviour can lead to a so-called "deadline avalanche." This occurs when the scheduler is under high load and deadlines are expiring more quickly than requests can be satisfied, causing the scheduler to degenerate to FIFO behaviour. To solve this, requests are normally issued in batches of up to `fifo_batch` requests. The deadline queues are only checked for expiries between batches, with the expired request starting a new batch.

The V(R) I/O scheduler patch for Linux 2.6.24 can be found in Appendix A.

3.3 CCISS queue depth

The hardware RAID controller used in this work is the HP *Smart Array* P600 (discussed in detail in Section 4.1.2) which uses the Linux CCISS driver. This device is capable of queueing up to

Parameter	Default	Description
`sync_expire`	500ms	Expiry time for synchronous requests.
`async_expire`	5s	Expiry time for asynchronous requests.
`fifo_batch`	16	Maximum requests per batch.
`rev_penalty`	10	Penalty for reversing head direction.

Table 3.1: Summary of V(R) tunables.

512 requests, therefore the I/O scheduler does not see more than a single concurrent request until the disk load exceeds 512 concurrent requests—a situation that cannot be considered typical. This has several implications for benchmarking I/O scheduler performance.

Firstly, since the disks (and indeed, the RAID controller itself) are doing I/O scheduling, a deep queue on the controller can negate the scheduling performed on the host. Moreover, with fewer requests at the host, the schedulers have less request information and are thus less effective. Secondly, the I/O schedulers cannot be compared under light load, since they all degenerate to FIFO.

To solve this problem, we modified the CCISS driver to introduce a `queue_depth` tunable, which can be used to restrict the queue depth of the device to any value between 1 and the maximum supported by the hardware (512 for the P600). The implementation of this tunable modifies the existing queue depth restriction to be variable at run-time, rather than a compile-time constant.

A patch to add the `queue_depth` tunable to the CCISS driver in Linux 2.6.24 is available in Appendix A.

3.4 Deadline and anticipatory tweaks

During the course of our initial investigation, several issues with the deadline and anticipatory schedulers were discovered and fixed. In this section, we describe the problems, how they were solved, and some benchmark results. Since benchmarking the fixes is not a primary goal of this work, the results shown are not intended to be a rigorous analysis but rather indicate the usefulness of the work.

All of these fixes were accepted into the Linux mainline kernel, and are present in version 2.6.24 and later. Hence, all the benchmarks and analysis in the main body of the work include these improvements. The descriptions that follow apply to Linux 2.6.23.

3.4.1 Resetting batch counter

The Linux deadline I/O scheduler issues requests in batches. To ensure acceptable interactive performance, several read batches are issued between each write batch. However, in between read batches, the batch counter would not be reset, causing the second and subsequent batches

to consist of a only a single request. Thus, deadline's goal of preferring reads over writes failed; rather than issuing n read batches in between writes, it would issue only 1 read batch plus an additional n requests.

We fixed this bug by forcing the batch count to reset whenever a new batch is started, regardless of what triggered it. The corresponding patch is available in Appendix A.

To measure the effect of this bug on performance, the following experiment was conducted. Using the FIO tool (discussed in Section 4.2.3), we generated a workload consisting of two processes issuing the same I/O pattern; 8 MiB sequential requests issued asynchronously in chunks of 16 KiB directly to the device (i.e., bypassing the page cache and file system), with one process writing and the other reading. The target device was a single 15 000 RPM, 72 GiB SCSI disk. We ran the workload over a period of 300s and measured the total I/O performed, then repeated the experiment after applying our patch. The results are shown in Table 3.2.

Batching algorithm	Total I/O (MiB)		Read/write ratio
	Read	Write	
Original	5 357	6 643	0.806
Modified	9 406	4 703	2.000

Table 3.2: Deadline throughput before and after fixing the batch count reset bug.

The measurements show that the expected read/write ratio of 2 (which is the default value of the `writes_starved` tunable) is not achieved by the original deadline batching algorithm. Indeed, the total write I/O *exceeds* the total read I/O. This is due to misalignment of read requests on batch boundaries, resulting in under-full batches. The fixed version of deadline achieves the desired ratio.

3.4.2 Batch start-sector

The deadline scheduler keeps read and write requests in separate sector-sorted binary trees so that finding the next request in a given direction is a constant-time operation. This is required for efficient batching. As a result of this separation, finding the next request after a data-direction switch requires an $O(\lg n)$ traversal of the other tree.

To avoid this costly operation, deadline simply starts new batches from the request with the lowest start sector. Since the request queues are implemented as min-trees, this is an $O(1)$ operation. However, this strategy leads to significant fairness disparity between low- and high-sector requests.

To rectify this problem, we modified the deadline batching algorithm such that new batches after a data-direction switch start with the request whose deadline is earliest. This solves the problem described above, while maintaining the $O(1)$ performance. We can achieve this by using the expiry-sorted queues; the request with the earliest deadline is simply the one at the head of the queue.

This modification was also applied to the anticipatory scheduler, but for a different reason. AS stores the next request for both synchronous and asynchronous requests, and upon switching data direction, continues where the previous batch (in the same direction) finished. However, after making a data-direction switch, the scheduler does not know the location of the request nearest the head, and finding this request would require an $O(\lg n)$ search through the request tree. Thus it makes little sense to "continue" batches across direction switches.

By having AS start new batches at the request with the earliest deadline, we reduce the probability that deadlines will expire. This can improve throughput, since expiries result in a less efficient schedule. Moreover, this change reconciles the deadline and anticipatory batching algorithms.

Benchmarking

To measure the effects of this bug and the effectiveness of the fix, we constructed the following benchmark. Using `FIO`, 10 processes are created, each doing direct, sequential, synchronous read I/O from sector offsets increasing by 7 GiB. Thus, process 1 reads from sector 0, process 2 from 7 GiB, process 3 from 14 GiB, and so on. In addition, a random writer process is created to cause data-direction switches. The hardware is the same used in Section 3.4.1.

The results of this benchmark are graphed in Figure 3.1. It clearly shows that, under the original algorithm, processes operating at one end of the disk receive significantly greater bandwidth than those reading from the remainder of the disk. In the fixed version, the bandwidths received by each process differ by less than 200 B/s.

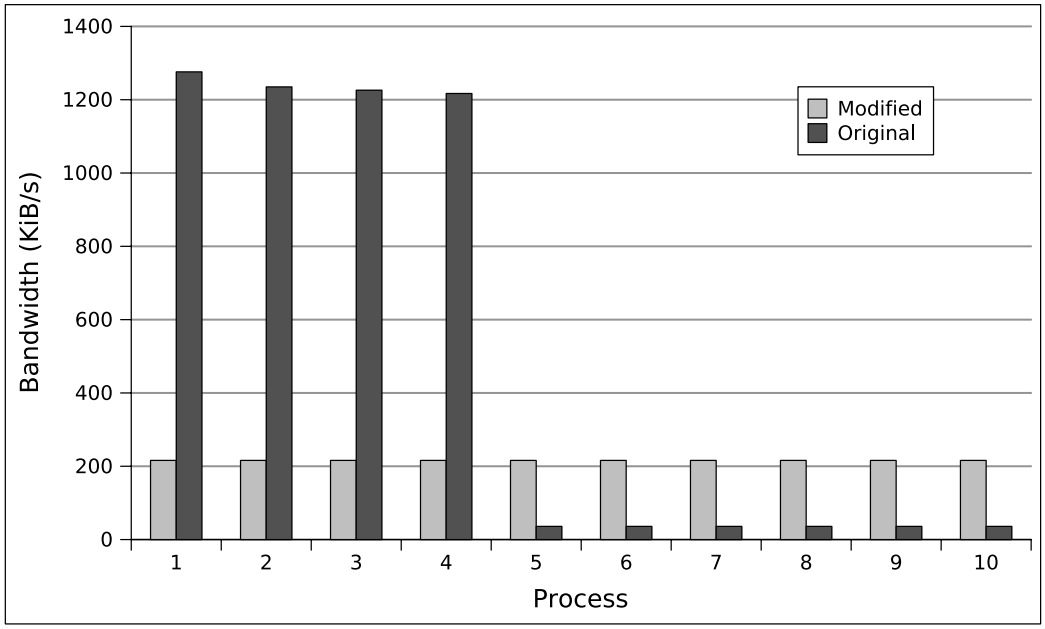

Figure 3.1: Deadline bandwidth distribution before and after applying our start-sector fix.

It is important to note that the aggregate bandwidth in this benchmark decreased after applying our fix. This is expected, since by concentrating request service to a small region of this disk, seeking is reduced. We consider this to be an acceptable trade-off, because the primary aim of the deadline scheduler is to provide reasonable (though non-strict) quality of service to all disk users. Given this goal, an order of magnitude bandwidth difference depending on the region of the disk accessed is clearly not acceptable.

A realistic example of where this bug may manifest is on a partitioned disk. Sustained access to the first partition may cause starvation of requests on other partitions. Worse still, on logical volumes backed by multiple physical disks (such as a hardware JBOD[1] array), entire disks may sit idle, severely hurting total throughput.

[1]JBOD (just a bunch of disks) is similar to RAID 0 but without data striping.

Chapter 4

Benchmarks

In this chapter, we describe the hardware and software system under which our benchmarks were conducted. Firstly, Section 4.1 details the hardware of the system, including the disk subsystem. Section 4.2 describes the two most important software components: the operating system and the workload-generators. Finally, in Section 4.3, we lay out the details of the benchmarks themselves.

4.1 Hardware

4.1.1 Host

The host machine is an HP Integrity rx2660 server with the following specifications:

Component	Description
Processor	1 Intel Itanium 2, 9010 "Montecito" at 1.6 GHz
RAM	2 GiB
SAS Controller	LSI Logic SAS1068 Fusion-MPT PCI-X

Table 4.1: Host system specifications.

4.1.2 Storage subsystem

Two pieces of storage hardware were benchmarked; a hardware RAID controller managing 8 physical disks, and a single disk connected to the host via a SAS (serial attached SCSI) adapter.

The RAID controller used is an HP Smart Array P600 which is connected via a SAS bus to an HP MSA50 disk enclosure holding 8 SAS disks (discussed in Section 4.1.3). The specifications of the RAID controller are as follows [HP08]:

While we had 10 disks available for testing, only 8 were used in the RAID array. Another was used for the single-disk host-controller benchmarks, and the remaining disk kept as a spare.

Component	Description
Controller memory	256 MiB, battery-backed
Queue depth (max.)	512
Disk interconnect	SAS (max. 1.5 GB/s)
Host interconnect	PCI-X (max. 1 GiB/s)
RAID levels	0, 1, 1+0, 5, 6

Table 4.2: RAID controller specifications.

4.1.3 Disks

The disks used for all benchmarking (both RAID and single-disk tests) are HP DG072A4951 72 GB SAS disks, the specifications of which are shown in Table 4.3.

Component	Description
Rotational speed	10 000 RPM
Controller memory	16 MiB
Queue depth (max)	64
Seek time (typical)	3.7 ms
Sustained transfer rate	55–89 MB/s
Heads	2
Platters	1
Zones	20

Table 4.3: Disk drive specifications [Hit08].

4.2 Software

4.2.1 Linux

The Linux 2.6.24 kernel was used for all benchmarks, minimally configured to support only the necessary features. The corresponding configuration file is available in Appendix B. On top of the default kernel we applied the FIFO, V(R), and CCISS queue depth patches.

The Linux distribution used on the benchmarking machine was Debian 4.0 “etch”.

4.2.2 Filesystems

For the filesystem-based benchmarks, we selected the EXT2 and XFS filesystems.

EXT2 has seen wide use in the Linux community since its introduction in 1994 [CTT94]. It is an FFS-like[1], block-allocating filesystem that uses bitmap-based free block management. It attempts to improve locality though the use of block groups, which cluster inodes and the corresponding data, and block preallocation, which over-allocates blocks during writes to improve contiguity. EXT2 was selected over the more recent EXT3 filesystem since EXT2 does not implement journalling, in contrast to the second filesystem, XFS.

Silicon Graphics' *XFS* filesystem was introduced into the Linux kernel in 2001 [BFH02]. Its primary goals are maximum scalability in storage capacity, filesystem size, and performance. It supports metadata-journalling and uses tree-based extent allocation. XFS tries to ensure maximum contiguity by using delayed allocation and large extents. It also performs write clustering which, similarly to request merging, results in more sequential I/O.

