Preemptive RAID Scheduling

Zoran Dimitrijević
Computer Science
UC, Santa Barbara
zoran@cs.ucsb.edu

Raju Rangaswami
Computer Science
UC, Santa Barbara
raju@cs.ucsb.edu

Edward Chang
Electrical Engineering
UC, Santa Barbara
echang@ece.ucsb.edu

ABSTRACT

Emerging video surveillance, environmental monitoring applications, and constantly evolving large scientific setups require large, high-performance, and reliable storage systems with guaranteed real-time data access. These systems are often implemented using redundant arrays of independent disks (RAID). In this paper we investigate the effectiveness of preemptive disk-scheduling algorithms to achieve better quality of service (QoS) in RAID systems. We present an architecture for QoS-aware RAID systems that use Semi-preemptible IO for servicing internal disk IOs. We show *when* and *how* to preempt IOs to improve the overall performance of the RAID system. We evaluate the benefits and estimate the overhead of our approach using a preemptible RAID simulator that we have implemented.

1. INTRODUCTION

Emerging applications such as video surveillance, large-scale sensor networks, and virtual reality require high-capacity, high-bandwidth RAID storage to support high-volume IOs. In addition to high throughput performance, increasing numbers of applications require real-time data delivery or short access time (response time). The deployment of high-bandwidth networks promised by research projects such as OptIPuter[18] will further magnify the access-time bottleneck of RAID. For applications with mission-critical, high-bandwidth data requirements, access-time reduction will inevitably become increasingly important.

What is the worst-case access time, and how can it be mitigated? On an idle disk, the access time is composed of a seek and a rotational delay. However, when the disk is servicing an IO, a new IO must wait at least until after the on-going IO has been completed. If the current IO involves a large volume of data transfer, the access time can be excessive. One solution to reduce the access time is to divide a large IO into a number of small IOs, and permit a high-priority IO to preempt the current IO. Indeed, a priority-based scheduling policy can reduce the access time for a mission-critical IO request. However, priority-based scheduling, if not performed carefully, can incur excessive overhead and thereby degrade the disk throughput unnecessarily.

In this paper, we introduce preemptive RAID scheduling, or Praid. Praid provides *preemption* mechanisms to allow an on-going IO to be preempted [4] and *resumption* mechanisms to resume a preempted IO on the same or a different disk. In addition to the mechanisms, we propose scheduling policies to decide whether and when to preempt, for maximizing the *yield*, or the total value, of the schedule. Since the yield of an IO is application- and user-defined, our scheduler maps external value propositions to internal yields, producing a schedule that can maximize total external value for all IOs, pending and current.

1.1 Illustrative Example

The purpose of this example is to show how preemptive scheduling works, and why it can outperform a traditional priority-based scheduling policy.

Suppose that the disk is servicing a large sequential write IO when a higher priority read IO arrives. The new IO can arrive at either time t_1 or t_2, as depicted in Figure 1. If the write IO has been buffered in a non-volatile RAID buffer[1], the IO can be preempted to service the read IO request. The preempted write IO is delayed, to be serviced at some later time. When the write IO is resumed, additional disk overhead is incurred. We refer to this overhead as a *preemption overhead*.

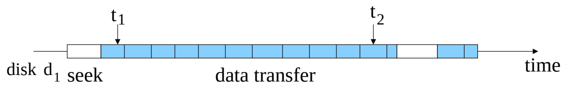

Figure 1: Sequential disk access.

Now, a priority-based scheduler will preempt the long sequential write access (and incur a preemption overhead) regardless of whether the read IO arrives at time t_1 or t_2. However, preempting the write IO at t_2 may not be profitable, since the write IO is nearly completed. Such a preemption is likely to be counter-productive—not gaining much in response time, but incurring preemption overhead. Our Praid scheme is able to discern whether and when a preemption should take place. □

The above example shows just one simple scenario where additional mechanisms and intelligent policies can yield further performance gains for RAID systems. In the rest of the paper, we will detail our preemption mechanisms and policies. Our experiments reported in Section 4 show that our Praid is indeed helpful in improving access time while maintaining high throughput, outperforming traditional schedulers.

1.2 Contributions

In addition to the overall approach, the specific contributions of this paper can be summarized as follows:

- *Preemption mechanisms.* We introduce two methods to preempt disk IOs in RAID systems—JIT-preemption and JIT-migration. Both methods are used by the preemptive schedulers presented in this paper to simplify decisions about the IO preemption.

[1]Most current RAID systems are equipped with a large non-volatile buffer. Write IOs are reported to the operating system as serviced, as soon as the data for the write IO is copied into this buffer.

- *Preemptible RAID policies.* We propose scheduling methods which aim to maximize the total QoS value (each IO is marked with its yield function) and use greedy approaches to decide whether the IO preemption is beneficial or not.

- *System architecture of the preemptible RAID system.* We present an architecture for QoS-aware RAID systems that use Semi-preemptible IO [4] for servicing their internal disk IOs. We implement a simulator for preemptible RAID systems (PraidSim).

The rest of this paper is organized as follows: Section 2 introduces the preemption methods used for preemptive RAID scheduling. Section 3 presents the preemptible-RAID system architecture and the scheduling framework. In Section 4, we present our experimental environment and evaluate different scheduling approaches using simulation. In Section 5 we present related research. We make concluding remarks and suggest directions for future work in Section 6.

2. PREEMPTION METHODS

Applications that access data sequentially, typically issuing large disk IOs of the order of megabytes, are candidates that can benefit from IO preemption capability. Examples of such applications are real-time audio/video streaming, background jobs like disk defragmentation/backup, high data-volume scientific applications, and virtual reality systems.

In this section we introduce methods for IO preemption and resumption. In the next section, we explain how to decide when the IO preemption is desirable and when it is not (scheduling decisions).

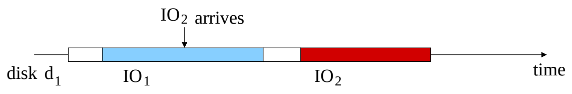

Figure 2: No preemption.

Figure 2 depicts the case when the disk scheduler decides that an ongoing IO (IO_1) should not be preempted. IO_2 can be serviced only after IO_1 completes. During the service of large IOs, a better schedule may become feasible, and IO preemption can lead to better overall performance of a RAID system. The preempted IO may be completed later on the same disk or a different one. Also, the scheduler may choose to drop the ongoing IO request entirely, since its completion may not be necessary or useful in the future (for example, unsuccessful speculative read, cache-prefetch operation, or a missed deadline for the preempted IO).

2.1 Semi-preemptible IO

Semi-preemptible IO [4] maps each IO request into multiple fast-executing (and hence short-duration) disk commands using three methods. (The ongoing IO request can be preempted between these disk commands.) Each of these three methods addresses the reduction of one of the following IO components: $T_{transfer}$ (denoting transfer time), T_{rot} (denoting rotational delay), and T_{seek} (denoting seek time).

