Workload Dependent Performance Evaluation of the Linux 2.6 I/O Schedulers

Dominique A. Heger, Steven L. Pratt, Proceeding of the 2004 Linux Symposium (OLS), Ottawa, 2004

Abstract

The goal of this study was to quantify I/O performance under varying workload conditions in environments ranging from single-CPU single-disk setups to SMP systems that utilize large, sophisticated RAID systems. Incorporating different I/O workload patterns, the focus was on quantifying the baseline, as well as the optimized performance behavior under different I/O scheduler and file system configurations. The analysis resulted in establishing performance metrics that outline the I/O performance behavior and guide the configuration, the setup, and the fine tuning process. The point of convergence in the analysis was the entire I/O stack, incorporating the major software and hardware I/O optimization features that are present in the I/O path.

Introduction

This study was initiated to quantify I/O performance in a Linux 2.6 environment. The I/O stack in general has become considerably more complex over the last few years. Contemporary I/O solutions include hardware, firmware, as well as software support for features such as request coalescing, adaptive prefetching, automated invocation of direct I/O, or asynchronous write-behind polices. From a hardware perspective, incorporating large cache subsystems on a memory, RAID controller, and physical disk layer allows for a very aggressive utilization of these I/O optimization techniques. The interaction of the different optimization methods that are incorporated in the different layers of the I/O stack is neither well understood nor been quantified to an extent necessary to make a rational statement on I/O performance. A rather interesting feature of the Linux operating system is the I/O scheduler [6]. Unlike the CPU scheduler, an I/O scheduler is not a necessary component of any operating system per se, and therefore is not an actual building block in some of the commercial UNIX® systems. This study elaborates how the I/O scheduler is embedded into the Linux I/O framework, and discusses the 4 (rather distinct) implementations and performance behaviors of the I/O schedulers that are available in Linux 2.6. Section 1 introduces the BIO layer, whereas Section 2 elaborates on the anticipatory (AS), the deadline, the noop, as well as the completely fair queuing (CFQ) I/O schedulers. Section 2 further highlights some of the performance issues that may surface based on which I/O scheduler is being utilized. Section 3 discusses some additional hardware and software components that impact I/O performance. Section 4 introduces the workload generator used in this study and outlines the methodology that was utilized to conduct the analysis. Section 5 discusses the results of the project. Section 6 provides some additional recommendations and discusses future work items.

1 I/O Scheduling and the BIO layer

The I/O scheduler in Linux forms the interface between the generic block layer and the low-level device driver's [2],[7]. The block layer provides functions that are utilized by the file systems and the virtual memory manager to submit I/O requests to block devices. These requests are transformed by the I/O scheduler and made available to the low-level device drivers. The device drivers consume the transformed requests and forward them (by using device specific protocols) to the actual device controllers that perform the I/O operations. As prioritized resource management seeks to regulate the use of a disk subsystem by an application, the I/O scheduler is considered an imperative kernel component in the Linux I/O path. It is further possible to regulate the disk usage in the kernel layers above and below the I/O scheduler. Adjusting the I/O pattern generated by the file system or the virtual memory manager (VMM) is considered as an option. Another option is to adjust the way specific device drivers or device controllers consume and manipulate the I/O requests.

The various Linux 2.6 I/O schedulers can be abstracted into a rather generic I/O model. The I/O requests are generated by the block layer on behalf of threads that are accessing various file systems, threads that are performing raw I/O, or are generated by virtual memory management (VMM) components of the kernel such as the *kswapd* or the *pdflush* threads. The producers of I/O requests initiate a call to *__make_request(),* which invokes various I/O scheduler functions such as *elevator_merge_fn()*. The *enqueue* functions in the I/O framework intend to merge the newly submitted block I/O unit (a *bio* in 2.6 or a *buffer_head* in the older 2.4 kernel) with previously submitted requests, and to sort (or sometimes just insert) the request into one or more internal I/O queues. As a unit, the internal queues form a single logical queue that is associated with each block device. At a later stage, the low-level device driver calls the generic kernel function *elv_next_request()* to obtain the next request from the logical queue. The *elv_next_request()* call interacts with the I/O scheduler's dequeue function *elevator_next_req_fn()*, and the latter has an opportunity to select the appropriate request from one of the internal queues. The device driver processes the request by converting the I/O submission into

(potential) scatter-gather lists and protocol-specific commands that are submitted to the device controller. From an I/O scheduler perspective, the block layer is considered as the producer of I/O requests and the device drivers are labeled as the actual consumers.

From a generic perspective, every read or write request launched by an application results in either utilizing the respective I/O system calls or in memory mapping (*mmap*) the file into a process's address space [14]. I/O operations normally result in allocating *PAGE_SIZE* units of physical memory. These pages are being indexed, as this enables the system to later on locate the page in the buffer cache [10]. A cache subsystem only improves performance if the data in the cache is being reused. Further, the read cache abstraction allows the system to implement (file system dependent) read-ahead functionalities, as well as to construct large contiguous (SCSI) I/O commands that can be served via a single direct memory access (DMA) operation. In circumstances where the cache represents pure (memory bus) overhead, I/O features such as direct I/O should be explored (especially in situations where the system is CPU bound).

In a general write scenario, the system is not necessarily concerned with the previous content of a file, as a *write()* operation normally results in overwriting the contents in the first place. Therefore, the write cache emphasizes other aspects such as asynchronous updates, as well as the possibility of omitting some write requests in the case where multiple *write()* operations into the cache subsystem result in a single I/O operation to a physical disk. Such a scenario may occur in an environment where updates to the same (or a similar) inode offset are being processed within a rather short time-span. The block layer in Linux 2.4 is organized around the *buffer_head* data structure [7]. The culprit of that implementation was that it is a daunting task to create a truly effective block I/O subsystem if the underlying *buffer_head* structures force each I/O request to be decomposed into 4KB chunks. The new representation of the block I/O layer in Linux 2.6 encourages large I/O operations. The block I/O layer now tracks data buffers by using *struct page* pointers. Linux 2.4 systems were prone to loose sight of the logical form of the writeback cache when flushing the cache subsystem. Linux 2.6 utilizes logical pages attached to inodes to flush dirty data, which allows multiple pages that belong to the same inode to be coalesced into a single *bio* that can be submitted to the I/O layer [2]. This approach represents a process that works well if the file is not fragmented on disk.

2 The 2.6 Deadline I/O Scheduler

The deadline I/O scheduler incorporates a per-request expiration-based approach and operates on 5 I/O queues [4]. The basic idea behind the implementation is to aggressively reorder requests to improve I/O performance while simultaneously ensuring that no I/O request is being starved. More specifically, the scheduler introduces the notion of a per-request deadline, which is used to assign a higher preference to read than write requests. The scheduler maintains 5 I/O queues. During the *enqueue* phase, each I/O request gets associated with a deadline, and is being inserted in I/O queues that are either organized by the starting logical block number (a sorted list) or by the deadline factor (a FIFO list). The scheduler incorporates separate sort and FIFO lists for read and write requests, respectively. The 5th I/O queue contains the requests that are to be handed off to the device driver. During a dequeue operation, in the case where the dispatch queue is empty, requests are moved from one of the 4 (sort or FIFO) I/O lists in batches. The next step consists of passing the head request on the dispatch queue to the device driver (this scenario also holds true in the case that the dispatch-queue is not empty). The logic behind moving the I/O requests from either the sort or the FIFO lists is based on the scheduler's goal to ensure that each read request is processed by its effective deadline, without starving the queued-up write requests. In this design, the goal of economizing the disk seek time is accomplished by moving a larger batch of requests from the sort list (logical block number sorted), and balancing it with a controlled number of requests from the FIFO list. Hence, the ramification is that the deadline I/O scheduler effectively emphasizes average read request response time over disk utilization and total average I/O request response time.