These two filesystems were chosen since they generate significantly different I/O patterns on the disk and hence test the I/O schedulers in different ways. Several reasons for this are:

- EXT2 allocates fixed-size blocks, whereas XFS allocates variable-size extents;
- XFS employs a journal which results in highly-sequential writes;
- EXT2 uses indirection blocks for managing data blocks, whereas XFS uses B+ trees;
- XFS employs a tree-based approach to free-space management. EXT2 uses bitmaps.

In all tests, the filesystem creation and mount options were the defaults provided by the `mkfs` and `mount` utilities.

4.2.3 FIO

FIO (the Flexible I/O Tester) is one of two tools we used for I/O load generation. It spawns a number of processes performing I/O loads specified by the user. Many aspects of the load can be specified, including: read/write ratio, synchronous or asynchronous, I/O depth, block size, sequentiality, direct or buffered requests, and think-time. It can operate on both filesystem and raw block devices.

FIO generates extensive per-process and aggregate performance statistics after a run, including: total I/O, run time, read and write request latency (min, max, mean, standard deviation), read and write bandwidth (min, max, mean, standard deviation), CPU usage, and I/O depth distribution.

We used FIO version 1.19. More information, including source code, can be found at the FIO website [Axb08b].

[1]FFS, the Berkely Fast File System, is a classic UNIX filesystem designed by McKusick et al. [MJLF84]

4.2.4 Postmark

The second benchmarking tool used was Postmark [Kat97]; a filesystem-level benchmark that generates a workload typical of an email or news server. Certain parameters of the load can be modified via a configuration file, generating the following I/O pattern:

1. A pool of text files and directories are created on the specified filesystem. The range of file sizes and the total number of files and directories can be configured;
2. A configurable number of transactions are performed, each consisting of:
   - a random-size file is created, or a randomly-selected file is deleted;
   - a randomly-selected file is fully read or appended with a random amount of data;
3. The directory hierarchy (including all files) is deleted.

The ratio of create/delete and read/append operations can be specified, as can the block size. In addition to configuring the I/O pattern, one can also select buffered or direct I/O. After completing the workload, Postmark generates a report showing: total run time, transaction rate, rate and number of creations, reads, appends and deletes, and the read and write throughput.

We ran our benchmarks on Postmark 1.51 using non-default parameters.

4.3 Tests conducted

To test the performance of the I/O schedulers, four different workloads were run against a number of disk configurations.

4.3.1 Disk configurations

The RAID levels tested were: 0, 1, 1+0, 5 and 6. This reflects the capabilities of the controller used. We also tested two configurations consisting of a single-disk: the first had the disk connected directly to the host's SAS controller, while the other was connected via the RAID controller in a 1-disk RAID 0 setup. These allow isolation of the effects of the RAID controller, such as its on-board memory and queueing capacity.

The P600 RAID controller has configurable parameters that may affect its performance. In our benchmarks, all these parameters remained at their default values, which are outlined in Table 4.4.

4.3.2 I/O schedulers

The generic FIO and Postmark benchmarks tested the following I/O schedulers:

- noop;

Parameter	Value	Description
Accelerator ratio	50/50	Ratio (read/write) of on-board controller memory used for read-caching and write-back buffering.
Drive write cache	false	Enables write-back buffering on the disk controllers.
Stripe size	*varies*	Size of stripe-units of the RAID array. Depends on the RAID level used (see Table 4.5).

Table 4.4: Default RAID parameters.

Level	Stripe size (KiB)
0	128
1	128
1+0	128
5	64
6	16

Table 4.5: Default RAID stripe-unit size.

- FIFO;
- FIFO with maximum queue depth ("bigq");
- deadline;
- V(R) configured for SCAN (`rev_penalty` = 0);
- V(R) configured for SSTF (`rev_penalty` = 1);
- AS;
- AS with anticipation disabled (`antic_expire` = 0);
- CFQ;
- CFQ with idle window disabled (`slice_idle` = 0).

This includes each of the Linux 2.6.24 schedulers in their default configurations. AS with anticipation disabled was tested to isolate the performance implications of anticipation. CFQ was tested with idle window disabled for the same reason. FIFO and V(R) in SCAN and SSTF were tested as these algorithms are frequently used as baselines for I/O scheduler performance evaluations.

The other benchmarks test a subset of these schedulers appropriate for the aims of the benchmark.

Tunables

All scheduler-specific tunables remained at their default values, with the exceptions listed above. Two of the generic block-layer tunables were modified from their default values during the benchmarks: `queue_depth` and `nr_requests`.

As discussed when motivating the CCISS queue depth modification (Section 3.3), a large queue is not suitable for benchmarking I/O schedulers. Thus, all the above tests used a depth of 32, except one of the FIFO runs, which used the maximum supported by the corresponding hardware (512 for the RAID controller, 64 for the single disk). This was done to simulate the situation where all requests are immediately offloaded to the hardware with no scheduling on the host.

The queue depth of 32 was chosen to be as small as possible while ensuring a high probability that no disk in the array becomes idle. Assuming a randomly-distributed set of 32 requests, the probability of idle disks (i.e. no requests assigned to them) is approximately 11%[2]. This translates to approximately 1.5% lost bandwidth, which we consider to be an acceptable trade-off in order to minimise perturbations of the host I/O schedule caused by RAID- and disk-controller scheduling.

The `nr_requests` tunable was set to 4096 to ensure the maximal number of requests are queued with the I/O scheduler. In the case where I/O is issued when `nr_requests` is exceeded, the request will not be processed until existing ones complete. This feature is designed to prevent denial of service attacks. However, since we are interested in the I/O schedulers and not the I/O subsystem *per se*, this tunable was increased to maximise the load at the scheduler level.

4.3.3 Micro-benchmarks

The first of our four benchmarks is a series of FIO micro-benchmarks testing each scheduler on 8-disk RAID 0, 5 and 6 arrays, RAID 1+0 with 2, 4 and 8 disks, and the two single-disk configurations.

In each test, a number of processes are spawned issuing the same I/O pattern. We tested 32, 64, 96, 128, 160, 192, 224, and 256 processes. Fewer than 32 processes were not considered, since no host I/O scheduling occurs under such light loads.

Two workloads were tested. The first consists of randomly-distributed 16 KiB reads, corresponding to the page size of the benchmarking machine. The second is a sequential test consisting of 1 MiB read requests issued at a random offset. In both cases, I/O is issued directly to the raw device (i.e. no filesystem), bypassing the page cache. Each test, corresponding to a scheduler, RAID setup, and I/O load combination, ran for 200 seconds. In total, this benchmark ran for 70 hours.

To simplify analysis, the workloads in this benchmark are highly artificial and ignore many factors that normally contribute to I/O performance. However, the literature [RR06, RW93b] suggests that they are typical of those seen at the disk on real systems.

[2]We were unable to determine idle-probability analytically; this figure was determined via simulation.

4.3.4 Postmark

The second benchmark used the Postmark tool to create a more realistic workload. It was run on top of the XFS and EXT2 filesystems using buffered I/O, testing the same RAID configurations as the FIO micro-benchmarks.

The benchmark was conducted as follows:

1. an XFS or EXT2 filesystem was created and mounted on the unpartitioned logical disk;
2. five directories were created at the top-level of the new filesystem;
3. five instances of Postmark were started, each using a different top-level directory.

We ran multiple instances of Postmark to ensure a sufficiently heavy load on the I/O scheduler. Five was chosen empirically using the `blktrace` [Axb08a] tool to measure average request load.

The Postmark configuration is shown in Table 4.6. The small file sizes were used for tests

Parameter	Value
File size (small)	512 B – 2 MiB
File size (large)	512 B – 4 MiB
Files	10 000
Sub-directories	100
Transactions	15 000

Table 4.6: Postmark benchmark configuration.

with a logical capacity of under 140 GiB; the single disks and the 2-disk RAID 1. The large sizes were used on the remaining benchmarks. This was done to use sufficient available disk capacity. Since this results in less sequentiality in disk requests, care must be taken in comparing results between disk systems.

The length of each benchmark is determined by the number of transactions; hence, the time of each run differs. The total time to run the Postmark benchmark was 334 hours.

4.3.5 Anticipation

The third benchmark was designed to test the applicability of anticipation on hardware RAID volumes. FIO was used as the load generator which created a sequential reader (64 sequential 16 KiB blocks) and a number of randomly-reading processes. Two parameters were varied for each scheduler/disk combination: the number of sequential readers (1, 4, 16, 32, 64, 96 and 128) and the think-time of the sequential process (0, 0.1, 0.5, 1, 2, 4, 6, 8, 10 and 20 ms). The remaining parameters were the same as the FIO micro-benchmarks: synchronous, direct, raw I/O, running for 200 seconds per data point.

The anticipatory scheduler was used exclusively in this benchmark, using four different configurations, listed in Table 4.7. In the single-disk host-controller test, a hardware-imposed

maximum queue depth of 64 was tested; the RAID-controller benchmarks used a maximum of 512.

Test	`antic_expire`	`queue_depth`
1	*default*	32
2	0	32
3	*default*	512/64
4	0	512/64

Table 4.7: Anticipation benchmark configurations.

In addition to the two single-disks configurations, we tested:

- RAID 0 with 2, 4 and 8 disks;
- RAID 1+0 with 2, 4 and 8 disks;
- RAID 5 with 3, 5 and 8 disks.

RAID 6 was not studied, since we did not consider it sufficiently different from RAID 5 to be worth the time required. In total, this benchmark ran for 171 hours.

4.3.6 Queue depth

The fourth and final benchmark was designed to investigate how performance reacts to different queue depths. Postmark was used on top of the XFS filesystem and deadline scheduler; the reasons for this choice will be explained in Section 5.4. As with the first Postmark benchmark, five instances were run in parallel, with the following configuration:

Parameter	Value
File size	512 B – 2 MiB
Files	4 000
Sub-directories	100
Transactions	15 000

Table 4.8: Queue depth Postmark configuration.

The queue depths tested were 1, 8, 16, 32, 64, 96, 128 and 256. We tested the following disk configurations:

- RAID 0 with 2, 4 and 8 disks;
- RAID 1+0 with 2, 4 and 8 disks;

- RAID 5 with 3, 5 and 8 disks;
- RAID 6 with 4, 6 and 8 disks.

We did not benchmark any single-disk configurations. The total run time for this benchmark was 203 hours.

Chapter 5

Results

This section presents and analyses the results of our benchmarks. Section 5.1 looks at the FIO micro-benchmarks, separately analysing the four workloads. Section 5.2 presents the results of the Postmark tests, and Section 5.3 analyses the FIO anticipation benchmarks. Finally, in Section 5.4 we look at the Postmark queue depth benchmark results.

5.1 Micro-benchmarks

In this section, we look at the results obtained in the FIO micro-benchmarks described in Section 4.2.3. The main figure of interest is aggregate read bandwidth, which is the sum of the average per-process bandwidths. Other interesting metrics are:

- Bandwidth standard deviation: deviation among the per-process bandwidths;
- Average read latency: per-read-request latency averaged over all processes. This is the average time a process must wait for an individual request to complete and is important for application interactivity;
- Maximum read latency: worst-case latency seen by any request from any process. Also important for interactivity;
- Read latency standard deviation: per-request latency calculated over all processes.

The complete set of graphs is available in Appendix C.

5.1.1 Random readers

The aggregate read-bandwidth results for this benchmark are graphed in Figure 5.1. On RAID volumes, large-queue FIFO achieves the highest throughput by a significant margin. For the single-disk benchmarks, V(R) generally performs the best, with FIFO (large-queue) achieving good results at low-load, but not scaling above 64 processes.