1. *Chunking* $T_{transfer}$. A large IO transfer is divided into a number of small chunk transfers, and preemption is made possible between the small transfers. If the IO is not preempted between the chunk transfers, chunking does not incur any overhead. This is due to the prefetching mechanism in current disk drives.

2. *Preempting* T_{rot}. By performing just-in-time (JIT) seek for servicing an IO request, the rotational delay at the destination track is virtually eliminated. The pre-seek slack time thus obtained is preemptible. This slack can also be used to perform prefetching for the ongoing IO request, or/and to perform seek splitting.

3. *Splitting* T_{seek}. Semi-preemptible IO splits a long seek into sub-seeks, and permits preemption between two sub-seeks.

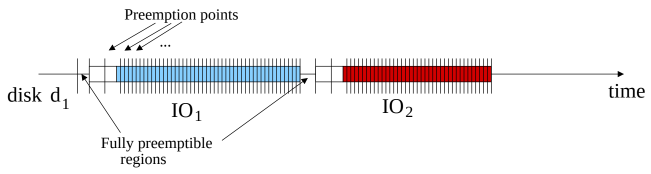

Figure 3: Possible preemption points for semi-preemptible IO.

Figure 3 shows the possible preemption points while servicing a semi-preemptible IO. Preemption is possible only after completion of any disk command or during the disk idle time. The regions before the JIT-seek operation are fully preemptible (since no disk command is issued). The seek operations are the least preemptible, and the data transfer phase is highly preemptible (preemption is possible after servicing each chunk, which is on the order of 0.5 ms).[2]

2.2 Delaying Preempted IOs

When the employed disk scheduler decides that preempting and delaying an ongoing IO would yield a better overall schedule, the IO is preempted using *JIT-preemption* and another IO is scheduled. This is a local decision, meaning that a request for the remaining portion of the preempted IO is placed back in the local queue, and resumed later on the same disk (or dropped completely).

2.2.1 The Method

Definition 2.2: *JIT-preemption* is a method for the preemption of an ongoing semi-preemptible IO at the points that minimize the rotational delay at the destination track (for the higher-priority IO which is serviced next). The scheduler decides when to preempt the ongoing IO using the knowledge about the available JIT-preemption points (these points are roughly one disk rotation apart).

Preemption:

The method relies on JIT-seek (described in Semi-preemptible IO [4]), which requires rotational delay prediction also used in other disk schedulers [10, 12]. JIT-preemption is similar to free-prefetching [12]. However, if the preempted IO will be completed later, then the JIT-preemption always yields useful data transfer (prefetching may or may not be useful). The second difference is that it can also be used for write IOs, although its implementation outside of disk firmware is more difficult for write IOs than it is for the read IOs [4].

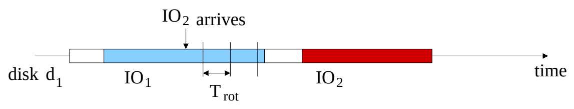

Figure 4: Possible JIT-preemption points.

[2]If we know in advance when to preempt the ongoing IO, we can choose the size for the last data-transfer chunk before preemption, and further tune the desired preemption point.

Figure 4 depicts the positions of possible JIT-preemption points. If IO_1 is preempted between these points, the resulting service time for IO_2 would be exactly the same as if the preemption is delayed until the next possible JIT-preemption point. This is because the rotational delay at the destination track varies depending on when the seek operation starts. The rotational delay is minimal at the JIT-preemption points, which are roughly one disk rotation apart.

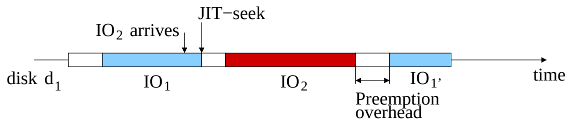

Figure 5: JIT-preemption during data transfer.

Figure 5 depicts the case when the ongoing IO_1 is preempted during its data transfer phase in order to service IO_2. In this case, the first available JIT-preemption point is chosen. The white regions represent the access-time overhead (seek time and rotational delay for an IO). Since JIT-seek minimizes rotational delay for IO_2, access-time overhead is reduced for the case with JIT-preemption (compared to the no-preemption case depicted in Figure 4).

Resumption:

The preemption overhead (depicted in Figure 5) is the additional seek time and rotational delay required to resume the preempted IO_1. Depending on the scheduling decision, IO_1 may be resumed immediately after IO_2 completes, at some later time, or never (it is dropped and does not complete). We explain scheduling decisions in detail later in Section 3.3.

2.3 Migrating Preempted IOs

RAID systems duplicate data for deliberate redundancy. If an ongoing IO can also be serviced at some other disk which holds a copy of the data, the scheduler has the option to preempt the IO and migrate its remaining portion to the other disk. In the traditional static-level RAIDs, this situation can happen in RAID levels 1 and 0/1 [1] (mirrored or mirrored/striped configuration). However, it might also happen in reconfigurable RAID systems (for example, HP AutoRAID [25]), in object-based RAID storage [13], or in non-traditional large-scale storage [8, 17].

2.3.1 The Method

When a scheduler decides to migrate the preempted IO to another disk with a copy of the data, it can choose to favor the ongoing IO (which will be preempted and migrated) or the higher-priority IO. The former uses *JIT-migration* and the latter uses JIT-preemption with migration, introduced earlier.

Definition 2.3: *JIT-migration* is a method for the preemption and migration of an ongoing semi-preemptible IO in a fashion that minimizes the service time for the preempted IO. The ongoing IO is preempted only when the destination disk for the migration is ready to perform data-transfer for the remaining portion of the IO.

Preemption:

Depending on the scheduling decision, we identify two different approaches: (1) use JIT-migration to preempt and migrate the ongoing IO only if it does not increase its service time, and (2) use JIT-preemption to preempt the ongoing IO and migrate its remaining portion to a disk with a replica of the data.

Figure 6 depicts the case when the ongoing IO (IO_1) is more important than the newly arrived IO (IO_2). However, if the disk with the replica is idle or servicing less important IOs, we can still reduce the service time for IO_2. As soon as IO_2 arrives, the scheduler can issue a speculative migration to another disk with a copy of the data. When the data transfer is ready to begin at the other disk, the scheduler can migrate the remaining portion of IO_1 at the desired moment. Since the disks are not necessarily rotating in unison, the IO_1 can be serviced only at approximately the same time when compared with the no-preemption case. The preemption delay for IO_1 depends on the queue at the disk with the replica. If the disk with the replica is idle, the delay will be of the order of 10 ms (equivalent to the access-time overhead).

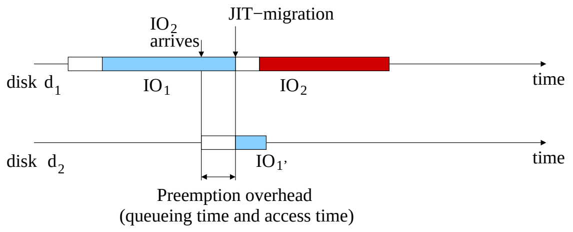

Figure 6: Preemption with JIT-migration.