To reiterate, the basic idea behind the deadline scheduler is that all read requests are satisfied within a specified time period. On the other hand, write requests do not have any specific deadlines associated with them. As the block device driver is ready to launch another disk I/O request, the core algorithm of the deadline scheduler is invoked. In a simplified form, the fist action being taken is to identify if there are I/O requests waiting in the dispatch queue, and if yes, there is no additional decision to be made what to execute next. Otherwise it is necessary to move a new set of I/O requests to the dispatch queue. The scheduler searches for work in the following places, BUT will only migrate requests from the *first source* that results in a hit. (1) If there are pending write I/O requests, and the scheduler has not selected any write requests for a certain amount of time, a set of write requests is selected (see tunables in Appendix A). (2) If there are expired read requests in the *read_fifo* list, the system will move a set of these requests to the dispatch queue. (3) If there are pending read requests in the sort list, the system will migrate some of these requests to the dispatch queue. (4) As a last resource, if there are any pending write I/O operations, the dispatch queue is being populated with requests from the sorted write list. In general, the definition of a *certain amount of time* for write request starvation is normally 2 iterations of the scheduler algorithm (see Appendix A). After two sets of read requests have been moved to the dispatch queue, the

scheduler will migrate some write requests to the dispatch queue. A set or batch of requests can be (as an example) 64 contiguous requests, but a request that requires a disk seek operation counts the same as 16 contiguous requests.

2.1 The 2.6 Anticipatory I/O scheduler

The anticipatory (AS) I/O scheduler's design attempts to reduce the per thread read response time. It introduces a controlled delay component into the dispatching equation [5],[9],[11]. The delay is being invoked on any new read request to the device driver, thereby allowing a thread that just finished its read I/O request to submit a new read request, basically enhancing the chances (based on locality) that this scheduling behavior will result in smaller seek operations. The tradeoff between reduced seeks and decreased disk utilization (due to the additional delay factor in dispatching a request) is managed by utilizing an actual *cost-benefit* analysis [9].

The next few paragraphs discuss the general design of an anticipatory I/O scheduler, outlining the different components that comprise the I/O framework. Basically, as a read I/O request completes, the I/O framework stalls for a brief amount of time, awaiting additional requests to arrive, before dispatching a new request to the disk subsystem. The focus of this design is on applications threads that rapidly generate another I/O request that could potentially be serviced before the scheduler chooses another task, and by doing so, *deceptive idleness* may be avoided [9]. Deceptive idleness is defined as a condition that forces the scheduler into making a decision too early, basically by assuming that the thread issuing the last request has momentarily no further disk request lined up, and hence the scheduler selects an I/O request from another task. The design discussed here argues that the fact that the disk remains idle during the short stall period is not necessarily detrimental to I/O performance. The question of whether (and for how long) to wait at any given decision point is key to the effectiveness and performance of the implementation. In practice, the framework waits for the shortest possible period of time for which the scheduler expects (with a high probability) the benefits of *actively waiting* to outweigh the costs of keeping the disk subsystem in an idle state. An assessment of the costs and benefits is only possible relative to a particular scheduling policy [11]. To elaborate, a seek reducing scheduler may wish to wait for contiguous or proximal requests, whereas a proportional-share scheduler may prefer weighted fairness as one of its primary criteria. To allow for such a high degree of flexibility, while trying to minimize the burden on the development efforts for any particular disk scheduler, the anticipatory scheduling framework consists of 3 components [9]. (1) The original disk scheduler, which implements the scheduling policy and is unaware of any anticipatory scheduling techniques. (2) An actual scheduler independent anticipation core. (3) An adaptive scheduler-specific anticipation heuristic for seek reducing (such as SPTF or C-SCAN) as well as any potential proportional-share (CFQ or YFQ) scheduler. The anticipation core implements the generic logic and timing mechanisms for waiting, and relies on the anticipation heuristic to decide if and for how long to wait. The actual heuristic is implemented separately for each disk scheduler, and has access to the internal state of the scheduler. To apply anticipatory scheduling to a new scheduling policy, it is merely necessary to implement an appropriate anticipation heuristic.

Any traditional work-conserving I/O scheduler operates in two states (known as idle and busy). Applications may issue I/O requests at any time, and these requests are normally being placed into the scheduler's *pool of requests*. If the disk subsystem is idle at this point, or whenever another request completes, a new request is being scheduled, the scheduler's *select* function is called, whereupon a request is chosen from the pool and dispatched to the disk device driver. The anticipation core forms a wrapper around this traditional scheduler scheme. Whenever the disk becomes idle, it invokes the scheduler to select a candidate request (still basically following the same philosophy as always). However, instead of dequeuing and dispatching a request immediately, the framework first passes the request to the anticipation heuristic for evaluation. A return value (result) of zero indicates that the heuristic has deemed it pointless to wait and the core therefore proceeds to dispatch the candidate request. However, a positive integer as a return value represents the waiting period in microseconds that the heuristic deems suitable. The core initiates a timeout for that particular time period, and basically enters a new wait state. Though the disk is inactive, this state is considered different from idling (while having pending requests and an active timeout). If the timeout expires before the arrival of any new request, the previously chosen request is dispatched without any further delay. However, new requests may arrive during the wait period and these requests are added to the pool of I/O requests. The anticipation core then immediately requests the scheduler to select a new candidate request from the pool, and initiates communication with the heuristic to evaluate this new candidate. This scenario may lead to an immediate dispatch of the new candidate request, or it may cause the core to remain in the wait state, depending on the scheduler's selection and the anticipation heuristic's evaluation. In the latter case, the original timeout remains in effect, thus preventing unbounded waiting situations by repeatedly re-triggering the timeout.

As the heuristic being used is disk scheduler dependent, the discussion here only generalizes on the actual implementation techniques that may be utilized. Therefore, the next few paragraphs discuss a shortest positioning time first (SPTF) based implementation,

where the disk scheduler determines the positioning time for each available request based on the current head position, and basically chooses the request that results into the shortest seek distance. In general, the heuristic has to evaluate the candidate request that was chosen by the scheduling policy. The intuition is that if the candidate I/O request is located close to the current head position, there is no need to wait on any other requests. Assuming synchronous I/O requests initiated by a single thread, the task that issued the last request is likely to submit the next request soon, and if this request is expected to be close to the current request, the heuristic decides to wait for this request [11]. The waiting period is chosen as the expected *YZ* percentile (normally around 95%) think-time, within which there is a *XZ* probability (again normally 95%) that a request will arrive. This simple approach is transformed and generalized into a succinct cost-benefit equation that is intended to cover the entire range of values for the head positioning, as well as the think-times. To simplify the discussion, the adaptive component of the heuristic consists of collecting online statistics on all the disk requests to estimate the different time variables that are being used in the decision making process. The expected positioning time for each process represents a weighted-average over the time of the *positing time* for requests from that process (as measured upon request completion). Expected median and percentile think-times are estimated by maintaining a *decayed frequency table* of request think-times for each process.

The Linux 2.6 implementation of the anticipatory I/O scheduler follows the basic idea that if the disk drive just operated on a read request, the assumption can be made that there is another read request in the pipeline, and hence it is worth the while to wait [5]. As discussed, the I/O scheduler starts a timer, and at this point there are no more I/O requests passed down to the device driver. If a (close) read request arrives during the wait time, it is serviced immediately and in the process, the actual distance that the kernel considers as *close* grows as time passes (the adaptive part of the heuristic). Eventually the *close* requests will dry out and the scheduler will decide to submit some of the write requests (see Appendix A).

2.2 The 2.6 CFQ Scheduler

The Completely Fair Queuing (CFQ) I/O scheduler can be considered to represent an extension to the better known Stochastic Fair Queuing (SFQ) implementation [12]. The focus of both implementations is on the concept of *fair allocation of I/O bandwidth* among all the *initiators of I/O requests*. An SFQ-based scheduler design was initially proposed (and ultimately being implemented) for some network scheduling related subsystems. The goal to accomplish is to distribute the available I/O bandwidth as equally as possible among the I/O requests. The implementation utilizes *n* (normally 64) internal I/O queues, as well as a single I/O dispatch queue. During an *enqueue* operation, the PID of the currently running process (the actual I/O request producer) is utilized to select one of the internal queues (normally hash based) and hence, the request is basically inserted into one of the queues (in FIFO order). During dequeue, the SFQ design calls for a round robin based scan through the non-empty I/O queues, and basically selects requests from the head of the queues. To avoid encountering too many seek operations, an entire round of requests is collected, sorted, and ultimately merged into the dispatch queue. In a next step, the head request in the dispatch queue is passed to the device driver. Conceptually, a CFQ implementation does not utilize a hash function. Therefore, each I/O process gets an internal queue assigned (which implies that the number of I/O processes determines the number of internal queues). In Linux 2.6.5, the CFQ I/O scheduler utilizes a hash function (and a certain amount of request queues) and therefore resembles an SFQ implementation. The CFQ, as well as the SFQ implementations strives to manage per-process I/O bandwidth, and provide fairness at the level of process granularity.