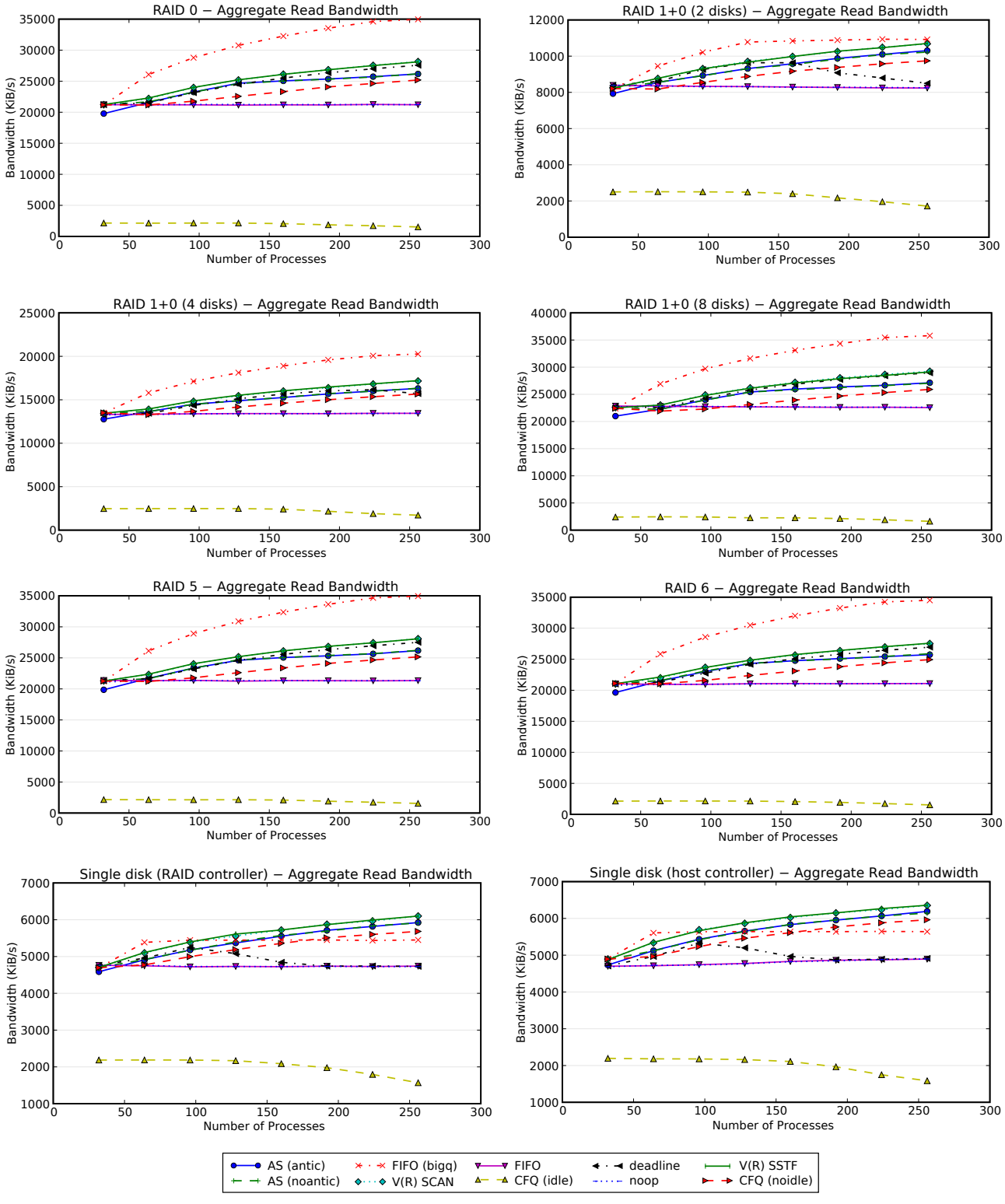

Figure 5.1: Random readers aggregate bandwidth.

In general, reordering schedulers display superior bandwidth performance and scaling compared with the non-reordering schedulers (noop and FIFO), regardless of the disk type or size. The performance of noop and FIFO is practically identical, since the probability of merges occurring for a random workload is low; less than 0.01% for 32 requests uniformly distributed over a 100 GiB array, and less than 0.6% for 256 concurrent requests.[1]

CFQ performs poorly in the random readers tests, notably so in the idle-enabled case, where it consistently performs 2 to 15 times worse than the other schedulers in bandwidth terms. The idle-enabled CFQ also displays very high maximum, average, and standard deviation of per-request service latency. When the idle window is disabled, CFQ performance improves dramatically. However, it still performs worse than AS, deadline, V(R) and FIFO (large queue) in most cases.

One would generally expect that an I/O scheduler should be able to extract greater bandwidth from a device as the workload increases (particularly when it is purely random), since a better schedule can be produced. However, with the idle window disabled, CFQ does not scale with increased load, and indeed scales negatively in some benchmarks.

Performance of the RAID 0, 5, and 6 arrays are very similar. This is expected, since the on-disk access patterns are very similar; no parity information has to be read or modified under a pure read load. Interestingly, the RAID 1+0 8-disk array also performs similarly, achieving only a marginal throughput increase of the order of 5%. This is despite the fact that the inherent redundancy of RAID 1+0 can be exploited to improve performance [CT96, San03].

The results of the two single-disk benchmarks are also similar, with the host-controller test achieving a very modest improvement in mean bandwidth, but approximately equal performance in other metrics.

The V(R) schedulers, SSTF and SCAN, perform practically the same under all tests. This is also true of the AS schedulers, which are indistinguishable with anticipation enabled or disabled. The latter can be explained by the anticipation heuristic detecting a workload which is not improved by anticipation. However, this is a poor test for the quality of the algorithm, since a purely random workload is a pathological case for an anticipatory scheduler and easily detectable.

The deadline scheduler exhibits interesting behaviour. On larger arrays it performs well, with similar throughput to V(R) and positive scaling. However, on smaller arrays and the single-disk tests, it scales negatively after reaching a certain process load. This point corresponds approximately to the load where the average request latency crosses 250ms. This may be due to the "deadline avalanche" problem discussed in Section 3.2, where the scheduler behaviour degenerates to FIFO. This hypothesis is further substantiated by the fact that per-process bandwidth

[1] The probability of merges given k disk blocks and n requests is $\left(\prod_{i=2}^{n}(k-i)\right)/k^{n-1}$, which can be derived from the solution to the birthday problem [Wei08].

variation drops sharply at the same load point, and that deadline performance approaches that of FIFO as load increases. It reaches FIFO performance at the point where average latency cross 500ms—the value of the read-request expiry time.

Interestingly, AS does not suffer from this problem, despite having a read expiry time of 125ms—4 times lower than deadline's. This is because of differences in their batching algorithms. While deadline and AS both check for expired requests after each batch, deadline's batches contain a single request under a random workload, whereas AS's batches are much larger. Hence AS schedules more requests between handling expiries, thus achieving higher throughput. However, deadline achieves better maximum latency figures.

5.1.2 Sequential readers

The throughput results of the FIO sequential-readers benchmark are graphed in Figure 5.2. CFQ with idle disabled performs well in the sequential readers benchmark, generally attaining the highest bandwidth except in the RAID 0 test, where it is outperformed by V(R). FIFO (large queue) tends to perform well at low loads, often performing the best at 64 processes or less. However, it does not scale well; not at all on the RAID 6 or small RAID 1+0 configurations, and even scaling negatively on RAID 5 and the host-controller single-disk test.

While the non-idling CFQ scheduler performs well, this is not so for the idle-enabled version, which displays bimodal behaviour. In some cases, it performs very well, about equal to the idle-disabled CFQ. In other cases, it performs very poorly as it does in the random readers test. There is no clear correlation between the benchmarks that evoke this behaviour and those that do not.

Noop and FIFO perform similarly, as they do under a random read workload, and for the same reason: no merging due to randomly-distributed requests. Indeed, merging is *less* likely in this case, since the issued requests are already larger than the maximum permitted, and thus will never be merged. These schedulers generally achieve a bandwidth comparable to other schedulers at low load, but do not scale. Their performance is approximately constant in most cases, scaling gently in others.

The V(R) schedulers, SSTF and SCAN, again behave quite similar to one another, generally somewhere toward the middle in the bandwidth ranking. The notable exception is RAID 0, where they are at least 10 MiB/s better than the other schedulers at loads above 150 processes, and second only to large-queue FIFO at lower loads.

In terms of latency, however, SSTF and SCAN perform poorly. They scale approximately linearly in standard deviation where most of the other schedulers scale sub-linearly. For maximum latency, they show anything up to 7 times that of other schedulers; up to 58s for the single disk connected via the host controller and 34s on the 2-disk RAID 1+0. This is almost certainly due to the lack of deadlines.

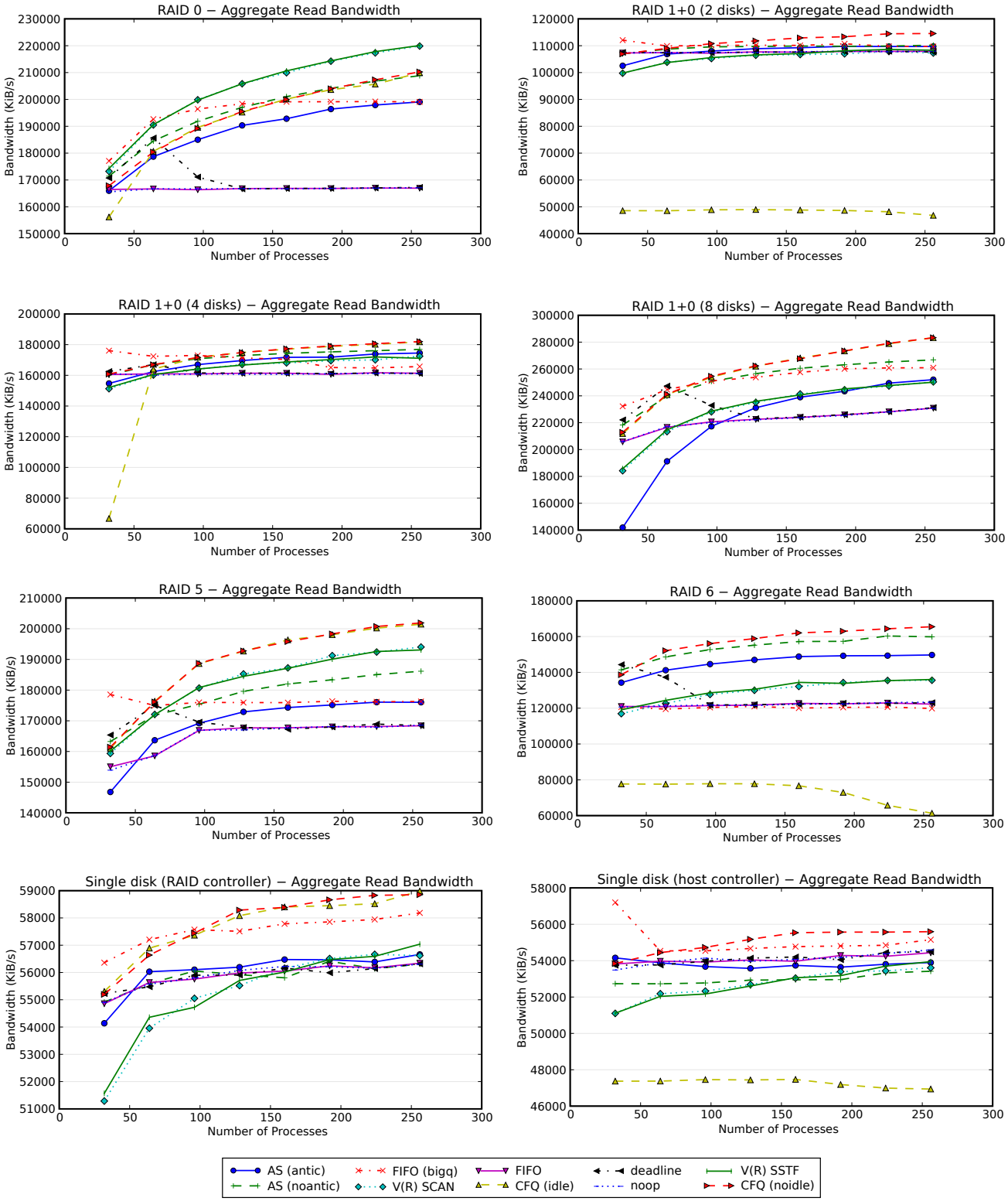

Figure 5.2: Sequential readers aggregate bandwidth.

In the RAID-based tests, AS performs better with anticipation disabled; approximately 5% on the larger arrays. However, on the single disk tests, the situation is reversed: the anticipatory version performs better, albeit by a smaller margin. This suggests that the anticipation algorithm, while fine for single-disks, is either not well suited for hardware RAID volumes, or needs to be tuned for such devices.

The deadline scheduler behaves poorly in this benchmark, with throughput equivalent to noop/FIFO at any appreciable load. This is the same scalability problem experience by deadline under a random-read workload, but occurs at a lower load level. This is because the average read latency is larger under a sequential workload due to larger requests.

5.2 Postmark

The results of the Postmark benchmark (see Section 4.2.4) are plotted in Figure 5.3 and 5.4, which shows the average throughput in transactions per second (TPS) per Postmark instance (with five instances in total). Table 5.1 shows the per-instance throughput averaged over all disk configurations. For XFS, the standard deviation of the per-instance bandwidths is at most 0.118 TPS, while for EXT2, it is at most 0.253. The complete standard deviation graphs can be found in Appendix C.