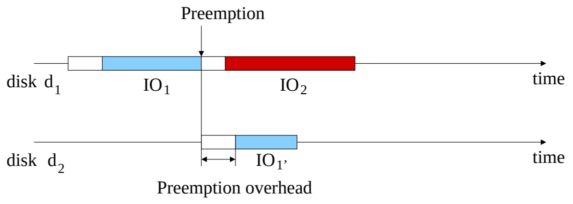

Figure 7: JIT-preemption with migration.

Figure 7 depicts the case when it is possible to use JIT-preemption to promptly service IO_2, while migrating IO_1 to another disk. Preemption overhead is in the form of additional seek time and rotational delay required for the completion of IO_1 at the replica disk.

Resumption:

In the case of JIT-migration, IO_1 is not preempted until the disk with the replica is ready to continue its data transfer. In the case of JIT-preemption with migration, the resumption is at a later time on the replica disk. In both cases, the preemption overhead exists only at the replica disk. This suggests that both these methods are able to improve the schedule in cases when it is hard to achieve an efficient load balancing.

3. PREEMPTIVE RAID SCHEDULING

In this section, we first present a high-level system architecture for RAID systems with the support for preemptive disk scheduling. We then explain the global (RAID) and local (single-disk) scheduling approaches.

3.1 PRAID System Architecture

Figure 8 depicts a simplified architecture of preemptible RAID systems. The main system components are the RAID controller, the attached disks, and the IO interface for servicing external IO requests. The components of the RAID controller are the RAID scheduler, the single-disk schedulers (one for each disk in the array), the RAID reconfiguration manager, and the RAID cache (including both the volatile read cache and the non-volatile write buffer).

External IOs are issued by the IO scheduler external to the RAID system (for example, the operating system's disk scheduler). These IOs are tagged with their QoS requirements, so that the RAID scheduler can optimize their scheduling. The external IOs may also

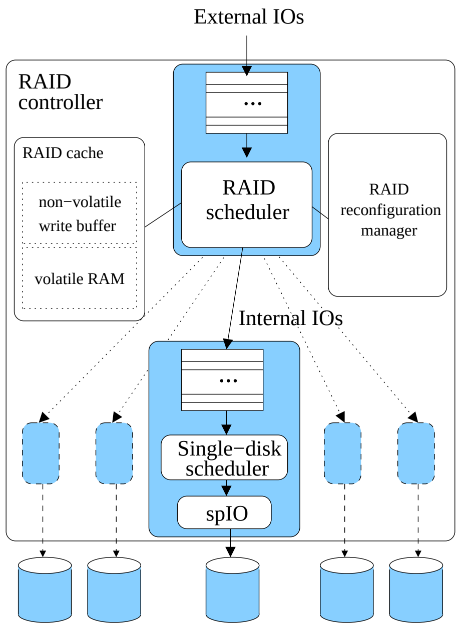

Figure 8: A simplified Preemptible RAID architecture.

be left untagged, making them best-effort IOs. We have extended a Linux kernel to enable such an IO interface.

The *RAID scheduler* maps *external IOs* to *internal IOs* and dispatches them to appropriate single-disk scheduling queues. Internal IOs are also generated by the RAID reconfiguration manager for various maintenance, reconfiguration, or failure-recovery procedures.

Internal IOs are IOs which reside in the scheduling queues of individual disks. They are tagged with internally generated QoS-value functions, and serviced using *Semi-preemptible IO*. The RAID scheduler and the local single-disk schedulers reside on the same RAID controller, and communication between them is fast and cheap.[3]

Single-disk schedulers make local scheduling decisions for internal IOs waiting to be serviced at a disk. Internal IOs are semi-preemptible, and single-disk schedulers can decide to preempt ongoing internal IOs. Since the communication between individual disk schedulers is efficient, single-disk schedulers in the same RAID group cooperate to improve the overall QoS-value for the entire system.

The *RAID cache* consists of both volatile memory for caching read IO data and non-volatile memory for buffering write IO data. The non-volatile memory is usually implemented as battery-backed RAM (hence, it does not lose its contents if the power is turned off). Today, most large-scale RAID systems invariably contain a large battery-backed RAM cache.

The *RAID reconfiguration manager* controls and optimizes the internal data organization within the RAID system. For instance, in the HP AutoRAID system [25], the reconfiguration manager can dynamically choose between RAID 0/1 and RAID 5 configurations for selected data to optimize IO performance. Another important function of the reconfiguration manager is to migrate data to a hot-swap in case of a disk failure. These operations of the RAID reconfiguration manager are also a source of internal IOs within the RAID system.

All the scheduling methods presented within this framework are designed to be implemented in the firmware for hardware RAID controllers or in the driver for software RAID systems.

[3]The assumption of efficient communication between the single-disk schedulers holds for most RAID systems implemented as a single box, which is typically the case for current RAID systems. We use this assumption for efficient migration of internal IOs from one disk to another.

3.2 Global RAID Scheduling

The global RAID scheduler is responsible for mapping external IOs to internal IOs and for dispatching internal IOs to appropriate single-disk scheduling queues.

3.2.1 External IOs

In this paper we refer to IO requests generated by a file system outside of the RAID system as external IOs. They can be tagged with the application-specified QoS class or can be left as regular, best-effort requests.[4]

Our approach for providing QoS hints to the disk scheduler is to enable applications to specify desired QoS parameters per each file descriptor. Internally, we pass the pointer to these QoS parameters along with each IO request in the disk queue. After the *open()* system call, file accesses get assigned the default best-effort QoS class. We introduce several new *ioctl()* commands which enable an application to set up different QoS parameters for its open files. These additional *ioctl()* commands are summarized in Table 1.

Ioctl command	Argument	Description
IO_GET_QOS	**struct ucsb_io ***	Get file's QoS
IO_BESTEFFORT		Set best-effort class
IO_QOS_CLASS	**int *class**	Set IO's QoS class
IO_PRIORITY	**int *priority**	Set IO's priority
IO_DEADLINE	**int *deadline**	Set IO's deadline

Table 1: Additional *ioclt()* commands.

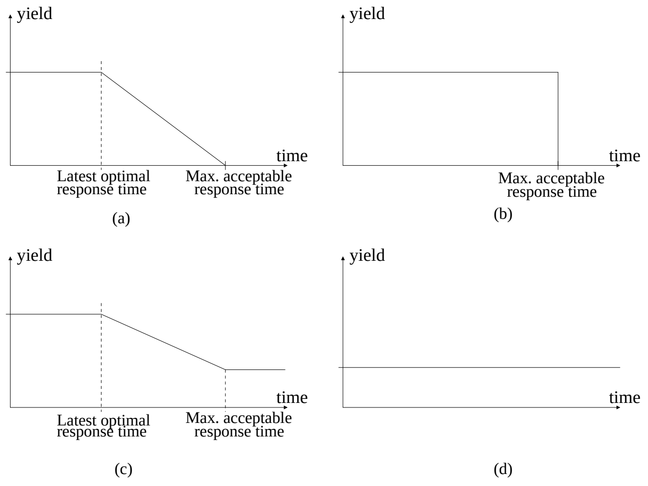

Figure 9: Yield functions: (a) interactive real-time IO, (b) hard real-time IO, (c) interactive best-effort IO, and (d) best-effort IO.