2.3 The 2.6 noop I/O scheduler

The Linux 2.6 noop I/O scheduler can be considered as a rather minimal overhead I/O scheduler that performs and provides basic merging and sorting functionalities. The main usage of the noop scheduler revolves around non disk-based block devices (such as memory devices), as well as specialized software or hardware environments that incorporate their own I/O scheduling and (large) caching functionality, and therefore require only minimal assistance from the kernel. Therefore, in large I/O subsystems that incorporate RAID controllers and a vast number of contemporary physical disk drives (TCQ drives), the noop scheduler has the potential to outperform the other 3 I/O schedulers as the workload increases.

2.4 I/O Scheduler - Performance Implications

The next few paragraphs augment on the I/O scheduler discussion, and introduce some additional performance issues that have to be taken into consideration while conducting an I/O performance analysis. The current AS implementation consists of several different heuristics and policies that basically determine when and how I/O requests are dispatched to the I/O controller(s). The elevator algorithm that is being utilized in AS *is similar* to the one used for the deadline scheduler. The main difference is that the AS implementation allows limited backward movements (in other words supports backward seek operations) [1]. A backward seek operation may occur while choosing between two I/O requests, where one request is located behind the elevator's current head position while the other request is ahead of the elevator's current position.

The AS scheduler utilizes the *lowest logical block* information as the yardstick for sorting, as well as determining the seek distance. In the case that the seek distance to the request behind the elevator is less than half the seek distance to the request in front of the elevator, the request behind the elevator is chosen. The backward seek operations are limited to a maximum of *MAXBACK* (1024 * 1024) blocks. This approach favors the forward movement progress of the elevator, while still allowing short backward seek operations. The expiration time for the requests held on the FIFO lists is tunable via the parameter's *read_expire* and *write_expire* (see Appendix A). When a read or a write operation expires, the AS I/O scheduler will interrupt either the current elevator sweep or the read anticipation process to service the expired request(s).

2.5 Read and Write Request Batches

An actual I/O batch is described as a set of read or write requests. The AS scheduler alternates between dispatching either read or write batches to the device driver. In a read scenario, the scheduler submits read requests to the device driver, as long as there are read requests to be submitted, and the read batch time limit (*read_batch_expire*) has not been exceeded. The clock on *read_batch_expire* only starts in the case that there are write requests pending. In a write scenario, the scheduler submits write requests to the device driver as long as there are pending write requests, and the write batch time limit *write_batch_expire* has not been exceeded. The heuristic used insures that the length of the write batches will gradually be shortened if there are read batches that frequently exceed their time limit.

When switching between read and write requests, the scheduler waits until all the requests from the previous batch are completed before scheduling any new requests. The read and write FIFO expiration time is only being checked when scheduling I/O for a batch of the corresponding (read or write) operation. To illustrate, the read FIFO timeout values are only analyzed while operating on read batches. Along the same lines, the write FIFO timeout values are only consulted while operating on write batches. Based on the used heuristics and policies, it is generally not recommended to set the read batch time to a higher value than the write expiration time, or to set the write batch time to a greater value than the read expiration time. As the IO scheduler switches from a read to a write batch, the I/O framework launches the elevator with the head request on the write expired FIFO list. Likewise, when switching from a write to a read batch, the I/O scheduler starts the elevator with the first entry on the read expired FIFO list.

2.6 Read Anticipation Heuristic

The process of read anticipation solely occurs when scheduling a batch of read requests. The AS implementation only allows *one read request at a time* to be dispatched to the controller. This has to be compared to either the *many write request* scenario or the *many read request* case if read anticipation is deactivated. In the case that read anticipation is enabled (*antic_expire* $\neq$ *0*), read requests are dispatched to the (disk or RAID) controller one at a time. At the end of each read request, the I/O scheduler examines the next read request from the sorted read list (an actual rb-tree) [1]. If the next read request belongs to the same process as the request that just completed, or if the next request in the queue is close (data block wise) to the just completed request, the request is being dispatched immediately. Otherwise, the statistics (average think-time and seek distance) available for the process that just completed are being examined (cost-benefit analysis). The statistics are associated with each process, but these statistics are not associated with a specific I/O device per se. To illustrate, the approach works more efficiently if there is a one-to-one correlation between a process and a disk. In the case that a process is actively working I/O requests on separate devices, the actual statistics reflect a combination of the I/O behavior across all the devices, skewing the statistics and therefore distorting the facts. If the AS scheduler guesses right, very expensive seek operations can be omitted, and hence the overall I/O throughput will benefit tremendously. In the case that the AS scheduler guesses wrong, the *antic_expire* time is wasted. In an environment that consists of larger (HW striped) RAID systems and tag command queuing (TCQ) capable disk drives, it is more beneficial to dispatch an entire batch of read requests and let the controllers and disk do their magic.

From a physical disk perspective, to locate specific data, the disk drive's logic requires the cylinder, the head, and the sector information [17]. The cylinder specifies the track on which the data resides. Based on the layering technique used, the tracks underneath each other form a cylinder. The head information identifies the specific read/write head (and therefore the exact platter). The search is now narrowed down to a single track on a single platter. Ultimately, the sector value reflects the sector on the track, and the search is completed. Contemporary disk subsystems do not communicate in terms of cylinders, heads and sectors. Instead, modern disk drives map a unique block number over each cylinder/head/sector construct. Therefore, that (unique) reference number identifies a specific cylinder/head/sector combination. Operating systems address the disk drives by utilizing these block numbers (logical block addressing), and hence the disk drive is responsible for translating the block number into the appropriate cylinder/head/sector value. The culprit is that it is *not guaranteed* that the physical mapping is actually sequential. But the statement can be made that there is a rather high probability that a logical block *n* is physically adjacent to a logical block *n+1*. The existence of the discussed sequential layout is paramount to the I/O scheduler performing as

advertised. Based on how the read anticipatory heuristic is implemented in AS, I/O environments that consist of RAID systems (operating in a hardware stripe setup) may experience a rather erratic performance behavior. This is due to the current AS implementation that is based on the notion that an I/O device has only one physical (seek) head, ignoring the fact that in a RAID environment, each physical disk has its own physical seek head construct. As this is not recognized by the AS scheduler, the data being used for the *statistics analysis* is getting skewed. Further, disk drives that support TCQ perform best when being able to operate on *n* (and not 1) pending I/O requests. The read anticipatory heuristic basically disables TCQ. Therefore, environments that support TCQ and/or consist of RAID systems may benefit from either choosing an alternate I/O scheduler or from setting the *antic_expire* parameter to *0*. The tuning allows the AS scheduler to behave similarly to the deadline I/O scheduler (the emphasis is on *behave* and not *performance*).

3 I/O Components that Affect Performance

In any computer system, between the disk drives and the actual memory subsystem is a hierarchy of additional controllers, host adapters, bus converters, and data paths that all impact I/O performance in one way or another [17]. Linux file systems submit I/O requests by utilizing *submit_bio()*. This function submits requests by utilizing the *request function* as specified during *queue creation*. Technically, device drivers do not have to use the I/O scheduler, however all SCSI devices in Linux utilize the scheduler by virtue of the SCSI mid-layer [1]. The *scsi_alloc_queue()* function calls *blk_init_queue()*, which sets the request function to *scsi_request_fn()*. The *scsi_request_fn()* function takes requests from the I/O scheduler (on dequeue), and passes them down to the device driver.

3.1 SCSI Operations

In the case of a simple SCSI disk access, the request has to be processed by the server, the SCSI host adapter, the embedded disk controller, and ultimately by the disk mechanism itself. As the OS receives the I/O request, it converts the request into a SCSI command packet. In the case of a synchronous request, the calling thread surrenders the CPU and transitions into a sleep state until the I/O operation is completed. In a next step, the SCSI command is transferred across the server's I/O bus to the SCSI host adapter. The host adapter is responsible for interacting with the target controller and the respective devices. In a first step, the host adapter selects the target by asserting its control line onto the SCSI-bus (as the bus becomes available). This phase is known as the SCSI selection period. As soon as the target responds to the selection process, the host adapter transfers the SCSI command to the target. This section of the I/O process is labeled as the command phase. If the target is capable of processing the command immediately, it either returns the requested data or the status information.