The first and most obvious observation is that XFS outperforms EXT2, on every scheduler, for the RAID-controller-based tests. However, for the host-controller test, EXT2 has a slight advantage of 0.3 TPS averaged over all schedulers. In contrast, XFS has an advantage of 1.8 TPS when the single disk is connected via the RAID controller. A possible explanation is that XFS makes better use of the large buffer and cache of the controller. This might be because XFS has better request locality (and thus has a higher cache hit rate), or because it is write-bound and thus benefits greatly from a large write-back buffer.

It is also clear that XFS scales with increasing array size better than EXT2. While XFS achieves a 35% maximum throughput improvement between a 4-disk and 8-disk RAID 1+0 array, EXT2 gains less than 2%.

Overall, deadline achieves the best throughput with a TPS of 4.12 averaged over all schedulers, filesystems, and hardware setups, and is always within 10% of the best-performing scheduler. Interestingly, deadline performs best for the single-disk RAID-controller benchmark, but this is not so for the host-controller test, for which default AS has the highest TPS. The latter is also the only test where the anticipation-enabled version of AS shows a significant advantage over AS with anticipation disabled. These results suggest that the RAID controller might be undermining the anticipation algorithm. We will have more to say about this in Section 5.3 when the anticipation benchmark results are analysed.

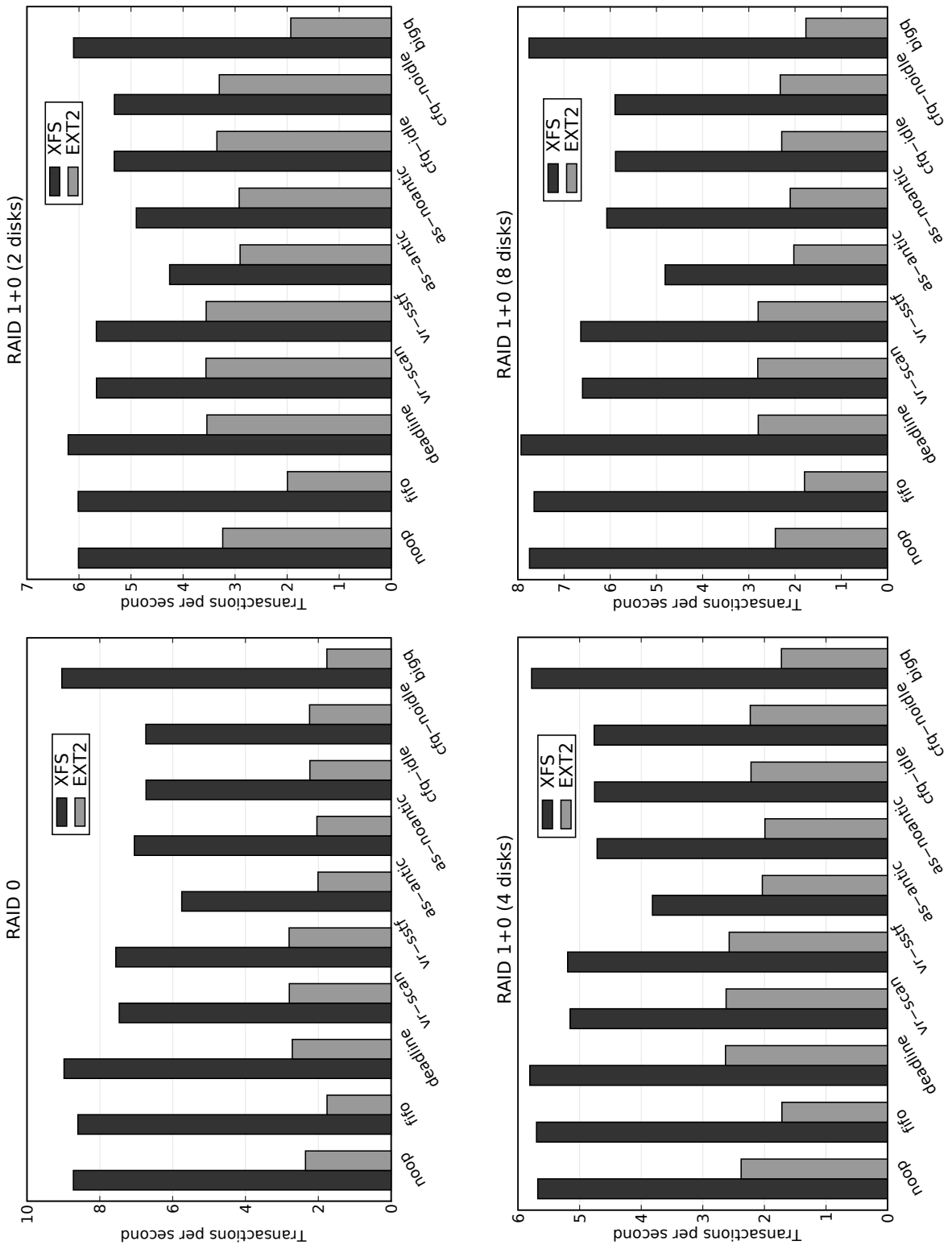

Figure 5.3: Postmark results 1.

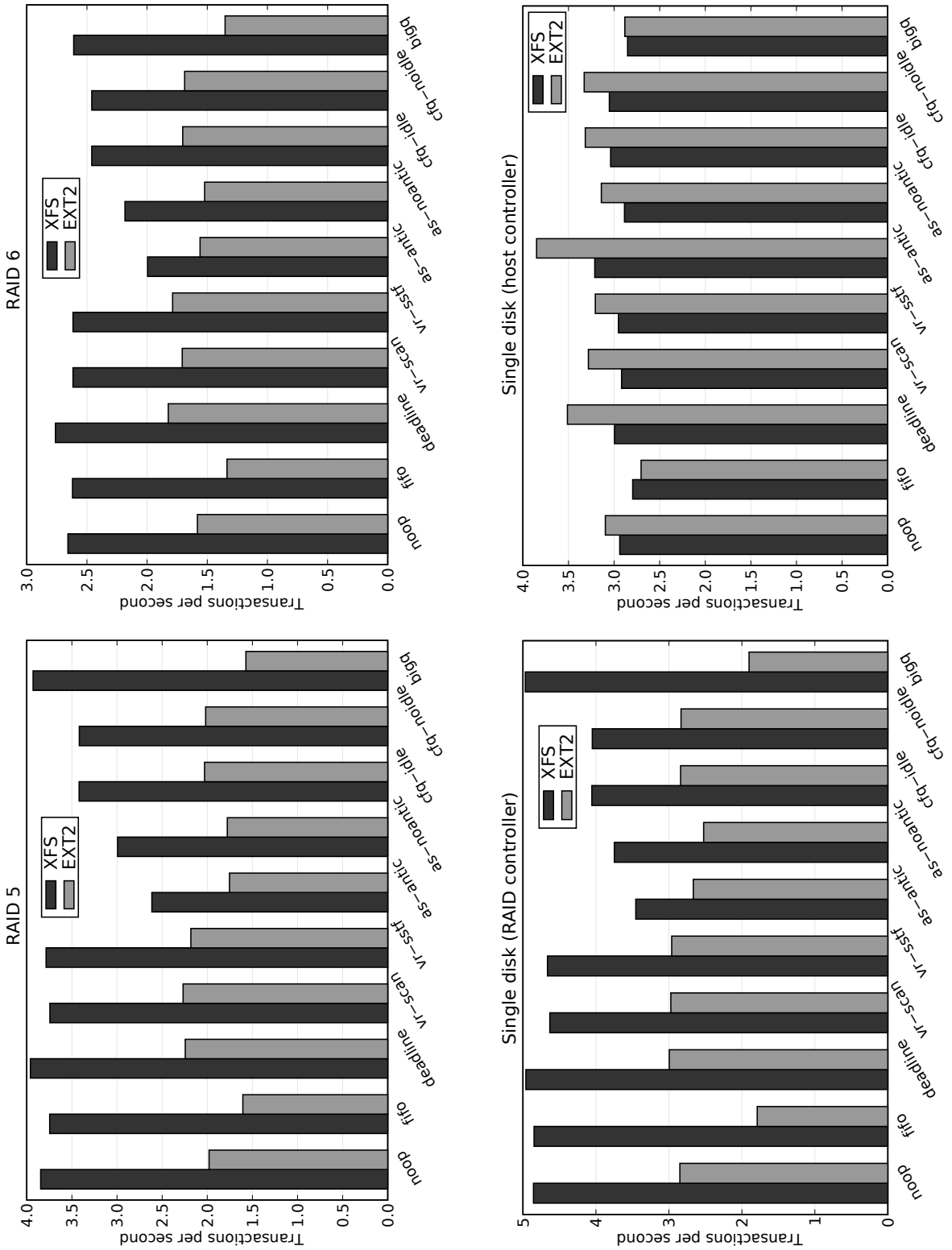

Figure 5.4: Postmark results 2.

The two FIFO tests (shallow and deep queue) show very similar performance results; averages of 3.54 and 3.62 TPS respectively, and a maximum difference of 0.44 TPS (occurring in the RAID 0 test). This suggests that the I/O schedulers on the RAID and disk controllers are ineffective, or do not benefit from a large pool of requests to schedule.

The most significant difference between noop and FIFO is that noop merges requests. In terms of performance, noop always performs at least as well as FIFO, and this is most clearly seen on EXT2 where FIFO always performs the worst. This is probably because XFS performs its own merging and EXT2 does not.

On XFS, the average throughput of FIFO is 5.25 TPS, while for noop, it is 5.30 TPS. The maximum difference between FIFO and noop in any one test is 0.14 TPS. Thus, merging does not appreciably improve performance in this case. On the other hand, the average performance of FIFO and noop on EXT2 is 1.84 and 2.49 TPS respectively, with a maximum difference of 1.10, and a minimum of 0.35. Clearly, merging is far more effective in this case, providing an average 36% throughput improvement.

In this particular benchmark, CFQ behaves almost identically with the idle window both disabled and enabled; an average of 3.48 TPS in both cases with a maximum difference of 0.03 TPS. Generally, CFQ performs better than AS, but not as well as V(R).

Finally, we note that V(R), in both SSTF and SCAN mode, perform practically the same, with a TPS of 3.80 averaged over all runs, and a maximum difference of only 0.09. Both algorithms tend to perform well, achieving an average of 0.32 TPS less than deadline.

	Throughput (TPS)	
Scheduler	EXT2	XFS
noop	2.49	5.31
FIFO	1.84	5.25
deadline	2.78	5.45
V(R) SCAN	2.75	4.85
V(R) SSTF	2.73	4.88
AS	2.35	3.74
AS (no antic.)	2.25	4.32
CFQ	2.50	4.46
CFQ (no idle)	2.50	4.46
FIFO (deep queue)	1.86	5.38

Table 5.1: Average per-instance Postmark throughput.

5.3 Anticipation

The bandwidth results of the anticipation benchmark are shown in Figures 5.5–5.10. A limited number of graphs are shown for each disk configuration; the full graphs, including all tested think-times and metrics, are available in Appendix C.

Table 5.2 shows, for each disk configuration, the maximum think-time for which anticipation enabled results in improved performance compared with anticipation disabled. Also tabulated is the approximate process load, for that think-time, where the performance of the two schedulers is equal.

Configuration	Think-time (μs)	Max. processes
Disk (host ctrl.)	2000	32
Disk (RAID ctrl.)	2000	22
RAID 1+0 (2)	1000	35
RAID 0 (2)	1000	28
RAID 0 (4)	1000	28
RAID 5 (3)	1000	12
RAID 1+0 (4)	500	28
RAID 5 (5)	500	10
RAID 1+0 (8)	100	16
RAID 0 (8)	100	4
RAID 5 (8)	100	4

Table 5.2: Maximum think-time for effective anticipation and the corresponding maximum random process load.

As the number of disks in the array increases, the effectiveness of anticipation decreases, irrespective of the RAID configuration. This is because, by design, AS idles the disk system at certain times. By increasing the array size, more disk resources go under-utilised when idling. At a certain array size, think-time and process load, the bandwidth lost due to idling outweighs the gains due to anticipation. For a given array configuration, we define the set of process loads and think-times for which anticipation improves performance as the *anticipation window*.