The yield function attached to an external IO determines the QoS value added to the system upon its completion. Figure 9 depicts four possible yield functions that we use in this paper. Functions (a)

[4]Most commodity operating systems still do not provide such an interface. However, several research prototypes have implemented QoS extensions for commodity operating systems [14, 19, 20, 5]

and (b) represent the case when a hard deadline is associated with servicing the IO. If the deadline is missed, the IO should be dropped since its completion does not yield any value.[5] Servicing best-effort IOs always yields some QoS value, and these IOs should not be dropped. We must point out that the yield functions presented here are not the only possible ones. The framework enables specifying one "user-defined" yield function for each QoS class.

To customize the yield function for each external IO ($v_{ext}(t)$), we use the generic yield function for each QoS class ($yield(t)$ depicted in Figure 9) and the four additional parameters. The additional parameters are: the time when the external IO is submitted (t_{start}), the IO size ($size$), the IO priority (p), and the IO deadline (T_{deadln}). In this paper we use a linear approach, giving more value to a larger and higher-priority IO. We customize the yield functions according to the following equation (P_{def} denotes the default priority):

$$v_{ext}(t) = size \times \frac{p - P_{def}}{P_{def}} \times yield\left(\frac{t - t_{start}}{T_{deadln}}\right) . \qquad (1)$$

3.2.2 *RAID Scheduler*

The most important task that the RAID scheduler performs is mapping external IOs to internal IOs. Internal IOs are also generated by the RAID reconfiguration manager, and scheduled to appropriate local-disk queues by the RAID scheduler. Each external IO (*parent IO*) is mapped to a set of internal IOs (*child IOs*). To perform this mapping, the RAID scheduler has to be aware of the low-level placement of data on the RAID system.

The RAID scheduler has a global view of the load on each of the disks in the array. For read IOs, the internal IO can be scheduled to any disk containing a copy of the data. The scheduler can choose the least-loaded disk or use a round-robin strategy. However, for write IOs, the internal IOs are dispatched to all disks where duplicate copies are located. To maintain a consistent view, the non-volatile RAID buffer is not freed until all internal IOs complete.

The RAID scheduler makes the following scheduling decisions to dispatch internal IOs to corresponding local-disk scheduling queues:

- *Read splitting.* To further reduce response time for interactive read requests, the RAID scheduler may split the read request into as many parts as there are disks with copies of the data, issuing each part to a different disk. The read request might be completed faster by utilizing all possible disks. However, this involves more disk-seek overhead.

- *Speculative scheduling.* Apart from dispatching read requests to the least-loaded disk, the RAID scheduler might also dispatch the same request with best-effort priority to other disks which hold a copy of the data requested. This is done in the hope that if a more loaded disk manages to clear its load earlier, then the read request can be serviced sooner. The first speculative read IO to finish removes (or preempts) all other internal IOs for the same data from the other disks' queues.

3.3 Local Disk Scheduling

Using a local disk scheduling algorithm, the single-disk schedulers dispatch internal (semi-preemptible) IOs and decide about IO preemptions.

[5]The option of dropping an IO request at the storage level is not widely used in today's systems. Additional handling might be needed at the user level. However, the current interface need not be changed, since systems can use the existing error-handling mechanisms.

3.3.1 *Internal IOs*

We refer to IO requests generated inside the RAID system as internal IOs. These IOs are generated by the RAID firmware and managed by the RAID system itself. Usually, multiple internal IO requests (for several disks) must be issued to service each external IO. The requests related to data parity management, RAID auto reconfiguration, data migration, and data recovery are independently generated by the RAID reconfiguration manager, and they are not associated with any external IO. Each internal IO is tagged with its own descriptor. The internal IO descriptor is summarized in Table 2. The deadline and the yield function for the parent IO are used to (1) give more local-scheduling priority to earlier deadlines and (2) drop the internal IO after its hard deadline expires.

Attribute	Description
Starting block	Logical number for 1st data block
IO Size	The internal IO size in disk blocks
Parent's IO value	The external IO value (from Eq. 1)
Parent's deadline	The external IO deadline
Parent's IO size	The remaining external IO size

Table 2: Internal IO descriptor.

3.3.2 *Single-disk Scheduler*

For external IOs whose value deteriorates rapidly with time, a disk scheduler may benefit if it preempts less urgent IOs. In traditional systems this is usually accomplished by bounding the size of disk IOs to relatively small values and using priority scheduling. However, this approach has two important shortcomings. First, it greatly increases the number of disk IOs in the scheduling queue, which complicates the implementation of sophisticated QoS-aware schedulers. Second, the schedulers rarely account for the overhead of disrupting sequential disk access, since they do not actually *preempt* the low-level disk IOs.

Here, we present a scheduler that uses an explicitly preemptible approach, and hence does not need to bound the low-level disk IO size. This approach can reduce the number of IOs in the scheduling queue by one or two orders of magnitude (for example, when the preemption is implicit, a single 8 MB IO would be split into eighty 128 kB low-level disk IOs). Also, since IOs are serviced using Semi-preemptible IO, if the scheduler chooses to preempt the ongoing IO, the expected waiting time will be substantially shorter than in the case of traditional schedulers with a bounded size for non-preemptible IOs [4].

The single-disk scheduler maintains a queue of all internal IOs for a particular disk. The components of the internal IO response time are *waiting time* and *service time*. The waiting time is the amount of time that the request spends in the scheduling queue. The service time is the time required by the disk to complete the scheduled request, consisting of the access latency (seek time and rotational delay) and the data transfer time.

Internal scheduling values:

When an internal IO is serviced, its completion yields some QoS value. However, it is hard to estimate this value. First, external QoS value is generated only after the completion of the last internal IO due for a parent external IO. Second, when performing write-back operations for buffered write IOs, their external QoS value has been already harvested. However, not servicing these internal IOs implies that servicing future write IOs will suffer when the write buffer gets filled up. Third, internally generated IOs must be serviced although their completion does not yield any additional external QoS value.

Although we do not know the QoS value generated due to internal read IO, we can estimate it using the following approach. When the scheduler decides to schedule the internal IO, it predicts the IO's service time ($T_{service}$).[6] Let $v_{ext}(t)$ be the value function for the parent IO, as defined in Equation 1. Let $size_{int}$ denote the size of the internal IO, and $size_{remain}$ denote the remaining size of the parent IO. Then, we can estimate the scheduling value for the internal read IO (v_{int_read}) using the following heuristic[7]:

$$v_{int\_read} = v_{ext}(t + T_{service}) \times \frac{size_{int}}{size_{remain}} \ . \qquad (2)$$

Figure 10 depicts the dynamic nature of scheduling value for internal write IOs. Unlike internal read IOs, the scheduling value of internal write IOs do not depend on the value of the corresponding external IOs. Whenever the system services a new external write IO, less space is available in the non-volatile write buffer, and performing write-back operations becomes more important. Hence, the scheduling value for write IOs increases. Whenever the last internal write IO for a particular external IO completes, its data is flushed from the non-volatile buffer, making more space available. This reduces the importance of write-back operations, and thereby decreases the scheduling value for internal write IOs.