In most circumstances, the request can only be processed immediately if the data is available in the target controller's cache. In the case of a *read()* request, the data is normally not available. This results into the target disconnecting from the SCSI bus to allow other SCSI operations to be processed. If the I/O operation consists of a *write()* request, the data phase is followed immediately by a command phase on the bus, as the data is transferred into the target's cache. At that stage, the target disconnects from the bus. After disconnecting from the bus, the target resumes its own processing while the bus can be utilized by other SCSI requests. After the physical I/O operation is completed on the target disk, the target controller competes again for the bus, and reconnects as soon as the bus is available. The reconnect phase is followed by a data phase (in the case of *read()* operation) where the data is actually being moved. The data phase is followed by another status phase to describe the results of the I/O operation. As soon as the SCSI host adapter receives the status update, it verifies the proper completion of the request and notifies the OS to interrupt the requesting worker thread. Overall, the simple SCSI I/O request causes 7 phase changes consisting of a select, a command, a disconnect, a reconnect, a data, a status, and a disconnect operation. Each phase consumes time and contributes to the overall I/O processing latency on the system.

3.2 SCSI Disk Fence

When discussing SCSI disks, it is imperative to understand the performance impact of a relatively obscure disk control parameter that is labeled as the fence. When a SCSI disk recognizes a significant delay (such as a seek operation) in a *read()* request, the disk will surrender the bus. At the point where the disk is ready to transfer the data, the drive will again contend for the bus so that the *read()* request can be completed. The fence parameter determines the time at which the disk will begin to contend for the SCSI bus. If the fence is set to 0 (the minimum), the disk will contend for the SCSI bus after the first sector has been transferred into the disk controller's memory. In the case where the fence is set to 255 (the maximum), the disk will wait until almost all the requested data has been accumulated in the controller's memory before contending for the bus.

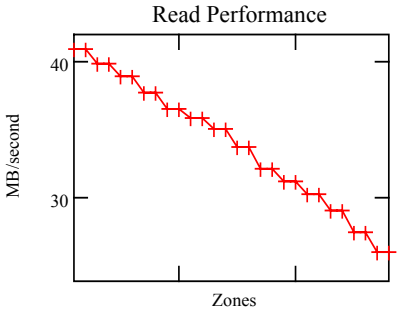
The performance implication of setting the fence to a low value is a reduced response time, but results in a data transfer that happens basically at disk speed. On the other hand, a high fence value will delay the start of the data transfer, but results in a data transfer that occurs at near burst speed. Therefore, in systems with multiple disks per adapter, a high fence value potentially increases overall throughput for I/O intensive workloads. A study by Shriver [15] observed

fairness in servicing sufficiently large I/O requests (in the 16KB to 128KB range), despite the fact that the SCSI disks have different priorities when contending for the bus. Although each process attempts to progress through its requests without any coordination with other processes, a convoy behavior among all the processes was observed. Namely, all disk drives received a request and transmitted the data back to the host adapter before any disk received another request from the adapter (a behavior labeled as *rounds*). The study revealed that the host adapter does not arbitrate for the bus, despite having the highest priority, as long as any disk is arbitrating.

3.3 Zone Bit Recording (ZBR)

Contemporary disk drives utilize a technology called Zone Bit Recording to increase capacity [17]. Incorporating the technology, cylinders are grouped into zones, based on their distance from the center of the disk. Each zone is assigned a number of sectors per track. The outer zones contain more sectors per track compared to the inner zones that are located closer to the spindle. With ZBR disks, the actual data transfer rate varies depending on the physical sector location.

Figure 1: ZBR Throughput Performance



Note: Figure 1 depicts the average throughput per zone, the benchmark revealed 14 distinct performance steps.

Given the fact that a disk drive spins at a constant rate, the outer zones that contain more sectors will transfer data at a higher rate than the inner zones that contain fewer sectors. In this study, evaluating I/O performance on an 18.4 GB Seagate ST318417W disk drive outlined the throughput degradation for sequential *read()* operations based on physical sector location. The ZCAV program used in this experiment is part of the Bonnie++ benchmark suite. Figure 1 outlines the average zone *read()* throughput performance. It has to be pointed out that the performance degradation is not gradual, as the benchmark results revealed 14 clear distinct performance steps along the throughput curve. Another observation derived from the experiment was that for this particular ZBR disk, the outer zones revealed to be wider than the inner zones. The Seagate specifications for this particular disk cite an internal transfer rate of 28.1 to 50.7 MB/second. The measured minimum and maximum throughput *read()* values of 25.99 MB/second and 40.84 MB/second, respectively are approximately 8.1% and 19.5% (13.8% on average) lower, and represent actual throughput rates. Benchmarks conducted on 4 other ZBR drives revealed a similar picture. On average, the actual system throughput rates were 13% to 15% lower than what was cited in the vendor specifications. Based on the conducted research, this text proposes a first-order ZBR approximation nominal disk transfer rate model (for a particular request size *req*) that is defined in Equation 1 as:

$$ntr_{zbr}(req) = 0.85 \cdot (tr_{max}) - \frac{req \cdot (tr_{max} - tr_{min})}{cap} \quad (1)$$

tr_{max} = maximum disk specific internal transfer speed

tr_{min} = minimum disk specific internal transfer speed

The suggested throughput regulation factor of 0.85 was derived from the earlier observation that throughput rates adjusted for factors such as sector overhead, error correction, or track and cylinder skewing issues resulted in a drop of approximately 15% compared to the manufacturer reported transfer rates. This study argues that the manufacturer reported transfer rates could be more accurately defined as *instantaneous bit rates* at the read-write heads. It has to be emphasized that the calculated throughput rates derived from the presented model will have to be adjusted onto the target system's ability to sustain the I/O rate.

The theories of progressive chaos imply that anything that evolves out of a perfect order will over time become disordered due to outside forces. The progressive chaos concept can certainly be applied to I/O performance. The dynamic allocation (as well as de-allocation) of file system resources contributes to the progressive chaos scenario encountered in virtually any file system designs. Form a device driver and physical disk drive perspective, the results of disk access optimization strategies are first, that the number of transactions per second is maximized and second, that the order in which the requests are being received is not necessarily the order the requests are getting processed. Thus, the response time of any particular request can not be guaranteed. A request queue may increase spatial locality by selecting requests in an order to minimize the physical arm movement (a workload transformation), but may also increase the perceived response time because of queuing delays (a behavior transformation). The argument made in this study is that the interrelationship of some the discussed I/O components has to be taken into consideration while evaluating and quantifying performance

4 I/O Schedulers and Performance

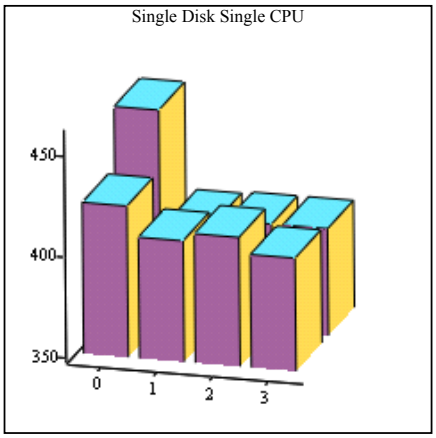
The main goal of this study was to quantify I/O performance (focusing on the Linux 2.6 I/O schedulers) under varying workload scenarios and hardware configurations. Therefore, the benchmarks were conducted on a single-CPU single-disk system, a midrange 8-way NUMA RAID-5 system, and a 16-way SMP system that utilized a 28-disk RAID-0 configuration. The reader is referred to Appendix B for a more detailed description of the different benchmark environments. As a workload generator, the study utilized the flexible file system benchmark (FFSB) infrastructure [8]. FFSB represents a benchmarking environment that allows analyzing I/O performance by simulating basically any I/O pattern imaginable. The benchmarks can be executed on multiple individual file systems, utilizing an adjustable number of worker threads, where each thread may either operate out of a combined or a thread-based I/O profile. Aging the file systems, as well as collecting systems utilization and throughput statistics is part of the benchmarking framework. Next to the more traditional sequential read and sequential write benchmarks, the study used a filer server, a web server, a mail server, as well as a metadata intensive I/O profile (see Appendix B). The file, as well as the mail server workloads (the actual transaction mix) was based on Intel's Iometer benchmark [18], whereas the mail server transaction mix was loosely derived from the SPECmail2001 I/O profile [19]. The I/O analysis in this study was composed of two distinct focal points. One emphasis of the study was on aggregate I/O performance achieved across the 4 benchmarked workload profiles, whereas a second emphasis was on the sequential read and write performance behavior. The emphasis on aggregate performance across the 4 distinct workload profiles is based on the claim made that an I/O scheduler has to provide adequate performance in a variety of workload scenarios and hardware configurations, respectively. All the conducted benchmarks were executed with the default tuning values (if not specified otherwise) in an *ext3* as well as an *xfs* file system environment. In this paper, the term response time represents the total run time of the actual FFSB benchmark, incorporating all the I/O operations that are executed by the worker threads.