Over all disk configurations, anticipation is never effective where think-time exceeds 2ms or the array size equals 8 disk. While the 8-disk arrays do show a very minor performance improvement through anticipation, the gains occur over a small range of loads. The 4- and 5-disk arrays, however, display significant bandwidth improvements under anticipation.

At think-times of 8ms and above, the large-queue runs perform approximately equal, as do the small-queue ones. This is because AS internally disables anticipation, which occurs when the think-time exceeds `antic_expire`, whose default value is 6.7ms. It can be seen that

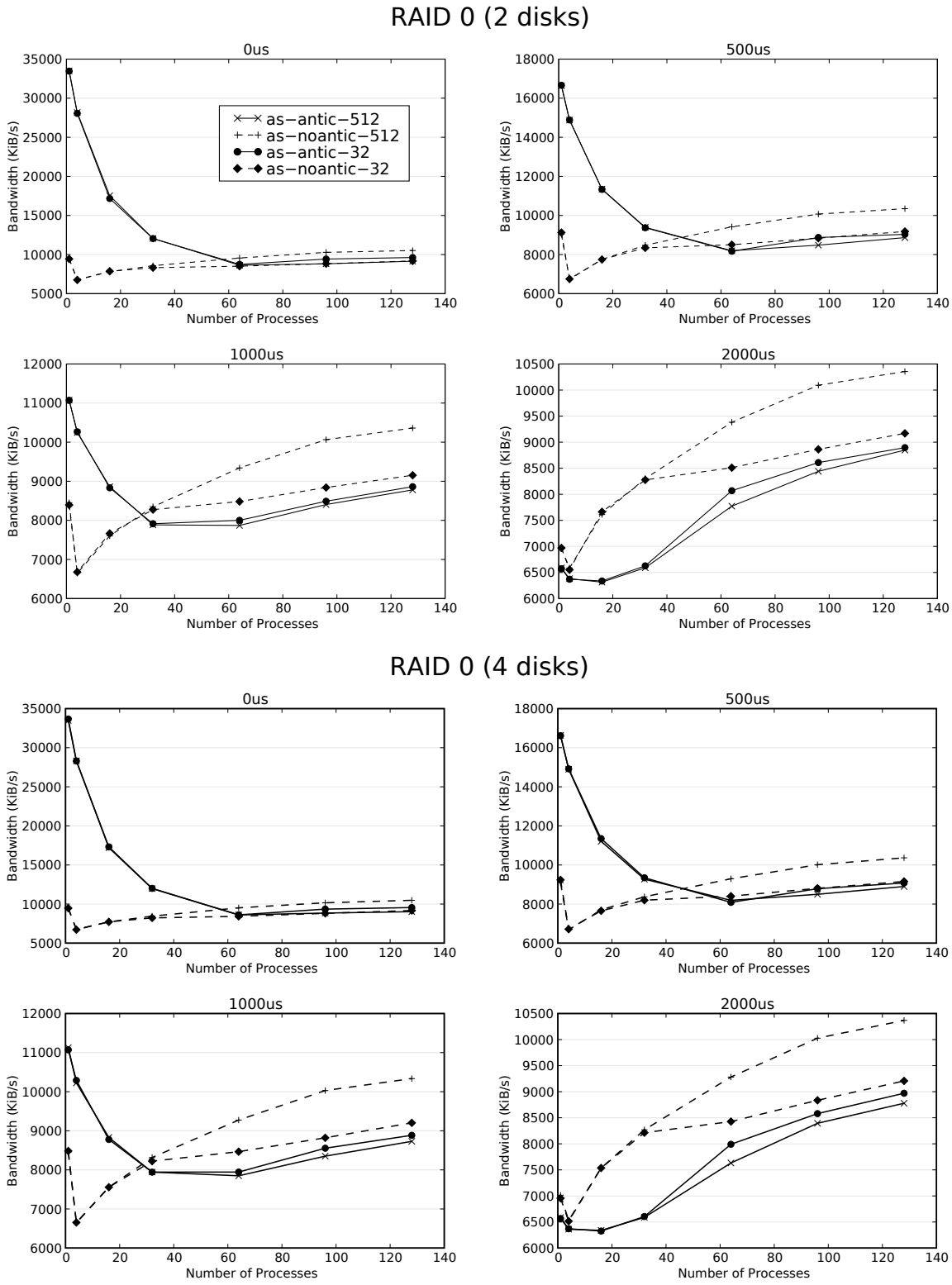

Figure 5.5: Anticipation bandwidth results 1.

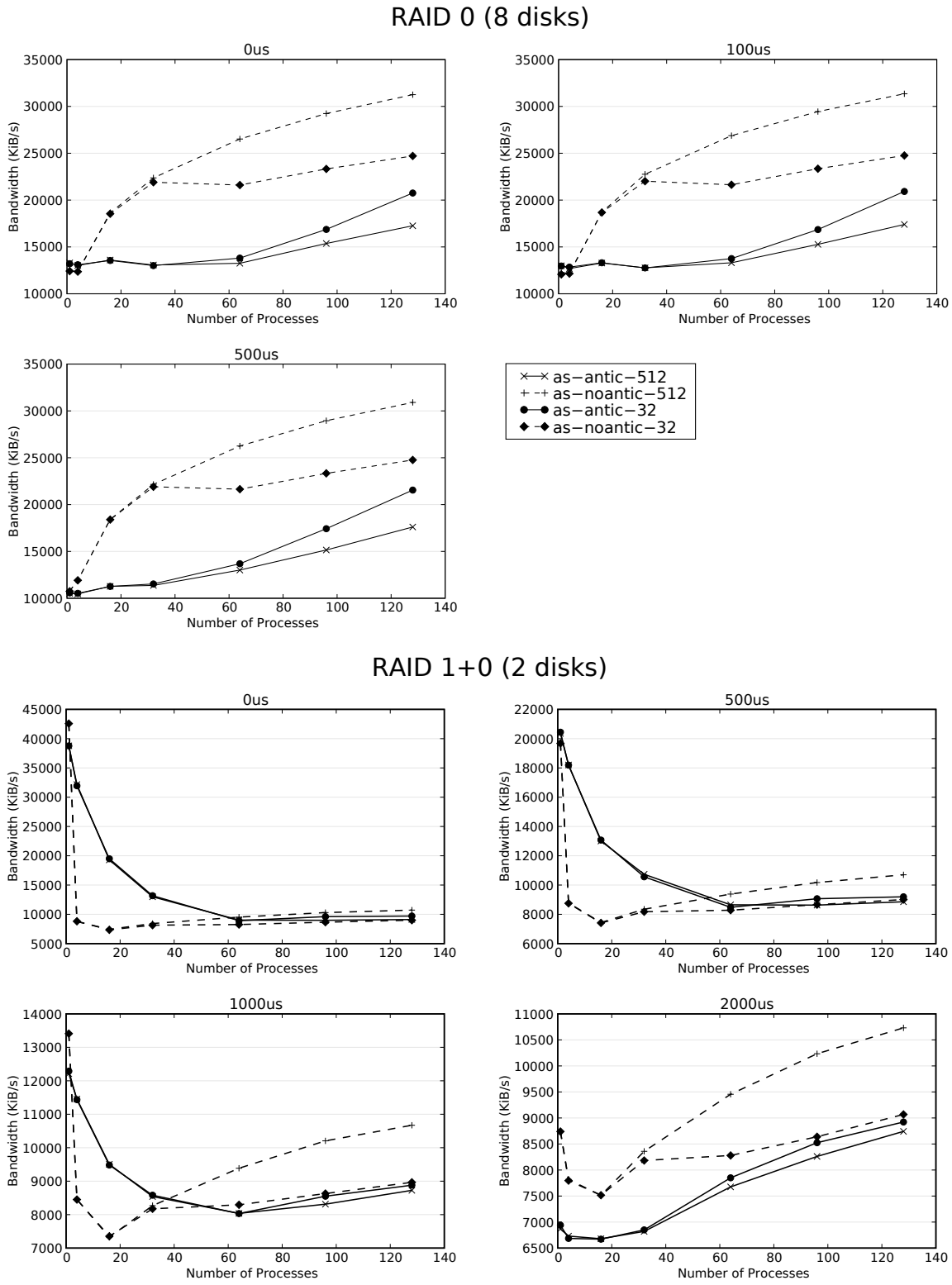

Figure 5.6: Anticipation bandwidth results 2.

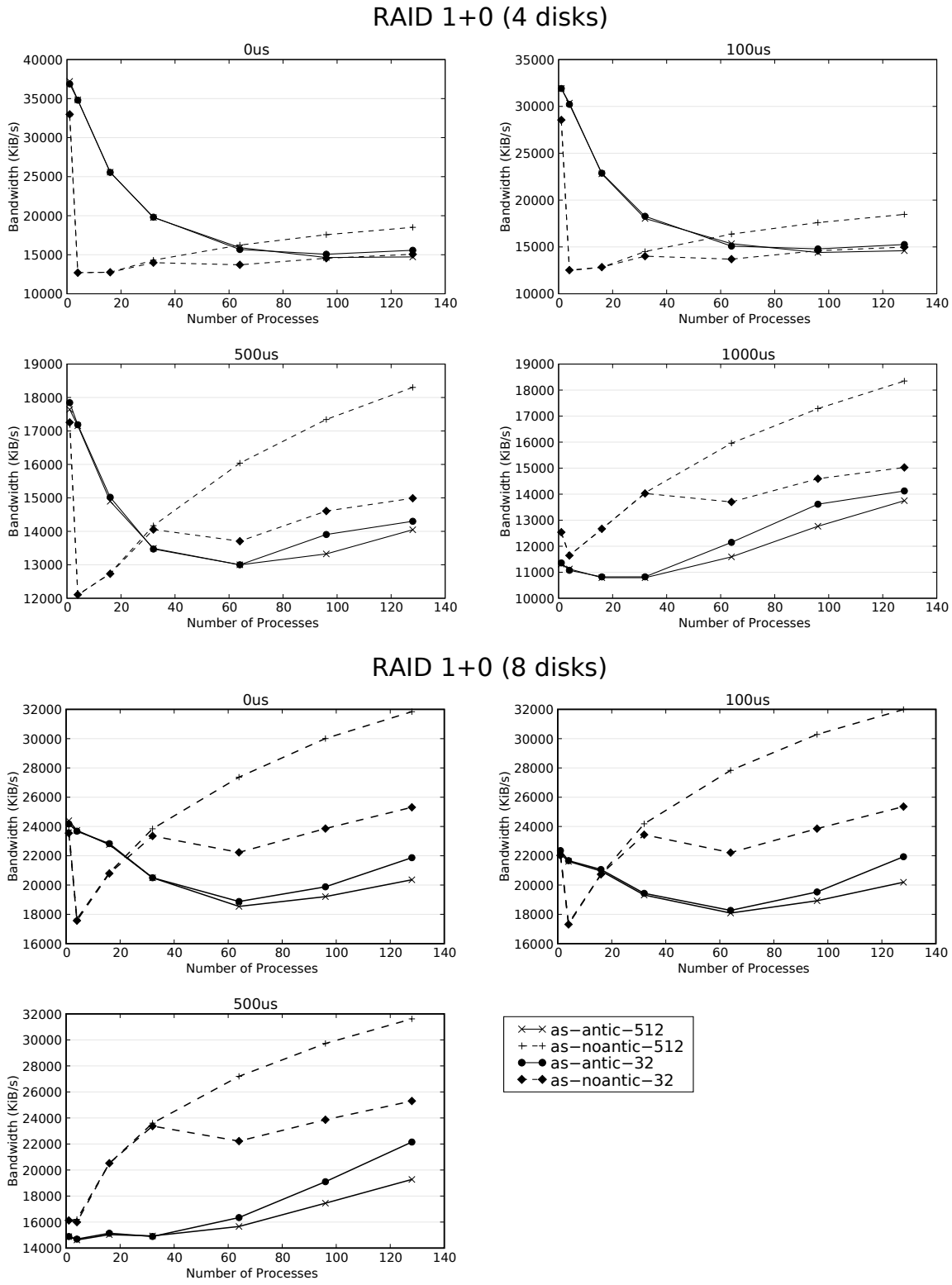

Figure 5.7: Anticipation bandwidth results 3.