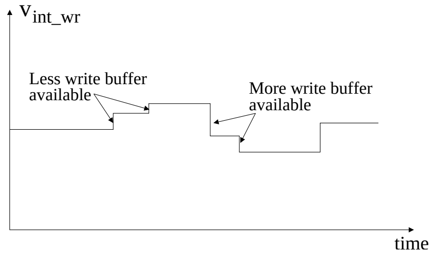

Figure 10: Scheduling value for internal write IOs.

In estimating the scheduling value for internal write IOs, we need to consider both the available non-volatile buffer space and all the pending external write IOs when the buffer is full. Let $I_{wr}(space)$ denote the value of freeing space in the non-volatile buffer (it is a function of the buffer utilization). Let $v^{i}_{ext}(t)$ denote the value of the i^{th} external write IO waiting to be buffered. Let $size_{remain\_wr}$ denote the remaining size of all of the internal IO's siblings (required to flush parent's data from the non-volatile buffer). We use the following heuristic to estimate the scheduling value of the internal write IOs:

$$v_{int\_wr} = \frac{(size_{int})^2}{size_{remain\_wr}} \times (I_{wr}(space) + Max\{v^{i}_{ext}(t)\}) \ . \qquad (3)$$

$I_{wr}(space)$ should assign a low value to write IOs when the buffer is nearly empty, giving higher priority to read IOs. When the buffer is nearly full, $I_{wr}(space)$ should give high value to write IOs, giving higher priority to write-back operations. We use the maximum value of all pending external write IOs to further increase the priority of internal write IOs when the non-volatile buffer is full.

Scheduling:

Scheduling IOs whose service yields various values and incurs differing kinds of overhead is a hard problem. In this paper we do not intend to ascertain which scheduling method is best. We choose a simple greedy approach which, when servicing common IO request types, converges to well-known schedulers. For scheduling decisions, the greedy method uses the scheduling value to calculate the *average yield* of an IO (y_{avg}), defined as

$$y_{avg} = \frac{v_{int\_\{read/wr\}}}{T_{service}} \ . \qquad (4)$$

Thus, the average yield is computed taking into consideration the total time required to service the IO, including the access delay. Figure 11 depicts the average yield (solid line) for two internal IOs serviced by the same disk. The dotted line denotes the yield for the same IOs when distributed over the useful data transfer periods latency. When the scheduler must choose an IO to service next from the queue, it services the IO with the maximum average yield. When all IOs are the same, this greedy approach gives the same schedule as the shortest-access-time-first (SATF) scheduler [10]. When all deadlines are the same, it converges to rate-monotonic scheduling [16].

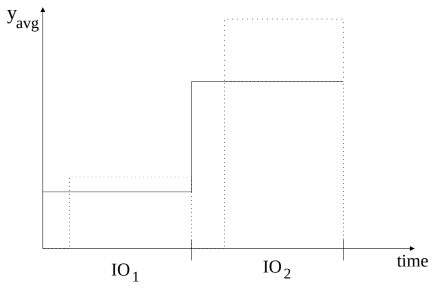

Figure 11: Average yield.

Preemption:

Whenever a new IO arrives, the scheduler checks whether preempting the ongoing IO (using the preemption methods introduced in Section 2), servicing the new IO, and immediately resuming the preempted IO, offers a better average yield than would be obtained without preemption. To calculate the average yield in either case, we must consider the yields due to both IOs. Let the ongoing IO be denoted as IO_1 and the newly arrived IO as IO_2. Let $T^{1}_{service-remain}$ denote the time required to service the remaining portion of IO_1 irrespective of whether it is preempted or not.[8] In either case, we use the following formulation to give us the average yield due to both IOs:

$$y_{avg} = \frac{v^{1}_{int} + v^{2}_{int}}{T^{1}_{service-remain} + T^{2}_{service}} \ . \qquad (5)$$

Notice that although we consider only the remaining time left to service the ongoing IO, we still include its entire yield, as opposed to including only the yield corresponding to the remaining portion of the IO. Indeed, the ongoing IO yields any value *only if* it is serviced entirely. We now present two preemption approaches - *conservative preemption* and *aggressive preemption* - that optimize for the long-term and short-term respectively.

Conservative Preemption:

The conservative approach makes a decision based on a long-term optimization criterion. Only if the preemption of the ongoing IO yields an overall average yield in the long term (given by

[6]Performing this prediction is free since it is already required for successfully implementing Semi-preemptible.

[7]As is usually the case with heuristics, this is just one of several possible.

[8]The value of $T^{1}_{service-remain}$ will be different depending on which case gets instantiated. It will include the preemption overhead in case the IO is preempted.

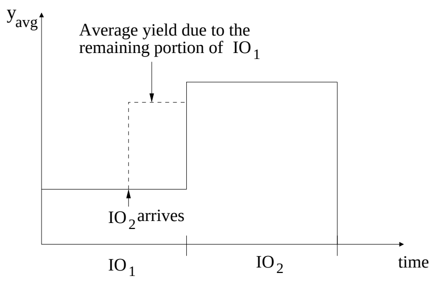

Figure 12: Conservative preemption.

Equation 5) greater than the no preemption case, the ongoing IO is preempted. Figure 12 depicts the case when even though the newly arrived IO (IO_2) offers a greater average yield than that of the remaining portion of the ongoing IO (IO_1), the conservative approach chooses not to preempt the ongoing IO. By not preempting the ongoing IO, an overall greater yield is obtained after both IOs have been serviced.

Aggressive Preemption:

Although the current IO offers a lesser average yield than the newly arrived IO, the conservative approach might conceivably choose not to preempt it. This happens because the conservative approach considers the overall average yield for servicing both IOs before making a decision, taking into consideration the preemption overhead. When the preemption overhead is considered within the framework of Equation 5, by not preempting the current IO (and thus eliminating preemption overhead) we obtain an overall better yield on the completion of the two IOs.

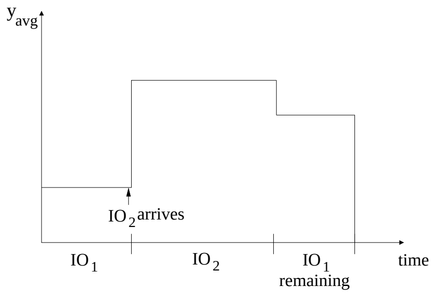

Figure 13: Aggressive preemption.

However, it is also conceivable that additional IO requests arrive in this period with higher priority than the ongoing IO. In this case, the best schedule might be simply to service all the higher priority IOs in the queue before finally servicing the ongoing IO. The aggressive preemption approach preempts the ongoing IO as soon as another IO with a higher average yield arrives. Figure 13 depicts the case when the aggressive approach preempts the ongoing IO in a greedy manner to immediately increase the average yield.