5 Single-CPU Single-Disk Setup

The normalized results across the 4 workload profiles revealed that the deadline, the noop, as well as the CFQ schedulers performed within 2% and 1% percent on *ext3* and *xfs* (see Figure 2). On *ext3*, the CFQ scheduler had a slight advantage, whereas on *xfs* the deadline scheduler provided the best aggregate (normalized) response time. On both file systems, the AS scheduler represented the least efficient solution, trailing the other I/O schedulers by 4.6% and 13% on *ext3* and *xfs*, respectively. Not surprisingly, among the 4 workloads benchmarked in a single disk system, AS trailed the other 3 I/O schedulers by a rather significant margin in the Web Server scenario (which reflects 100% random read operations).

On sequential read operations, the AS scheduler outperformed the other 3 implementations by an average of 130% and 127% on *ext3* and *xfs*. The sequential read results clearly support the discussion in this paper on where the design focus for AS was directed. In the case of sequential write operations, AS revealed the most efficient solution on *ext3*, whereas the noop scheduler provided the best throughput on *xfs*. The performance delta (for the sequential write scenarios) among the I/O schedulers was 8% on *ext3* and 2% on *xfs* (see Appendix C).

Figure 2: Aggregate Response Time (Normalized)



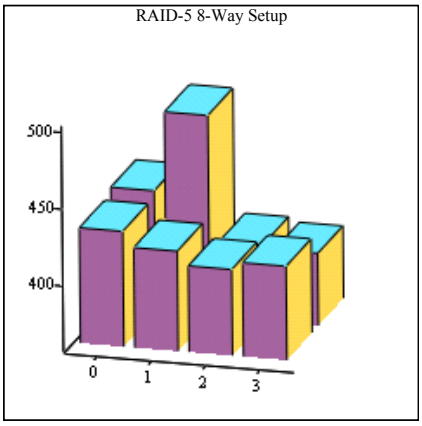
Note: In Figure 2, the x-axis depicts the I/O schedulers, 0 = AS, 1 = deadline, 2 = noop, and 3 = CFQ. The front row reflects the ext3 setup, whereas the back row shows xfs. The y-axis discloses the aggregate (normalized) response time over the 4 benchmarked profiles per I/O scheduler.

5.1 8-Way RAID-5 Setup

In the RAID-5 environment, the normalized response time values (across the 4 profiles) disclosed that the deadline scheduler provided the most efficient solution on *ext3* as well as *xfs* (see Figure 3 and Figure 4). While executing in an *ext3* environment, all 4 I/O schedulers were within 4.5%, with the AS I/O scheduler trailing noop and CFQ by approximately 2.5%. On *xfs*, the study clearly disclosed a profound AS I/O inefficiency while executing the metadata benchmark. The delta among the schedulers on *xfs* was much larger than on *ext3*, as the CFQ, noop, and AS implementations trailed the deadline scheduler by 1%, 6%, and 145%, respectively (see Appendix C). As in the single disk setup, the AS scheduler provided the most efficient sequential read performance. The gap between AS and the other 3 implementations shrunk though rather significantly compared to the single disk scenarios. The average sequential read throughput (for the other 3 schedulers) was approximately 20% less on both *ext3* and *xfs*, respectively. The sequential write performance was dominated by the CFQ scheduler's

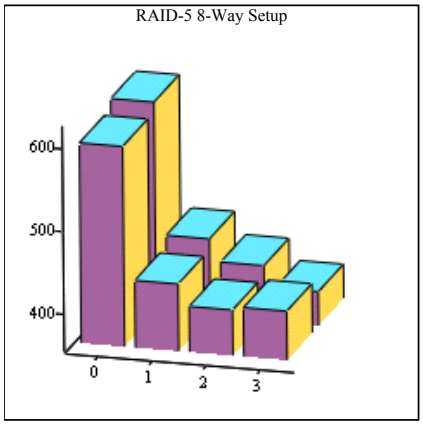
response time that outperformed the other 3 solutions. The delta between the most (CFQ) and the least efficient implementation was 22% (AS) and 15% (noop) on *ext3* and *xfs*, respectively (see Appendix C).

Figure 3: ext3 - Aggregate Response Time (Normalized)



Note: In Figure 3, the x-axis depicts the I/O schedulers, 0 = AS, 1 = noop, 2 = deadline, and 3 = CFQ (for ext3). The front-row reflects the non-tuned, and the back-row the tuned environments. The y-axis discloses the normalized response time (over the 4 profiles) per I/O scheduler.

Figure 4: xfs - Aggregate Response Time (Normalized)



Note: In Figure 3, the x-axis depicts the I/O schedulers, 0 = AS, 1 = noop, 2 = deadline, and 3 = CFQ (for xfs). The front-row reflects the non-tuned, and the back-row the tuned environments. The y-axis discloses the normalized response time (over the 4 profiles) per I/O scheduler.

In a second phase, all the I/O scheduler setups were tuned by adjusting the (per block device) tunable *nr_requests* (I/O operations in fly) from its default value of 128 to 2,560. The results revealed that the CFQ scheduler reacted in a rather positive way to the adjustment, and ergo was capable to provide on *ext3* as well as on *xfs* the most efficient solution. The tuning resulted into decreasing the response time for CFQ in all the conducted (workload profile based) benchmarks on both file systems (see Appendix C). While CFQ benefited from the tuning, the results for the other 3 implementations were inconclusive. Based on the profile, the tuning either resulted in a gain or a loss in performance. As CFQ is designed to operate on larger sets of I/O requests, the results basically reflect the design goals of the scheduler [1]. This is in contrast to the AS implementation, where by design, any read intensive workload can not directly benefit from the change. On the other hand, in the case sequential write operations are being executed, AS was capable of taking advantage of the tuning as the response time decreased by 7% and 8% on *ext3* and *xfs*, respectively. The conducted benchmarks revealed another significant inefficiency behavior in the I/O subsystem, as the write performance (for all the schedulers) on *ext3* was significantly lower (by a factor of approximately 2.1) than on *xfs*. The culprit here is the *ext3* reservation code. Ext3 patches to resolve the issue are available from *kernel.org*.

5.2 16-Way RAID-0 Setup

Utilizing the 28 disk RAID-0 configuration as the benchmark environment revealed that across the 4 workload profiles, the deadline implementation was able to outperform the other 3 schedulers (see Appendix C). It has to be pointed out though that the CFQ, as well as the noop scheduler, slightly outperformed the deadline implementation in 3 out of the 4 benchmarks. Overall, the deadline scheduler gained a substantial lead processing the Web server profile (100% random read requests), outperforming the other 3 implementations by up to 62%. On *ext3*, the noop scheduler reflected the most efficient solution while operating on sequential read and write requests, whereas on *xfs*, CFQ and deadline dominated the sequential read and write benchmarks. The performance delta among the schedulers (for the 4 profiles) was much more noticeable on *xfs* (38%) than on *ext3* (6%), which reflects a similar behavior as encountered on the RAID-5 setup. Increasing *nr_requests* to 2,560 on the RAID-0 system led to inconclusive results (for all the I/O schedulers) on ext3 as well as xfs. The erratic behavior encountered in the tuned, large RAID-0 environment is currently being investigated.

5.3 AS Sequential Read Performance

To further illustrate and basically back up the claim made in Section 2 that the AS scheduler design views the I/O subsystem based on a notion that an I/O device has only one physical (seek) head, this study analyzed the sequential read performance in different hardware setups. The results were being compared to the CFQ scheduler. In the single disk setup, the AS implementation is capable of approaching the capacity of the hardware, and therefore provides optimal throughput performance. Under the same workload conditions, the CFQ scheduler substantially hampers throughput performance, and does not allow the system to fully

utilize the capacity of the I/O subsystem. The described behavior holds true for the *ext3* as well as the *xfs* file system. Hence, the statement can be made that in the case of sequential read operations and CFQ, the I/O scheduler (and not the file system per se) reflects the actual I/O bottleneck. This picture is being reversed as the capacity of the I/O subsystem is being increased.