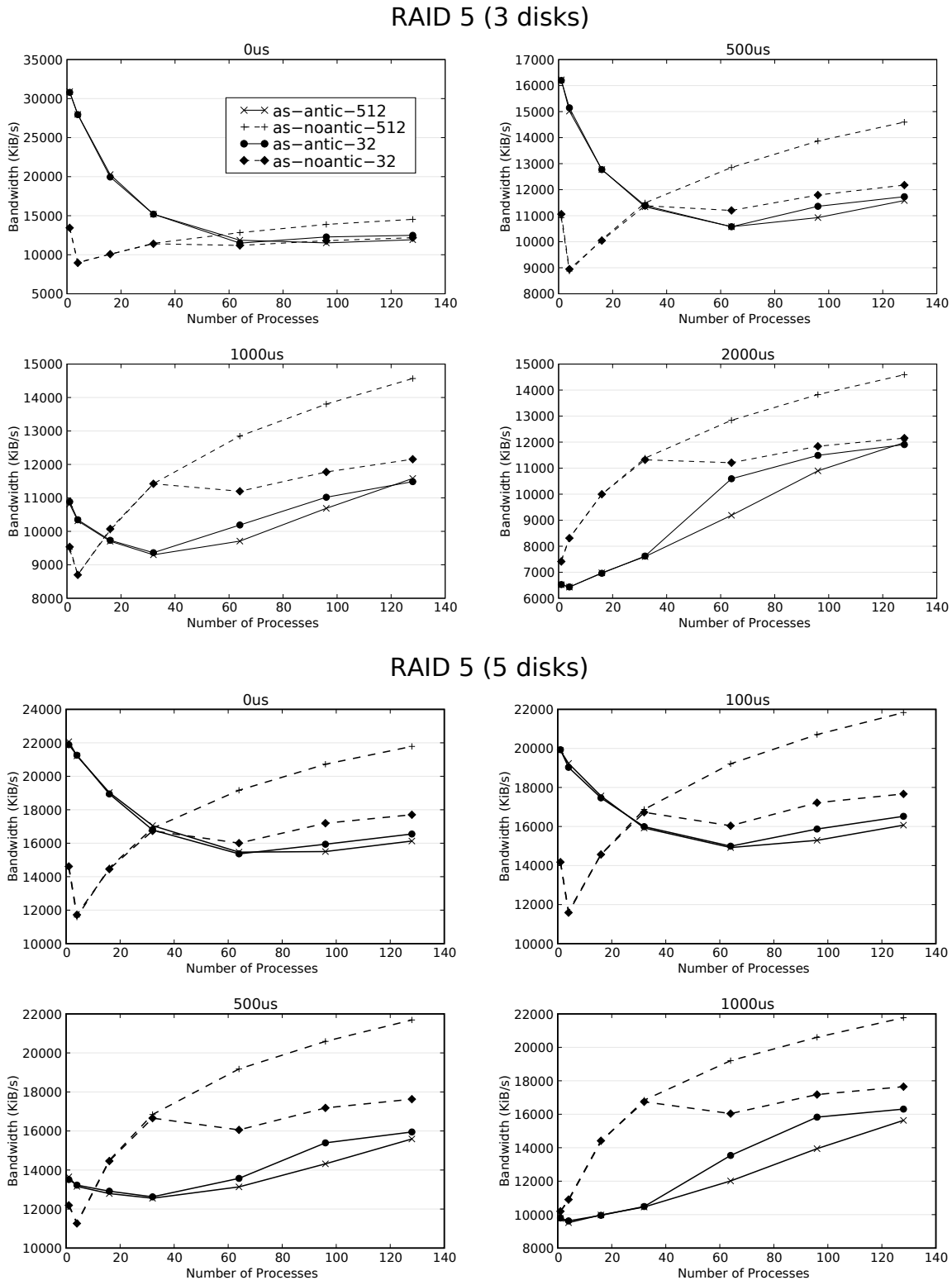

Figure 5.8: Anticipation bandwidth results 4.

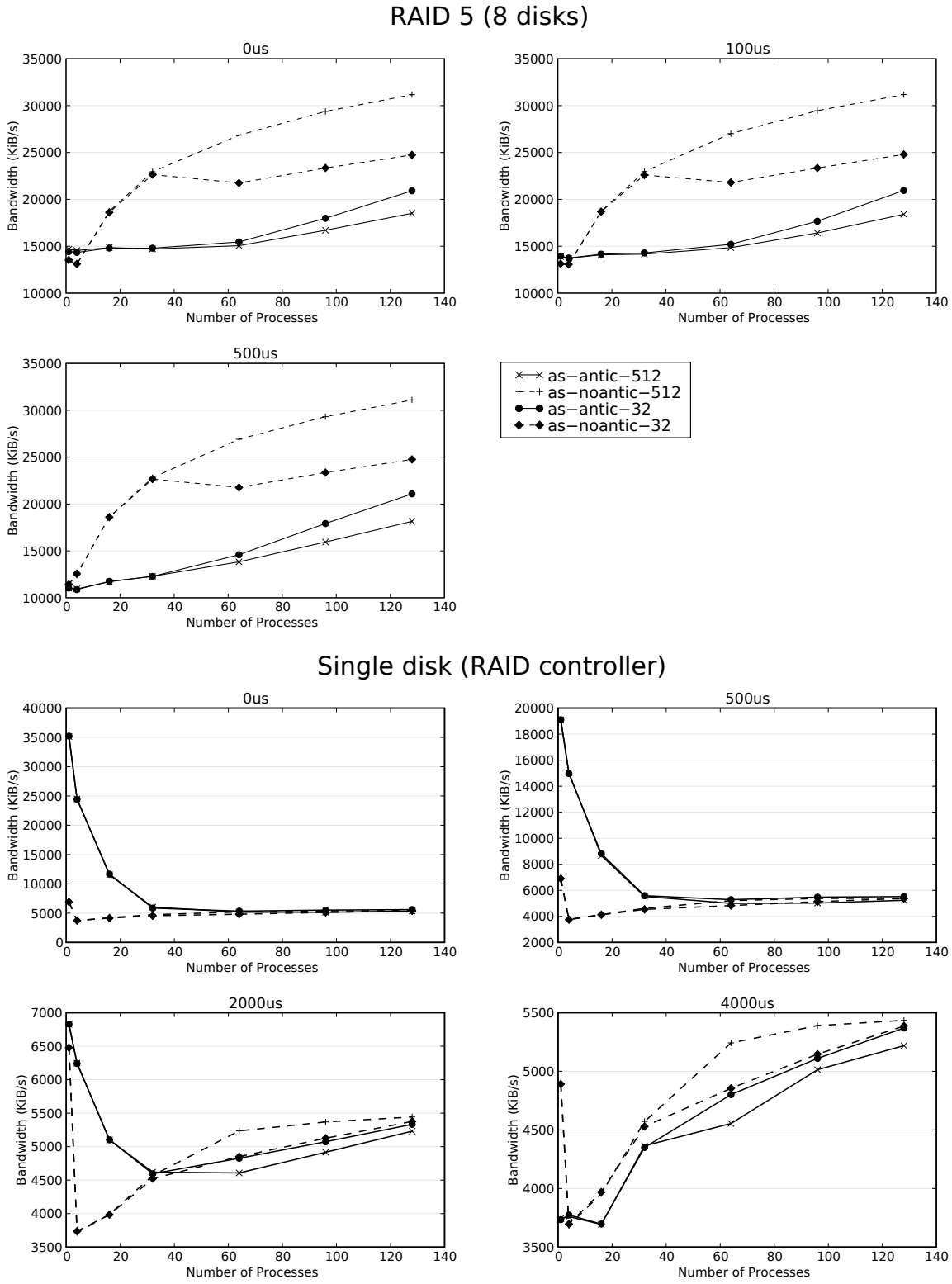

Figure 5.9: Anticipation bandwidth results 5.

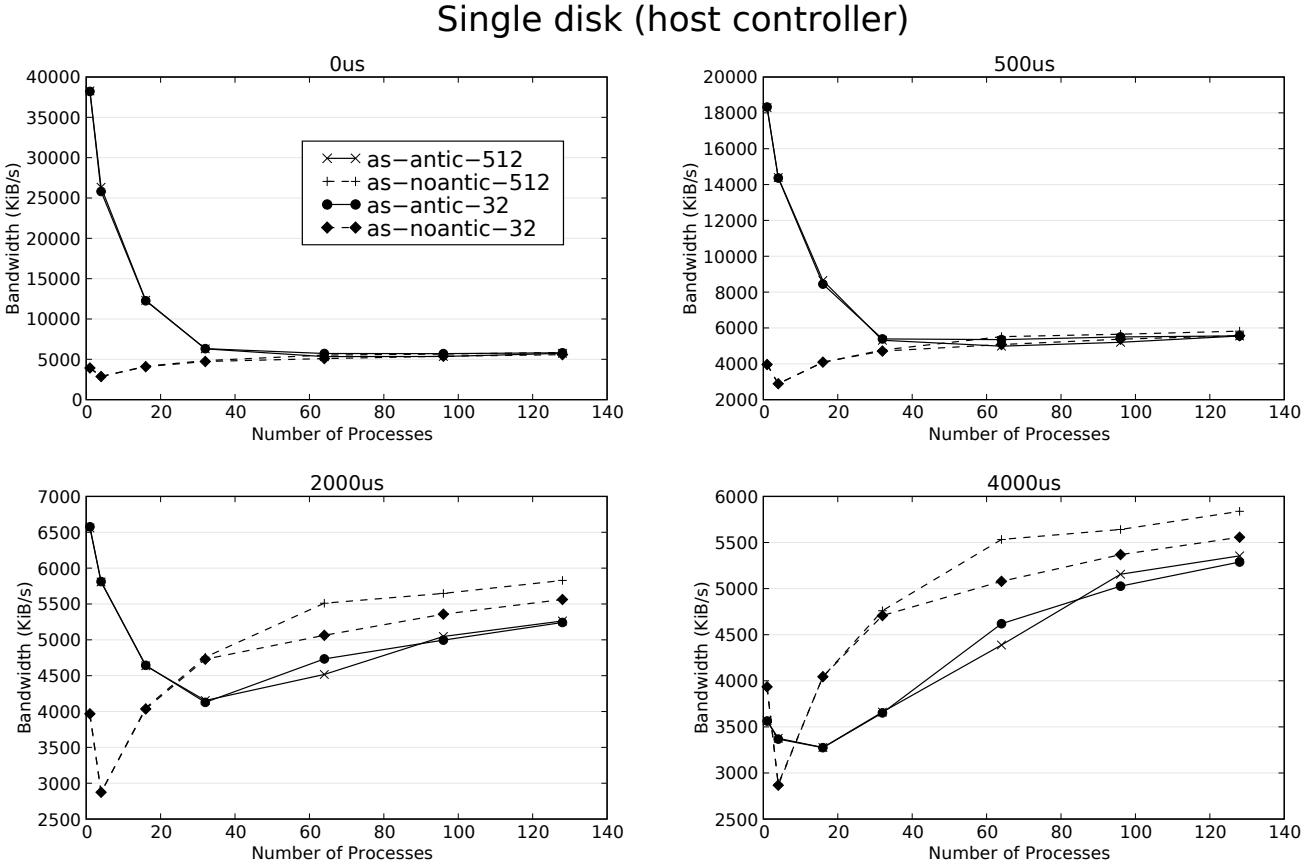

Figure 5.10: Anticipation bandwidth results 6.

the anticipation-enabled schedulers still take a small performance penalty, despite anticipation being internally disabled.

Roughly speaking, the anticipation-disabled curves scale positively with increasing load. This is not generally true of the anticipation-enabled results, which often scale negatively to a point, and scale positively thereafter. Two observations lead to the conclusion that this is not due to the anticipation algorithm itself:

1. The curves are quite smooth (accounting for the sparseness of the data points). One would expect a discontinuity at the point where anticipation disables;

2. The anticipation-enabled curves do not subsequently reach the performance of the anticipation-disabled curves.

A likely explanation is that, after a certain process load, the extra bandwidth gained from additional request concurrency (and hence a better schedule) offsets the bandwidth loss from bad anticipation.

At a 32-process load, the two anticipation-disabled curves—identical up to this point—begin to diverge significantly. This is because the small-queue saturates, causing performance to plateau, while the large-queue curve continues to grow smoothly. These results mirror what we saw in the FIO micro-benchmarks (Section 5.1).

The same phenomenon is not seen in the anticipation-enabled curves. Within the anticipation window, performance is largely independent of the queue depth, even above 32 processes. However, for larger process loads, the small-queue scheduler achieves greater throughput than the large-queue, and tends to approach the performance of the anticipation-disabled small-queue curve.

5.4 Queue depth

The deadline I/O scheduler and XFS filesystem were chosen for this benchmark after analysing the results of Section 5.2, the first Postmark benchmark. This combination represents the best average performance over all tests run. The results of the queue depth benchmark are graphed in Figure 5.11, shown in transactions per second (TPS) per process.