Finally, to support cascading preemptions, we simply return the preempted IO to the scheduling queue. The predicted average yield, according to Equation 4, increases for the remaining portion of preempted IOs for which the data has been partially transfered before preemption. This is necessary in order to maintain the feasibility of the greedy approach, since actual value is generated only after the whole IO completes. This approach also prevents thrashing due to cascading preemptions. Cascading preemptions happen only when the average yield for all IOs in the cascade is maximum.[9]

4. EXPERIMENTAL EVALUATION

In this study we have relied on simulation to validate our preemptive scheduling methods. *Semi-preemptible IO* [4] shows that it is feasible to implement preemption methods necessary for preemptive RAID scheduling outside of disk firmware. In this study we used the previous work in disk modeling and profiling [4, 7, 11] to build an accurate simulator for preemptible RAID systems (PraidSim). We evaluate the PRAID system using several microbenchmarks and for two simulated real-time streaming applications.

4.1 Experimental Setup

We use PraidSim to evaluate preemptive RAID scheduling algorithms. PraidSim is implemented in C++ and uses Disksim [7] to simulate disk accesses. We do not use the RAID simulator implemented in Disksim, but write our own simulator for QoS-aware RAID systems based on the architecture presented in Section 3. PraidSim can either generate a simulated workload for external IOs or perform a trace-driven simulation. We have chosen to simulate only the chunking and JIT-seek methods from *Semi-preemptible IO*. The seek-splitting method only helps in reducing the maximum IO waiting time. The chunking method relies only on optimal chunk size for a particular disk, which is easy to profile for both IDE and SCSI disks [4]. JIT-seek is used for JIT-preemption and has been previously implemented in several schedulers [4, 11].

Parameter name	Description
RAID level	RAID 0, RAID 0/1, or RAID 5
Number of disks	Number of disks in the disk array
Mirrors	Number of mirror disks
Disksim model	Name of the parameter file for Disksim disks
Striping unit	Size of the striping unit in disk blocks (512 B)
Write IOs	Write IO arrival rate and random distribution
Read IOs	Read IO deadlines, arrival rate and rand. dist.
Interactive IOs	Interactive IO arrival rate and rand. dist.
Scheduling	SCAN or FIFO for each IO class
Preemption	Preempt writes, reads, or no preemption
Interactivity	Preemption criteria for interactive IOs
Write priority	Buffer size and dynamic QoS value for writes
Chunk size	Chunk size for *Semi-preemptible IO*

Table 3: Summary of PraidSim parameters.

Table 3 summarizes the configurable parameters in PraidSim. Internal RAID configuration is chosen by specifying the RAID level, number of disks in the array, number of mirror replicas, stripe size, and the name of the simulated disk for Disksim. For this paper we used the Quantum Atlas 10K disk model. The IO arrival rate is specified with the arrival rate and random distribution for write IOs, deadline read IOs, and interactive read IOs; or by specifying a trace file. The next set of parameters is used to specify the PraidSim scheduling algorithm for non-interactive read and write IOs, the preemption decisions, methods for scheduling interactive reads, and dynamic value for internal write IOs. The chunk size parameter specifies the chunk size used to schedule semi-preemptible IOs. For all experiments in this paper we used chunk size of 20 kB.

4.2 Microbenchmarks

Our microbenchmarks aimed to answer the following questions:

- Does *preempting* non-interactive IOs always improve the quality of service?

[9]Since we use a greedy approach, starvation is possible. To handle starvation, we can make a simple modification to our internal scheduling value, forcing it to increase with time.

- How does *preemption* help when interactive operations consist of several IOs in a cascade?
- What is the overhead of preempting and delaying *write* IOs to promptly service read requests?

4.2.1 Preemption Decisions

In order to show that decisions about preempting sequential disk accesses are not trivial for all applications, we performed the following experiment. We varied the size of non-interactive IOs and measured both the response time for interactive IOs and the throughput for non-interactive IOs. We fix the arrival rate for interactive IOs to 10 req/s, and keep disk fully utilized with non-interactive IOs. The size of the interactive requests is 100 kB.

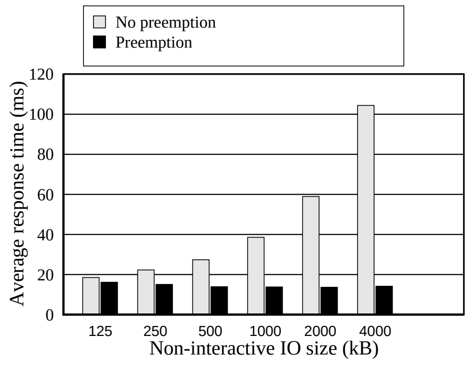

Figure 14: Average response time for interactive IOs vs. non-interactive IO size.

Figure 14 depicts the average response time for interactive IOs for preempt-never and preempt-always approaches. For small IO sizes the benefit of preemption is of the order of $5 - 10$ ms. However, for large non-interactive sequential IOs, the preemption yields improvements of the order of 100 ms. The preemptive approach also provides less variation in response times, which is very important for interactive systems. Figure 15 shows the difference in throughput between the preempt-never and preempt-always approaches. The main question is whether the trade-off between improved response time and reduced throughput yields better quality of service.

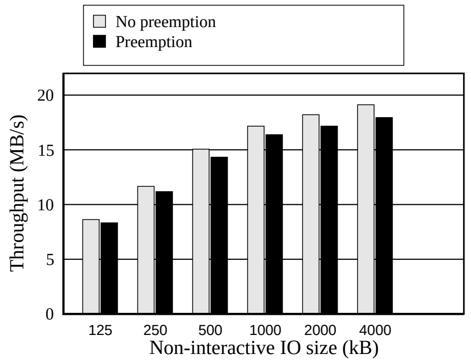

Figure 15: Disk throughput vs. non-interactive IO size.

Figure 16 depicts the improvements in aggregate interactive value (for all external interactive IOs) of the preempt-always over the preempt-never approach. We use a yield function for interactive real-time IOs from Figure 9(a) in Section 3.2.1. If non-interactive IOs are small, the preempt-always approach does not offer any improvement, since all interactive IOs can be serviced before their deadlines even without preemptions. For large sizes of non-iteractive IOs and short (100 ms) deadlines, preempt-always yielded up to 2.8 times more value than the non-preemptive approach (180% improvement). However, even for large non-interactive IOs, if the deadlines are of the order of 200 ms, then the preempt-always approach makes only marginal improvements over the preempt-never approach.

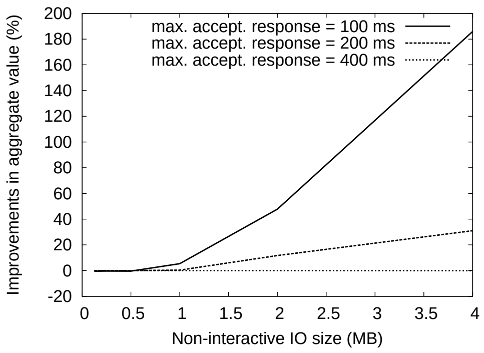

Figure 16: Improvements in aggregate interactive value.