Table 1: AS vs. CFQ Sequential Read Performance

HW Setup	AS Throughput	CFQ Throughput
Single Disk	52 MB/sec	23 MB/sec
RAID-5	46 MB/sec	39.2 MB/sec
RAID-0	31 MB/sec	158 MB/sec

As depicted in Table 1, the CFQ scheduler approaches first, the throughput of the AS implementation in the benchmarked RAID-5 environment and second, is capable of approaching the capacity of the hardware in the large RAID-0 setup. In the RAID-0 environment, the AS scheduler only approaches approximately 17% of the hardware capacity (180 MB/sec). To reiterate, the discussed I/O behavior is reflected in the *ext3* as well as the *xfs* benchmark results. From any file system perspective, performance should not degrade if the size of the file system, the number of files stored in the file system, or the size of the individual files stored in the file system increases. Further, the performance of a file system is supposed to approach the capacity of the hardware (workload dependent of course). This study clearly outlines that in the discussed workload scenario, the 2 benchmarked file systems are capable of achieving these goals, but only in the case the I/O schedulers are exchanged depending on the physical hardware setup. The fact that the read-ahead code in Linux 2.6 has to operate as efficiently as possible (in conjunction with the I/O scheduler and the file system) has to be considered here as well.

5.4 AS verses deadline Performance

Based on the benchmarked profiles and hardware setups, the AS scheduler provided in most circumstances the least efficient I/O solution. As the AS framework represents an extension to the deadline implementation, this study explored the possibility of tuning AS to approach deadline behavior. The tuning consisted of setting *nr_requests* to 2,560, *antic_expire* to 0, *read_batch_expire* to 1,000, *read_expire* to 500, *write_batch_expire* to 250, and *write_expire* to 5,000. Setting the *antic_expire* value to 0 (by design) basically disables the anticipatory portion of the scheduler. The benchmarks were executed utilizing the RAID-5 environment, and the results were compared to the deadline performance results reported this study. On *ext3*, the non-tuned AS version trailed the non-tuned deadline setup by approximately 4.5% (across the 4 profiles). Tuning the AS scheduler resulted into a substantial performance boost, as the benchmark results revealed that the tuned AS implementation outperformed the default deadline setup by approximately 6.5% (see Appendix C). The performance advantage was squandered though while comparing the tuned AS solution against the deadline environment with *nr_requests* set to 2,560. Across the 4 workload profiles, deadline again outperformed the AS implementation by approximately 17%. As anticipated, setting *antic_expire* to 0 resulted into lower sequential read performance, stabilizing the response time at deadline performance (see Appendix C). On *xfs*, the results were (based on the rather erratic metadata performance behavior of AS) inconclusive. One of the conclusions is that based on the current implementation of the AS code that collects the statistical data, the implemented heuristic is not flexible enough to detect any prolonged random I/O behavior, a scenario where it would be necessary to *deactivate* the *active wait* behavior. Further, setting *antic_expire* to 0 should force the scheduler into deadline behavior, a claim that is not backed up by the empirical data collected for this study. One explanation for the discrepancy is that the short backward seek operations supported in AS are not part of the deadline framework. Therefore, depending on the actual physical disk scheduling policy, the AS backward seek operations may be counterproductive from a performance perspective.

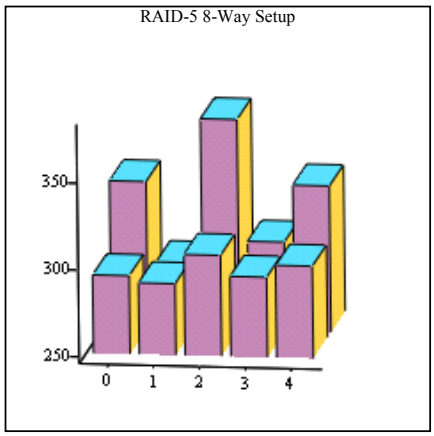
5.5 CFQ Performance

The benchmarks conducted revealed that the tuned CFQ setup provided the most efficient solution for the RAID-5 environment (see Section 5.1). Therefore, the study further explored varies ways to improve the performance of the CFQ framework. The CFQ I/O scheduler in Linux 2.6.5 resembles a SFQ implementation, which operates on a certain number or internal I/O queues and hashes on a per process granularity to determine where to place an I/O request. More specifically, the CFQ scheduler in 2.6.5 hashes on the thread group id (*tgid*), which represents the process *PID* as in *POSIX.1* [1]. The approach chosen was to alter the CFQ code to hash on the Linux *PID*. This code change introduces fairness on a per thread (instead of per process) granularity, and therefore alters the distribution of the I/O requests in the internal queues. In addition, the *cfq_quantum* and *cfq_queued* parameters of the CFQ framework were exported into user space.

In a first step, the default *tgid* based CFQ version with *cfq_quantum* set to 32 (default equals to 8) was compared to the *PID* based implementation that used the same tuning configuration. Across the 4 profiles, the *PID* based implementation reflected the more efficient solution, processing the I/O workloads approximately 4.5% and 2% faster on ext3 and xfs, respectively. To further quantify the performance impact of the different hash methods (*tgid* verses *PID* based), in a second step, the study compared the default Linux 2.6.5 CFQ setup to the *PID* based code that was configured with *cfq_quantum* adjusted to 32

(see Appendix C). Across the 4 profiles benchmarked on *ext3*, the new CFQ scheduler that hashed on a *PID* granularity outperformed the status quo by approximately 10%. With the new method, the sequential read and write performance improved by 3% and 4%, respectively. On *xfs* (across the 4 profiles), the *tgid* based CFQ implementation proved to be the more efficient solution, outperforming the *PID* based setup by approximately 9%. On the other hand, the *PID* based solution was slightly more efficient while operating on the sequential read (2%) and write (1%) profiles. The ramification is that based on the conducted benchmarks and file system configurations, certain workload scenarios can be processed more efficiently in a tuned, *PID* hash based configuration setup.

Figure 5: Mixed Workload Behavior



Note: In Figure 5, the x-axis depicts the I/O schedulers, 0 = default CFQ, 1 = CFQ with PID, 2 = AS, 3 = deadline, and 4 = noop. The front row reflects the xfs, whereas the back row depicts the ext3 based environment. The y-axis discloses the *actual* response time for the mixed workload profile.

To further substantiate the potential of the proposed *PID* based hashing approach, a mixed I/O workload (consisting of 32 concurrent threads) was benchmarked. The environment used reflected the RAID-5 setup. The I/O profile was decomposed in 4 subsets of 8 worker threads, each subset executing either 64KB sequential read, 4KB random read, 4KB random write, or 256KB sequential write operations (see Figure 4). The benchmark results revealed that in this mixed I/O scenario, the *PID* based CFQ solution (tuned with *cfq_quantum* = 32) outperformed the other I/O schedulers by at least 5% and 2% on *ext3* and *xfs*, respectively (see Figure 5 and Appendix C). The performance delta among the schedulers was greater on *ext3* (15%) than on *xfs* (6%).

6 Conclusions and Future Work

The benchmarks conducted on varying hardware configurations revealed a strong (setup based) correlation among the I/O scheduler, the workload profile, the file system, and ultimately I/O performance. The empirical data disclosed that most tuning efforts resulted in reshuffling the scheduler *performance ranking*. The ramification is that the choice of an I/O scheduler has to be based on the workload pattern, the hardware setup, as well as the file system used. To reemphasize the importance of the discussed approach, an additional benchmark was conducted utilizing a Linux 2.6 SMP system, the jfs file system, and a large RAID-0 configuration, consisting of 84 RAID-0 systems (5 disks each). The SPECsfs [20] benchmark was used as the workload generator. The focus was on determining the highest throughput achievable in the RAID-0 setup by only substituting the I/O scheduler between SPECsfs runs. The results revealed that the noop scheduler was able to outperform the CFQ, as well as the AS scheduler. The result reverses the order, and basically contradicts the ranking established for the RAID-5 and RAID-0 environments benchmarked in this study. On the smaller RAID systems, the noop scheduler was not able to outperform the CFQ implementation in any random I/O test. In the large RAID-0 environment, the 84 rb-tree data structures that have to be maintained (from a memory as well as a CPU perspective) in CFQ represent a substantial, noticeable overhead factor.