A queue depth of 32 is the optimal point for about half of the RAID arrays, with the others experiencing optimality at queue depths of 16 or 64 requests. However, in many cases, there is a significant performance drop at 64 requests and greater.

The optimal queue depth over all configurations is 32 requests. It achieves an average of 3.78 TPS, while 16 achieves 3.73 TPS, and 64 3.59 TPS. This experimentally justifies our reasoning in Section 4.3.2 for the selection of a 32-deep queue.

Interestingly, there is no obvious correlation between optimal queue depth and RAID level or array size. Moreover, each RAID level and array size exhibits approximately the same behaviour; a sharp ramp-up from a unit queue depth to the optimal point at 16, 32 or 64 queue depth, and a gentle ramp-down from that point to 256. The performance at 256 requests achieves an average of 80% of that at the optimal point.

The exception to this is RAID 6; with 8 disks, the performance at 256 queue depth is greater than that at 128. This may be due to the amount of work required by RAID 6 arrays for write requests; a larger queue depth means the controller has more information and can avoid more partial-stripe writes, which are very expensive. One would expect to see similar behaviour for other RAID 5 and 6 arrays, which is not the case. An alternative hypothesis is that, because RAID 6 has the smallest stripe-unit size of all the array configurations, the benefits of merging are somewhat lost; requests end up being split-up again in the RAID controller. These is no further evidence to substantiate either of these claims.

RAID 6 is also interesting in that it is the only array type to show multiple maxima in the performance curves.

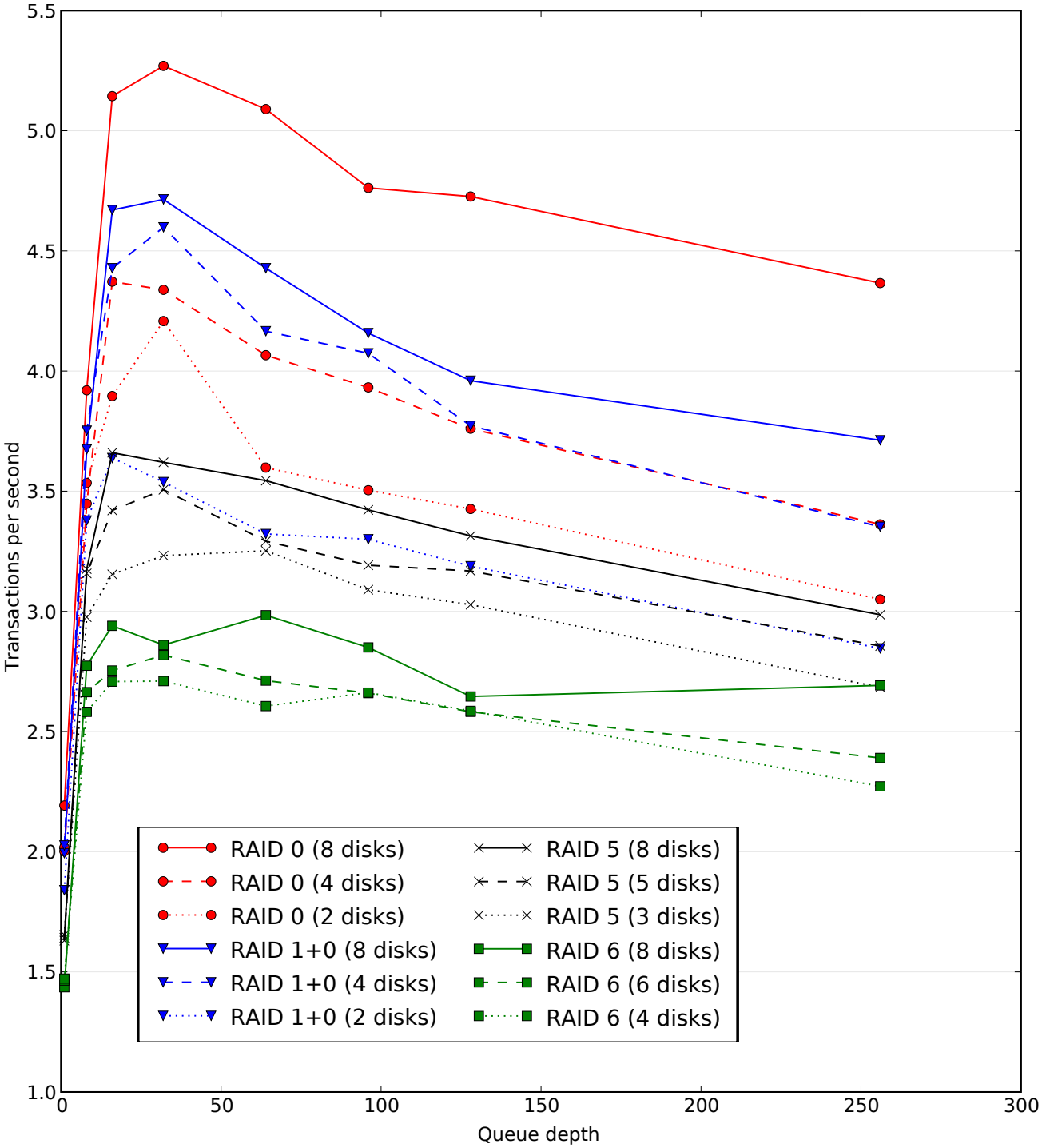

Figure 5.11: Postmark queue depth results.

Chapter 6

Discussion

We have shown that I/O scheduling is indeed a useful technique for improving performance on hardware RAID systems. The classic techniques of seek minimisation, request merging and anticipation, all apply to hardware RAID arrays of small to medium size. However, as other studies have found, I/O scheduler performance is heavily dependant on the workload.

6.1 Seek minimisation

Our results suggest that seek minimisation leads to acceptable performance over a wide range of workloads and disk configurations. While there are several instances where seek-minimising schedulers are substantially outperformed by FCFS-like algorithms, these can be attributed to poor heuristics relating to features like deadlines and anticipation. The pure SSTF and SCAN implementations show consistently good performance in all cases.

For single-disk storage, the literature is very clear about the advantages of seek-optimising schedulers. Our results indicate this class of algorithms does not harm performance on RAID-based systems, and indeed can improve performance on some workloads. As such, a good general-purpose, scalable I/O scheduler should implement seek-minimisation.

Contrary to the literature, our results do not indicate any substantial throughput or latency differences between SCAN and SSTF—the two classic seek-reduction techniques—either on RAID or on single disks. However, one could argue that SCAN-like algorithms are a superior choice for a scalable I/O scheduler, since they are generally preferred on single-disk systems and are inherently free from starvation.

6.2 Merging

The value of merging is clearly demonstrated in the generic Postmark benchmark when run on an EXT2 filesystem. It shows a throughput improvement in every case, approximately 35% on average. The same results are not seen in our micro-benchmarks, simply because the workloads do not issue mergeable requests. Most importantly, in no case does merging *reduce* throughput.

On closer inspection, this is not necessarily an obvious result. Because of data striping, requests may need to be split in the RAID controller at the cost of processor and memory capacity—particularly when small stripe-unit sizes are used. Quantifying this effect could be done by varying the stripe-unit size while holding all other conditions constant. However, we did not have time to conduct such an experiment, so we relegate it to future work. Based on the available evidence, the performance loss due to splitting is negligible, either in absolute terms, or compared with the gains of merging in the host.

Queue depth While the results are unambiguous about the importance of merging, doing it successfully is hindered by an intrinsic feature of RAID hardware: command queueing. Queueing is necessary on hardware RAID, since otherwise the enormous parallelism benefits of multiple physical disks are lost. On the other hand, deep queues mean requests are issued to the device sooner and in greater numbers, leaving less room for merging at the host.

Our queue depth benchmark shows this clearly. On average, a queue depth of 32 is the optimal balance between request concurrency and merge ratio. Amazingly, this is largely independent of the number of disks forming the array. For a 2-disk RAID 0 array, a queue depth of 8 results in a negligible probability ($\approx 1\%$) of under-utilisation, yet a 32-deep queue performs 20% better. Some unknown effects are clearly at play.

We have been unable to determine a model for selecting the optimal queue depth. However, we have seen that choosing a queue depth that is slightly larger than optimal does not have a drastic performance effect, while an overly-small one does. Thus, until a model can be determined, queue depth for a specific RAID-based storage system should be determined by experimentation with a modest safety margin applied.

In summary, merging is an important feature for RAID-friendly I/O scheduling, but it must be supported by a sufficiently-small queue depth on the controller to see any real performance advantage.

6.3 Anticipation

We have seen, perhaps surprisingly, that anticipation is a useful technique even on moderately large arrays. We have determined that the maximum array size amenable to anticipation is approximately six disks. However, this is certainly dependent on the performance characteristics of the individual disks. We conjecture that the effectiveness of anticipation depends on the ratio of sequential throughput to seek time. This is because anticipation trades the combined random throughput of many disks (which is proportional to the seek time) for the sequential throughput of a single disk. Since only a single type of disk was available to us, we were unable to verify this claim.

Outside the anticipation window, our results show that the penalty for bad anticipation, especially at low to medium loads, is quite substantial. This is more pronounced on RAID than a single disk, since a greater amount of potential bandwidth can be lost. As such, the anticipation

heuristic for a RAID-aware scheduler must be highly conservative. It is also important, more so on RAID than single disks, that the heuristic is carefully tuned for the hardware and workload used.

Queue depth is largely irrelevant as far as anticipation is concerned. Neither the performance gains or the effective load range are dependent on queue depth, since AS must artificially force a unit queue depth by issuing requests one at a time to ensure they are satisfied in the desired order. Outside the anticipation window, the anticipation heuristic should ideally disable anticipation, in which case our non-anticipatory results apply. Thus, the decision to enable anticipation should not be affected by the queue depth.

6.4 Linux schedulers

We have found deadline to yield the best performance overall, especially at low- and medium-level workloads. However, under heavy workloads, deadline degrades to FCFS behaviour. This can be solved by simply increasing the values of `read_expire` and `write_expire` when it is known that the device will be under high load.

The root of the problem is deadline's static expiries; when the average request latency exceeds the expiry time, most requests will expire since the deadlines can not be met. A potential improvement to deadline is to dynamically scale expiries as a function of the average request latency. This means the deadline avalanche problem is avoided, while resulting in tighter deadlines when under low load.

We have shown that AS's anticipation can yield significant performance improvements. However, the cost of incorrect anticipation is quite high, and our results suggest that the anticipation heuristic is poorly tuned, even in the case of a single disks—the very situation AS is designed for.

Since the non-anticipatory version of AS shows no compelling reasons to be used over deadline, AS should be used only on small arrays with workloads that are known to be amenable to anticipation. With appropriate tuning of the anticipation heuristic, we believe AS could be a good general-purpose scheduler.

CFQ has been shown to perform relatively well in many cases. However, we identified an implementation issue which can lead to very poor results. The cause of this behaviour is that CFQ, by design, does not disable its idle window feature for processes doing random I/O. The motivation for this is that fair distribution of disk time is improved by allowing a small window in which a process can issue more I/O before the next process is serviced. This only occurs on devices that are not queueing-capable to prevent high-end disk hardware performance suffering. A bug in the CFQ implementation prevents detection of queueing-capable devices when presented with certain workloads.

As a work-around for this problem, we suggest that the idle window feature should be manually disabled by setting `slice_idle`=0 whenever using CFQ on hardware RAID, regardless of the workload or RAID configuration. There is no evidence to suggest this will negatively affect performance. CFQ with idle window disabled outperforms the idle-enabled version in all of our benchmarks. Moreover, in normal operation, CFQ internally disables the idle window on queueing-capable hardware like RAID.

Finally, the noop scheduler has been shown to perform well on some workloads, and poorly on others. In particular, it does not scale well on purely random or sequential workloads. Thus, noop should only be used on arrays servicing light to medium workloads.

6.5 Limitations

The main limitation of this work is that only one particular RAID controller, HP's P600, was subject to investigation. We also tested only a single type of disk. As such, it is difficult to determine which conclusions are valid only for the specific controller, and which apply to hardware RAID in general. Features such as I/O scheduling, prefetching, and cache management, could vary wildly between models and manufacturers. The performance characteristics of the disks, such as seek time and transfer speed, could also have an effect.

Moreover, we were not able to fully explore the configuration options of the P600 controller which are present in many hardware RAID devices. This includes parameters such as such as stripe-unit size and buffer/cache sizes, which we left at their default values.