Figure 17 shows the deference in aggregate values for all serviced IOs between the preempt-always and the preempt-never approaches. For cases when the non-interactive requests yield the same as or greater value than the interactive IOs, the preempt-always approach degrades the aggregate value when a disk services small non-interactive IOs (up to approximately 2 MB in this example). For cases when interactive requests are substantially more important than the non-interactive ones, the difference in aggregate value for all IOs converges to the curve presented in Figure 16. Simple priority-based scheduling cannot easily handle both cases.

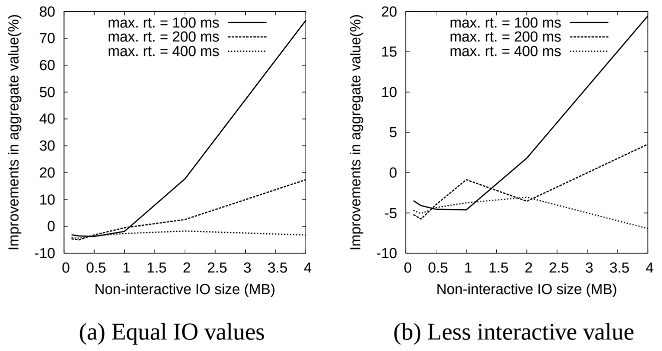

Figure 17: Differences in aggregate values for all IOs between the preempt-always and preempt-never approaches: (a) non-interactive and interactive IOs are equally important and (b) non-interactive IOs are more important (their value is five times greater).

4.2.2 Response Time for Cascading Interactive IOs

In order to show how preemptions help when an interactive operation consists of issuing multiple IO requests in a cascade[10], we performed the following experiment. The background, non-interactive workload consists of both read and write IOs, each external IO being 2 MB long. We use the RAID 0/1 configuration with 8 disks.

[10] In case when an application cannot issue the next IO unless it gets the data for the previous one. A typical example for applications where multiple queries must be performed for a single interactive operation is video surveillance.

The sizes of internal IOs are between 0 and 2 MB and the interactive IOs are 100 kB each. As soon as one interactive IO completes, we issue the next IO in the cascade, measuring the time required to complete all cascading IOs. Figure 18 depicts the effect of cascading interactive IOs on average response time for the whole operation. If the maximum acceptable response time for interactive operations is around 100 ms, the preemptive approach can service six cascading IOs, whereas the non-preemptive approach can service only two.

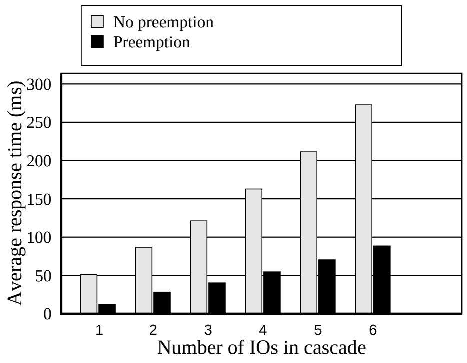

Figure 18: Response time for cascading interactive IOs.

4.2.3 Overhead of Delaying Write IOs

In order to show the overhead of preempting and delaying write IOs (to reduce the latency for read IOs), we performed the following experiment. We varied the arrival rate for read requests and plotted the overhead in terms of reduced RAID idle time and increased buffering requirements. We compared the following three scheduling policies: (1) SCAN scheduling without priorities, (2) SCAN scheduling with priorities for reads but without preemptions, and (3) SCAN scheduling with write preemptions.

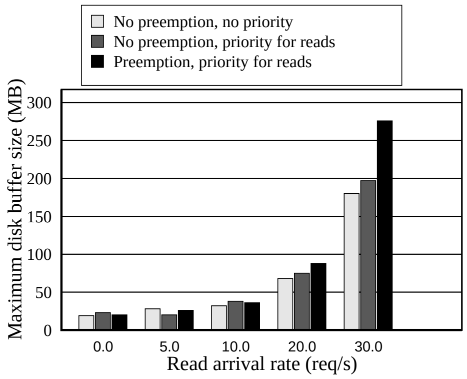

Figure 19: RAID write-buffer requirements.

Figure 19 depicts the RAID write-buffer requirements for different read arrival rates. In this case, we used RAID level 0/1, 4+4 disks, each external read IO was 1 MB, and the external write rate was 50 MB/s (100 MB/s internally). We can see that independently of the scheduling criteria, if the available disk idle time is small, then the required buffer size increases exponentially. This is a well-known property of real-time streaming systems. The additional write-buffer requirement is acceptable for a range of read arrival rates in the system with preemptions. The real system needs to control the number of preemptions as well as the read/write priorities depending on available RAID idle time. Figure 20 depicts the average disk idle-time for different read arrival rates.

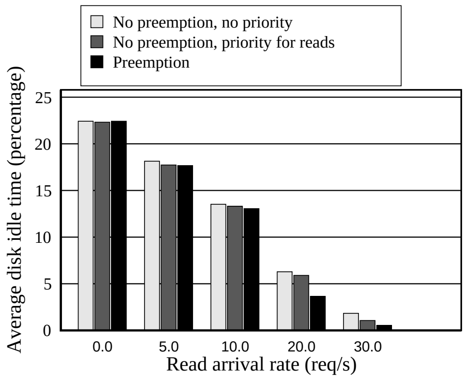

Figure 20: Average RAID idle-time.

4.3 Write-intensive Real-time Applications

In this section we discuss the benefits of using the preemptive RAID scheduling for write-intensive real-time streaming applications. We generated a workload similar to that of a video surveillance system which services read and write streams with real-time deadlines. In addition to IOs for real-time streams, we also generate interactive read IOs. We present results for a typical RAID 0/1 (4+4 disks) configuration with a real-time write rate of 50 MB/s (internally 100 MB/s) and a real-time read rate of 10 MB/s. Interactive IO arrival rate is 10 req/s. The external non-interactive IOs are 2 MB each, and interactive IOs are 1 MB each. The workload corresponds to a video surveillance system with 50 dvd-quality write video streams, 20 real-time read streams, and 10 interactive operations performed each second.

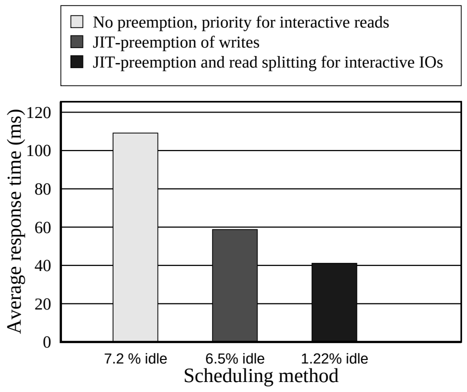

Figure 21: Average interactive read response times.