The ramification is that there is no silver bullet (a.k.a. I/O scheduler) that consistently provides the best possible I/O performance. While the AS scheduler excels on small configurations in a sequential read scenario, the non-tuned deadline solution provides acceptable performance on smaller RAID systems. The CFQ scheduler revealed the most potential from a tuning perspective on smaller RAID-5 systems, as increasing the *nr_requests* parameter provided the lowest response time. As the noop scheduler represents a rather *light-way* solution, large RAID systems that consist of many individual logical devices may benefit from the reduced memory, as well as CPU overhead encountered by this solution. On large RAID systems that consist of many logical devices, the other 3 implementations have to maintain (by design) rather complex data structures as part of the operating framework. Further, the study revealed that the proposed *PID* based and tunable CFQ implementation reflects a valuable alternative to the standard CFQ implementation. The empirical data collected on a RAID-5 system supports that claim, as true fairness on a per thread basis is being introduced.

Future work items include analyzing the rather erratic performance behavior encountered by the AS scheduler on *xfs* while processing a metadata intensive workload profile. Another focal point is an in-depth analysis of the inconsistent *nr_requests* behavior observed on large RAID-0 systems. Different hardware setups will be used to aid this study. The anticipatory heuristics of the AS code used in Linux 2.6.5 is the target of another study, aiming at enhancing the *adaptiveness* of the (status quo) implementation based on certain workload conditions.

Additional research in the area of the proposed *PID* based CFQ implementation, as well as branching the I/O performance study out into even larger I/O subsystems represent other work items that will be addressed in the near future.

Legal Statement

References

1. The Linux Source Code
2. Arcangeli, A., "*Evolution of Linux Towards Clustering*", EFD R&D Clamart, 2003
3. Axboe, J., "*Deadline I/O Scheduler Tunables*", SuSE, EDF R&D, 2003
4. Corbet, J., "*A new deadline I/O scheduler*". http://lwn.net/Articles/10874
5. Corbet, J., "*Anticipatory I/O scheduling*". http://lwn.net/Articles/21274
6. Corbet, J., "*The Continuing Development of I/O Scheduling*", http://lwn.net/Articles/21274
7. Corbet, J., "*Porting drivers to the 2.5 kernel*", Linux Symposium, Ottawa, Canada, 2003
8. Heger, D., Jacobs, J., McCloskey, B., Stultz, J., "*Evaluating Systems Performance in the Context of Performance Paths*", IBM Technical White Paper, Austin, 2000
9. Iyer, S., Drushel, P., "*Anticipatory Scheduling – A disk scheduling framework to overcome deceptive idleness in synchronous I/O*", SOSP 2001
10. Lee Irwin III, W., "*A 2.5 Page Clustering Implementation*", Linux Symposium, Ottawa, 2003
11. Nagar, S., Franke, H., Choi, J., Seetharaman, C., Kaplan, S., Singhvi, N., Kashyap, V., Kravetz, M., "*Class-Based Prioritized Resource Control in Linux*", 2003 Linux Symposium,
12. McKenney, P., "*Stochastic Fairness Queueing*", INFOCOM, 1990
13. Molnar, I., "*Goals, Design and Implementation of the new ultra-scalable O(1) scheduler*". (sched-design.txt).
14. Mosberger, D., Eranian, S., "*IA-64 Linux Kernel, Design and Implementation*", Prentice Hall, NJ, 2002
15. Shriver, E., Merchant, A., Wilkes, J., "*An Analytic Behavior Model with Readahead Caches and Request Reordering",* Bell Labs, 1998.
16. Wienand, I., "*An analysis of Next Generation Threads on IA64*", HP, 2003
17. Zimmermann, R., Ghandeharizadeh, S., "*Continuous Display Using Heterogeneous Disk-Subsystems*", ACM Multimedia, 1997
18. http://www.iometer.org/
19. http://www.specbench.org/osg/mail2001
20. http://www.specbench.org/sfs97r1/docs/chapter1.html

Appendix A: Deadline Tunables

The ***read_expire*** parameter (which is specified in milliseconds) is part of the actual deadline equation. As already discussed, the goal of the scheduler is to insure (basically guarantee) a start service time for a given I/O request. As the design focuses manly on read requests, each actual read I/O that enters the scheduler is assigned a deadline factor that consists of the current time plus the *read_expire* value (in milliseconds).

The ***fifo_batch*** parameter governs the number of request that are being moved to the dispatch queue. In this design, as a read request expires, it becomes necessary to move some I/O requests from the sorted I/O scheduler list into the block device's actual dispatch queue. Hence the *fifo_batch* parameter controls the batch size based on the cost of each I/O request. A request is qualified by the scheduler as either a *seek* or a *stream* request. For additional information, please see the discussion on the *seek_cost* as well as the *stream_unit* parameters.

The ***seek_cost*** parameter quantifies the cost of a seek operation compared to a *stream_unit* (expressed in Kbytes). The ***stream_unit*** parameter dictates how man Kbytes are used to describe a single stream unit. A stream unit has an associated cost of 1, hence if a request consists of XY Kbytes, the actual cost can be determined as cost = (XY + *stream_unit* – 1)/ *stream_unit*. To reemphasize, the combination of the *stream_unit*, *seek_cost*, and *fifo_batch* parameters, respectively, determine how many requests are potentially being moved as an I/O request expires.

The ***write_starved*** parameter (expressed in number of dispatches) indicates how many times the I/O scheduler assigns preference to read over write requests. As already discussed, when the I/O scheduler has to move requests to the dispatch queue, the preference scheme in the design favors read over write requests. However, the write requests can not be staved indefinitely, hence after the read requests were favored for *write_starved* number of times, write requests are being dispatched.

The ***front_merges*** parameter controls the request merge technique used by the scheduler. In some circumstances, a request may enter the scheduler that is contiguous to a request that is already in the I/O queue. It is feasible to assume that the new request may have a correlation to either the front or the back of the already queued request. Hence, the new request is labeled as either a front or a back merge candidate. Based on the way files are laid out, back merge operations are more common than front merges. For some workloads, it is unnecessary to even consider front merge operations, ergo setting the *front_merges* flag to 0 disables that functionality. It has to be pointed out that despite setting the flag to 0, front merges may still happen due to the cached *merge_last* hint component. But as this feature represents an almost 0 cost factor, this is not considered as an I/O performance issue.

AS Tunables

The parameter ***read_expire*** governs the timeframe until a read request is labeled as *expired*. The parameter further controls to a certain extent the interval in-between *expired* requests are serviced. This approach basically equates to determining the timeslice a single reader request is allowed to use in the general presence of other I/O requests. The approximation 100 * ((*seek time* / *read_expire*) + 1) describes the percentile of *streaming read efficiency* a physical disk should receive in a environment that consists of multiple concurrent read requests.

The parameter ***read_batch_expire*** governs the time assigned to a batch (or set) of read requests prior to serving any (potentially) pending write requests. Obviously, a higher value increases the priority allotted to read requests. Setting the value to less than *read_expire* would reverse the scenario, as at this point the write requests would be favored over the read requests. The literature suggests setting the parameter to a multiple of the *read_expire* value. The parameters ***write_expire*** and ***write_batch_expire***, respectively, describe and govern the above-discussed behavior for any (potential) write requests.

The ***antic_expire*** parameter controls the maximum amount of time the AS scheduler will *idle* before moving on to another request. The literature suggests initializing the parameter slightly higher for *large seek time* devices.

Appendix B: Benchmark Environment

The benchmarking was performed in a Linux 2.6.4 environment. For this study, the CFQ I/O scheduler was back-ported from Linux 2.6.5 to 2.6.4.

1. 16-way 1.7Ghz Power4+™ IBM p690 SMP system configured with 4GB memory. 28 15,000-RPM SCSI disk drives configured in a single RAID-0 setup that used Emulex LP9802-2G Fiber controllers (1 in use for the actual testing). System was configured with the Linux 2.6.4 operating system.

2. 8-way NUMA system. IBM x440 with Pentium™ IV Zeon 2.0GHz processors and 512KB L2 cache subsystem. Configured with 4 qla2300 fiber-cards (only one was used in this study). The I/O subsystem consisted of 2 FAStT700 I/O controllers and utilized 15,000-RPM SCSI 18GB disk drives. The system was configured with 1GB of memory, setup as a

RAID-5 (5 disks) configuration, and used the Linux 2.6.4 operating system.

3. Single CPU system. IBM x440 (8-way, only one CPU was used in this study) with Pentium IV Zeon 1.5GHz processor, and 512k L2 cache subsystem. The system was configured with a Adaptec aic7899 Ultra160 SCSI adapter and a single 10,000 RPM 18GB disk. The system used the Linux 2.6.4 operating system and was configured with 1GB of memory.