Another limitation is the broadness of the benchmarks. We ran two benchmarks to test general performance characteristics: FIO micro-benchmarks and Postmark. These do not cover a wide spectrum of real-world workloads, so important results may have been missed. Partly, this is due to the lack of available tools.

In studying the benchmark results, we focused primarily on throughput as a measure of performance. CPU and memory overheads can also be important. This is particularly true in large I/O systems like hardware RAID, which can move a much larger amount of data, and thus has to deal with more interrupts, longer queues, etc.

We did not directly measure CFQ's fairness properties, nor did we investigate I/O priorities.

Due to hardware availability issues, the maximum array size we were able to test was 8 disks. However, RAID arrays can be much larger than this. In particular, the P600 can support up to 38 drives per controller. However, we can extrapolate many of the results obtained to larger arrays, with a reasonable degree of confidence, where the measurements have shown scaling limits within the configurations tested. One example is anticipation, which we have shown to be largely ineffective on 8-disk arrays. It is reasonable to assume that the same is true for larger arrays.

Despite our aims to study I/O scheduling on RAID hardware in the abstract, all our benchmarks were run on a single operating system: Linux. Hence, we were testing particular implementations of I/O scheduling algorithms. This problem is somewhat mitigated by the use of custom-built implementations of classic scheduling algorithms, which were not complicated by the requirements of real-world I/O schedulers. However, we are still at the mercy of peculiarities in Linux's I/O stack, such as optimisations like queue plugging[1], readahead, buffering and caching policies, and indeed implementation bugs.

6.6 Future work

There are many limitations, described above, that must be resolved before I/O scheduling on hardware RAID can be considered a "solved" problem. We have also uncovered sub-optimal performance in Linux's I/O scheduling, such as the poor anticipation heuristic and bugs in the CFQ implementation, which we would like to resolve. However, several more fundamental questions have been uncovered as a result of our work.

Write-back considered harmful One issue is that of write-back buffering, which complicates I/O scheduling. The problems all stem from the fact that write-back buffers generate completion events when the request is committed to the buffer, not when it is written to the (non-volatile) disk medium.

Write-back buffers pose a problem for resource usage accounting, because the actual time to service a write request is unknown. Measuring short-term (i.e. before buffer saturation) write bandwidth is also impossible. Thus, truly fair queueing is not possible. Real-time I/O scheduling is also complicated, because the maximum latency depends on the number of outstanding requests on the device. Under a write-back buffering scheme, this is not known.

Write-back can also compromise robustness, particularly to filesystems, since it is not known when a write request is physically written to the disk medium. Such problems are currently ignored or solved at the cost of performance. Several other performance problems are caused by write-back buffering. For instance, anticipatory scheduling does not function correctly if the idle time is actually used to issue requests that have been queued in the write-back buffer. Moreover, write-through buffers are immune to the adversary effect (see Section 2.1.3).

These issues are compounded in hardware RAID systems, since they often have very large write-back buffers; up to 512 MiB in the P600. Some hardware, including the P600, attempt to eliminate some of these problems by using expensive battery-backed write-back buffers.

We believe that most advantages of disk-level write-back buffering are provided by command queueing, so the write-back buffer can be replaced by queued write-through buffers without adverse performance implications.

[1]Queue plugging delays new I/O streams until multiple requests have been accumulated, with the aim of producing a more efficient schedule.

Propagating information up the I/O stack One of the most challenging factors with host I/O scheduling on hardware RAID is the lack of information about the underlying device made available to the scheduler. This is a problem even on simpler hardware, with features like command queueing becoming commonplace.

Performance optimisations may be possible in the I/O scheduler by providing it with such information as:

- type of underlying hardware;
- RAID topology and stripe-unit boundaries;
- queue depth.

Knowing the stripe-unit boundaries, for instance, allows us to prevent merging of requests that would be split in the RAID controller anyway. This may lead to performance gains, especially on low-end RAID hardware. Additionally, knowing the full details of the RAID topology, specifically the logical to physical block mapping, may allow per-disk anticipatory scheduling, whereas current anticipation techniques only work over entire logical devices.

Software RAID Software RAID, while providing similar properties to end-users of I/O systems, is significantly different from hardware RAID from an I/O scheduling perspective, with a range of new optimisations available.

Because the volume is managed purely in software, the I/O scheduler can have far more knowledge about the RAID volume which can be used to produce a better schedule. Moreover, scheduling policies can be applied over entire logical volumes, like in the hardware RAID cause, but also on the physical disks forming the array. This allows, for instance, fairness provided at the volume level, while performance is provided at the disk level where it is most effective.

Chapter 7

Conclusions

We have studied I/O scheduling on hardware RAID arrays and found that it is indeed a useful technique for improving performance. In particular, both sorting and merging can lead to non-trivial performance advantages. Anticipation can also be successfully applied to hardware RAID, but is only effective on arrays of approximately six disks or less.

Contrary to intuition, reducing the RAID device's queue depth is important to performance, because it allows more effective merging to take place on the host system. We found a queue depth of 32 to be optimal for the specific hardware tested, which is largely independent of the RAID level or size.

Of the four Linux I/O schedulers, we have found deadline to yield the best performance overall, especially at low- and medium-level workloads. While deadline degrades to FCFS behaviour under heavy workloads, we have suggested a simple solution to resolve this problem.

We have shown that, while anticipation can yield significant performance improvements, the cost of incorrect anticipation is high. Moreover, Linux's AS scheduler is poorly tuned, and should be avoided unless the workload is known to be anticipation-friendly.

While CFQ has been shown to perform relatively well in many cases, we have identified a serious performance issue, and suggested a work-around.

We have also developed two new I/O schedulers for the Linux kernel, FIFO and V(R), and contributed several performance enhancements.

Bibliography

[ABZ02] M. Andrews, M.A. Bender, and L. Zhang. New algorithms for disk scheduling. *Algorithmica*, 32(2):277–301, 2002.

[Axb08a] Jens Axboe. blktrace homepage. `http://git.kernel.dk/?p=blktrace.git;a=summary`, May 2008.

[Axb08b] Jens Axboe. FIO homepage. `http://git.kernel.dk/?p=fio.git;a=summary`, May 2008.

[BFH02] R. Bryant, R. Forester, and J. Hawkes. Filesystem performance and scalability in Linux 2.4.17. In *Proceedings of the 2002 Annual USENIX Technical Conference, FREENIX Track*, pages 259–274, 2002.

[CKR72] E. G. Coffman, L. A. Klimko, and Barbara Ryan. Analysis of scanning policies for reducing disk seek times. *SIAM Journal on Computing*, 1(3):269–279, 1972.

[CLG+94] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, and D.A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[CP90] P.M. Chen and D.A. Patterson. Maximizing performance in a striped disk array. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 322–331, 1990.

[CT96] S. Chen and D. Towsley. A performance evaluation of RAID architectures. *IEEE Transactions on Computers*, 45(10):1116–1130, 1996.

[CTT94] R. Card, T. Tso, and S. Tweedie. Design and implementation of the second extended filesystem. In *Proceedings of the First Dutch International Symposium on Linux*, pages 90–367, 1994.

[GD87] R. Geist and S. Daniel. A continuum of disk scheduling algorithms. *ACM Transactions on Computer Systems*, 5(1):77–92, 1987.

[GH03] E. Grochowski and R.D. Halem. Technological impact of magnetic hard disk drives on storage systems. *IBM Systems Journal*, 42(2):338–346, 2003.

[Hit08] Hitachi Global Storage Technologies. Ultrastar C10K147 data sheet, March 2008.

[Hof80] M. Hofri. Disk scheduling: FCFS vs. SSTF revisited. *Communications of the ACM*, 23(11):645–653, 1980.

[HP08] Hewlett-Packard. QuickSpecs HP Smart Array P600 controller, March 2008.

[HSA02] J. Hsieh, C. Stanton, and R. Ali. Performance evaluation of software RAID vs. hardware RAID for Parallel Virtual File System. In *Proceedings of the 9th International Conference on Parallel and Distributed Systems*, pages 307–313, 2002.

[ID01] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

[JW91] D.M. Jacobson and J. Wilkes. Disk scheduling algorithms based on rotational position. Technical Report HPL-CSP-91-7rev1, Hewlett-Packard Laboratories, 1991.

[Kat97] J. Katcher. Postmark: A new file system benchmark. Technical Report TR3022, Network Appliance, 1997.

[LSG02] C.R. Lumb, J. Schindler, and G.R. Ganger. Freeblock scheduling outside of disk firmware. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, pages 275–288, January 2002.

[McK90] P.E. McKenney. Stochastic fairness queueing. In *Proceedings of the 9th IEEE INFOCOM*, pages 733–740, 1990.

[MJLF84] M.K. McKusick, W.N. Joy, S.J. Leffler, and R.S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, 1984.

[PGK88] D.A. Patterson, G.A. Gibson, and R.H Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM International Conference on Management of Data*, Chicago, Illinois, United States, 1988.

[PH04] S. Pratt and D. Heger. Workload dependent performance evaluation of the Linux 2.6 I/O schedulers. In *Proceedings of the 2004 Ottawa Linux Symposium*, pages 425–448, 2004.

[RLLR07] A. Riska, J. Larkby-Lahet, and E. Riedel. Evaluating block-level optimization through the IO path. In *Proceedings of the 2007 Annual USENIX Technical Conference*, pages 247–260, 2007.

[RP03] L. Reuther and M. Pohlack. Rotational-position-aware real-time disk scheduling using a dynamic active subset (DAS). In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pages 374–385, 2003.

[RR06] A. Riska and E. Riedel. Disk drive level workload characterization. In *Proceedings of the 2006 Annual USENIX Technical Conference*, 2006.

[RW93a] C. Ruemmler and J. Wilkes. Modelling disks. Technical Report HPL-93-68, Hewlett-Packard Laboratories, 1993.

[RW93b] C. Ruemmler and J. Wilkes. UNIX disk access patterns. In *Proceedings of the 1993 Winter USENIX Technical Conference*, pages 405–420, 1993.

[San03] P. Sanders. Asynchronous scheduling of redundant disk arrays. *IEEE Transactions on Computers*, 52(9):1170–1184, 2003.

[SCO90] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *Proceedings of the 1990 Winter USENIX Technical Conference*, pages 313–323, 1990.

[Tan01] A.S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 2nd edition, 2001.

[TP72] T.J. Teorey and T.B. Pinkerton. A comparative analysis of disk scheduling policies. *Communications of the ACM*, 15(3), 1972.

[Wei08] E. W. Weisstein. Birthday problem. From *MathWorld* – A Wolfram Web Resource. `http://mathworld.wolfram.com/BirthdayProblem.html`, 2008.

[WGP94] B.L. Worthington, G.R. Ganger, and Y.N. Patt. Scheduling algorithms for modern disk drives. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):241–251, 1994.

Appendix A

Source Code

Source code is available on the accompanying disk in the `appendix_a` directory. The supplied files are:

- `fifo-iosched.patch`: adds the FIFO I/O scheduler to Linux 2.6.24;
- `vr-iosched.patch`: adds the V(R) I/O scheduler to Linux 2.6.24;
- `as-fixes.patch`: the modifications we made to the anticipatory I/O scheduler. Applies to Linux 2.6.23, and is included in Linux 2.6.24 mainline;
- `deadline-fixes.patch`: the modifications we made to the deadline I/O scheduler. Applies to Linux 2.6.23, and is included in Linux 2.6.24 mainline;
- `cciss-qdepth.patch`: adds the `queue_depth` tunable to the CCISS device driver in Linux 2.6.24.

Appendix B

Configuration Files

The `appendix_b` directory contains the configurations used for the Linux kernel build and Postmark benchmarks.

- `linux-2.6.24.config`: Linux configuration used on the benchmarking machine;
- `postmark-large.conf`: "large file" Postmark configuration for the benchmarks described in Section 4.2.4 and analysed in Section 5.2;
- `postmark-small.conf`: "small file" Postmark configuration;
- `postmark-qdepth.conf`: Postmark configuration for the queue depth benchmark (Section 4.3.6 and Section 5.4).

Appendix C

Benchmark Results

The complete set of graphs for our benchmark results are available in the `appendix_c` directory. The included files are:

- `fio-microbenchmarks.pdf`: results of FIO micro-benchmarks discussed in Section 5.1;
- `fio-anticipation.pdf`: results of FIO anticipation benchmarks discussed in Section 5.3;
- `postmark.pdf`: Postmark benchmark results analysed in Section 5.2.