Figure 21 depicts the improvements in the response times for interactive IOs and the overhead in reduced RAID idle time. The system was able to satisfy all real-time streaming requirements in all four cases. Using the JIT-preemption method, our system decreased the interactive response time from 110 ms to 60 ms, by reducing the RAID idle-time from 7.2% to 6.5%. The read-splitting method from Section 3.2.2 further decreases the response time (by reducing the data-transfer component on a single disk) with the substantially larger effect on reduced average disk idle time.

4.4 Read-intensive Applications

Figure 22 depicts the average response times for interactive read requests for read-intensive real-time streaming applications. The setup is the same as for write-intensive applications in the previous section, but the system services only read IOs. The streaming rate for non-interactive reads is 129 MB/s. The interactive IOs

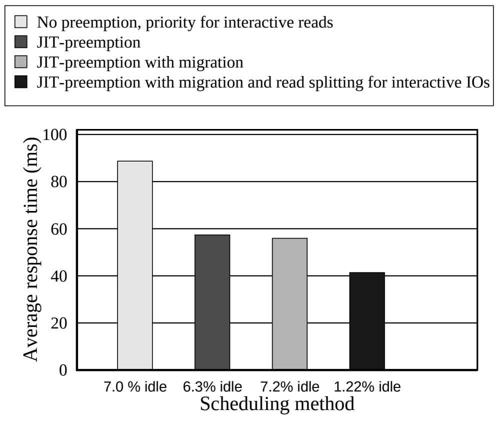

Figure 22: Average interactive read response times.

are 1 MB each, and their arrival rate is 10 req/s. The improvements in average response times were similar to those in our write-intensive experiment. The JIT-preemption with migration didn't substantially improve the average response for interactive IOs, but the better load-balancing compensated for the JIT-preemption reduction in idle time.

5. RELATED WORK

Before the pioneering work of [3, 14, 21], it was assumed that the nature of disk IOs was inherently non-preemptible. Preemptible RAID scheduling is based on detailed knowledge of low-level disk characteristics. A number of scheduling approaches rely on these low-level characteristics [4, 9, 11, 15]. RAID storage was the focus of a number of important studies including [2, 6, 21, 22, 25]. In his recent keynote speech at FAST 2003, John Wilkes et al. [23, 24] stressed the importance of providing quality-of-service scheduling in storage systems.

While most current commodity operating systems do not provide sufficient support for real-time disk access, several research projects are committed to implementing real-time support for commodity operating systems [14, 19]. Molano et al. [14] presented their design and implementation of a real-time file system for RT-Mach. Sundaram et al. [19] presented their QoS extensions for Linux operating system (QLinux).

6. CONCLUSION

In this paper we have investigated the effectiveness of IO preemptions to provide better disk scheduling for RAID storage systems. We first introduced methods for preemptions and resumptions of disk IOs—JIT-preemption and JIT-migration. We then proposed an architecture for QoS-aware RAID systems and a framework for preemptive RAID scheduling. We implemented a simulator for such systems (PraidSim). Using simulation, we evaluated benefits and estimated the overhead involved with the preemptive scheduling decisions. Our evaluation showed that using IO preemptions can provide better overall system QoS for several important applications.

We plan to further this work in the following two directions. First, based on the existing Linux QoS extensions, we plan to implement the preemptive scheduler for software RAIDs. Second, we plan to investigate the effectiveness of preemptive scheduling approaches in cluster-based storage systems.

7. REFERENCES

[1] 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*, 26(2):145–185, June 1994.

[2] 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*, 26(2):145–185, June 1994.

[3] S. J. Daigle and J. K. Strosnider. Disk scheduling for multimedia data streams. *Proceedings of the IS&T/SPIE*, February 1994.

[4] Z. Dimitrijevic, R. Rangaswami, and E. Chang. Design and implementation of Semi-preemptible IO. *Proceeding of Usenix FAST*, March 2003.

[5] Z. Dimitrijevic, R. Rangaswami, M. Sang, K. Ramachandran, and E. Chang. UCSB-IO: Linux kernel extensions for QoS disk access. *http://www.cs.ucsb.edu/∼zoran/ucsb-io*, 2003.

[6] A. L. Drapeau, K. Shirriff, E. K. Lee, J. H. Hartman, E. L. Miller, S. Seshan, R. H. Katz, K. Lutz, and D. A. Patterson. RAID-II: A high-bandwidth network file server. *Proceedings of the ACM ISCA*, 1994.

[7] G. R. Ganger, B. L. Worthington, and Y. N. Patt. The disksim simulation environment version 2.0 reference manual. *Reference Manual*, December 1999.

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *ACM SOSP*, October 2003.

[9] S. Iyer and P. Druschel. Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous I/O. *18th Symposium on Operating Systems Principles*, September 2001.

[10] D. M. Jacobson and J. Wilkes. Disk scheduling algorithms based on rotational position. *HPL Technical Report*, February 1991.

[11] C. R. Lumb, J. Schindler, and G. R. Ganger. Freeblock scheduling outside of disk firmware. *Proceedings of the Usenix FAST*, January 2002.

[12] C. R. Lumb, J. Schindler, G. R. Ganger, and D. F. Nagle. Towards higher disk head utilization: Extracting free bandwith from busy disk drives. *Proceedings of the OSDI*, 2000.

[13] M. Mesnier, G. R. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, August 2003.

[14] A. Molano, K. Juvva, and R. Rajkumar. Guaranteeing timing constraints for disk accesses in RT-Mach. *Proceedings of the Real Time Systems Symposium*, 1997.

[15] F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Robust, portable I/O scheduling with the disk mimic. *USENIX 2003 Annual Technical Conference*, June 2003.

[16] L. Sha, R. Rajkumar, and S. Sathaye. Generalized rate monotonic scheduling theory, a framework of developing real-time systems. *Proceedings of The IEEE*, January 1994.

[17] M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Evolving RPC for active storage. *Proceedings of ASPLOS-X*, 2002.

[18] L. L. Smarr, A. A. Chien, T. DeFanti, J. Leigh, and P. M. Papadopoulos. The OptIPuter. *Communications of the ACM*, November 2003.

[19] V. Sundaram, A. Chandra, P. Goyal, P. Shenoy, J. Sahni, and H. Vin. Application performance in the qlinux multimedia operating system. *ACM Multimedia*, 2000.

[20] E. Thereska, J. Schindler, J. Bucy, B. Salmon, C. R. Lumb, and G. R. Ganger. A framework for building unobtrusive disk maintenance applications. *Proceedings of the Third Usenix FAST*, March 2004.

[21] A. Thomasian. Priority queueing in raid5 disk arrays with an nvs cache. *Proceedings of MASCOTS*, 1995.

[22] F. Tobagi, J. Pang, R. Baird, and M. Gang. Streaming raid-a disk array management system for video files. *First ACM Conference on Multimedia*, August 1993.

[23] J. Wilkes. Traveling to Rome: QoS specifications for automated storage system management. *Proceedings of Intl. Workshop on Quality of Service (IWQoS'2001)*, June 2001.

[24] J. Wilkes. Data services – from data to containers. *Keynote speech at Usenix FAST*, March 2003.

[25] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–36, February 1996.