Workload Profiles

1. ***Web Server Benchmark***. The benchmark utilized 4 worker threads per available CPU. In a first phase, the benchmark created several hundred thousand files ranging from 4KB to 64KB. The files were distributed across 100 directories, The goal of the create phase was to exceed the size of the memory subsystem by creating more files than what can be cached by the system in RAM. Each worker thread executed 1,000 random read operations on randomly chosen files. The workload distribution in this benchmark was derived from Intel's Iometer benchmark.

2. ***File Server Benchmark***. The benchmark utilized 4 worker threads per available CPU. In a first phase, the benchmark created several hundred thousand files ranging from 4KB to 64KB. The files were distributed across 100 directories. The goal of the create phase was to exceed the size of the memory subsystem by creating more files than what can be cached by the system in RAM. Each worker thread executed 1,000 random read or write operations on randomly chosen files. The ratio of read to write operations on a per thread basis was specified as 80% to 20%, respectively. The workload distribution in this benchmark was derived from Intel's Iometer benchmark.

3. ***Mail Server Benchmark***. The benchmark utilized 4 worker threads per available CPU. In a first phase, the benchmark created several hundred thousand files ranging from 4KB to 64KB. The files were distributed across 100 directories. The goal of the create phase was to exceed the size of the memory subsystem by creating more files than what can be cached by the system in RAM. Each worker thread executed 1,000 random read, create, or delete operations on randomly chosen files. The ratio of read to create to delete operations on a per thread basis was specified as 40% to 40% to 20%, respectively. The workload distribution in this benchmark was (loosely) derived from the SPECmail2001 benchmark.

4. ***MetaData Benchmark***. The benchmark utilized 4 worker threads per available CPU. In a first phase, the benchmark created several hundred thousand files ranging from 4KB to 64KB. The files were distributed across 100 directories. The goal of the create phase was to exceed the size of the memory subsystem by creating more files than what can be cached by the system in RAM. Each worker thread executed 1,000 random create, write (append), or delete operations on randomly chosen files. The ratio of create to write to delete operations on a per thread basis was specified as 40% to 40% to 20%.

(i) ***Sequential Read Benchmark***. The benchmark utilized 4 worker threads per available CPU. In a first phase, the benchmark created several hundred 50MB files in a single directory structure. The goal of the create phase was to exceed the size of the memory subsystem by creating more files than what can be cached by the system in RAM. Each worker thread executed 64KB sequential read operations, starting at offset 0 reading the entire file up to offset 5GB. This process was repeated on a per worker thread basis 20 times on randomly chosen files.

(ii) ***Sequential Write (Create) Benchmark***. The benchmark utilized 4 worker threads per available CPU. Each worker thread executed 64KB sequential write operations up to a target file size of 50MB. This process was repeated on a per worker-thread basis 20 times on newly created files.

Appendix C: Raw Data Sheets (Mean Response Time in Seconds over 3 Test Runs)

Table 2: Single Disk Single CPU – Mean Response Time in Seconds (AS, deadline, noop, CFQ)

	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
File Server	610.9	574.6	567.7	579.1	613.5	572.9	571.3	569.9
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
MetaData	621	634.1	623.6	597.5	883.8	781.8	773.3	771.7
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Web Server	531.4	502.1	498.3	486.8	559	462.7	461.6	462.9
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Mail Server	508.9	485.3	522.5	505.5	709.3	633	648.5	650.4
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Seq. Read	405	953.2	939.4	945.4	385.2	872.8	881.3	872.4
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Seq. Write	261.3	276.5	269.1	282.6	225.7	222.6	220.9	222.4

Table 3: RAID-5 8-Way Setup – No Tuning - Mean Response Time in Seconds (AS, deadline, noop, CFQ)

	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
File Server	77.2	81.2	86.5	82.7	83.8	90.3	96.6	90.7
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
MetaData	147.8	148.4	133	145.3	205.8	90.8	101.6	100.8
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Web Server	70.2	58.4	66.2	59.2	82.1	81.3	78.8	75.2
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Mail Server	119.2	114.8	115.3	119.3	153.9	92.1	100.7	92.2
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Seq. Read	517.5	631.1	654.1	583.5	515.8	624.4	628.7	604.5
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Seq. Write	1033.2	843.7	969.5	840.5	426.6	422.3	462.6	400.4

Table 4: RAID-5 8-Way Setup – nr_requests = 2,560 - Mean Response Time in Seconds (AS, deadline, noop, CFQ)

	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
File Server	78.3	72.1	87.1	70.7	94.1	75	89.2	76
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
MetaData	127.1	133	137.3	124.9	189.1	101.1	104.6	99.3
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Web Server	62.4	58.8	75.3	57.5	79.4	72.83	80.6	71.7
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Mail Server	110.2	92.9	118.8	99.6	152.5	100.2	95.1	81
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Seq. Read	523.8	586.2	585.3	618.7	518.5	594.8	580.7	594.4
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Seq. Write	968.2	782.9	1757.8	813.2	394.3	395.6	549.9	436.4

Table 5: RAID-5 8-Way – Default AS, Default deadline, and Tuned AS Comparison - Mean Response Time in Seconds

	AS – ext3	DL – ext3	AS Tuned – ext3	AS - xfs	DL - xfs	AS Tuned - xfs
File Server	77.2	81.2	72.1	83.8	90.3	84.5
	AS Default	DL Default	AS Tuned	AS Default	DL Default	AS Tuned
MetaData	147.8	148.4	133.7	205.8	90.8	187.4
	AS Default	DL Default	AS Tuned	AS Default	DL Default	AS Tuned
Web Server	70.2	58.4	62	82.1	81.3	75.9
	AS Default	DL Default	AS Tuned	AS Default	DL Default	AS Tuned
Mail Server	119.2	114.8	103.5	153.9	92.1	140.2
	AS Default	DL Default	AS Tuned	AS Default	DL Default	AS Tuned
Seq. Read	517.5	631.1	634.5	515.8	624.4	614.1
	AS Default	DL Default	AS Tuned	AS Default	DL Default	AS Tuned
Seq. Write	1033.2	843.7	923.4	426.6	422.3	389.1

Table 6: RAID-5 8-Way– Default CFQ, PID Hashed CFQ & cfq_quantum=32, Default CFQ & cfq_quantum=32 - Mean RT

	CFQ – ext3	PID & Tuned – ext3	CFQ Tuned – ext3	CFQ - xfs	PID & Tuned - xfs	CFQ Tuned - xfs
File Server	70.7	71.1	70.6	76	75.9	74.3
	CFQ	PID & Tuned	CFQ Tuned	CFQ	PID & Tuned	CFQ Tuned
MetaData	124.9	122	125.1	99.3	92.9	97.4
	CFQ	PID & Tuned	CFQ Tuned	CFQ	PID & Tuned	CFQ Tuned
Web Server	57.5	55.8	58	71.7	73	72.5
	CFQ	PID & Tuned	CFQ Tuned	CFQ	PID & Tuned	CFQ Tuned
Mail Server	99.6	94.5	93.3	81	93.6	93.3
	CFQ	PID & Tuned	CFQ Tuned	CFQ	PID & Tuned	CFQ Tuned
Seq. Read	618.7	599.5	595.4	594.4	583.7	604.1
	CFQ	PID & Tuned	CFQ Tuned	CFQ	PID & Tuned	CFQ Tuned
Seq. Write	813.2	781.1	758.4	436.4	432.1	414.6

Table 7: RAID-0 16 – Default I/O Schedulers, No Tuning, Mean Response Time (in seconds)

	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
File Server	44.5	40	41.9	40.8	42.5	43	45.9	42.5
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
MetaData	66.7	64.6	66.2	64	101.8	71.7	72.4	66.7
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Web Server	43.4	38.2	37.9	42.9	68.3	42.8	69.3	64.5
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Mail Server	60.3	58.5	58.7	58.1	100.3	66.2	65.8	65.1
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Seq. Read	2582.1	470.4	460.2	510.9	2601.2	541	576.1	511.2
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Seq. Write	1313.8	1439.3	1171.1	1433.5	508.5	506.2	508.5	509.8

Table 8: RAID-5 8-Way – Mixed Workload Behavior (CFQ-T = PID & cfq_quantum = 32), Response Time (in seconds)

	CFQ	CFQ-T	AS	DL	NO
Mixed ext3	334.1	288.1	371.2	301.2	333.5
	CFQ	CFQ-T	AS	DL	NO
Mixed xfs	295	291	308.4	296	302.8